

**RT-11 System  
Reference Manual**

For additional copies, order No. DEC-11-ORUGA-A-D  
from Software Distribution Center, Digital Equipment  
Corporation, Maynard, Mass.

The "HOW TO OBTAIN SOFTWARE INFORMATION" page, located at the back of this document, explains the various services available to DIGITAL software users.

The postage prepaid "READER'S COMMENTS" form on the last page of this document requests the user's critical evaluation. All comments received are acknowledged and will be considered when subsequent documents are prepared.

Copyright (C) 1973 by Digital Equipment Corporation

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 manual.

The software described in this document is furnished to the purchaser under a license for use on a single computer system and can be copied (with inclusion of DIGITAL's copyright notice) only for use in such system, except as may otherwise be provided in writing by DIGITAL.

Digital Equipment Corporation assumes no responsibility for the use or reliability of its software on equipment that is not supplied by DIGITAL.

The following are trademarks of Digital Equipment Corporation, Maynard, Massachusetts:

CDP	DIGITAL	KA10	PS/8
COMPUTER LAB	DNC	LAB-8	QUICKPOINT
COMTEX	EDGRIN	LAB-8/e	RAD-8
COMSYST	EDUSYSTEM	LAB-K	RSTS
DDT	FLIP CHIP	OMNIBUS	RSX
DEC	FOCAL	OS/8	RTM
DECCOMM	GLC-8		SABR
DECTAPE	IDAC	PDP	TYPESET 8
DIBOL	IDACS	PHA	UNIBUS
	INDAC		

## PREFACE

The RT-11 System Reference Manual provides the information necessary for the user of utility programs and the experienced system programmer.

Chapter 1 covers the overview and monitor keyboard commands necessary to implement user programs. The utility programs (Editor, PIP, MACRO, Linker, ODT) are described in Chapters 3 through 7.

Chapter 8, Programmed Requests, is of particular interest to the experienced programmer who wishes to make use of the services of the monitor in an assembly language program.

Chapters 9 and 10 explain the 8K Assembler and Expand programs. The Appendices summarize RT-11 commands and error messages previously described and introduce the PIPC and PATCH programs.

BASIC/RT11 language features are summarized in Appendix L. A more detailed explanation is available in the BASIC/RT11 Language Reference Manual, DEC-11-ORBMA-A-D.



## CONTENTS

CHAPTER 1.	RT-11 OVERVIEW	1-1
1.1	HARDWARE CONFIGURATION	1-2
1.2	SYSTEM SOFTWARE COMPONENTS	1-2
CHAPTER 2.	MONITOR	2-1
2.1	START PROCEDURE	2-1
2.2	KEYBOARD COMMUNICATION	2-3
2.2.1	Commands to Allocate System Resources	2-4
2.2.1.1	DATE Command	2-4
2.2.1.2	INITIALIZE Command	2-4
2.2.1.3	ASSIGN Command	2-5
2.2.1.4	CLOSE Command	2-6
2.2.2	Commands to Manipulate Core Images	2-6
2.2.2.1	GET Command	2-6
2.2.2.2	EXAMINE Command	2-7
2.2.2.3	DEPOSIT Command	2-7
2.2.2.4	BASE Command	2-8
2.2.2.5	SAVE Command	2-8
2.2.3	Commands to Start a Program	2-10
2.2.3.1	RUN Command	2-10
2.2.3.2	START Command	2-11
2.2.3.3	REENTER Command	2-11
2.3	RT-11 FILE SPECIFICATION	2-12
2.3.1	Physical Device Names	2-12
2.3.2	File Names and Extensions	2-13
2.4	ERROR MESSAGES	2-15
CHAPTER 3	TEXT EDITOR	3-1
3.1	CALLING AND USING EDIT	3-1
3.2	COMMANDS	3-2
3.2.1	Key Commands	3-2
3.2.2	Editing Commands	3-3
3.2.2.1	Command Repetition	3-8
3.2.2.2	Core Usage	3-10
3.2.2.3	Input/Output Commands	3-10
3.2.2.4	Commands to Move Location Pointer	3-15
3.2.2.5	Search Commands	3-17
3.2.2.6	Commands to Modify the Text	3-19
3.2.2.7	Utility Commands	3-23

3.3	ERROR MESSAGES	3-27
3.4	EDIT EXAMPLE	3-29
CHAPTER 4	PERIPHERAL INTERCHANGE PROGRAM (PIP)	4-1
4.1	CALLING AND USING PIP	4-1
4.2	PIP COMMANDS	4-2
4.2.1	Copy Commands	4-4
4.2.2	Multiple Copy Commands	4-5
4.2.3	Delete Command	4-7
4.2.4	Rename Command	4-8
4.2.5	Extend Command	4-9
4.2.6	Directory List Commands	4-9
4.2.7	Directory Initialization Command	4-11
4.2.8	Compress Command	4-12
4.2.9	Bootstrap Copy Command	4-12
4.2.10	Boot Command	4-13
4.2.11	Version Command	4-13
4.3	ERROR MESSAGES	4-13
CHAPTER 5	MACRO ASSEMBLER	5-1
5.1	SOURCE PROGRAM FORMAT	5-1
5.1.1	Statement Format	5-2
5.1.1.1	Label Field	5-2
5.1.1.2	Operator Field	5-3
5.1.1.3	Operand Field	5-4
5.1.1.4	Comment Field	5-4
5.1.2	Format Control	5-4
5.2	SYMBOLS AND EXPRESSIONS	5-5
5.2.1	Character Set	5-5
5.2.1.1	Separating and Delimiting Characters	5-6
5.2.1.2	Illegal Characters	5-7
5.2.1.3	Operator Characters	5-8
5.2.2	MACRO Symbols	5-9
5.2.2.1	Permanent Symbols	5-9
5.2.2.2	User-Defined and MACRO Symbols	5-9
5.2.3	Direct Assignment	5-11
5.2.4	Register Symbols	5-12
5.2.5	Local Symbols	5-13
5.2.6	Assembly Location Counter	5-14
5.2.7	Numbers	5-16
5.2.8	Terms	5-17
5.2.9	Expressions	5-18
5.3	RELOCATION AND LINKING	5-19
5.4	ADDRESSING MODES	5-20
5.4.1	Register Mode	5-20

5.4.2	Register Deferred Mode	5-21
5.4.3	Autoincrement Mode	5-21
5.4.4	Autoincrement Deferred Mode	5-22
5.4.5	Autodecrement Mode	5-22
5.4.6	Autodecrement Deferred Mode	5-22
5.4.7	Index Mode	5-22
5.4.8	Index Deferred Mode	5-23
5.4.9	Immediate Mode	5-23
5.4.10	Absolute Mode	5-23
5.4.11	Relative Mode	5-24
5.4.12	Relative Deferred Mode	5-24
5.4.13	Table of Mode Forms and Codes	5-25
5.4.14	Branch Instruction Addressing	5-25
5.4.15	EMT and TRAP Addressing	5-26
5.5	ASSEMBLER DIRECTIVES	5-26
5.5.1	Listing Control Directives	5-27
5.5.1.1	.LIST and .NLIST	5-27
5.5.1.2	Page Headings	5-33
5.5.1.3	.TITLE	5-33
5.5.1.4	.SBTTL	5-33
5.5.1.5	.IDENT	5-36
5.5.1.6	Page Ejection	5-36
5.5.2	Functions: .ENABL and .DSABL Directives	5-36
5.5.3	Data Storage Directives	5-37
5.5.3.1	.BYTE	5-38
5.5.3.2	.WORD	5-39
5.5.3.3	ASCII Conversion of One or Two Characters	5-40
5.5.3.4	.ASCII	5-40
5.5.3.5	.ASCIZ	5-42
5.5.3.6	.RAD50	5-42
5.5.4	Radix Control	5-43
5.5.4.1	.RADIX	5-43
5.5.4.2	Temporary Radix Control: †D, †o, and †B	5-44
5.5.5	Location Counter Control	5-45
5.5.5.1	.EVEN	5-45
5.5.5.2	.ODD	5-45
5.5.5.3	.BLKB and .BLKW	5-46
5.5.6	Numeric Control	5-47
5.5.6.1	.FLT2 and .FLT4	5-47
5.5.6.2	Temporary Numeric Control: †F and †C	5-48
5.5.7	Terminating Directives	5-49
5.5.7.1	.END	5-49
5.5.7.2	.EOT	5-50
5.5.8	Program Boundaries Directive: .LIMIT	5-50
5.5.9	Program Section Directives	5-50
5.5.9.1	.ASECT and .CSECT	5-52
5.5.10	Symbol Control: .BLOBL	5-53
5.5.11	Conditional Assembly Directives	5-54
5.5.11.1	Subconditionals	5-56
5.5.11.2	Immediate Conditionals	5-57
5.5.11.3	PAL-11R and PAL-11S Conditional Assembly Directives	5-58

5.6	MACRO DIRECTIVES	5-59
5.6.1	MACRO Definition	5-59
5.6.1.1	.MACRO	5-59
5.6.1.2	.ENDM	5-59
5.6.1.3	.MEXIT	5-60
5.6.1.4	MACRO Definition Formatting	5-61
5.6.2	Macro Calls	5-61
5.6.3	Arguments to Macro Calls and Definitions	5-61
5.6.3.1	Macro Nesting	5-62
5.6.3.2	Special Characters	5-63
5.6.3.3	Numeric Arguments Passed as Symbols	5-64
5.6.3.4	Number of Arguments	5-65
5.6.3.5	Automatically Created Symbols	5-65
5.6.3.6	Concatenation	5-66
5.6.4	.NARG, .NCHR, and .NTYPE	5-67
5.6.5	.ERROR and .PRINT	5-68
5.6.6	Indefinite Repeat Block: .IRP and .IRPC	5-69
5.6.7	Repeat Block: .REPT	5-72
5.6.8	Macro Libraries: .MCALL	5-72
5.7	OPERATING PROCEDURES	5-73
5.8	ERROR MESSAGES	5-74
CHAPTER 6	LINKER	6-1
6.1	INTRODUCTION	6-1
6.2	ABSOLUTE AND RELOCATABLE PROGRAM SECTIONS	6-1
6.3	GLOBAL SYMBOLS	6-2
6.4	INPUT AND OUTPUT	6-2
6.4.1	Object Module	6-2
6.4.2	Load Module	6-2
6.4.3	Load Map	6-3
6.5	USING OVERLAYS	6-5
6.6	OPERATING PROCEDURES	6-9
6.6.1	Command String	6-9
6.6.1.1	Switches	6-10
6.7	ERROR HANDLING AND MESSAGES	6-15
CHAPTER 7	ODT	7-1
7.1	RELOCATION	7-1
7.2	RELOCATABLE EXPRESSIONS	7-2
7.3	COMMAND SUMMARY	7-3



7.4	COMMANDS AND FUNCTIONS	7-6
7.4.1	Printout Formats	7-6
7.4.2	Opening, Changing and Closing Locations	7-7
7.4.3	Breakpoints	7-12
7.4.4	Running the Program, r;G and r;P	7-13
7.4.5	Single-Instruction Mode	7-14
7.4.6	Searches	7-15
7.4.7	The Constant Register, r;C	7-16
7.4.8	Core Block Initialization, ;F and ;I	7-16
7.4.9	Calculating Offsets, r;O	7-17
7.4.10	Relocation Register Commands, r;nR, ;nR, ;R	7-17
7.4.11	The Relocation Calculators, nR and n!	7-18
7.4.12	ODT's Priority Level, \$P	7-19
7.4.13	ASCII Input and Output, r;nA	7-20
7.5	ERROR DETECTION	7-20
7.6	PROGRAMMING CONSIDERATIONS	7-21
7.6.1	Functional Organization	7-21
7.6.2	Breakpoints	7-24
7.6.3	Search	7-27
7.6.4	Terminal Interrupt	7-28
7.7	OPERATING PROCEDURES	7-28
7.7.1	Return to Monitor, CTRL/C	7-30
7.7.2	Terminate Search, CTRL/U	7-30
CHAPTER 8	PROGRAMMED REQUESTS	8-1
8.1	SYSTEM CONCEPTS	8-2
8.2	TYPES OF PROGRAMMED REQUESTS	8-7
8.3	PROGRAMMED REQUEST USAGE	8-11
8.3.1	.DATE	8-11
8.3.2	.CLOSE	8-12
8.3.3	.CSIGEN	8-13
8.3.4	.CSISPC	8-15
8.3.5	.DELETE	8-19
8.3.6	.DSTATUS	8-20
8.3.7	.ENTER	8-21
8.3.8	.EXIT	8-23
8.3.9	.FETCH	8-24
8.3.10	.HRESET	8-25
8.3.11	.LOCK	8-25
8.3.12	.LOOKUP	8-26
8.3.13	.PRINT	8-27
8.3.14	.QSET	8-28
8.3.15	.RCTRLO	8-29
8.3.16	.READ/.WRITE, .READC/.WRITC, .READW/.WRITW	8-30
8.3.17	.RELEAS	8-35
8.3.18	.RENAME	8-36
8.3.19	.REOPEN	8-37

8.3.20	.SAVESTATUS	8-38
8.3.21	.SETTOP	8-39
8.3.22	.SRESET	8-40
8.3.23	.TTYIN/.TTINR	8-41
8.3.24	.TTYOUT/.TTOUTR	8-42
8.3.25	.UNLOCK	8-44
8.3.26	.WAIT	8-44
8.3.27	.WRITC	8-45
8.3.28	.WRITE	8-46
8.3.29	.WRITW	8-47
<b>CHAPTER 9</b>	<b>EXPAND UTILITY PROGRAM</b>	<b>9-1</b>
9.1	LANGUAGE	9-1
9.2	RESTRICTIONS	9-1
9.3	OPERATION	9-3
9.4	ERROR MESSAGES	9-4
<b>CHAPTER 10</b>	<b>ASSEMBL, THE 8K ASSEMBLER</b>	<b>10-1</b>
10.1	OPERATING PROCEDURES	10-1
10.2	ERROR MESSAGES	10-4
<b>APPENDICES</b>		
A.	RT-11 System Build	A-1
B.	Physical Device Names	B-1
C.	Monitor Summary	C-1
D.	RT-11 Error Messages	D-1
E.	MACRO Character Sets	E-1
F.	MACRO Assembly Language and Assembler	F-1
G.	System Macro File	G-1
H.	Summary of Monitor Programmed Requests	H-1
I.	Editor Summary	I-1
J.	Linker Summary	J-1
K.	ODT Summary	K-1
L.	BASIC/RT-11 Summary	L-1
M.	Macro Permanent Symbol Table	M-1
N.	Peripheral Interchange Program--Cassette (PIPC) N-1	
O.	PATCH Program	O-1
P.	Fundamentals of Programming the PDP-11	P-1
Q.	Cassette Standards	Q-1
R.	PIP Command Summary	R-1
<b>GLOSSARY</b>		<b>GLOSSARY-1</b>
<b>INDEX</b>		<b>X-1</b>

## CHAPTER 1

### RT-11 OVERVIEW

The RT-11 System is a powerful programming and operating system designed for the PDP-11 series of computers. This system permits the use of a wide range of peripherals and up to 28K of core. RT-11 offers a versatile Keyboard Monitor which provides complete user control of the system from the terminal keyboard.

Besides the Monitor facilities, RT-11 includes a library of system programs which allow the user to develop programs using high level or assembly language. The following is a brief summary of the RT-11 system programs:

1. The Text Editor, EDIT, is used to create or modify source files for use as input to language processing programs such as the assembler or BASIC. EDIT contains powerful text manipulation commands for quick and easy editing of an ASCII input file.
2. The MACRO-11 Assembler brings the capabilities of macros to the RT-11 system. The Assembler accepts source files written in the MACRO language and generates a relocatable object module which is processed by the Linker before loading and execution.
3. EXPAND is used in 8K systems or in larger systems for very large programs to expand certain macros in an assembly language program so the program can be assembled by ASEMBL, the 8K assembler.
4. The Linker essentially fixes (i.e., makes absolute) the values of global symbols and converts the relocatable object modules of assembled programs and subroutines into a load module which can be loaded and executed by RT-11. The Linker also produces a load map (which contains the assigned absolute addresses) and provides automatic overlay capabilities to very large programs.
5. The Peripheral Interchange Program, PIP, is the RT-11 file maintenance and utility program and can be used to transfer files between devices which are part of the RT-11 system, to rename or delete files and to obtain directory listings.
6. PIPC (Peripheral Interchange Program for Cassettes) is used to transfer files between cassettes and other RT-11 system devices, delete cassette files and transfer cassette directories.
7. ODT (On-line Debugging Technique) aids in debugging assembled and linked object programs. It can be used to print the contents of specified locations, execute all or part of the object program and search the object program for bit patterns.

8. PATCH is a utility program used to make modifications to core image files. PATCH can be used on files which do or do not have overlays.

### 1.1 HARDWARE CONFIGURATIONS

The minimum RT-11 configuration is a PDP-11 series computer with 8K of core, a block-replacable systems device and a terminal.

The following devices are supported by the RT-11 system:

- RK11 disk
- TC11 DEctape
- Line printer (LP11 or LS11)
- Terminal (LA30, VT05 or LT33)
- PC11 High Speed Reader/Punch

The TA11 Cassette is supported by the utility program PIPC.

RT-11 operates in environments from 8K to 28K with no user interaction necessary in changing core size. The same system DEctape or disk operates on any PDP-11 family processor with 8K to 28K of core and makes use of all core available.

### 1.2 SYSTEM SOFTWARE COMPONENTS

The main software components of the RT-11 system are:

- Resident Monitor (RMON)
- Keyboard Monitor (KMON)
- User Service Routines (USR) and Command String Interpreter (CSI)
- Device Handlers
- System Programs

The Resident Monitor is the only permanently core resident part of RT-11. The programmed requests for all services of RT-11 are handled by RMON. RMON contains:

- Terminal service
- System device handler
- EMT processor
- Monitor error routine
- System I/O tables and data base

The Keyboard Monitor provides communication between the user and the RT-11 executive routines by accepting commands from the terminal keyboard. The commands enable the user to create logical names for devices, and run system and user programs. The User Service Routine performs the following operations: loads device handlers, opens files for READ or WRITE operations, and creates new files. The Command String Interpreter is part of the USR and can be called by any program to obtain command strings and open files for the program. For normal RT-11 usage, the USR function is unseen by the user and need not be of concern.

Device handlers for the RT-11 system are treated as files which are resident on the system device. These handlers transfer data to and from peripheral devices. New handlers can be added to the system as files on the systems device and can be interfaced to the system by modifying a few monitor tables. No program other than PIP should access these files.

The collection of system programs, as mentioned earlier, contains the Editor (EDIT), Peripheral Interchange Programs (PIP and PIPC), assembler (MACRO and ASEMBL), On-line Debugging Technique (ODT), Linker (LINK), EXPAND, and PATCH.



## CHAPTER 2

### MONITOR

The Monitor is the hub of the RT-11 system, providing access to system and user programs, performing input and output functions.

The user communicates with the Monitor through programmed requests and keyboard commands.

The keyboard commands are used to load and run programs, start or restart programs at specific addresses, modify the contents of memory and assign and deassign alternate device names (refer to paragraph 2.2).

Programmed requests are program instructions which pass arguments to the Monitor and request monitor services. These commands allow user assembly language programs to use available Monitor routines to open, create and close files (refer to Chapter 8).

#### 2.1 START PROCEDURE

After the system has been built (refer to Appendix A for build procedures), the Monitor can be loaded into core from disk or DECTape with one of the hardware bootstraps as follows:

1. Mount the systems device on unit 0.
2. If a disk is being used, be sure the WRITE protect light is not lit.
3. If a DECTape unit is being used, set the WRITE ENABLE/WRITE LOCK switch to WRITE ENABLE and the REMOTE/OFF/LOCAL switch to REMOTE.

If the hardware configuration includes the BM792-YB bootstrap:

1. Set the Switch Register to 173100 (the address of the ROM Bootstrap Loader).
2. Press the LOAD ADDR switch.
3. Set the Switch Register to the address of the word count register of disk or DECTape on which the Monitor resides (177406 for RK11/RK05 or 177344 for DECTape).
4. Press the START switch.

If the hardware configuration includes the MR11-DB bootstrap:

1. Set the Switch Register to  
773110 for disk or  
773120 for DECTape.

2. Press the LOAD ADDR switch.
3. Press the START switch.

If the hardware bootstrap is not available, enter one of the following bootstraps via the Switch Register. For either bootstrap, set the Switch Register to 1000 and press the LOAD ADDR switch. Then set the Switch Register to the first value shown for the appropriate bootstrap and press the DEPOSIT switch. Continue depositing the values shown being especially careful with those values marked with an asterisk (\*). When all the values have been entered, load address 1000 and press the START switch.

<u>DEctape</u>	<u>Disk</u>
12700	12700
177344	177406
12710	12710
177400	177400
12740	12740
4002	5*
5710	105710
100376	100376
12710	5007
3*	
105710	
100376	
12710	
5*	
105710	
100376	
5007	

The Monitor loads into core and prints its identification message followed by a dot (.) on the terminal to indicate it is ready to accept a command.

The Keyboard Monitor outputs certain characters on the terminal to indicate that it is ready to accept a command or file specification. These characters are:

<u>Character</u>	<u>Meaning</u>
.	the Keyboard Monitor is waiting for a keyboard command. (Refer to section 2.2.)
*	a program is waiting for a file specification. (Refer to section 2.3.)

The † character is also output on the terminal by RT-11, when the following devices are being used:

PR:	indicates that the paper tape reader is about to read a tape. Typing any character (which does not echo) starts tape.
-----	---



TT: indicates that the program is ready for input from the terminal keyboard. Type a CTRL/Z, and a carriage return to mark EOF. TT: does not echo the characters typed when used for input to EDIT or BASIC.

If the line printer is off-line for any reason, the system waits for the user to turn it on-line. Turning the printer on-line is sufficient to resume (or begin) printer output.

## 2.2 KEYBOARD COMMUNICATION

The keyboard commands provide the communication with the RT-11 Monitor and allow allocation of system resources (INITIALIZE, ASSIGN, DEASSIGN and CLOSE commands), manipulation of core images (GET, EXAMINE, DEPOSIT, and SAVE commands), starting of programs (RUN, R, START, REENTER commands). These commands are explained in paragraphs 2.2.1 through 2.2.3 and summarized in Appendix C. Keyboard commands can, in most cases, be abbreviated to the first two characters of the command if desired. The keyboard commands require a space between the command and the first argument. Through the keyboard, the user can communicate with

the monitor,  
 a user program running under RT-11, or  
 an RT-11 system program (Assembler, Editor, PIP, etc.)

The special functions of certain terminal keys used to communicate with the Keyboard Monitor are explained in Table 2-1.

Table 2-1

Special Function Keys

Key	Function
CTRL/C	(Typed by holding down the CTRL key while typing the C key.) Echoes ↑ C on terminal. Interrupts execution of the user program and returns to Monitor command level if the program is waiting for terminal input. Otherwise, CTRL/C must be typed twice. Monitor commands run to completion before a single CTRL/C takes effect. Two CTRL/C's echo two dots; cause the program being executed to be aborted and control returns to KMON.
CTRL/O	Inhibits printing on the terminal until completion of current output or until another CTRL/O is typed. When the first CTRL/O is typed, ↑ O is output on the terminal. The second CTRL/O re-enables printing.
CTRL/U	Deletes the current line and echoes an ↑ U at the terminal. CTRL/U does not delete data past the first CR/LF combination encountered to the left.
RUBOUT	Deletes the last character from the current line and echoes a backslash and the character deleted. Each succeeding RUBOUT typed deletes and echoes another character up to the last CR/LF typed.

The Keyboard Monitor has a "type ahead" feature which allows a command or file specification (up to 80 characters) to be typed while another command is executing. This terminal input is stored in a buffer and executed when the previous command is completed. When typing ahead, a single CTRL/C causes a return to the Monitor when the previous command completes execution. A double CTRL/C returns control to the Monitor immediately.

If "type ahead" input exceeds 80 characters, a bell rings and no characters are accepted in the buffer until part of the buffer is executed or entries are deleted. If "type ahead" is used in conjunction with an EXIT command from the Editor or BASIC, there is no terminal echo of the characters but they are stored in the buffer. "Type ahead" is particularly useful in specifying multiple command lines to the assembler.

Refer to section 2.3 for a description of the file specification (device, file names, extensions, etc.) format to be used with certain keyboard commands.

## 2.2.1 Commands to Allocate System Resources

### 2.2.1.1 DATE Command

The DATE (DA) command enters the specified date to the system. This date is assigned to new directory entries and listings until a new DATE command is issued.

The form of the command is:

```
DATE dd-mm-yy
```

followed by the RETURN key.

Where dd-~~mm~~-yy is the day, month and year to be entered. dd is a decimal number in the range 1-31; ~~mm~~ is the first three characters of the name of the month, and yy, 73-99.

Example:

```
.DA 1-APR-73   Enter the date 1-APR-73 as the current system
               date.
```

### 2.2.1.2 INITIALIZE Command

The INITIALIZE (IN) command resets all system tables to zero, removes all user device assignments, sets all handlers non-resident, and clears the core control block. (The core control block is a monitor

maintained area in bytes 360-377. It provides the means for describing into which areas of core a program will load. It is manipulated by the GET, RUN, R, SAVE and INIT commands. IN also stops all I/O in progress by executing a hardware RESET instruction.

The form of the command is:

INITIALIZE

followed by the RETURN key. The INITIALIZE command is generally used prior to running a user program.

Example:

```
.IN          Initialize system
.RUN PROG1   Execute PROG1.SAV
```

#### 2.2.1.3 ASSIGN Command

The ASSIGN (AS) command assigns the specified user-defined device name as an alternate name for the device specified.

The form of the command is

ASSIGN dev udev

followed by the RETURN key.

Where dev is one of the standard device names. (Refer to section 2.3.1.) If the standard device name is not specified, but a user device name (udev) is, an error message is printed.

udev is 1-3 alphanumeric characters to be used in a program to refer to the specified device.

If a user device name is not specified, any previous synonym assigned to the device is eliminated. If neither device nor synonym is specified, all previous assignments are eliminated.

The AS command is used when a program refers to a device which is not available at run time. Only one user-defined name can be assigned per AS command but several AS commands can be used to assign different names to the same device.

Examples:

```
.AS DT1 INP   Whenever the program encounters a reference
               to device INP, it uses device DT1.

.AS           Removes all previous user device name
               assignments.
```

#### 2.2.1.4 CLOSE Command

The CLOSE (CL) command closes all currently open files. This action can be used to preserve a tentative output file, which was not closed by a program. The CLOSE command is most frequently used after the execution of a single CTRL/C.

The form of the command is

```
CLOSE
```

followed by the RETURN key.

The CL command makes temporary entries in a directory permanent.

Example:

```
.CL          Close any open files.
```

#### 2.2.2 Commands to Manipulate Core Images

##### 2.2.2.1 GET Command

The GET (GE) command loads the specified core image file (core image format, not ASCII or binary) into core from the specified device and sets the core control block. If a portion of the program being loaded overlays the Keyboard Monitor, that portion of the program is placed in the overlay area on the systems device.

The form of the GET command is

```
GET dev:filnam.ext
```

followed by the RETURN key.

If a file name extension is not specified to the GET command, the extension .SAV is assumed. The GET command is typically used to load a program into core for modification and debugging.

The GET command cannot be used to load overlay segments of programs; it may only be used to load the root segment. (Refer to Chapter 6, Linker.)

Multiple GETs can be used to build a core image of several programs. Identical locations are required by any of the programs, the latest program overlays the previous one.

Examples:

```
.GE DT3:FILE1.SAV  Loads the file FILE1.SAV into core from  
DT3.  
.GE NAME1          Loads the file NAME1.SAV from device DK.
```

### 2.2.2.2 EXAMINE Command

The EXAMINE (E) command prints the contents of the specified location(s) in octal on the console terminal. The form of the E command is:

```
E location
    or
E location m-location n
```

followed by the RETURN key.

Where location is an octal address which is added to the relocation base value (refer to paragraph 2.2.2.4) to get the actual address examined. Any non-octal digit is accepted as a terminator of an address.

If more than one location is specified, (location m-location n) the contents of location m through location n are printed. If the second location specified (location n) is not greater than the first location specified only the contents of the first location are printed.

If no location is specified, the contents of location 0 are printed. Examination of locations in the monitor area are illegal.

Examples:

```
.E 1000          Prints contents of location 1000 (plus the
                base if other than 0).
XXXXXXXX

.E 1001-1012     Prints the contents of locations 1000 (plus
                the base if other than 0) through 1013.
XXXXXXXX XXXXXX XXXXXX XXXXXX XXXXXX XXXXXX
```

In the above examples xxxxxx represents octal numbers printed by the monitor.

### 2.2.2.3 DEPOSIT Command

The DEPOSIT command (D) deposits the specified value(s) starting at the location given.

The form of the D command is:

```
D location=value
    or
D location=value1,value2,...,valuen
```

followed by the RETURN key.

Where location is an octal address which is added to the relocation base value to get the actual address where the values are deposited. Any non-octal digit is accepted as a terminator of an address.

value is the new contents of location.

If multiple values are specified (value1,...,valuen), they are deposited beginning at the location specified. The Deposit command accepts word or byte addresses but executes the command as though a word address were specified.

Any character that is not an octal digit may be used to separate the locations and values in a Deposit command.

An error results when the address specified references a location within the resident monitor.

Examples:

```
.D 1000=3705
```

```
.D 7000=240,240,240
```

#### 2.2.2.4 BASE Command (B)

The BASE command (B) sets the relocation base. The relocation base is added to the address specified in an E (examine) or D (deposit) command to obtain the address of the location to be opened. The form of the command is

```
B_ or B location
```

Where location is an octal address used to determine the address of the location to be opened. Note that a space must follow the B command even if an address is not specified. Any non-octal digit is accepted as a terminator of an address.

This command is useful when referencing linked modules. The base address can be set to the address where the module of interest is loaded.

Example:

```
.B set base to 0  
.B 200 set base to 200
```

#### 2.2.2.5 SAVE Command

The SAVE (SA) command writes the area(s) of user core specified by the parameter list or by the core control block (if no parameter list is given) into the named file in save image format.

The SAVED file can then be referenced with a GET, RUN or R command.

The SAVE command does not write the overlay segments of programs; rather it saves only the root segment. (Refer to Chapter 6, Linker.)

The form of the command is

SAVE dev:filnam.ext parameters

followed by the RETURN key.

Where dev: is one of the standard RT-11 block replaceable device names. If no device is specified, DK is assumed.

file.ext is the name to be assigned to the file being saved. If the file name is omitted, an error message is output.

If no file name extension is specified, the extension .SAV is automatically added by the system.

parameters are core locations to be saved. RT-11 transfers core in 256-word blocks. If the locations specified make a block of less than 256 words, enough additional locations are transferred to make a 256-word block. If no locations are specified, the number of locations given in the core control block are transferred.

Parameters can be specified in the following format:

core area 1,...,core area n; start address, stack, JSW

Where core area 1, is an octal number or numbers separated by commas.

...,area n If more than one number is specified, the second number must be greater than the first. If the following parameters (start address, etc.) are to be specified, they must be separated from the core areas by a semicolon(;).

start addr user program starting address

stack user program initial stack (1000 if none given)

JSW user program Job Status Word

If any of the parameters are specified, all parameters should be specified; otherwise those not specified are given the default value 0.

If the JSW is set to:

20000(8) the program is restartable

200(8) a halt occurs on a hard error

Examples:

```
.SA FILE1 10000-11000, 14000-14100
    Saves locations 10000(8) through
    11777(8) (11000 starts the first word of
    a new block, therefore the whole block,
    up to 12000, is stored) and 14000(8)
    through 14777(8) on DK with the name
    FILE1.SAV.

.SA DT1:NAM.NEW 10000
    Saves locations 10000 through 10777 on
    DT1 with the name NAM.NEW.

.SA SY:PRAM 1000-5777; 1000, 10000, 0
    Saves locations 1000 through 5777; 1000
    is the user program start address; 10000
    is the user program initial stack; and
    the JSW is set to 0.
```

### 2.2.3 Commands to Start a Program

#### 2.2.3.1 RUN Command

The RUN (RU) command loads the specified core image file into core and starts execution at the address specified in the core control block.

The form of the command is

```
RUN dev:filnam.ext
```

followed by the RETURN key.

Where dev: is one of the RT-11 standard block replaceable device names. If dev: is not specified, the device is assumed to be DK.

filnam.ext is the file to be executed. If a file name extension is not specified, the extension .SAV is automatically assumed to be included in the file name.

The RUN command is a combination of the GET command and the START command (with no address specified).

Examples:

```
.RUN DT1:SRCH.SAV  Loads and executes the file SRCH.SAV
                   from DT1.

.RU PROG          Loads PROG.SAV from DK and executes the
                   file.
```

R Command:

This command is essentially the same as RUN except that the file specified must be on the system device (SY:).



The form of the command is

R filnam.ext

and no device is specified. If a file name extension is not specified, the extension .SAV is assumed.

Examples:

.R XY1.SAV      Loads and executes XY1.SAV from SY.

.R SRC            Loads and executes SRC.SAV from SY.

#### 2.2.3.2 START Command

The START (ST) command begins execution of the program currently in core at the specified address. The stack pointer is set to the user stack area as specified in the core control block. START does not clear and reset core areas.

The form of the command is

START address

followed by the RETURN key.

Where address      is an octal number representing any 16 bit address.

If the address given does not exist or is not an even address, an illegal address trap occurs.

If no address is given, the program's start address from the core control block is used.

Example:

.START 10000      Starts execution of the program currently in core at location 10000.

#### 2.2.3.3 REENTER Command

The REENTER (RE) command starts the program at its reentry address (the START address minus two). REENTER does not clear or reset any core areas and is generally used to avoid reloading the same program for repetitive execution.

The form of the command is

REENTER

and the RETURN key. If the reenter bit (bit 13) in the job status word (location 44) is not set ( $\neq 1$ ), the RE command is illegal.

The RE command generally reenters the program at the command level.

If desired, the reentry address can be set to a routine in the user program which will initialize the tables and stack, release device handlers etc. and then continue normal operation.

### 2.3 RT-11 FILE SPECIFICATION

The format of the general command string file specification for RT-11 is:

```
dev:filnam.ext[n],dev:filnam.ext,...=dev:filnam.ext,.../s/s
```

where dev: is the two-character device name (refer to section 2.3.1)

filnam.ext is the name of the file (six alphanumeric characters followed optionally by a three character extension) (refer to section 2.1.2).

[n] is the length (decimal) desired (output files only).

/S is one or more switches which may be specified to the calling program. These switches may be placed anywhere in the command string.

Up to three output files are specified first, followed by an equal sign (=) and up to six input files. The left angle bracket (<) can also be used to separate the output and input files. If no output files are specified, the = (or <) can be omitted. Each file specified must be separated from the next by a comma. The [] construction can only be used to the left of the = sign. This construction allows specification of an output file size, and thus the brackets must follow immediately after the filnam.ext.

(Refer to the appropriate Chapters for details of file specifications for each program.)

#### 2.3.1 Physical Device Names

Each device in an RT-11 file specification (dev:) is referenced by means of a standard two-character device name. Table 2-2 lists the names and related device.

Table 2-2

## Permanent Device Names

Permanent Name	I/O Device
SYn	System device, the device and unit from which the system is bootstrapped. N is an integer in the range 0-7.
DTn	DECTape n, where n is a unit number (an integer in the range 0 to 7, inclusive).
DK	The default storage device for all files. DK is usually the systems device but the assignment can be changed with the Assign Command. Usually DK is the disk on a single disk system or DT0 on a DECTape system.
TT	Terminal keyboard and printer.
LP	Line printer.
PP	High-speed paper tape punch.
RKn	RK disk cartridge drive n, where n is in the range 0-7 inclusive.
CTn	Cassette n where n is 0 or 1. (Used with PIPC only.)
PR	High-speed paper tape reader.

When no device is specified, the last device named on the same side of the command string is used. If the first file in a list (either input or output) has no explicit device, DK: is used.

In addition to the fixed names shown in Table 2-1, each device can be assigned logical names. This logical name takes precedence over the physical name and thus provides device independence. With this feature a program that is coded to use a specific device does not need to be rewritten if the device is unavailable. For example DK: is normally disk unit 0 but that name could be assigned to DECTape unit 0 with a monitor command.

Refer to Paragraph 2.2.1.3 (Assign command) for details on assigning logical names to devices.

### 2.3.2 File Names and Extensions

Files are referenced symbolically by a name of up to six alphanumeric characters followed, optionally, by a period and an extension of three alphanumeric characters. In a filename, excess characters (more than six) cause an error message. The extension to a file name is generally used as an aid for remembering the format of a file. In most cases, it is a good practice to conform to the standard file name extensions for RT-11. If an extension is not specified for an output file, some system programs assign default extensions. If an extension for an input file is not specified, the system searches for that file name with the default extension. Table 2-3 lists the standard extensions used in RT-11.

Table 2-3

File Name Extensions

Extension	Meaning
.BAD	Files with bad (unreadable) blocks
.BAK	Editor backup file
.BAS	BASIC source file (BASIC input)
.DAT	BASIC data file
.LDA	Absolute binary file
.LST	MACRO listing file (MACRO output)
.MAC	MACRO or EXPAND source file (input)
.MAP	Map file (Linker output)
.OBJ	Relocatable binary file (Macro output, Linker input)
.PAL	Output file of EXPAND (the macro expander program)
.SAV	Core image or SAVE file; default for R, RUN, SAVE and GET keyboard monitor commands; also default for output of Linker
.SYS	System files and handlers

Examples:

```
.RUN DK:PROG
```

Executes the file PROG.SAV (on device DK), if found.

```
.RUN DK:PROG.A
```

Executes PROG.A (on device DK), if found.

If a file name is to be used without an extension where the Monitor or a utility program assumes a default extension, a . must be entered after the file name to indicate that the file has no extension. For example, to run the file TEST type

```
.RUN TEST.
```

If the period after the file name is not specified, RT-11 attempts to run the file TEST.SAV.

## 2.4 ERROR MESSAGES

The following error messages can be output by the Keyboard Monitor.

<u>Message</u>	<u>Meaning</u>
?ADDR?	Address out of range in E or D command.
?DAT?	The DATE command had no argument or the argument was illegal.
?FIL NOT FND?	File specified in R, RUN, or GET command not found.
?FILE?	No file named where one is expected.
?HANDLR?	Attempting a close with no handler in core. The file cannot be closed.
?ILL CMD?	Illegal keyboard monitor command or command line too long.
?ILL DEV?	Illegal or nonexistent device.
?OVR COR?	Attempt to GET or RUN a file that is too big.
?PARAMS?	Bad save parameters.
?SV FIL I/O ER?	I/O error on .SAV file in SAVE (output) or R, RUN, or GET (input) command.
?SY I/O ER?	I/O error on system device (usually reading or writing scratch area).

The following messages are output by the RT-11 Monitor when an unrecoverable error has occurred. Control passes to the Keyboard Monitor. The program in which the error occurred cannot be restarted with the RE command. To execute the program again, use the RUN command.

The format for Monitor error messages is:

?M-text?

<u>Message</u>	<u>Meaning</u>
?M-USR ILL?	The USR was illegally called from an I/O completion routine.
?M-NO DEV?	An I/O operation was requested on a channel, but no device handler was in core.
?M-DIR I/O ERR?	An error has occurred while the USR was reading/writing a device directory. Usually due to WRITE LOCKed device, or there may be a hardware problem. Retry the operation.
?M-ILL HAND LD?	An attempt was made to load a device handler over the USR.

<u>Message</u>	<u>Meaning</u>
?M-OVLY ERR?	The system attempted to read an overlay segment on channel 17 but was unsuccessful. (can only occur if program was linked with the overlay feature requested.)
?M-HAND LD FAIL?	Failed reading a device handler from the system device. There may be a hardware problem. Retry the operation.
?M-DIR OVFL0?	No more directory segments are available for expansion. Occurs when .ENTER causes directory extension.

CHAPTER 3  
TEXT EDITOR

The text editor (EDIT) is used to create and modify ASCII source files so that these files can be used as input to other system programs such as the Assembler or BASIC. Controlled by user commands from the keyboard, EDIT reads ASCII files from any device, makes specified changes and writes ASCII files to any device. The Editor considers a file to be divided into logical units called pages. A page of text is generally 50-60 lines long (delimited by form feed characters) and corresponds approximately to a physical page of a program listing. The Editor reads one page of text at a time from the input file into its internal buffer where the page becomes available for editing. The editing commands are used to:

- Read a "page" of text from the input file.
- Locate text to be changed.
- Execute and verify the changes.
- Output the page to the output file,
- Proceed in the same manner to the end of the input file.

### 3.1 CALLING AND USING EDIT

To call EDIT from the system device type:

R EDIT

and the RETURN key in response to the dot (.) printed by the Monitor, EDIT responds with an asterisk (\*) indicating it is in command mode and awaiting a user command string.

To restart the Editor without reloading, type the Monitor REENTER command as follows:

```
↑C  
.REENTER  
*
```

When the REENTER command is executed, the text buffers are re-initialized and EDIT is ready for another editing session. Using the REENTER command is equivalent to starting EDIT with the RUN command, but saves the time and device motion associated with reloading the Editor. If several consecutive editing sessions are planned, it is suggested that the first session be started with a .R EDIT command and the following sessions with REENTER Commands.

## 3.2 COMMANDS

### 3.2.1 Key Commands

The EDIT key commands listed in Table 3-1 are the same as those used for the Monitor, with the addition of CTRL/X. Control commands are typed by holding down the CTRL key while typing the appropriate character.

Table 3-1  
EDIT Key Commands

Key	Explanation
ALTMODE	Echoes \$. A single ALTMODE terminates a text string. A double ALTMODE executes the command string. For example,  *GMOV A,B\$-1D\$\$
CTRL/C	Echoes at the terminal as @C and a carriage return. Terminates execution of EDIT commands and returns to Monitor command mode. A double CTRL/C is necessary when I/O is in progress. The REENTER command may be used to restart the editor but the contents of the text buffers are lost.  ↑C .REENTER *
CTRL/O	Echoes ↑O and a carriage return. Inhibits printing on the terminal until completion of the current command string or a second CTRL/O.
CTRL/U	Echoes ↑U and a carriage return. Deletes all the characters on the current terminal input line. (Equivalent to typing RUBOUT back to the beginning of the line.)
RUBOUT	Deletes the last character from the current line and echoes a backslash and the character deleted. Each succeeding RUBOUT typed deletes and echoes another character. For example,  CALL ENDMAC ;CLSO\OS\OSE MACRO  The first non-RUBOUT key typed causes another backslash to be printed, thereby enclosing the deleted characters.
TAB	Spaces to the next tab stop. Tab stops are positioned every eight spaces on the terminal.

(continued on next page)



Table 3-1 (Cont.)  
EDIT Key Commands

Key	Explanation
CTRL/X	<p>Echoes ↑X and a carriage return. CTRL/X causes the entire command string to be ignored, and EDIT types another "*". For example,</p> <pre data-bbox="641 336 763 420">*IABCD) EFGH ↑X *</pre> <p>A CTRL/U would only cause deletion of EFGH; CTRL/X erases the whole command.</p>

### 3.2.2 Editing Commands

EDIT operates in two modes, command mode and text mode. The mode of operation determines the action performed on the characters in a command string.

In command mode, EDIT accepts the key commands described in paragraph 3.2.1 or any of the editing commands described in this section.

EDIT automatically enters text mode when it encounters a command which requires a text string (for example, the Insert command). The next successive alphanumeric characters in the command are taken as text until an ALTMODE character is encountered. EDIT then returns to command mode and processes the next characters in the command string as a command.

EDIT commands fall into five categories:

<u>Category</u>	<u>Commands</u>
Input/Output	Edit Backup Edit Read Edit Write End File Exit List Next Read Verify Write
Pointer location	Advance Beginning Jump
Search	Find Get Position
Text modification	Change Delete eXchange Insert Kill

<u>Category</u>	<u>Commands</u>
Utility	Execute Macro Macro Save Unsave Edit Version

The general format for EDIT commands is:

argument command text \$

where argument can be:

An integer in the range -16383 to +16383 and may be preceded, except where noted, by a + or -. If no sign is used, the number is assumed to be positive. Where a number is expected but not specified, it is assumed to be 1.

A 0 to represent "the beginning of the current line".

A / to represent "the end of the current buffer".

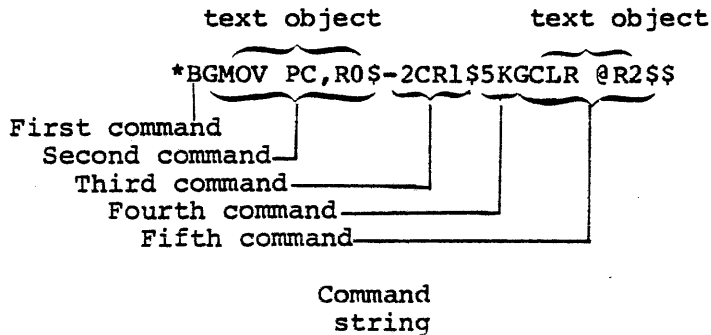
An = to represent -n where n is the length of the last text argument used.

command is a one- or two-letter command as described in the following sections.

text is a string of successive ASCII characters terminated by a single ALTMODE character (echoed as \$).

All EDIT command strings are terminated by two successive ALTMODE characters. Spaces, carriage returns and line feeds within a command string are ignored unless they appear in a text string. Commands such as Insert or eXchange can contain text strings that are several lines long. Each line is terminated with a CR/LF and the command is terminated with a double ALTMODE.

Several commands can be strung together and executed in sequence. For example,



Execution of a command string begins when the double ALTMODE is typed and proceeds from left to right.

With the exception of ALTMODE, all characters are legal in a text string. Carriage return and line feed characters in text strings do not terminate that command string. Two successive ALTMODE's must be typed in to terminate a command string.

Upon receipt of the double ALTMODE, EDIT begins execution of the command string. Prior to executing any commands, the Editor first scans the entire command string for errors in command format (illegal arguments, illegal combinations of commands, etc.). If an error of this type is found, an error message of the form

?ERROR MESSAGE?

is printed and no commands are executed. If the command string is syntactically correct, execution is started. Execution errors are still possible (buffer overflow, I/O errors, etc.), and if such an error occurs, a message of the form

?\*ERROR MESSAGE\*?

is printed; note the asterisks. In this case, all commands preceding the one in error are executed, while the command in error and those following are not executed.

Except when part of a text string, spaces, carriage return, line feed, and single ALTMODES are ignored and may be used to enhance the clarity of command strings. For example,

```
*BGMOV R0$=CCLR R1$AV$$
```

may be typed as

```
*B$ GMOV R0$  
=CCLR R1$  
A$ V$$
```

with equivalent execution.

If a command being typed is within ten characters of exceeding the command space available, the message

```
*CB ALMOST FULL*
```

is typed. (CB = Command Buffer.) If the command can be completed within ten characters, finish typing the command; otherwise, type the ALTMODE key to execute the command line already completed. The message repeats each time a character is entered in one of the last ten spaces.

Most EDIT commands require one or more arguments which specify the location where the command is to be executed and the number of times it is to be executed. For instance, the List command needs a specification of the number of lines to be listed and the Delete command needs a specification of where and how many characters to delete.

Most EDIT commands function with respect to a current character pointer. This character pointer is normally located between the most recent character operated upon and the next character in the buffer. Most commands use this pointer as an implied argument. Certain EDIT commands change the current position of the pointer.

#### NUMERIC ARGUMENTS

Edit commands are line-oriented or character-oriented depending on the arguments they accept. Line oriented commands operate on entire lines of text. Character-oriented commands operate on individual characters independent of what or where they are.

For character oriented commands, a numeric argument specifies the number of characters that are involved in the operation. Positive arguments represent the number of characters in the forward direction (towards the end of the buffer); negative arguments the number of characters in the backward direction (towards the beginning of the buffer). Carriage return and line feed characters are treated the same as any other character.

```
MOV    #VECT,R2(CR)(LF)
↑CLR   @R2(CR)(LF)
```

where ↑ represents the current position of the pointer, and the spaces (4) between MOV and # represent a TAB character.

The EDIT command -2J(jump) backs up the pointer two characters.

```
MOV    #VECT,R2(CR)(LF)
CLR    @R2(CR)↑(LF)
```

The command 10J(jump) advances the pointer ten characters and places it between the CR and LF characters following the second line.

```
MOV    #VECT,R2(CR)(LF)
CLR    @R2(CR)↑(LF)
```

Finally, to place the pointer after the "C" in the first line, a -14J(jump) command is used.

```
MOV    #VECT,R2(CR)(LF)
CLR    @R2(CR)(LF)
```

For line oriented commands, a numeric argument represents the number of lines involved in the operation. Positive arguments represent the number of lines forward (toward the end of the buffer); this is accomplished by counting carriage return/line feed combinations beginning at the pointer. Hence, if the pointer is at the beginning of a line, a line-oriented command argument of +1 represents the entire line between the current pointer and the terminating line feed. If the current pointer is in the middle of the line, an argument of +1 represents only the portion of the line between pointer and the terminating line feed.

For example, assume a buffer of

```
MOV    PC,R1 (CR) (LF)
ADD    #DRIV-.,R1 (CR) (LF)
MOV    #VECT,R2 (CR) (LF)
CLR    @R2 (CR) (LF)
```

The command to advance the pointer one line (1A) causes the following change:

```
MOV    PC,R1 (CR) (LF)
↑ADD    #DRIV-.,R1 (CR) (LF)
MOV    #VECT,R2 (CR) (LF)
CLR    @R2 (CR) (LF)
```

The command 2A(advance) moves the pointer over 2 carriage return/line feed combinations:

```
MOV    PC,R1 (CR) (LF)
ADD    #DRIV-.,R1 (CR) (LF)
MOV    #VECT,R2 (CR) (LF)
↑CLR    @R2 (CR) (LF)
```

Negative line arguments reference lines in the backward direction (toward the beginning of the buffer). This is accomplished by counting backwards from the pointer across n carriage return/line feed combinations and starting the reference immediately after the n+1 CRLF, i.e., at the beginning of the -nth line. This means that if you are at the beginning of the line, a line argument of -1 means "the previous line", but if the pointer is in the middle of a line, an argument of -1 means the preceding 1 1/2 lines. Assume the buffer contains

```
MOV    PC,R1 (CR) (LF)
ADD    #DRIV-.,R1 (CR) (LF)
MOV    #VECT,R2 (CR) (LF)
CLR    ↓@R2 (CR) (LF)
```

A command of -1A(advance) backs the pointer up 1 1/2 lines.

```
MOV    PC,R1 (CR) (LF)
ADD    #DRIV-.,R1 (CR) (LF)
↑MOV    #VECT,R2 (CR) (LF)
CLR    @R2 (CR) (LF)
```

Now a command of -1A(advance) backs it up only 1 line.

```
MOV    PC,R1 (CR) (LF)
↑ADD    #DRIV-.,R1 (CR) (LF)
MOV    #VECT,R2 (CR) (LF)
CLR    @R2 (CR) (LF)
```

## NON-NUMERIC ARGUMENTS

Besides numeric arguments, both line-oriented and character-oriented commands allow the general arguments

- 0 Beginning of Current Line - Used with a command to operate on text from the pointer to the beginning of a line or to move the pointer to the beginning of the current line.
- / End of Text Buffer - Used with a command to operate on buffer contents from the pointer to the end of the buffer or move the pointer to the end of the buffer.
- = Length of Last Text Object - Equal to -n where n is the length of the last text argument executed. (Legal only with Jump, Delete and Change commands).

### 3.2.2.1 Command Repetition

Portions of a command string may be executed more than once by enclosing the desired portion in angle brackets (<>) and preceding the left angle bracket with the number of iterations desired. The structure is

```
command1 command2 n<command3 command4> command5 $$
```

In the above example, commands 1 and 2 are executed, then 3 and 4 are executed n times one after the other. Finally, command 5 is executed once and the command line is finished. The iteration argument must be a positive number (1 to 16,383), and if not specified is assumed to be 1. If the number is negative or too large, an error message is printed. Iteration brackets may be nested up to 20 levels. Command lines are checked to make certain the brackets are correctly used and match prior to execution.

Essentially, enclosing a portion of a command string in iteration brackets is equivalent to typing that portion of the string n times, where n is the iteration argument. For example

```
*BGAAA$3<-DIB$-J>V$$
```

is equivalent to typing

```
*BGAAA$-DIB$-J-DIB$-J-DIB$-JV$$
```

and

```
*B3<2<AD>V>$
```

is equivalent to typing

```
*BADADVADADVADADV$$
```

The following bracket structures are examples of legal usage:

```
<<<<<<>>>>
<<<>>><<>>
```

The following bracket structures are examples of illegal combinations and cause an error message:

```
<><>
<<<>>
```

Command execution proceeds left to right remembering the iteration counts until a right bracket is encountered. EDIT then returns to the last left bracket encountered, decrements the counter and executes the commands within the brackets. When the counter is decremented to 0, EDIT looks for the next iteration count to the left and repeats the same procedures. The overall effect is that EDIT works its way to the inner most brackets and then works its way back again. The most common use for iteration commands is for commands, such as Unsave, which do not accept repeat counts. For example:

```
3<U>$$
```

As an example, assume a file called SAMP (stored on device DK) is to be read and the first four occurrences of the instruction MOV #200,R0 on each of the first five pages changed to MOV #244,R4. The following command line is entered.

```
ERSAMP$5<R 4<BGMOV #200,R0$=J 3<G0$=C4>>>$$
```

The command line contains three 'sets' of iteration loops (A,B,C) and is executed as follows:

Execution initially proceeds from left to right; the file SAMP is opened for input, and the first page is read into memory. The pointer is moved to the beginning of the buffer and a search is initiated for the character string MOV #200,R0. When the string is found, the pointer is positioned at the end of the string, but the =J command moves the pointer back so that it is positioned immediately preceding the string. At this point, execution has passed through each of the first two 'sets' of iteration loops (A,B) once. The innermost loop (C) is next executed three times, changing the 0's to 4's. Control now moves back to pick up the second iteration of loop B, and again moves from left to right. When loop C has executed three times, control again moves back to loop B. When loop B has executed a total of 4 times, control moves back to the second iteration of loop A, and so forth until all iterations have been satisfied.

### 3.2.2.2 Core Usage

The core area available to EDIT is divided into four logical buffers as follows:

MACRO BUFFER
SAVE BUFFER
FREE CORE
COMMAND INPUT BUFFER
TEXT BUFFER

The Text Buffer contains the current page of text being edited.

The Command Input Buffer holds the command currently being typed at the terminal.

The Save Buffer contains text stored with the Save (S) command.

The Macro Buffer contains any command string macro entered with the Macro (M) command.

The Macro and Save Buffers are not allocated space until an M or S command is executed. Once an M or S command is executed, a OM or OU (Unsave) command must be executed to return that space to the free area.

The size of each buffer automatically expands and contracts to accommodate the text being entered; if there is not enough space available to accommodate required expansion of any of the buffers, a "?\*NO ROOM\*?" error message is typed.

### 3.2.2.3 Input/Output Commands

These commands allow files to be created, opened for editing, listed or closed. Pages of the files can be read into memory for processing. Once editing is completed and the page is written to the output file, that page of text is unavailable for further editing until the file is closed and reopened.

#### EDIT READ

The Edit Read command opens an existing file for input, and prepares it for editing.

The form of the command is:

ERdev:filnam.ext\$

The string argument (dev:filnam.ext) is limited to 19 characters and specifies the file to be opened. If no device is specified, DK: is assumed. If any file is currently open for input, that file is closed.



Edit Read does not input a page of text nor does it affect the contents of the Save buffer.

Edit Read can be used repetitively on the same file to reposition EDIT at the beginning of the file. The first Read command following any Edit Read command inputs the first page of the file.

Examples:

```
*ERDT1:SAMP.MAC$$   Opens SAMP.MAC on device DT1: for input.
*ERSOURCE$$         Opens SOURCE on device DK: for input.
```

#### EDIT WRITE

The Edit Write command sets up a new file for output of edited text. Any current output files are closed and a new file with the specified name is opened on the specified device.

The form of the command is

```
EWdev:filnam.ext[n]$
```

The string argument (dev:filnam.ext[n]) is limited to 19 characters and is the name to be assigned to the output file being opened. If dev: is not specified, DK: is assumed. [n] is optional, and represents the length of the file to be opened. If not specified, the largest possible space is used.

If a file with the same name already exists on the device, the old file is destroyed when an EXit, End File or another Edit Write command is executed.

The EW command does not output any text nor does it affect the contents of any of the buffers.

Examples:

```
*EWDK:TEST.MAC$$   Opens the file TEST.MAC on device DK:
                   for output.
*EWFILE1.BAS[11]$$ Opens the file FILE1.BAS (11 blocks) on
                   the device DK: for output.
```

#### EDIT BACKUP

The Edit Backup command is used to open a file for editing without deleting the old copy while assigning the same name to the new copy. The name and extension specified in the command is assigned to the new output file and the old version is preserved with the name specified and the extension .BAK. When an Exit or End File command is executed, any existing file with the current name and the extension .BAK is deleted. The input file (now the old version) is assigned the extension .BAK. The output file is closed and assigned the name specified in the EB command. This renaming of files takes place whenever an Exit, End File, Edit Read, Edit Write or Edit Backup command is executed after a previous EB command has been successfully executed.

The format of the command is:

EBdev:filnam.ext[n]\$

The device designation, file name and extension are limited to 19 characters. If dev: is not specified, DK: is assumed.

Examples:

\*EBSY:BAS1.MAC\$\$ Opens BAS1.MAC on SY. When editing is complete, the old BAS1.MAC becomes BAS1.BAK and the new file becomes BAS1.MAC. Any previous version of BAS1.BAK is deleted.

\*EBBAS2.BAS[15]\$\$ Opens BAS2.BAS on DK (15 blocks). When editing is complete, the old BAS2.BAS is labeled BAS2.BAK and the new file becomes BAS2.BAS. Any previous version of BAS2.BAK is deleted.

In EB, ER and EW commands, leading spaces between the command and the file name are illegal (the file name is considered to be a text string). All dev:file.ext specifications for EB, ER and EW commands conform to the RT-11 conventions for file naming and are identical to file names used in command strings to the other system programs.

## READ

The Read command (R) moves the next page of text from the input file (previously specified in an ER or EB command) and appends it to the current contents, if any, of the text buffer.

The form of the command is:

R

There are no arguments to the R command and the pointer is not moved. Read inputs text until one of the following conditions is met.

1. A form feed character, signifying the end of the page, is encountered. At this point, the form feed will be the last character in the buffer.
2. The text buffer is within 500 characters of being full. When this condition occurs, Read inputs up to the next carriage return/line feed (CR/LF) combination.
3. An end-of-file condition is detected.

An error message is printed if the READ exceeds the core available, or if no input is available.

The maximum number of characters which can be brought into core with a R command is approximately 6,000 for an 8K system. Each additional 4K of core allows approximately 8,000 additional characters to be input.

## WRITE

The Write command moves lines of text from the text buffer to the output file (specified in the EW or EB command). The format of the command is

- nW Write n lines of text to the output file.
- 0W Write the text from the beginning of the current line to the pointer.
- /W Write the text from the pointer to the end of the buffer.

Write accepts all legal line-oriented arguments and does not move the pointer. If the buffer is empty when the write is executed, no characters are output.

### Examples:

- \*5W\$\$ Write the next 5 lines of text starting at the pointer, to the current output file.
- \*-2W\$\$ Write the previous 2 lines of text, starting at the pointer, to the current output file.
- \*B/W\$\$ Write the entire text buffer to the current output file.

## NEXT

The Next command writes the current text buffer to the output file, clears the buffer and reads in the next page of the input file. This command is equivalent to a combination of the Beginning, Write, Delete and Read commands (B/W/DR). The Next command can be repeated n times by specifying an argument before the command. The command format is:

nN

Next accepts only positive arguments (n) and leaves the pointer at the beginning of the buffer. An error message is printed if fewer than n pages are available on the input file. Next can be used to space forward, in page increments, through the input file.

### Examples:

- \*2N\$\$ Write the contents of the current text buffer to the output file. Read and write the next page of text. Then read another page into the text buffer.

## LIST

The List command prints the specified number of lines on the terminal. The format of the command is:

nL	Print n lines on the terminal beginning at the pointer.
0L	Print from the beginning of the current line up to the pointer.
/L	Print from the pointer to the end of the buffer.

List accepts all legal line-oriented arguments and does not move the pointer.

Examples:

*-2L\$\$	Print the previous 2 lines.
*4L\$\$	Print 4 lines beginning at the pointer.

Assuming the pointer location as follows:

```
MOV 5(R1),@R2
ADD  R1,(R2)+
```

the command:

*-1L\$\$	Prints the previous 1 1/2 lines up to the pointer.
----------	--

```
MOV 5(R1),@R2
ADD
```

### VERIFY

The Verify command prints the current text line on the terminal. The position of the pointer within the line has no effect and the pointer does not move. The command format is:

V

and there are no legal arguments. The V command is equivalent to a 0LL (List) command.

Example:

*V\$\$	The command causes the current line of text to be printed.
ADD R1,(R2)+	

### END FILE

The End File command closes the current output file. The End File does no input/output to the text buffers and does not move the pointer. The buffer contents are not affected.

The form of the command is:

EF

There are no legal arguments for the EF command. EF is used when the output file is to be closed as it stands, with no further output desired. Note that an implied EF command is included in EW. and EB commands.

#### EXIT

The Exit command is used to terminate editing and return control to the Monitor. It performs consecutive Next commands until the end of the input file is reached, then the input and output files are closed.

The command format is:

EX

There are no legal arguments for the EX command. Essentially, EXit is used to copy the remainder of the input file into the output file and return to the monitor. EXIT is legal only when there is an output file open. If an output file is not open and it is desired to terminate the editing session, return to the Monitor with CTRL/C.

#### NOTE

An EF or EX command is necessary in order to make an output file permanent. If CTRL/C is used to return to the Monitor without a prior execution of an EF command, the current output file is not saved.

As an example of the contrasting uses of EF and EX commands, the input file, SAMPLE, which contains several pages of text can be edited to make the first and second pages of the file into separate files called SAM1 and SAM2 as follows:

```
*EWSAM1$$  
*ERSAMPLE$$  
*RNEF$$  
*EWSAM2$$  
*NEF$$  
*EWSAMPLE$EX$$
```

#### 3.2.2.4 Commands to Move Location Pointer

##### BEGINNING

The Beginning command moves the current location pointer to the beginning of the text buffer. The command format is:

B

and there are no arguments.

Example:

Assuming the buffer contains:

```
MOVB 5(R1),@R2
ADD R1,(R2)+
CLR @R2
MOVB↑ 6(R1),@R2
```

the command:

```
*B$$
```

moves the pointer to:

```
↑MOVB 5(R1),@R2
```

### JUMP

The Jump command moves the pointer over the specified number of characters in the text buffer. The form of the command is:

nJ	Move the pointer n characters.
0J	Move the pointer to the beginning of the current line (equivalent to 0A).
/J	Move the pointer to the end of the text buffer (equivalent to /A).
=J	Move the pointer backward n characters, where n equals the length of the last text argument used.

Jump accepts all legal character-oriented arguments. Negative arguments move the pointer toward the beginning of the buffer, positive arguments toward the end. Jump treats CR, LF and form feed characters the same as any other character, counting one buffer position for each.

Examples:

```
*3J$$           Moves the pointer ahead three characters
```

```
*-4J$$          Moves the pointer back four characters
```

### ADVANCE

The Advance command moves the pointer the specified number of lines and leaves it at the beginning of the line.

The form of the command is:

nA	Advance the pointer over n carriage return/line feeds.
----	--

0A	Advance the pointer to the beginning of the current line. (Equivalent to 0J)
----	--

/A Advance the pointer to the end of the text buffer. (Equivalent to /J)

Advance accepts all legal line-oriented arguments. Advance and Jump commands perform the same function, but the first is line oriented, while the second is character-oriented.

Examples:

\*3A\$\$ Move the pointer ahead three lines

Assuming the buffer contains:

CLR @R2  
↑

the command

\*0A\$\$ Moves the pointer to

↑CLR @R2

### 3.2.2.5 Search Commands

#### GET

The Get command starts at the pointer and searches the current text buffer for the nth occurrence of the specified text string. If the search is successful, the pointer is left immediately following the nth occurrence of the text string. If the search fails, an error message is printed and the pointer is left at the end of the text buffer. The format of the command is:

nGtext\$

The argument (n), if specified, must be positive. The text string may be any length and immediately follows the G command. The search is made on the portion of the text between the pointer and the end of the buffer.

Example:

Assuming the buffer contains:

↑MOV PC,R1  
ADD #DRIV-.,R1  
MOV #VECT,R2  
CLR @R2  
MOVB 5(R1),@R2  
ADD R1,(R2)+  
CLR @R2  
MOVB 6(R1),@R2

The command:

\*GADD\$\$

positions the pointer at:

```
ADD↑ #DRIV-.,R1
```

The command:

```
*3G@R2$$
```

positions the pointer at:

```
ADD R1,(R2)+  
CLR @R2↑
```

To position the pointer at the beginning of the desired text, use the GET command in combination with the =J command:

```
*GTEST$=J$$
```

This command combination places the pointer before instead of after the text.

## FIND

The Find command starts at the current pointer and searches the entire input file for the nth occurrence of the text string. If the nth occurrence of the text string is not found in the current buffer, a Next command is automatically performed and the search is continued on the new text in the buffer. If the search is successful, the pointer is left immediately following the nth occurrence of the text string. If the search fails (i.e., the end-of-file is detected for the input file and the nth occurrence of the text string has not been found), an error message is printed and the pointer is left at the beginning of an empty text buffer.

The form of the command is:

```
nFtext$
```

The argument (n), if specified, must be positive.

An F command specifying a nonexistent search string can be used to copy all remaining text from the input file to the output file, instead of the EXIT command (which returns to the Monitor when execution is complete).

Find is a combination of Get and Next commands.

Example:

```
*2FMOVB 6(R1),@R2$$
```

Searches the entire input file for the second occurrence of the text string. Each unsuccessfully searched buffer is written to the output file.



## POSITION

The Position command searches the input file for the nth occurrence of the text string. If the desired text string is not found in the current buffer, the buffer is cleared and a new page is read from the input file. The format of the command is:

nPtext\$

The argument (n), if specified, must be positive. When a P command is executed the current contents of the buffer are searched from the location of the pointer to the end of the buffer. If the search is unsuccessful, the buffer is cleared and a new page of text is read and the cycle is continued.

If the search is successful, the pointer is positioned after the nth occurrence of the text. If it is not, the pointer is left at the beginning of an empty text buffer.

The Position command is a combination of the Get, Delete and Read commands.

The Position command is most useful as a means of placing the location pointer in the input file. For example, if the aim of the editing session is to create a new file out of the second half of the input file, a Position search will save time.

The difference between the Find and Position command is that Find writes the contents of the searched buffer to the output file while Position deletes the contents of the buffer after it is searched.

Example:

\*PADD R1,(R2)+\$\$ Searches the entire input file for the specified string ignoring the unsuccessfully searched buffers.

### 3.2.2.6 Commands to Modify the Text

#### INSERT

The Insert command inserts the specified text in the text buffer, starting at the current pointer position. The location pointer is positioned after the last character of the insert. The command format is:

Itext\$

There are no arguments to the Insert command, and the text string is limited only by the size of the text buffer and the space available. All characters except ALTMODE are legal in the text string. ALTMODE terminates the text string.

## NOTE

Forgetting to type the I command will cause the text entered to be executed as commands.

EDIT automatically protects against overflowing the text buffer during an Insert. If the I command is the first command in a multiple command line, EDIT ensures that there will be enough space for the Insert to be executed at least once.

If repetition of the command exceeds the available core, an error message is printed.

Example:

*IMOV	#BUFF,R2	Inserts the specified text at
MOV	#LINE,R1	the current location of the
MOVB	-1(R2),R0\$\$	pointer and leaves the pointer
		after R0.
*		

## DELETE

The Delete command removes the specified number of characters from the text buffer. Characters are deleted starting at the pointer and upon completion the pointer is positioned at the first character following the deleted text.

The form of the command is:

nD	Delete n characters.
0D	Delete from pointer to beginning of current line (equivalent to 0K).
/D	Delete from pointer to end of text buffer (equivalent to /K).
=D	Delete n characters to the left of the pointer, where n equals the length of the last text argument used.

Delete accepts all legal character-oriented arguments. Positive arguments delete toward the end of the buffer, negative arguments toward the beginning.

Example:

*-2D\$\$	Deletes the two characters immediately preceding the pointer.
----------	---

Assuming a buffer of:

ADD	R1,(R2)+
CLR	↑@R2

the command:

\*0D\$\$

would leave the buffer with:

```
ADD      R1,(R2)+
↑@R2
```

To delete a given item, combine a search with the =D command:

\*FMONEY\$=D\$\$

deletes the next occurrence of "money" from the file being edited.

### KILL

The Kill command removes n lines from the text buffer. Lines are deleted starting at the location pointer and the pointer is positioned at the beginning of the line following the deleted text. The command format is:

nK	Delete n lines from the text buffer.
OK	Delete from the beginning of the current line to the pointer (equivalent to 0D).
/K	Delete from the pointer to the end of the text buffer (equivalent to /D).

Kill accepts all legal line-oriented arguments. Positive arguments Kill toward the end of the buffer; negative arguments toward the beginning.

Examples:

*2K\$\$	Deletes two lines starting at the current location of the pointer.
---------	--

Assuming a buffer of:

```
ADD      R1,(R2)+
CLR↑    @R2
MOVB    6(R1),@R2
```

The command:

\*/K\$\$

Alters the contents of the buffer to:

```
ADD      - R1,(R2)+
CLR↑
```

Kill and Delete commands perform the same function, except that Kill is line oriented and Delete is character oriented.

## CHANGE

The Change command replaces *n* characters, starting at the pointer, with the specified text string.

The form of the command is:

- `nCtext$` Replace *n* characters with the specified text.
- `0Ctext$` Replace the characters from the beginning of the line up to the pointer with the specified text (equivalent to `0X`).
- `/Ctext$` Replace the characters from the pointer to the end of the buffer with the specified text (equivalent to `/X`).
- `=Ctext$` Replace *n* characters to the left of the pointer with the indicated text string, where *n* represents the length of the last text argument used.

The size of the text is limited only by the size of the text buffer and the space available. All characters are legal except `ALTMODE` which terminates the text string.

The Change command is identical to a Delete followed by an Insert (`nDItext$`), and accepts all legal character-oriented arguments.

If there is space available so a Change command can be typed in, it will be executed at least once. (It must be the first command of string). If repetition of the command exceeds the available core, an error message is printed.

Examples:

`*5C#VECT$$` Replaces the five characters to the right of the pointer with `#VECT`.

Assuming a buffer of:

```
CLR      @R2
MOV↑    5(R1),@R2
```

the command:

```
*0CADDB$$
```

would leave the buffer with:

```
CLR      @R2
ADDB↑   5(R1),@R2
```

To replace a given text string with another, combine a search with the `=C` command.

```
*GAAA$=CBBBB$
```

replaces the next occurrence of "AAAA" with "BBBB".

## EXCHANGE

The Exchange command replaces *n* lines with the text string starting at the pointer.

The form of the command is:

```
nXtext$   Replace n lines with the specified text.
0Xtext$   Replace the current line from the beginning to the
           pointer with the specified text (equivalent to
           0C).
/Xtext$   Replace the lines from the pointer to the end of
           the buffer with the specified text (equivalent to
           /C).
```

All characters are legal in the text string except ALTMODE which terminates the text.

The Exchange command is identical to a Kill command followed by an Insert (nKIttext\$), and accepts all legal line-oriented arguments.

If there is space available so the X command can be typed in, it will be executed at least once provided it is the first command in the string. If repetition of the command exceeds the available core, an error message is printed.

Example:

```
*2XADD R1,(R2)+   Replaces the two lines to
CLR @R2           the right of the pointer location
$$               with the text string.
*
```

The Change and Exchange commands perform the same function; Change is character oriented, Exchange is line oriented.

### 3.2.2.7 Utility Commands

#### SAVE

The Save command starts at the pointer and copies the specified number of lines into the Save Buffer.

The form of the command is:

```
nS
```

The argument (*n*) must be positive.

The pointer position does not change and the contents of the text buffer are not altered. Each time a Save is executed, the previous contents of the Save Buffer, if any, are destroyed. If the Save command causes the Save Buffer to exceed the core available, an error message is printed.

Example:

Assuming the text buffer contains the following assembly language subroutine,

```
↑;SUBROUTINE MSGTYP
;WHEN CALLED, EXPECTS R0 TO POINT TO AN
;ASCII MESSAGE THAT ENDS IN A ZERO BYTE,
;TYPES THAT MESSAGE ON THE USER TERMINAL

      .ASECT
      .=1000
MSGTYP:  TSTB (%0)           ;DONE?
        BEQ MDONE         ;YES-RETURN
MLOOP:  TSTB @#177564      ;NO-IS TERMINAL READY?
        BPL MLOOP        ;NO-WAIT
        MOVB (%0)+,@#177566 ;YES PRINT CHARACTER
        BR MSGTYP        ;LOOP
MDONE:  RTS %7           ;RETURN
```

The command:

\*14S

Stores the entire subroutine in the Save Buffer, so it may be inserted in a program when needed.

The Save command is useful for moving blocks of text or inserting the same block of text in several places.

### UNSAVE

The Unsave command inserts the entire contents of the Save Buffer in the text buffer at the pointer location. The pointer is positioned following the inserted text.

The form of the command is:

U     Insert in the text buffer the contents of the Save Buffer.

OU    Clear the Save Buffer and reclaim the area for text.

Zero is the only legal argument to the U command.

The contents of the Save Buffer are not destroyed by the Unsave command and may be Unsaved as many times as desired.

If the Unsave command exceeds the core available, an error message is displayed.

### MACRO

The Macro command is a special case command which inserts a command string into the EDIT Macro Buffer. The Macro command is of the form:

M/command string/

where / represents the delimiter character. The delimiter is always the first character following the M command, and may be any character which does not appear in the macro command string.

Starting with the character following the delimiter, EDIT places the macro command string characters into its internal Macro Buffer until the delimiter is encountered again. At this point, EDIT returns to command mode.

The Macro command does not execute the macro string; it merely allows specification of the macro for later execution by the Execute Macro (EM) command. Macro does not affect the contents of the Text Buffer or Save Buffer.

All characters except the delimiter are legal macro command string characters, including single ALTMODE's to terminate text commands. All commands, except the M and EM commands, are legal in a command string macro. That is, EDIT macro operations are not recursive.

All arguments except 0 are illegal with the M command. A OM command clears the macro buffer and reclaims the space for use as part of the text buffer.

Typing the M command immediately followed by two identical characters (assumed to be delimiters) and two ALTMODE characters also clears the Macro Buffer. For example:

```
M//$$
```

Example:

```
*M/GRO$-C1$/$$      Stores a macro to change R0 to R1.
```

#### Note

Be careful to chose infrequently used characters as macro delimiters; use of frequently used characters can lead to inadvertent errors. For example,

```
*M GMOV R0$=CADD R1$ $$
```

It was intended that the macro be GMOV R0\$=CADD R1\$ but since the delimiter character (the character following the M) is a space, the space following MOV is the second delimiter, terminating the macro. EDIT then returns an error when the 0\$= becomes an illegal command structure.

#### EXECUTE MACRO

The Execute Macro command executes the command string specified in the last Macro command.

The form of the command is:

```
nEM
```

The argument (n) must be positive.

The macro is executed n times and control returns to the next command in the string.

Examples:

```
*M/BGRO$-C1$/$$  
*B1000EM$$  
?*SRCH FAIL IN MACRO*?
```

Executes the MACRO stored in the previous instruction, which returns an error message when the end of buffer is reached. The macro effectively changed all occurrences of R0 in the text buffer to R1.



\*IMOV PC,R1\$2EMICLR @R2\$\$      Inserts MOV PC,R1; executes  
the command in the MACRO  
buffer twice, then inserts  
CLR,@R2.

### EDIT VERSION

The Edit Version command displays the version number of the Editor on the console terminal.

The form of the command is

EV\$\$

Example:

```
*EV$$
V01-24
*
```

### 3.3 ERROR MESSAGES

Prior to execution, EDIT checks the command string for syntactical errors. If any errors exist, they are reported with a message of the form

?message?

If no syntactical errors are found, execution of the commands begins. If errors are found during command execution, the form of the message output is

?\*message\*?

If the error is detected within a command macro, the message format is

?message IN MACRO?

or

?\*message IN MACRO\*?

depending on when it is detected.

Table 3-2 lists the EDIT error messages.

Table 3-2

## EDIT Error Messages

Message	Explanation
?CB FULL?	Command exceeds the space allowed for a command string.
?ILL ARG?	The argument specified was illegal with that command. A negative argument was specified where a positive one is expected or argument exceeds the range + or - 16,383.
?ILL CMD?	EDIT does not recognize the command specified.
?ILL MAC?	Delimiters were improperly used, or an attempt was made to enter an M command, or an EM command within a macro.
?*DIR FULL*?	No room in device directory for output file.
?*EOF*?	Attempted a Read, Next or file searching command and no data was available.
?*FILE FULL*?	Available space for an output file is full. Type a CTRL/C and a Close command to save the data already written.
?*FILE NOT FND*?	Attempted to open a nonexisting file for editing.
?*HDW ERR*?	A hardware error occurred during I/O. May be caused by WRITE LOCKed device. Try again.
?*ILL DEV*?	Attempted to open a file on an illegal device.
?*ILL NAME*?	File name specified in EB, EW, or ER is illegal.
?*NO FILE*?	Attempted to read or write when no file is open.
?*NO ROOM*?	Attempted to Insert, Save, Unsave, Read, Next, Change or Exchange when there was not enough room in the appropriate buffer. Delete unwanted buffers to create more room or write text to the output file.
?*SRCH FAIL*?	The text string specified in a Get, Find or Position command was not found in the available data.
?"<>"ERR?	Iteration brackets are nested too deeply or used illegally or brackets are not matched.

### 3.4 EDIT EXAMPLE

The following example illustrates the use of some of the EDIT commands to change a program stored on the device DK. Sections of the terminal output are coded by letter and corresponding explanations follow the example.

```

A.  { .R EDIT
      *ERDK:TEST1.MAC$$
      *EWDK:TEST2.MAC$$
      *R$$

      { */L$$
        ;TEST PROGRAM

B.  { START:  MOV #1000,%6      ;INITIALIZE STACK
      MOV #MSG,%0      ;POINT R0 TO MESSAGE
      JSR PC,MSGTYP    ;PRINT IT
      HALT             ;STOP
      MSG:  .ASCII/IT WORKS/
            .BYTE 15
            .BYTE 12
            .BYTE 0

C.  *B 1J 5D$$

D.  { *GPROGRAM$$
      *0L$$
      ;PROGRAM *I TO TEST SUBROUTINE MSGTYP. TYPES
E.  { ;"THE TEST PROGRAM WORKS"
      ;ON THE TELETYPE\EPYTELET\TERMINAL$$

F.  { *F.ASCII/$$
      *8CTHE TEST PROGRAM WORKS$$

G.  { *P.BYTE ↑X
      *F.BYTE 0$V$$
      .BYTE 0

H.  { *I
      .END
      $B/L$$
      ;PROGRAM TO TEST SUBROUTINE MSGTYP. TYPES
      ;"THE TEST PROGRAM WORKS"
      ;ON THE TERMINAL
      START:  MOV #1000,%6      ;INITIALIZE STACK
      MOV #MSG,%0      ;POINT R0 TO MESSAGE
      JSR PC,MSGTYP    ;PRINT IT
      HALT             ;STOP
      MSG:  .ASCII/THE TEST PROGRAM WORKS/
            .BYTE 15
            .BYTE 12
            .BYTE 0
            .END

I.  { *EX$$
      .
  
```

- A The EDIT program is called and prints an \*. The input file is TEST1.MAC; the output file is TEST2.MAC and the first page of input is read.
- B The buffer contents are listed.
- C Be sure the pointer is at the beginning of the buffer. Advance pointer one character (past the ;) and delete the "TEST".
- D Position pointer after PROGRAM and verify the position by listing up to the pointer.
- E Insert text. RUBOUT used to correct typing error.
- F Search for .ASCII/ and change "IT WORKS" to "THE TEST PROGRAM WORKS".
- G CTRL/X typed to cancel P command. Search for ".BYTE 0" and verify location of pointer with V command.
- H Insert text. Return pointer to beginning of buffer and list entire contents of buffer.
- I Close input and output files after copying the current text buffer as well as the rest of input file into output file. EDIT returns control to the Monitor.

## CHAPTER 4

### PERIPHERAL INTERCHANGE PROGRAM (PIP)

The Peripheral Interchange Program (PIP) is the file transfer and maintenance utility for RT-11. PIP is used to transfer files between devices, merge and delete files, and list, zero, and compress directories.

#### 4.1 CALLING AND USING PIP

To call PIP from the system device type:

```
R PIP
```

in response to the dot printed by the Keyboard Monitor. The Command String Interpreter then prints an asterisk at the left margin of the terminal and waits to receive a line of I/O files and command switches. PIP accepts up to six input file names and three output file names; command switches generally are placed at the end of the command string but may follow any file name in the string.

Since PIP performs file transfers for all file types (ASCII, formatted binary, image or SAVE format), there are no assumed extensions assigned by PIP to file names for either input or output files. All extensions, where present, must be explicitly specified.

Following completion of a PIP operation, the Command String Interpreter again prints an asterisk at the left margin and waits for another PIP I/O specification line. Typing CTRL/C returns control to the Keyboard Monitor.

PIP follows the standard file specification syntax explained in paragraph 2.2 with one exception--the asterisk character (\*) can be used in a file specification to replace a file name or extension.

The asterisk may be used to replace a file name and/or extension as follows:

<u>Specification to be replaced</u>	<u>Form</u>
file name	*.ext
extension	filnam.*
file name and extension	*.*

Use of the asterisk (called the "wild card") in a file specification means "all". For instance, "\*.MAC" means all files with the extension .MAC. The wild card character is legal only in the following cases:

1. Input file specification for the copy and multiple copy commands (no switch, /I, and /A). For example,

\*FILE=\*.MAC/I

\*SOURCE.MAC=\*.MAC/A

2. File specification for the delete and extend commands (/D and /T). For example,

\*TEST.\*/D

\*DATA.\*[100]=/T

3. Input and output file specifications for the rename command (/R). For example,

\*\*TST=\*.BAK/R

4. Input and output file specifications for the multiple copy command (/X). For example,

\*DT0:\*.\*=DT1:\*/X

Operations on files specified by the \* are performed in the order the files appear in the directory.

System files with the extension .SYS are ignored when the wild card character is used unless the /Y switch is specified.

Examples:

**BAK/D	Causes all files with the extension .BAK to be deleted regardless of their file names.
*FILEA.*/D/Y	Causes all files with the name FILEA to be deleted, regardless of extension.
**.*=*/X/Y	Transfers all files regardless of file name or extension.

#### 4.2 PIP COMMANDS

The various commands allowed on a PIP I/O specification line are summarized in Table 4-1. If no command switch is specified, PIP assumes the operation is a file transfer in image mode.

Table 4-1

## PIP COMMANDS

Switch	Explanation
/A	Copies file(s) in ASCII mode (ignores nulls; converts to 7-bit ASCII).
/B	Copies files in formatted binary. <i>(Temporary copy)</i>
/D	Deletes file(s) from specified device.
/E	Lists the entire directory including unused spaces and their sizes.
/F	Prints the short directory (file names only) of the specified device.
/G	Ignores any input errors which occur during a file transfer and continues copying.
/I or no switch	Copies file(s) in image mode (byte by byte).
/L	Lists the entire directory of the specified device.
/N	Used with /Z to specify the number of directory blocks to allocate to the directory.
/O	Bootstraps the specified device (DT0 or RK0 only).
/R	Renames the specified file.
/S	Compresses the file on the specified directory device so all free area is combined.
/T	Extends number of blocks allocated for a file.
/U	Copies the specified bootstrap file into absolute blocks 0 and 2 of the specified device.
/V	Outputs the version number of the PIP program being used to the terminal.
/X	Copies files individually (without concatenation).
/Y	Causes system files to be operated on by the command specified. Attempted modifications or deletions of .SYS files without /Y are null operations, and cause the message ?NO SYS ACTION? to be printed.
/Z	Zeroes (initializes) the directory of the specified device and allows specification of directory size and directory entry size when used with /N.

#### 4.2.1 Copy Commands

A file specification without a command switch copies files onto the destination device in image mode (byte by byte) and is used to transfer core image (Save format) files and any files other than ASCII or formatted binary. For example:

\*ABC<XYZ                   Copies XYZ onto the same device (DK) and assigns the name ABC.

\*SY:BACK<PR:/I           Transfers a tape from the paper tape reader to the system device under the name BACK.

The /A switch is used to copy file(s) in ASCII mode as follows:

\*DT1:F1<F2/A           Copies F2 onto device DT1 in ASCII mode and assigns the name F1.

Nulls are ignored in an ASCII mode file transfer. The /B switch is used to transfer formatted binary files. The formatted binary copy switch should be used for .OBJ files produced by the assembler and .LDA files produced by the Linker. When doing formatted binary transfers, PIP verifies checksums and prints the message ?CHK SUM? if a check sum error occurs.

\*DK:PIP.OBJ<PR:/B       Transfers a formatted binary file from the papertape reader to device DK and assigns the name PIP.OBJ.

To concatenate more than one file into a single file use the following format:

\*DK:AA<DT1:BB,CC,DD/I   Transfers files BB, CC and DD to the device DK as one file and assigns the name AA.

\*DT3:MERGE<DT2:FILE2,FILE3/A   Merges ASCII files, FILE2 and FILE3, on DT2 into one ASCII file, MERGE, on device DT3.

Errors which occur during the copy operation (such as a parity error) cause PIP to output an error message and return for another command string.

The /G switch is used to copy files and ignore input errors. For example:

\*ABC<DT1:TOP/G           Copies file TOP in image mode from device DT1 to device DK and assigns the name ABC.

\*DT2:COMB<DT1:F1,F2/A/G   Copies files F1 and F2 in ASCII mode from device DT1 to device DT2 as one file with the name COMB. Ignores input errors.



The wild card character may be used in the input file specification of transfer operations. Be sure to use the /Y switch if System files (.SYS) are to be copied. For example:

\*DT1:PROG1<\*.SAV/I Copies, in image mode, all files with the .SAV extension from device DK to device DT1 assigns the name PROG1.

\*DT2:NN3<ITEM1.\*,ITEM2/A Copies, in ASCII mode, all files labeled ITEM1 and file ITEM2 from device DK to device DT2 assigns the name NN3. .SYS files are ignored.

\*MARK<DT3:\*.\*/G/Y Copies, in image mode, all files from device DT3 to device DK; assigns the name MARK; ignores any input errors.

The file allocation scheme for RF-11 normally allots half the largest available space for a new file. Therefore, although the directory for a given device may show a free area of 200 blocks, PIP may return an ?OUT ER? message when a transfer is attempted to that device with a file greater than 100 blocks long. Transfers in this situation can be accomplished in either of two ways:

1. Use the [n] construction on the output file to specify the desired length.
2. Use the /X switch during the transfer to force PIP to allocate the correct number of blocks for the output file.

For example:

Assume we know from prior directory listings that there is a 200 block <unused> space on DT1, and that File A is 150 blocks long.

```
.R PIP
*DT1:A=A
?OUT ER? File longer than 100 blocks.
*DT1:A[150]=A
or Either command will cause a transfer.
DT1:A=A/X
*
```

#### 4.2.2 Multiple Copy Commands

The /X switch allows the transfer of several files at a time onto the destination device as individual files. The /A, /G, /B and /Y switches can be used with /X.

Examples:

```
*FILE1,FILE2,FILE3<DT1:FILEA,FILEB,FILEC/X
Copies, in image mode, FILEA, FILEB and
FILEC from device DT1 to device DK as
separate files called FILE1, FILE2 and
FILE3.
```

\*DT2:F1.\*<F2.\* /X      Copies, in image mode, all files named F2 from device DK to device DT2 (except files with .SYS or .BAD extensions) as separate files; assigns the name F1 and keeps the old extensions.

\*DT1:\*. \* <DT2:\*. \* /X      Copies, in image mode, all files on device DT2 to device DT1 (except files with .SYS or .BAD extension) as separate files keeping the same names and extensions.

\*DT1:FILE1,FILE2<FILEA.\* /A/G/X      Copies, in ASCII mode, all files named FILEA (except files with .SYS or .BAD extension) from device DK to device DT1 as separate files; assigns the names FILE1 and FILE2. The files are transferred in the order they are encountered in the directory; if there are more than two files named FILEA, no transfer takes place and PIP returns to command mode.

\*DT0:\*.SYS<\*.SYS /X/Y      Copies the system files from device DK to device DT0.

File transfers performed via normal transfers place the new file in the largest available area on the disk. The /X switch, however, places the copied files in the first free place large enough to accommodate it. Therefore, use the /X switch when it is desired to place a file in the first slot available.

For example, /X would be useful as an alternative to a [] specification to transfer a 150 block file into a 200 block area:

\*A[150]=B  
\*A/X=B

Example:

Directory of DT1:

9-MAY-73		
MONITR.SYS	32	5-MAY-73
<UNUSED>	2	
PR.SYS	2	5-MAY-73
<UNUSED>	438	

To copy file PP.SYS (2 blocks long) from DT0: to DT1:, the command:

\*DT1:PP.SYS=DT0:PP.SYS/Y

can be entered, and the new directory is:

```

          9-MAY-73
MONITR.SYS  32   5-MAY-73
<UNUSED>    2
PR.SYS      2   5-MAY-73
PP.SYS      2   9-MAY-73
<UNUSED>   436

```

If the command:

```
*DT1:PP.SYS=DT0:PP.SYS/Y/X
```

is entered, the new directory is

```

          9-MAY-73
MONITR.SYS  32   5-MAY-73
PP.SYS      2   9-MAY-73
PR.SYS      2   5-MAY-73
<UNUSED>   438

```

#### 4.2.3 Delete Command

The /D switch is used to delete one or more files from the specified device. The wild card character (\*) can be used in the file specification in a Delete command.

Only six files can be specified in a delete operation if each file to be deleted is individually named (i.e., if the wild card character is not used).

When a file is deleted, the information is not destroyed, the file name is merely removed from the directory. If a file has been deleted but not overwritten, it can be recovered with the /T switch by specifying a command of the form

```
x[n]=/T
```

where x is the name desired and n is the length of the deleted file.

For example,

```

*/E
  4-JUN-73
A      .MAC      18      3-JUN-73
B      .MAC      17      3-JUN-73
C      .MAC      19      3-JUN-73
<UNUSED>      512      3-JUN-73
512 FREE BLOCKS
*B.MAC/D
*/E
  4-JUN-73
A      .MAC      18      3-JUN-73
<UNUSED>      17      3-JUN-73
C      .MAC      19      3-JUN-73
529 FREE BLOCKS

```

File B.MAC could now be recovered by

```
*B.MAC[17]=/T
```

The /T switch looks for the first unused area large enough to accommodate the requested file length. If the file to be recovered is in the first area large enough to accommodate the size specified, the above is sufficient. If not, all larger unused spaces preceding the desired file would have to be given dummy names before the recovery could be made. The dummy names would be given as above.

For instance, assume the above example with the exception that A.MAC has a 33 block unused file before it.

```
*/E
  4-JUN-73
<UNUSED>          33
A.      MAC      18
<UNUSED>          17
C.      MAC      512
```

A recovery of B.MAC would require

```
*DUMMY[33]=/T
*B.MAC[17]=/T
```

If the 33 block unused area were not named prior to B.MAC, the first 17 blocks of the 33 block area would have become B.MAC.

Examples:

```
*FILE1.SV/D      Deletes FILE1.SV from device DK.

*DT1:*.*/D      Deletes all files from device DT1 except
                those with the .SYS or .BAD extension.
                If there is a file with a .SYS
                extension, the message ?NO SYS ACTION?
                is printed to remind the user that .SYS
                files have not been deleted.

**.MAC/D        Deletes all files with the .MAC
                extension from device DK.

*DT1:B1,DT2:R1,DT3:AA/D
                Deletes the files specified from the
                associated device.

**.* /D/Y      Deletes all files from device DK.
```

#### 4.2.4 Rename Command

The /R switch is used to rename the file given as input with the name given in the output specification. Only one file may be renamed in a Rename operation unless the wild card character is used. The /Y switch must be used in conjunction with /R to rename .SYS files.

The Rename command is particularly useful when a file on disk or DECTape contains bad blocks. By renaming the file with a .BAD extension, the file permanently resides in that area of the device. Once a file is given a .BAD extension it can not be renamed or moved

during a compress operation. .BAD files are not renamed in wild card operations.

Examples:

```
*DT1:F1<DT1:F0/R    Renames F0 to F1 on device DT1.  
*FILE1.*<FILE2.*R  Renames all files on device DK with the  
                    name FILE2 to FILE1 (except files with  
                    .SYS or .BAD extension), retaining the  
                    original extensions.
```

#### 4.2.5 Extend Command

The /T switch is used to increase the number of blocks allocated for the specified file. The /T switch requires a numeric argument of the form [n] where n is a decimal number which specifies the number of blocks in the file at the completion of the extend operation.

The format of a /T switch is:

```
dev:filnam.ext[n]=/T
```

A file can be extended only if it is followed on the specified device by an unused area of sufficient size to accommodate the additional length of the extended file.

It may be necessary to create this space by moving other files on the device (use PIP's /X switch).

Using the /T switch when specifying a file that does not currently exist creates a file of the specified length.

Error messages are printed if the /T command would make the specified file smaller (?EXT NEG?) or if there is insufficient space following the file (?ROOM?).

Examples:

```
*ABC[200]=/T        Assigns 200 blocks to file ABC on device  
                    DK.  
*DT1:XYZ.*[100]=/T Assigns 100 blocks to all files named  
                    XYZ on device DT1.
```

#### 4.2.6 Directory List Commands

The /L switch lists the entire directory of the specified device. The listing contains the current date, all files with their associated lengths and dates, and total free blocks on the device. The file lengths and number of free blocks are decimal values.

If no output device is specified, the directory is output to the terminal (TT:).

Examples:

\*DT1:/L                    Outputs full directory of device DT1 to the terminal.

\*DIRECT<DT3:/L           Outputs full directory of device DT3 to a file, DIRECT, on the device DK.

\*\* .MAC/L                 Lists a directory of files with the .MAC extension.

The /E switch lists the entire directory including the unused areas and their sizes in blocks (decimal).

Examples:

\*/E                        Outputs to the terminal a complete directory of the device DK including size of unused areas.

\*LP:<DT1:/E               Outputs to the line printer a complete directory of device DT1 including size of unused areas.

The /F switch lists only the current date, file names and total free blocks in the directory, omitting the file lengths and associated dates.

Examples:

\*/F                        Outputs a file name directory of the device DK to the terminal.

\*LP:</F                    Outputs file name directory of the device DK to the line printer.

The /L, /E and /F commands have no effect on the files on the specified device.

Use of the asterisk (\*) or file names in a file specification with an /L, /F, or /E switch lists those files from the directory of the specified device.

If a file which exists on the device specified is included in a command with a /L, /E or /F switch, the file name and optionally the date and file length are output. For example,

F1.SAV/L

causes

```
      4-JUN-73
F1    .SAV    18    3-JUN-73
<UNUSED>    512
512 FREE BLOCKS
```

to be output if the file exists on device DK:

Directory listings of directories with multiple segments contain blank lines at segment boundaries.

#### 4.2.7 Directory Initialization Command

The /Z switch clears and initializes the directory on the specified device. The /Z should always be used to create an empty file directory before using a DECTape or disk for the first time.

The form of command is:

```
/Z:n
```

where n is an octal number specifying the number of extra words per directory entry. If n is not specified, no extra words are allocated, and 70 entries can be made in a directory block. When extra words are allocated, the formula for determining the number of entries per directory block is:

$$507 \div ((\# \text{ of extra words}) + 7)$$

For example, if the switch /Z:1 is used, 63 entries can be made per block.

When /Z is specified, PIP replies:

```
device/Z ARE YOU SURE?
```

For example,

```
DT1:/Z ARE YOU SURE?
```

Answer Y and a carriage return. If answer begins with a character other than Y it is considered to be no.

Example:

```
DT1:/Z
*DT1:/Z ARE YOU SURE?Y
Zeros the directory on device DT1, and
allocates no extra words for the
directory.
```

The /N switch is used with /Z to specify the number of directory blocks to allocate to the directory. The form of the switch is:

```
/N:n
```

where n is an octal number. If n is not specified, four blocks are allocated. The maximum number of blocks which can be allocated is  $37_8$ .

Example:

```
*/Z:2/N:6
Zeroes the directory on device DK,
allocates two extra words per directory
entry and allocates six directory
blocks.
```

#### 4.2.8 Compress Command

The /S switch is used to compress the directory and files on the specified device so all the free (unused) blocks are condensed into one area. /S can also be used to copy DECTapes and disks (/S will not copy the bootstrap file in absolute blocks 0 and 2).

/S does not move files with the .BAD extension. This feature provides protection against reusing bad blocks which may occur on a disk. Files containing bad blocks can be renamed with the .BAD extension and then left in place when a /S is executed.

If a compress operation is performed on the systems device, the message

?REBOOT?

is printed to indicate that it may be necessary to reboot the system.

If .SYS files were not moved during the compress operation, it is not necessary to reboot the system.

Rebooting the system in response to the ?REBOOT? warning message should ONLY be done AFTER the operation which generated the message is complete. ?REBOOT? does not signify that the system should be rebooted immediately; the user should wait for the "\*" signifying that PIP is ready for another command before rebooting.

If the command attempts to compress a large device to a smaller one, an error results and the directory of the smaller device is zeroed.

Examples:

\*SY:/S                      Compresses the files on the system device SY:

\*DT1:A<DT2:/S              Transfers and compresses the files from device DT2 to device DT1. Device DT2 is not changed. The file name A is a dummy specification required by the Command String Interpreter.

#### 4.2.9 Bootstrap Copy Command

The bootstrap copy command (/U) copies the bootstrap portion of the specified file into absolute blocks 0 and 2 of the specified device.

Examples:

\*DK:A<DK:MONITR.SYS/U      Writes the bootstrap file MONITR.SYS in blocks 0 and 2 of the device DK. A is a dummy file name.



\*DT0:F2<DK:MONITR.SYS/U

Writes the bootstrap file MONITR.SYS into blocks 0 and 2 of the device DT0. The file name F2 is a dummy specification required by the Command String Interpreter.

#### 4.2.10 Boot Command

The boot command reboots the system, reinitializing monitor tables and returning the system to the monitor level. The boot command performs the same operation as a hardware bootstrap.

Example:

\*DK:/O Reboots the device DK.

If a boot command is specified on a non-file structured device, the message

?BAD BOOT?

is printed.

#### 4.2.11 Version Command

The Version command (/V) outputs a version number message to the terminal of the form

PIP V00-00

### 4.3 ERROR MESSAGES

The following error messages can be output by PIP.

<u>Errors</u>	<u>Meaning</u>
?BAD BOOT?	Attempted to bootstrap a non-file structured device.
?CHK SUM?	A checksum error occurred in a formatted binary transfer.
?COR OVR?	Core overflow - too many devices and/or file specifications (usually *.* operations) and no room for buffers.
?DEV FUL?	No room on device for file.
?ER RD DIR?	Unrecoverable error reading directory. Make sure device is ready. Try reformatting device.

<u>ERRORS</u>	<u>Meaning</u>
?ER WR DIR?	Unrecoverable error writing directory. Try again.
?EXT NEG?	A /T command attempted to make file smaller.
?FIL NOT FND?	File not found in delete, copy, rename operation.
?ILL DEV?	Illegal or nonexistent device.
?ILL SWT?	Illegal switch or switch combination.
?IN ER?	Unrecoverable error reading file. Try again (This error is ignored during /G operation.)
?OUT ER?	Unrecoverable error writing file. Perhaps a hardware or checksum error; try recopying file. Also, caused by an attempt to compress a larger device to a smaller one or not enough room when creating a file. The system takes the largest space available and divides it in half before attempting to insert the file. Try the [] construction or /X switch.
?OUT FIL?	Illegal output file specification or missing output file.
?ROOM?	Insufficient space following file specified with a /T switch.

The following warning messages are output by PIP.

?NO SYS ACTION?	The /Y switch was not included with a command specified on a .SYS file. The command is executed for all but the .SYS files. A *.* transfer is most likely to cause this message.
?REBOOT?	.SYS files have been transferred, renamed, compressed or deleted from the systems device. It may be necessary to reboot the system. If any of the .SYS files in use by the current system (MONITR.SYS and handler files) have been physically moved on the system device, it is necessary to reboot the system immediately. If not, this message can be ignored. If the cause of the message was a /S operation, the system need be rebooted only if there was an empty space before any of the .SYS files. The need to reboot can be permanently avoided by placing all .SYS files at the beginning of the systems device, then avoiding their involvements in PIP operations by avoiding the /Y switch.

## CHAPTER 5

### MACRO ASSEMBLER

MACRO is the RT-11 Assembler for system configurations of 12K or more. Users with 8K configurations must use ASEMBL and EXPAND and should read this chapter and Chapters 9 and 10 before using ASEMBL and EXPAND. The MACRO features not supported by ASEMBL are indicated in this chapter. Many of the features not available in ASEMBL are supported by EXPAND.

Some notable features of MACRO are:

1. Program control of assembly functions.
2. Device and file name specifications for input and output files
3. Error listing on command output device
4. Alphabetized, formatted symbol table listing
5. Relocatable object modules
6. Global symbols for linking between object modules
7. Conditional assembly directives
8. Program sectioning directives
9. User defined macros
10. Comprehensive set of system macros
11. Extensive listing control

#### 5.1 SOURCE PROGRAM FORMAT

A source program is composed of a sequence of source lines, where each line contains a single assembly language statement. Each line is terminated by a line feed character (which increments the line count by 1) or a form feed character (which increments both the line count and page count by 1).

Since Edit automatically appends a line feed to every carriage return character, the user need not concern himself with the statement terminator. However, a carriage return character not followed by a statement terminator generates an error flag. A legal statement terminator not immediately preceded by a carriage return causes the Assembler to insert a carriage return character for listing purposes.

An assembly language line can contain up to 132(10) characters (exclusive of the statement terminator). Beyond this limit, excess characters are ignored and generate an error flag.

### 5.1.1 Statement Format

A statement can contain up to four fields which are identified by order of appearance and by specified terminating characters. The general format of a MACRO-11 assembly language statement is:

```
label:   operator operand ;comments
```

The label and comment fields are optional. The operator and operand fields are interdependent; either may be omitted depending upon the contents of the other.

The Assembler interprets and processes these statements one by one, generating one or more binary instructions or data words or performing an assembly process. A statement contains one of these fields and may contain all four types. Blank lines are legal.

Some statements have one operand, for example:

```
CLR R0
```

while others have two, for example:

```
MOV #344,R2
```

An assembly language statement must be complete on one source line. No continuation lines are allowed. (If a continuation is attempted with a line feed, the Assembler interprets this as the statement terminator.)

MACRO source statements may be formatted with Edit such that use of the TAB character causes the statement fields to be aligned. For example:

<u>Label</u> <u>Field</u>	<u>Operator</u> <u>Field</u>	<u>Operand</u> <u>Field</u>	<u>Comment</u> <u>Field</u>
MASK=-10			;REGISTER EXPRESSION
REGEXP:			;MUST BE ABSOLUTE
	ABSEXP		
REGTST:	BIT	#MASK,VALUE	;3 BITS?
	BEQ	REGEX	;YES, OK
REGERR:	ERROR	R	;NO, ERROR
REGEX:	MOV	#DEFFLG!REGFLG,MODE	
	BIC	#MASK,VALUE	
	BR	ABSERX	

#### 5.1.1.1 Label Field

A label is a user-defined symbol which is assigned the value of the current location counter and entered into the user-defined symbol table. The value of the label may be either absolute or relocatable, depending on whether the location counter value is currently absolute or relocatable. In the latter case, the absolute value of the symbol is assigned by Link, i.e., the stated relocatable value plus the relocation constant.

A label is a symbolic means of referring to a specific location within a program. If present, a label always occurs first in a statement and must be terminated by a colon. For example, if the current location is absolute 100(8), the statement:

```
ABCD:    MOV  A,B
```

assigns the value 100(8) to the label ABCD. Subsequent reference to ABCD references location 100(8). In this example if the location counter were relocatable, the final value of ABCD would be 100(8)+K, where K is the location of the beginning of the relocatable section in which the label ABCD appears.

More than one label may appear within a single label field; each label within the field has the same value. For example, if the current location counter is 100(8), the multiple labels in the statement:

```
ABC:      $DD:      A7.7:      MOV  A,B
```

cause each of the three labels ABC, \$DD, and A7.7 to be equated to the value 100(8). (By convention, \$ and . characters are reserved for use in system software symbols.)

The first six characters of a label are significant. An error code is generated if different labels share the same first six characters.

A symbol used as a label may not be redefined within the user program. An attempt to redefine a label results in an error flag in the assembly listing.

#### 5.1.1.2 Operator Field

An operator field follows the label field in a statement, and may contain a macro call, an instruction mnemonic, or an assembler directive. The operator may be preceded by none, one or more labels and may be followed by one or more operands and/or a comment. Leading and trailing spaces and tabs are ignored.

When the operator is a macro call, the Assembler inserts the appropriate code to expand the macro. When the operator is an instruction mnemonic, it specifies the instruction to be generated and the action to be performed on any operand(s) which follow. When the operator is an Assembler directive, it specifies a certain function or action to be performed during assembly.

An operator is legally terminated by a space, tab, or any non-alphanumeric character (symbol component).

Consider the following examples

```
MOV A,B   (space terminates the operator MOV)
MOV@A,B   (@ terminates the operator MOV)
```

When the statement line does not contain an operand or comment, the operator is terminated by a carriage return followed by a line feed or form feed character.

A blank operator field is interpreted as a .WORD assembler directive (See Section 5.5.3.2).

#### 5.1.1.3 Operand Field

An operand is that part of a statement which is manipulated by the operator. Operands may be expressions, numbers, or symbolic or macro arguments (within the context of the operation). When multiple operands appear within a statement, each is separated from the next by one of the following characters: comma, tab, space or paired angle brackets around one or more operands (see Section 5.2.1.1). An operand may be preceded by an operator, label or another operand and followed by a comment.

The operand field is terminated by a semicolon when followed by a comment, or by a statement terminator when the operand completes the statement. For example:

```
LABEL:   MOV A,B ;COMMENT
```

The space between MOV and A terminates the operator field and begins the operand field; a comma separates the operands A and B; a semicolon terminates the operand field and begins the comment field.

#### 5.1.1.4 Comment Field

The comment field is optional and may contain any ASCII characters except null, rubout, carriage return, line feed, vertical tab or form feed. All other characters, even special characters with a defined usage, are ignored by the Assembler when appearing in the comment field.

The comment field may be preceded by one, any, none or all of the other three field types. Comments must begin with the semicolon character and end with a statement terminator.

Comments do not affect assembly processing or program execution, but are useful in source listings for later analysis, debugging, or documentation purposes.

#### 5.1.2 Format Control

Horizontal or line formatting of the source program is controlled by the space and tab characters. These characters have no effect on the assembly process unless they are embedded within a symbol, number, or ASCII text; or unless they are used as the operator field terminator. Thus, these characters can be used to provide an orderly source program. A statement can be written:

```
LABEL:MOV(SP)+,TAG;POP VALUE OFF STACK
```

or, using formatting characters, it can be written:

LABEL: MOV (SP)+,TAG ;POP VALUE OFF STACK

which is easier to read in the context of a source program listing.

Vertical formatting, i.e., page size, is controlled by the form feed character. A page of n lines is created by inserting a form feed (type the CTRL/FORM keys on the keyboard) after the nth line. (See also Section 5.5.1.6 for a description of page formatting with respect to macros and Section 5.5.1.3 for a description of assembly listing output.)

## 5.2 SYMBOLS AND EXPRESSIONS

This section describes the various components of legal MACRO expressions; the Assembler character set, symbol construction, numbers, operators, terms and expressions.

### 5.2.1 Character Set

The following characters are legal in MACRO source programs:

1. The letters A through Z. Both upper and lower case letters are acceptable, although, upon input, lower case letters are converted to upper case letters. Lower case letters can only be output by sending their ASCII values to the output device. This conversion is not true for .ASCII, .ASCIZ, ' (single quote) or " (double quote) statements if .ENABL LC is in effect.
2. The digits 0 through 9.
3. The characters . (period or dot) and \$ (dollar sign) which are reserved for use in system program symbols.
4. The following special characters:

<u>Character</u>	<u>Designation</u>	<u>Function</u>
carriage return		formatting character
line feed		
form feed		source statement terminators
vertical tab		
:	colon	label terminator
=	equal sign	direct assignment indicator
%	percent sign	register term indicator
tab		item or field terminator
space		item or field terminator
#	number sign	immediate expression indicator
@	at sign	deferred addressing indicator
(	left parenthesis	initial register indicator
)	right parenthesis	terminal register indicator

Character	Designation	Function
,	comma	operand field separator
;	semicolon	comment field indicator
<	left angle bracket	initial argument or expression indicator
>	right angle bracket	terminal argument or expression indicator
+	plus sign	arithmetic addition operator or auto increment indicator
-	minus sign	arithmetic subtraction operator or auto decrement indicator
*	asterisk	arithmetic multiplication operator
/	slash	arithmetic division operator
&	ampersand	logical AND operator
!	exclamation	logical inclusive OR operator
"	double quote	double ASCII character indicator
'	single quote	single ASCII character indicator
↑	up arrow	universal unary operator, argument indicator
\	backslash	macro numeric argument indicator (not available in ASEMBL)

#### 5.2.1.1 Separating and Delimiting Characters

Reference is made in the remainder of the chapter to legal separating characters and legal argument delimiters. These terms are defined below in Tables 5-1 and 5-2.

Table 5-1

Legal Separating Characters

Character	Definition	Usage
space	one or more spaces and/or tabs	A space is a legal separator only for argument operands. Spaces within expressions are ignored (see Section 5.2.8).
	comma	A comma is a legal separator for both expressions and argument operands.



Table 5-2

## Legal Delimiting Characters

Character	Definition	Usage
<...>	paired angle brackets	Paired angle brackets are used to enclose an argument, particularly when that argument contains separating characters. Paired angle brackets may be used anywhere in a program to enclose an expression for treatment as a term.
↑\...\	Up arrow construction where the up arrow character is followed by an argument bracketed by any paired printing characters.	This construction is equivalent in function to the paired angle brackets and is generally used only where the argument contains angle brackets.

Where argument delimiting characters are used, they must bracket the first (and, optionally, any following) argument(s). The character < and the characters ↑\, where \ is any printing character, can be considered unary operators which cannot be immediately preceded by another argument. For example:

```
.MACRO TEM <AB>C
```

indicates a macro definition with two arguments, while

```
.MACRO TEL C<AB>
```

has only one argument. The closing >, or matching character where the up arrow construction is used, acts as a separator. The opening argument delimiter does not act as an argument separator.

Angle brackets can be nested as follows:

```
<A<B>C>
```

which reduces to:

```
A<B>C
```

and which is considered to be one argument in both forms.

### 5.2.1.2 Illegal Characters

A character can be illegal in one of two ways:

1. A character which is not recognized as an element of the MACRO character set is always an illegal character and causes immediate termination of the current line at that point, plus

the output of an error flag in the assembly listing. For example:

```
LABEL←*A: MOV A,B
```

Since the backarrow is not a recognized character, the entire line is treated as a:

```
.WORD LABEL
```

statement and is flagged in the listing.

2. A legal MACRO character may be illegal in context. Such a character generates a Q error on the assembly listing.

### 5.2.1.3 Operator Characters

Legal unary operators under MACRO are as follows:

<u>Unary Operator</u>	<u>Explanation</u>		<u>Example</u>
+	plus sign	+A	(positive value of A, equivalent to A)
-	minus sign	-A	(negative, 2's complement, value of A)
↑	up arrow, universal unary operator (this usage is described in greater detail in Sections 5.5.4.2 and 5.5.6.2).	↑F3.0	(interprets 3.0 as a one word floating-point number).
		↑C24	(interprets the one's complement of the binary representation of 24(8))
		↑D127	(interprets 127 as a decimal number)
		↑O34	(interprets 34 as an octal number)
		↑B11000111	(interprets 11000111 as a binary value)

The unary operators as described above can be used adjacent to each other in a term. For example:

```
↑C↑O12
-↑O5
```

Legal binary operators under MACRO are as follows:

<u>Binary Operator</u>	<u>Explanation</u>	<u>Example</u>
+	addition	A+B
-	subtraction	A-B
*	multiplication	A*B (16-bit product returned)
/	division	A/B (16-bit quotient returned)
&	logical AND	A&B
!	logical inclusive OR	A!B

All binary operators have the same priority. Items can be grouped for evaluation within an expression by enclosure in angle brackets. Terms in angle brackets are evaluated first, and remaining operations are performed left to right. For example:

```
.WORD 1+2*3 ;IS 11 OCTAL
.WORD 1<2*3> ;IS 7 OCTAL
```

### 5.2.2 MACRO Symbols

There are three types of symbols: permanent, user-defined and macro. MACRO maintains three types of symbol tables: the Permanent Symbol Table (PST), the User Symbol Table (UST) and the Macro Symbol Table (MST). The PST contains all the permanent symbols and is part of the MACRO Assembler load module. The UST and MST are constructed as the source program is assembled; user-defined symbols are added to the table as they are encountered.

#### 5.2.2.1 Permanent Symbols

Permanent symbols consist of the instruction mnemonics (Appendix F, paragraph F.3) and assembler directives (sections 5.5 and 5.6, Appendix F, paragraph F.4). These symbols are a permanent part of the Assembler and need not be defined before being used in the source program.

#### 5.2.2.2 User-Defined and MACRO Symbols

User-defined symbols are those used as labels (Section 5.1.1.1) or defined by direct assignment (Section 5.2.3). These symbols are added to the User Symbol Table as they are encountered during the first pass of the assembly. Macro symbols are those symbols used as macro names (Section 5.6.1). These symbols are added to the Macro Symbol Table as they are encountered during the assembly.

User-defined and macro symbols can be composed of alphanumeric characters, dollar signs, and periods only; any other character is illegal.

The \$ and . characters are reserved for system software symbols (e.g., .READ, a system macro) and it is recommended that \$ and . not be inserted in user-defined or macro symbols.

The following rules apply to the creation of user-defined and macro symbols:

1. The first character must not be a number (except in the case of local symbols, see Section 5.2.5).
2. Each symbol must be unique within the first six characters.
3. A symbol can be written with more than six legal characters, but the seventh and subsequent characters are only checked for legality, and are not otherwise recognized by the Assembler.
4. Spaces, tabs, and illegal characters must not be embedded within a symbol.

The value of a symbol depends upon its use in the program. A symbol in the operator field may be any one of the three symbol types. To determine the value of the symbol, the Assembler searches the three symbol tables in the following order:

1. Macro Symbol Table
2. Permanent Symbol Table
3. User-Defined Symbol Table

A symbol found in the operand field is sought in the

1. User-Defined Symbol Table
2. Permanent Symbol Table

in that order. The Assembler never expects to find a macro name in an operand field.

These search orders allow redefinition of Permanent Symbol Table entries as user-defined or macro symbols. The same name can also be assigned to both a macro and a label.

User-defined symbols are either internal or external (global). All user-defined symbols are internal unless explicitly defined as being global with the `.GLOBL` directive (see Section 5.5.10).

Global symbols provide links between object modules. A global symbol which is defined as a label is generally called an entry point (to a section of code). Such symbols are referenced from other object modules to transfer control throughout the load module (which may be composed of a number of object modules).

Since `MACRO` provides program sectioning capabilities (Section 5.5.9), two types of internal symbols must be considered:

1. symbols that belong to the current program section; and
2. symbols that belong to other program sections.

In both cases, the symbol must be defined within the current assembly; the significance of the distinction is critical in evaluating expressions involving type (2) above (see Section 5.2.9).

### 5.2.3 Direct Assignment

A direct assignment statement associates a symbol with a value. When a direct assignment statement defines a symbol for the first time, that symbol is entered into the user symbol table and the specified value is associated with it. A symbol may be redefined by assigning a new value to a previously defined symbol. The latest assigned value replaces any previous value assigned to a symbol.

The general format for a direct assignment statement is:

symbol = expression

Symbols take on the relocatable or absolute attribute of their defining expression. However, if the defining expression is global, the symbol is not global unless explicitly defined as such in a .GLOBL directive (see Section 5.5.10).

For example:

```
A = 1                ;THE SYMBOL A IS EQUATED TO THE
                    ;VALUE 1.

B = 'A-1&MASKLOW    ;THE SYMBOL B IS EQUATED TO THE
                    ;VALUE OF THE EXPRESSION

C: D = 3            ;THE SYMBOL D IS EQUATED TO 3.

E: MOV #1,ABLE      ;LABELS C AND E ARE EQUATED TO THE
                    ;LOCATION OF THE MOV COMMAND
```

The following conventions apply to direct assignment statements:

1. An equal sign (=) must separate the symbol from the expression defining the symbol value.
2. A direct assignment statement is usually placed in the operator field and may be preceded by a label and followed by a comment.
3. Only one symbol can be defined by any one direct assignment statement.
4. Only one level of forward referencing is allowed.

Example of two levels of forward referencing (illegal):

```
X = Y
Y = Z
Z = 1
```

X and Y are both undefined throughout pass 1. X is undefined throughout pass 2 and causes a U error flag in the assembly listing.

#### 5.2.4 Register Symbols

The eight general registers of the PDP-11 are numbered 0 through 7 and can be expressed in the source program as:

```
%0
%1
.
.
%7
```

where the digit indicating the specific register can be replaced by any legal term which can be evaluated during the first assembly pass.

It is recommended that the programmer create and use symbolic names for all register references. A register symbol may be defined in a direct assignment statement, among the first statements in the program. A register symbol can not be defined after the statement which uses it. The defining expression of a register symbol must be absolute. For example:

```
R0=%0           ;REGISTER DEFINITION
R1=%1
R2=%2
R3=%3
R4=%4
R5=%5
SP=%6
PC=%7
```

The symbolic names assigned to the registers in the example above are the conventional names used in all PDP-11 system programs. Since these names are fairly mnemonic, it is suggested the user follow this convention. Registers 6 and 7 are given special names because of their special functions, while registers 0 through 5 are given similar names to denote their status as general purpose registers.

All register symbols must be defined before they are referenced. A forward reference to a register symbol causes phase errors in an assembly.

The % character can be used with any term or expression to specify a register. (A register expression less than 0 or greater than 7 is flagged with an R error code.) For example:

```
CLR %3+1
```

is equivalent to

```
CLR %4
```

and clears the contents of register 4, while

```
CLR 4
```

clears the contents of memory address 4.

In certain cases a register can be referenced without the use of a register symbol or register expression; these cases are recognized through the context of the statement. An example is shown below:

```
JSR 5,SUBR      ;FIRST OPERAND FIELD MUST ALWAYS BE A
                 ;REGISTER
```

### 5.2.5 Local Symbols

Local symbols are specially formatted symbols used as labels within a given range.

Local symbols provide a convenient means of generating labels for branch instructions, etc. Use of local symbols reduces the possibility of multiply-defined symbols within a user program and separates entry point symbols from local references. Local symbols, then, are not referenced from other object modules or even from outside their local symbol block.

Local symbols are of the form n\$, where n is a decimal integer from 1 to 127, inclusive, and can only be used on word boundaries. Local symbols include:

```
1$
27$
59$
104$
```

Within a local symbol block, local symbols can be defined and referenced. However, a local symbol cannot be referenced outside the block in which it is defined. There is no conflict with labels of the same name in other local symbol blocks.

Local symbols 64\$ through 127\$ can be generated automatically as a feature of the macro processor (see Section 5.6.3.5 for further details). When using local symbols the user is advised to first use the range from 1\$ to 63\$.

A local symbol block is delimited in one of the following ways:

1. The range of a single local symbol block can consist of those statements between two normally constructed symbolic labels. (Note that a statement of the form

```
LABEL=.
```

is a direct assignment, does not create a label in the strict sense, and does not delimit a local range.)

2. The range of a local symbol block is terminated upon encountering a .CSECT directive.
3. The range of a single local symbol block can be delimited with the .ENABL LSB and the first symbolic label or .CSECT directive following the .DSABL LSB directives. The default for LSB is off.

For examples of local symbols and local symbol blocks, see Figure 5-1.

The maximum offset of a local symbol from the base of its local symbol block is 128 decimal words. Symbols beyond this range are flagged with an A error code.

#### 5.2.6 Assembly Location Counter

The period (.) is the symbol for the assembly location counter. When used in the operand field of an instruction, it represents the address of the first word of the instruction. When used in the operand field of an assembler directive, it represents the address of the current byte or word. For example:

```
A:  MOV #.,R0      ;. REFERS TO LOCATION A,  
                        ;I.E., THE ADDRESS OF THE  
                        ;MOV INSTRUCTION.
```

(# is explained in Section 5.4.9).

At the beginning of each assembly pass, the Assembler clears the location counter. Normally, consecutive memory locations are assigned to each byte of object data generated. However, the location where the object data is stored may be changed by a direct assignment altering the location counter:

```
.=expression
```

Similar to other symbols, the location counter symbol has a mode associated with it, either absolute or relocatable. However, the mode cannot be external. The existing mode of the location counter cannot be changed by using a defining expression of a different mode.



<u>Line Number</u>	<u>Octal Expansion</u>	<u>Source Code</u>	<u>Comments</u>
1		.SBTTL	SECTOR INITIALIZATION
2			
3	000000'	.CSECT	IMPURE ;IMPURE STORAGE AREA
4	000000	IMPURE:	
5	000000'	.CSECT	IMPPAS ;CLEARED EACH PASS
6	000000	IMPPAS:	
7	000000'	.CSECT	IMPLIN ;CLEARED EACH LINE
8	000000	IMPLIN:	
9			
10	000000'	.CSECT	XCTPRG ;PROGRAM INITIALIZATION
11	00000	XCTPRG:	
12	00000 012700	MOV	#IMPURE,R0
	000000'		
13	00004 005020	1\$: CLR	(R0) + ;CLEAR IMPURE AREA
14	00006 022700	CMP	#IMPTOP,R0
	000040'		
15	00012 101374	BHI	1\$
16			
17	000000'	.CSECT	XCTPAS ;PASS INITIALIZATION
18	00000	XCTPAS:	
19	00000 012700	MOV	#IMPPAS,R0
	000000'		
20	00004 005020	1\$: CLR	(R0) + ;CLEAR IMPURE PART
21	00006 022700	CMP	#IMPTOP,R0
	000040'		
22	00012 101374	BHI	1\$
23			
24	000000'	.CSECT	XCTLIN ;LINE INITIALIZATION
25	00000	XCTLIN:	
26	00000 012700	MOV	#IMPLIN,R0
	000000'		
27	00004 005020	1\$: CLR	(R0) +
28	00006 022700	CMP	#IMPTOP,R0
	000040'		
29	00012 101374	BHI	1\$
30			
31	000000'	.CSECT	MIXED ;MIXED MODE SECTOR

Figure 5-1

Assembly Source Listing of MACRO Code Showing Local Symbol Blocks

The mode of the location counter symbol can be changed by the use of the .ASECT or .CSECT directive as explained in Section 5.5.9.

The expression defining the location counter must not contain forward references or symbols that vary from one pass to another.

Examples:

```
                .ASECT
                .=500                ;SET LOCATION COUNTER TO
                                      ;ABSOLUTE 500
FIRST:  MOV  .+10,COUNT              ;THE LABEL FIRST HAS THE VALUE
                                      ;500(8)
                                      ;.+10 EQUALS 510(8). THE
                                      ;CONTENTS OF THE LOCATION
                                      ;510(8) WILL BE DEPOSITED
                                      ;IN LOCATION COUNT.
                .=520                ;THE ASSEMBLY LOCATION COUNTER
                                      ;NOW HAS A VALUE OF
                                      ;ABSOLUTE 520(8).
SECOND: MOV  .,INDEX                ;THE LABEL SECOND HAS THE
                                      ;VALUE 520(8)
                                      ;THE CONTENTS OF LOCATION
                                      ;520(8), THAT IS, THE BINARY
                                      ;CODE FOR THE INSTRUCTION
                                      ;ITSELF, WILL BE DEPOSITED IN
                                      ;LOCATION INDEX.
                .CSECT
                .=.+20              ;SET LOCATION COUNTER TO
                                      ;RELOCATABLE 20 OF THE
                                      ;UNNAMED PROGRAM SECTION.
THIRD:  .WORD 0                    ;THE LABEL THIRD HAS THE
                                      ;VALUE OF RELOCATABLE 20.
```

Storage area may be reserved by advancing the location counter. For example, if the current value of the location counter is 1000, the direct assignment statement

```
                .=.+100
```

reserves 100(8) bytes of storage space in the program. The next instruction is stored at 1100.

### 5.2.7 Numbers

The MACRO Assembler assumes all numbers in the source program are to be interpreted in octal radix unless otherwise specified. The assumed radix can be altered with the .RADIX directive (see Section 5.5.4.1) or individual numbers can be treated as being of decimal, binary, or octal radix (see Section 5.5.4.2).

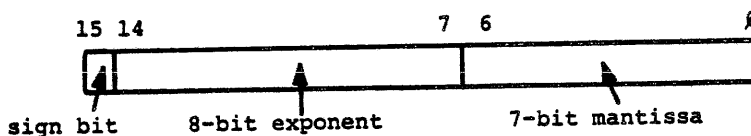
Octal numbers consist of the digits 0 through 7 only. A number not specified as a decimal number and containing an 8 or 9 is flagged with an N error code and treated as a decimal number.

Negative numbers are preceded by a minus sign (the Assembler translates them into two's complement form). Positive numbers may be preceded by a plus sign, although this is not required.

A number which is too large to fit into 16 bits (177777<n) is truncated from the left and flagged with a T error code in the assembly listing.

Numbers are always considered absolute quantities (that is, not relocatable).

The single-word floating-point numbers which can be generated with the  $\uparrow$ F operator (see Section 5.5.4.2) are stored in the following format:



Refer to PDP-11/45 Processor Handbook for details of the floating-point format.

### 5.2.8 Terms

A term is a component of an expression. A term may be one of the following:

1. A number, as defined in Section 5.2.7, whose 16-bit value is used.
2. A symbol, as defined earlier. Symbols are interpreted according to the following hierarchy:
  - a. a period causes the value of the current location counter to be used,
  - b. a permanent symbol whose basic value is used and whose arguments (if any) are ignored,
  - c. user defined symbols,
  - d. an undefined symbol is assigned a value of zero and inserted in the user-defined symbol table.
3. An ASCII conversion using either an apostrophe followed by a single ASCII character or a double quote followed by two ASCII characters which results in a word containing the 7-bit ASCII value of the character(s). (This construction is explained in greater detail in Section 5.5.3.3.)
4. A term may also be an expression or term enclosed in angle brackets. Any quantity enclosed in angle brackets is evaluated before the remainder of the expression in which it

is found. Angle brackets are used to alter the left to right evaluation of expressions (to differentiate between  $A*B+C$  and  $A*(B+C)$ ) or to apply a unary operator to an entire expression ( $-<A+B>$ , for example).

### 5.2.9 Expressions

Expressions are combinations of terms joined together by binary operators and which reduce to a 16-bit value. The operands of a .BYTE directive (see Section 5.5.3.1) are evaluated as word expressions before truncation to the low-order eight bits. Prior to truncation, the high-order byte must be zero or all ones (when byte value is negative, the sign bit is propagated). The evaluation of an expression includes the evaluation of the mode of the resultant expression; that is, absolute, relocatable or external. Expression modes are defined further below.

Expressions are evaluated left to right with no operator hierarchy rules except that unary operators take precedence over binary operators. A term preceded by a unary operator can be considered as containing that unary operator. (Terms are evaluated, where necessary, before their use in expressions.) Multiple unary operators are valid and are treated as follows:

--A

is equivalent to:

-<+<-A>>

A missing term, expression or external symbol is interpreted as a zero. A missing operator is interpreted as +. A Q error flag is generated for each missing term or operator. For example:

TAG ! LA 177777

is evaluated as

TAG ! LA+177777

with a Q error flag on the assembly listing line.

The value of an external expression is the value of the absolute part of the expression; e.g.,  $EXT+A$  has a value of A. This is modified by the Linker to become  $EXT+A$ .

Expressions, when evaluated, are either absolute, relocatable, or external. For the programmer writing position-independent code, the distinction is important.

1. An expression is absolute if its value is fixed. An expression whose terms are numbers and ASCII conversions will have an absolute value. A relocatable expression minus a relocatable term, where both items belong to the same program section, is also absolute.

2. An expression is relocatable if its value is fixed relative to a base address but will have an offset value added when linked. Expressions whose terms contain labels defined in relocatable sections and periods (in relocatable sections) will have a relocatable value.
3. An expression is external (or global) if its value is only partially defined during assembly and is completed at link time. An expression whose terms contain a global symbol not defined in the current program is an external expression. External expressions have relocatable values at execution time if the global symbol is defined as being relocatable or absolute if the global symbol is defined as absolute.

### 5.3 RELOCATION AND LINKING

The output of the MACRO Assembler is an object module which must be processed by Link before loading and execution. (refer to Chapter 6 for details.) The Linker essentially fixes (i.e., makes absolute) the values of external or relocatable symbols and turns the object module into a load module.

To enable the Linker to fix the value of an expression, the Assembler issues certain directives to the Linker together with required parameters. In the case of relocatable expressions, the Linker adds the base of the associated relocatable section (the location in memory of relocatable 0) to the value of the relocatable expression provided by the Assembler. In the case of an external expression, the value of the external term in the expression is determined by the Linker (since the external symbol must be defined in one of the other object modules which are being linked together) and adds it to the value of the external expression provided by the Assembler.

All instructions that are to be modified (as described in the previous paragraph) are marked with an apostrophe in the assembly listing (see also Appendix P, section P.2). Thus, the binary text output looks as follows:

```

005065   CLR   EXTERNAL(5)           ;VALUE OF EXTERNAL SYMBOL
000000'                                     ;ASSEMBLED ZERO; WILL BE
                                           ;MODIFIED BY THE LINKER.

005065   CLR   EXTERNAL+6(5)      ;THE ABSOLUTE PORTION OF THE
000006'                                     ;EXPRESSION (000006) IS ADDED
                                           ;BY THE LINKER TO THE VALUE
                                           ;OF THE EXTERNAL SYMBOL

005065   CLR   RELOCATABLE(5)    ;ASSUMING WE ARE IN A
000040'                                     ;RELOCATABLE
                                           ;SECTION AND THE VALUE OF
                                           ;RELOCATABLE IS RELOCATABLE 40

```

## 5.4 ADDRESSING MODES

The program counter (PC, register 7 of the eight general registers) always contains the address of the next word to be fetched; i.e., the address of the next instruction to be executed, or the second or third word of the current instruction.

In order to understand how the address modes operate and how they assemble, the action of the program counter must be understood. The key rule is:

Whenever the processor implicitly uses the program counter to fetch a word from memory, the program counter is automatically incremented by two after the fetch.

That is, when an instruction is fetched, the PC is incremented by two, so that it is pointing to the next word in memory; and, if an instruction uses indexing (Sections 5.4.7, 5.4.9 and 5.4.11) the processor uses the program counter to fetch the base from memory. Hence, using the rule above, the PC increments by two, and now points to the next word.

The following conventions are used in this section:

1. Let E be any expression as defined in section 5.2.
2. Let R be a register expression. This is any expression containing a term preceded by a % character or a symbol previously equated to such a term.

Examples:

```
R0 = %0      ;GENERAL REGISTER 0
R1 = R0+1    ;GENERAL REGISTER 1
R2 = 1+%1    ;GENERAL REGISTER 2
```

3. Let ER be a register expression or an expression in the range 0 to 7 inclusive.
4. Let A be a general address specification which produces a 6-bit mode address field as described in Sections 3.1 and 3.2 of the PDP-11 Processor Handbook (both 11/20 and 11/45 versions).

The addressing specifications, A, can be explained in terms of E, R, and ER as defined above. Each is illustrated with the single operand instruction CLR or double operand instruction MOV.

### 5.4.1 Register Mode

The register contains the operand.

Format for A: R

```
Examples:   R0=%0      ;DEFINE R0 AS REGISTER 0
            CLR R0     ;CLEAR REGISTER 0
```

#### 5.4.2 REGISTER DEFERRED MODE

THE REGISTER CONTAINS THE ADDRESS OF THE OPERAND.

FORMAT FOR A: @R OR (ER)

EXAMPLES:            CLR @R1                    ;BOTH INSTRUCTIONS CLEAR  
                     CLR (1)                    ;THE WORD AT THE ADDRESS  
   ;CONTAINED IN REGISTER 1

#### 5.4.3 Autoincrement Mode

The contents of the register are incremented immediately after being used as the address of the operand. (See note below.)

Format for A: (ER)+

Examples:            CLR (R0)+                    ;EACH INSTRUCTION CLEARS  
                     CLR (R0+3)+                ;THE WORD AT THE ADDRESS  
                     CLR (2)+                    ;CONTAINED IN THE SPECIFIED  
   ;REGISTER AND INCREMENTS  
   ;THAT REGISTER'S CONTENTS  
   ;BY TWO

#### NOTE

Both JMP and JSR instructions using non-deferred autoincrement mode, autoincrement the register before its use on the PDP-11/20 and 11/05 (but not on the PDP-11/40 or 11/45). In double operand instructions of the addressing form %R,(R)+ or %R,-(R) where the source and destination registers are the same, the source operand is evaluated as the autoincremented or autodecremented value; but the destination register, at the time it is used, still contains the originally intended effective address. In the following two examples, as executed on the PDP-11/20, R0 originally contains 100.

MOV R0,(0)+            ;THE QUANTITY 102 IS MOVED  
   ;TO LOCATION 100

MOV R0,-(0)            ;THE QUANTITY 76 IS MOVED  
   ;TO LOCATION 76

The use of these forms should be avoided as they are not compatible with the PDP-11/05, 11/40 and 11/45.

A Z error code is printed with each instruction which is not compatible among all members of the PDP-11 family. This is merely a warning code.

#### 5.4.4 Autoincrement Deferred Mode

The register contains the pointer to the address of the operand. The contents of the register are incremented after being used.

Format for A: @ (ER) +

Example: CLR @(3) + ;CONTENTS OF REGISTER 3 POINT  
;TO ADDRESS OF WORD TO BE  
;CLEARED BEFORE BEING  
;INCREMENTED BY TWO

#### 5.4.5 Autodecrement Mode

The contents of the register are decremented before being used as the address of the operand (see note under autoincrement mode).

Format for A: -(ER)

Examples: CLR -(R0) ;DECREMENT CONTENTS OF  
CLR -(R0+3) ;REGISTERS 0, 3 AND 2 BY TWO  
CLR -(2) ;BEFORE USING AS ADDRESSES OF  
;WORDS TO BE CLEARED.

#### 5.4.6 Autodecrement Deferred Mode

The contents of the register are decremented before being used as the pointer to the address of the operand.

Format for A: @-(ER)

Example: CLR @-(2) ;DECREMENT CONTENTS OF  
;REGISTER 2 BY TWO BEFORE  
;USING AS POINTER  
;TO ADDRESS OF WORD TO BE  
;CLEARED.

#### 5.4.7 Index Mode

The value of an expression E is stored as the second or third word of the instruction. The effective address is calculated as the value of E plus the contents of register ER. The value E is called the base.

Format for A: E(ER)

Examples: CLR X+2(R1) ;EFFECTIVE ADDRESS IS X+2 PLUS  
;THE CONTENTS OF REGISTER 1.  
CLR -2(3) ;EFFECTIVE ADDRESS IS -2 PLUS  
;THE CONTENTS OF REGISTER 3.



#### 5.4.8 Index Deferred Mode

An expression plus the contents of a register gives the pointer to the address of the operand.

Format for A: @E(ER)

Example: CLR @14(4) ;IF REGISTER 4 HOLDS 100 AND  
;LOCATION 114 HOLDS 2000,  
;LOCATION 2000 IS CLEARED.

#### 5.4.9 Immediate Mode

The immediate mode allows the operand itself to be stored as the second or third word of the instruction. It is assembled as an autoincrement of register 7, the PC.

Format for A: #E

Examples: MOV #100,R0 ;MOVE AN OCTAL 100 TO REGISTER  
;0  
MOV #X, R0 ;MOVE THE VALUE OF SYMBOL X TO  
;REGISTER 0

The operation of this mode is explained as follows:

The statement MOV #100,R3 assembles as two words. These are:

0 1 2 7 0 3  
0 0 0 1 0 0

Just before this instruction is fetched and executed, the PC points to the first word of the instruction. The processor fetches the first word and increments the PC by two. The source operand mode is 27 (autoincrement the PC). Thus, the PC is used as a pointer to fetch the operand (the second word of the instruction) before being incremented by two, to point to the next instruction.

#### 5.4.10 Absolute Mode

Absolute mode is the equivalent of immediate mode deferred. @#E specifies an absolute address which is stored in the second or third word of the instruction. Absolute mode is assembled as an autoincrement deferred of register 7, the PC.

Format for A: @#E

Examples: MOV @#100,R0 ;MOVE THE VALUE OF THE  
;CONTENTS OF LOCATION 100 TO  
;REGISTER 0.  
CLR @#X ;CLEAR THE CONTENTS OF THE  
;LOCATION WHOSE ADDRESS IS X.

#### 5.4.11 Relative Mode

Relative mode is the normal mode for memory references.

Format for A: E

```
Examples:      CLR 100           ;CLEAR LOCATION 100.
               MOV X,Y         ;MOVE CONTENTS OF LOCATION X
                                   ;TO LOCATION Y.
```

Relative mode is assembled as index mode, using register 7, the PC, as the index register. The base of the address calculation, which is stored in the second or third word of the instruction, is not the address of the operand (as in index mode), but the number which, when added to the PC, becomes the address of the operand. Thus, the base is X-PC, which is called an offset. The operation is explained as follows:

If the statement MOV 100,R3 is assembled at absolute location 20, the assembled code is:

```
Location 20:      0 1 6 7 0 3
Location 22:      0 0 0 0 5 4
```

The processor fetches the MOV instruction and adds two to the PC so that it points to location 22. The source operand mode is 67; that is, indexed by the PC. To pick up the base, the processor fetches the word pointed to by the PC and adds two to the PC. The PC now points to location 24. To calculate the address of the source operand, the base is added to the designated register. That is,  $BASE+PC=54+24=100$ , the operand address.

Since the Assembler considers "." as the address of the first word of the instruction, an equivalent index mode statement would be:

```
MOV 100--4(PC),R3
```

This mode is called relative because the operand address is calculated relative to the current PC. The base is the distance or offset (in bytes) between the operand and the current PC. If the operator and its operand are moved in memory so that the distance between the operator and data remains constant, the instruction will operate correctly anywhere in core.

#### 5.4.12 Relative Deferred Mode

Relative deferred mode is similar to relative mode, except that the expression, E, is used as the pointer to the address of the operand.

Format for A: @E

```
Example:      MOV @X,R0 ;MOVE THE CONTENTS OF THE
                                   ;LOCATION WHOSE ADDRESS IS IN
                                   ;X INTO REGISTER 0.
```

#### 5.4.13 Table of Mode Forms and Codes

Each instruction takes at least one word. Operands of the first six forms listed below, do not increase the length of an instruction. Each operand in one of the other modes, however, increases the instruction length by one word.

<u>Form</u>	<u>Mode</u>	<u>Meaning</u>
R	0n	Register mode
@R or (ER)	1n	Register deferred mode
(ER)+	2n	Autoincrement mode
@(ER)+	3n	Autoincrement deferred mode
-(ER)	4n	Autodecrement mode
@-(ER)	5n	Autodecrement deferred mode

where n is the register number.

Any of the following forms adds one word to the instruction length:

<u>Form</u>	<u>Mode</u>	<u>Meaning</u>
E (ER)	6n	Index mode
@E(ER)	7n	Index deferred mode
#E	27	Immediate mode
@#E	37	Absolute memory reference mode
E	67	Relative mode
@E	77	Relative deferred reference mode

where n is the register number. Note that in the last four forms, register 7 (the PC) is referenced.

#### NOTE

An alternate form for @R is (ER). However, the form @(ER) is equivalent to @0(ER).

The form @#E differs from the form E in that the second or third word of the instruction contains the absolute address of the operand rather than the relative distance between the operand and the PC. Thus, the instruction CLR @#100 clears absolute location 100 even if the instruction is moved from the point at which it was assembled. See the description of the .ENABLE AMA function in Section 5.5.2, which directs the assembly of all relative mode addresses as absolute mode addresses.

#### 5.4.14 Branch Instruction Addressing

The branch instructions are one word instructions. The high byte contains the op code and the low byte contains an 8-bit signed offset (7 bits plus sign) which specifies the branch address relative to the PC. The hardware calculates the branch address as follows:

1. Extend the sign of the offset through bits 8-15.

2. Multiply the result by 2. This creates a word offset rather than a byte offset.
3. Add the result to the PC to form the final branch address.

The Assembler performs the reverse operation to form the byte offset from the specified address. Remember that when the offset is added to the PC, the PC is pointing to the word following the branch instruction; hence the term -2 in the calculation.

Byte offset = (E-PC)/2 truncated to eight bits.

Since PC = .+2, we have

Byte offset = (E-.+2)/2 truncated to eight bits.

#### NOTE

It is illegal to branch to a location specified as an external symbol, or to a relocatable symbol from within an absolute section, or to an absolute symbol or a relocatable symbol or another program section from within a relocatable section.

#### 5.4.15 EMT and Trap Addressing

The EMT and TRAP instruction do not use the low-order byte of the word. This allows information to be transferred to the trap handlers in the low-order byte. If EMT or TRAP is followed by an expression, the value is put into the low-order byte of the word. However, if the expression is too big (>377(8)) it is truncated to eight bits and a T error flag is generated.

#### 5.5 ASSEMBLER DIRECTIVES

Directives are statements which cause the Assembler to perform certain processing operations.

Assembler directives can be preceded by a label, subject to restrictions associated with specific directives, and followed by a comment. An assembler directive occupies the operator field of a MACRO source line. Only one directive can be placed on any one line. Zero, one, or more operands can occupy the operand field; legal operands differ with each directive and may be either symbols, expressions, or arguments.

## 5.5.1 Listing Control Directives

### 5.5.1.1 .LIST and .NLIST

Listing options can be specified in the text of a MACRO program through the .LIST and .NLIST directives. These are of the form:

```
.LIST arg
.NLIST arg
```

where: arg represents one or more optional arguments.

When used without arguments, the listing directives alter the listing level count. The listing level count causes the listing to be suppressed when it is negative. The count is initialized to zero, incremented for each .LIST and decremented for each .NLIST. For example:

```
.MACRO LTEST ;LIST TEST
;A-THIS LINE SHOULD LIST
.NLIST
;B-THIS LINE SHOULD NOT LIST
.NLIST
;C-THIS LINE SHOULD NOT LIST
.LIST
;D-THIS LINE SHOULD NOT LIST (LEVEL NOT BACK TO ZERO)
.LIST
;E-THIS LINE SHOULD LIST (LEVEL BACK TO ZERO)
.ENDM

LTEST ;CALL THE MACRO

;A-THIS LINE SHOULD LIST
;E-THIS LINE SHOULD LIST (LEVEL BACK TO ZERO)
```

The primary purpose of the level count is to allow macro expansions to be selectively listed and yet exit with the level returned to the status current during the macro call.

The use of arguments with the listing directives does not affect the level count; however, .LIST and .NLIST can be used to override the current listing control. For example:

```
.MACRO XX
.
.
.LIST ;LIST NEXT LINE
X=.
.NLIST ;DO NOT LIST REMAINDER
;OF MACRO EXPANSION
.
.
.ENDM
.NLIST ME ;DO NOT LIST MACRO EXPANSIONS
XX
X=.
```

Allowable arguments for use with the listing directives are as follows (these arguments can be used singly or in combination):

<u>Argument</u>	<u>Default</u>	<u>Function</u>
SEQ	list	Controls the listing of source line sequence numbers.
LOC	list	Controls the listing of the location counter (this field would not normally be suppressed).
BIN	list	Controls the listing of generated binary code.
BEX	list	Controls listing of binary extensions; that is, those locations and binary contents beyond the first binary word (per source statement). This is a subset of the BIN argument.
SRC	list	Controls the listing of the source code.
COM	list	Controls the listing of comments. This is a subset of the SRC argument and can be used to reduce listing time and/or space where comments are unnecessary.
MD	list	Controls listing of macro definitions and repeat range expansions (has no effect in ASEMBL).
MC	list	Controls listing of macro calls and repeat range expansions (has no effect in ASEMBL).
ME	no list	Controls listing of macro expansions (has no effect in ASEMBL).
MEB	no list	Controls listing of macro expansion binary code. A .LIST MEB causes only those macro expansion statements producing binary code to be listed. This is a subset of the ME argument (has no effect in ASEMBL).
CND	list	Controls the listing of unsatisfied conditions and all .IF and .ENDC statements. This argument permits conditional assemblies to be listed without including unsatisfied code.
LD	no list	Controls listing of all listing directives having no arguments (those used to alter the listing level count).

<u>Argument</u>	<u>Default</u>	<u>Function</u>
TOC	list	Control listing of table of contents on pass 1 of the assembly (see Section 5.5.1.4 describing the .SBTTL directive). The full assembly listing is printed during pass 2 of the assembly.
TTM	Teletype mode	Control listing output format (has no effect in ASEMBL). The TTM argument (the default case) causes output lines to be truncated to 72 characters. Binary code is printed with the binary extensions below the first binary word. The alternative (.NLIST TTM) to Teletype mode is line printer mode, which is shown in Figure 5-2.
SYM	list	Controls the listing of the symbol table for the assembly.

An example of an assembly listing as sent to a 132 column line printer is shown in Figure 5-2. Notice that binary extensions for statements generating more than one word are spread horizontally on the source line. An example of an assembly listing as sent to a terminal is shown in Figure 5-3. Notice that binary extensions for statements generating more than one word are printed on subsequent lines.

Figure 5-4 illustrates a symbol table listing. With the exception of local symbols and macro names, all user-defined symbols are listed in the symbol table. The characters following the symbols listed have special meanings as follows:

=	the symbol is assigned in a direct assignment statement,
%	the symbol is a register symbol,
R	the symbol is relocatable,
G	the symbol is global.

The final value of the symbol is expressed in octal. If the symbol is undefined six asterisks are printed in place of the octal number.

CSECT numbers are listed if the symbol is in a named CSECT. All CSECTs are listed at the end of the table with their lengths and corresponding number.

1	001766				GETLIN:		;GET AN INPUT LINE
2	001766					SAVREG	
3	001772	016700	000020'		1\$:	MOV	FFCNT,R0 ;ANY RESERVED FF'S?
4	001776	001420				BEQ	31\$ ; NO
5	002000	060067	000022'			ADD	R0,PAGNUM ;YES, UPDATE PAGE NUMBER
6	002004	012767	177777	000026'		MOV	#-1,PAGEXT
7	002012	005067	000012'			CLR	LINNUM ;INIT NEW CREF SEQUENCE
8	002016	005067	000020'			CLR	FFCNT
9	002022	005067	000016'			CLR	SEQEND
10	002026	005767	000000'			TST	PASS
11	002032	001402				BEQ	31\$
12	002034	005067	000010'			CLR	LPPCNT
13	002040	012702	001712'		31\$:	MOV	#LINBUF,R2
14	002044	010267	000012'			MOV	R2,LCBEG
15	002050	012767	002116'	000014'		MOV	#LINEND,LCENDL ;SET UP BEGINNING
16						.IF	NDF XSML ; AND END OF LINE MARKERS
17	002056	005767	000200'			TST	SMLCNT ;IN SYSTEM MACRO?
18	002062	001145				BNE	40\$ ; YES, SPECIAL
19						.ENDC	
20						.IF	NDF XMACRO
21	002064	016701	002214'			MOV	MSBMRP,R1 ;ASSUME MACRO IN PROGRESS
22	002070	001166				.IFTF	
23						BNE	10\$ ;BRANCH IF SO
24	002072	012701	000756'			MOV	#SRCBUF,R1
25	002076					.WAIT	#SRCLNK
26	002104	005267	000012'			INC	LINNUM
27	002110	116700	000753'			MOVB	SRCHDR+3,R0 ;GET CODE BYTE
28	002114	032700	000047			BIT	#047,R0 ;ANYTHING BAD?
29	002120	001403				BEQ	32\$ ; NO
30	002122					ERROR	L ;YES, ERROR
31	002130	106100			32\$:	ROLB	R0 ;EOF?
32	002132	100014				BPL	2\$ ; NO
33	002134	056767	000006'	000004'		BIS	CSISAV,ENDFLG
34	002142	001003				BNE	34\$

5-30

Figure 5-2

Example of MACRO Line Printer Listing  
(132 column Line Printer)



```

.MAIN.      RT-11 MACRO VM01-01      22-JUL-73 PAGE 28

1 001766          GETLIN:                      ;GET AN INPUT LINE
2 001766          SAVREG
3 001772  016700 1$:      MOV      FFCNT,R0      ;ANY RESERVED FF'S?
      000020'
4 001776  001420          BEQ      31$          ; NO
5 002000  060067          ADD      R0,PAGENUM    ;YES, UPDATE PAGE NUMBER
      000022'
6 002004  012767          MOV      #-1,PAGEXT
      177777
      000026'
7 002012  005067          CLR      LINNUM      ;INIT NEW CREF SEQUENCE
      000012'
8 002016  005067          CLR      FFCNT
      000020'
9 002022  005067          CLR      SEQEND
      000016'
10 02026  005067          TST      PASS
      000000'
11 02032  001402          BEQ      31$
12 02034  005067          CLR      LPPCNT
      000010'
13 02040  012702 31$:    MOV      #LINBUF,R2
      001712'
14 02044  010267          MOV      R2,LDBEGL    ;SET UP BEGINNING
      000012'
15 02050  012767          MOV      #LINEND,LCENDL ; AND END OF LINE MARKERS
      002116'
      000014'

16          .IF NDF XSML
17 02056  005767          TST      SMLCNT      ;IN SYSTEM MACRO?
      000200'
18 02062  001145          BNE      40$          ;YES, SPECIAL
19          .ENDC
20          .IF NDF XMACRO
21 02064  016701          MOV      MSBMRP,R1    ;ASSUME MACRO IN PROGRESS
      002214'
22 02070  001166          BNE      10$          ;BRANCH IF SO
23          .IFTF
24 02072  012701          MOV      #SRCBUF,R1
      000756'

25 02076          .WAIT  #SRCLNK
26 02104  005267          INC      LINNUM
      000012'
27 02110  116700          MOVB    SRCHDR+3,R0    ;GET CODE BYTE
      000753'
28 02114  032700          BIT      #047,R0      ;ANYTHING BAD?
      000047
29 02120  001403          BEQ      32$          ; NO
30 02122          ERROR  L          ;YES, ERROR
31 02130  106100 32$:    ROLB    R0          ;EOF?
32 02132  100014          BPL      2$          ; NO
33 02134  056767          BIS      CSISAV,ENDFLG
      000006'
      000004'
34 02142  001003          BNE      34$

```

Figure 5-3 Example of Page Heading from MACRO Terminal Listing (same format as for 80-column line printer)

ABSEXP	***** G	ASSEM	***** G	RINCHN	000004	
BINDAT	002122R	004 BLKTRL	002110R	004 BPMB	000020	
BUFTBL	000360RG	003 CHAN	002162R	004 CHRPNY	***** G	
CMILEN	000123	CNTTAL	000344RG	003 CONT	000030RG	010
CORERR	001140R	010 CPL	000120	CR	000015	
CRFBUF	002066RG	004 CRFC	000020	CRFCMN	000012	
CRFDAT	002152R	004 CRFE	000040	CRFFLG	000002R	007
CRFLEN	000014	CRFM	000004	CHFP	000010	
CRFPNT	000060R	003 CRFS	000002	CRFTAB	000026R	003
CRFTST	000004RG	007 CTLYBL	000000R	003 DATE	000030R	010
DATTIM	001004RG	004 DEFEXT	000230R	003 DETAB	000164R	003
DNC	***** G	EOMASK	***** G	EDMCSI	***** G	
EMTERR	000052	ENDP1	***** G	ENDP2	***** G	
ENOSMT	000334R	010 ERR	001102R	010 ENRB	000072R	010
ERRCNT	***** G	FB	000010	FIN	001036RG	010
FINMSG	001017R	004 FINMS1	001041R	004 FINMS2	001057R	004
FINP1	000626R	010 FINP2	000626R	010 FINSML	001336RG	010
FRECOR	000006R	007 GETPLI	001170RG	010 GETH50	***** G	
GSARG	***** G	MORTTL	001071RG	004 IMPURT	000042R	007
IMPURS	000000R	007 INIOF	000076R	010 INIP1	000542R	010
INIP2	000572R	010 INISML	001272RG	010 INITI	000604R	010
IOFTBL	000012RG	007 IOLTBL	000330R	003 IO.EOF	000004 G	
IO.ERR	000010 G	IO.NNU	000001	IO.TTY	000002 G	
LCMASK	***** G	LCMCSI	***** G	LF	000012	
LINBUF	***** G	LINLEN	000200	LNTAB	000062R	003
LPP	000074	LSTBUF	001324R	LSTCHN	000002	
LSTDAT	002112R	004 LSTLEN	000265	MACPB	***** G	
MACP1	***** G	MACP2	***** G	MAXCMN	000010	
MCEXEC	000000	MONLOW	000054	MONTEL	000240R	003
MOVBTT	***** G	OBJBUF	001016R	004 OBJLEN	000052	
OCTLEN	000060	PASS	***** G	PC	000007	
PIN	001744R	010 PID	001752R	010 POUT	001734R	010
PRGLIM	000224R	003 PROSW	***** G	PTRTBL	002106R	004
PUTKB	***** G	PUTKBL	***** G	PUTLP	***** G	
RECNUM	000026R	007 RELCHN	000006	RELDAT	002132R	004
RLDBUF	001674R	004 RLDOLEN	000052	RT11	000000	
R0	000000	R1	000001	R2	000002	
R3	000003	R4	000004	R5	000005	
SAVREG	***** G	SERROR	001112R	010 SETON	000744R	010
SHDBLK	000320R	003 SHBLK	001000R	004 SHLBUF	001746R	004
SHLCHN	000010	SHLDAT	002142R	004 SHLLEN	000120	
SMLSW	000010R	007 SP	000000	SPACE	000040	
SRCBUF	001120R	004 SRCCHN	000000	SRCOAT	002102R	004
SRCLEN	000204	STANT	000002RG	010 STKFDG	000074	
STKLM	000002R	004 STKSAV	000000R	004 STLLEN	000100	
SMLOOK	000514R	010 SWNEXT	000270R	010 SWRC	000476R	010
SWRD	000430R	010 SWRE	000434R	010 SWRL	000366R	010
SWRN	000362R	010 SWTAS	000000R	005 SWTBL	000212R	003
SWTDOO	000330R	010 SWTEND	000220R	003 SWTERH	000340R	010
SWTFLG	000000R	007 SYSUIC	000401	SYTTP	000000RG	004
YAB	000011	THPCNT	000014	TSTSTK	001116RG	010
TTLEN	000040	USRLOC	000046	VT	000013	
WINST	004240	XBAH	000000	XEDARS	000000	
XEDCOR	000000	XEDPIC	000000	XMIT0	***** G	
XCLOUT	002050RG	010 XREAD	001552RG	010 XREADN	001552RG	010
XWAIT	002046RG	010 XWRITE	001346RG	010 XWRITE	001346RG	010

, 485,	000000	000
	000000	001
DPURE	000000	002
DPURES	000374	003
MIXED3	002176	004
SNTSE3	000000	005
SNTSEC	000000	006
IMPURS	000042	007
MAINS	002134	010
ERRORS DETECTED: 0		
FREE CORE: 12064, WORDS		

Figure 5-4 Symbol Table

### 5.5.1.2 Page Headings

The MACRO Assembler outputs each page in the format shown in Figure 5-3 (Terminal listing). On the first line of each listing page the Assembler prints (from right to left):

1. title taken from .TITLE directive
2. assembler version identification
3. the date (not in 8K version)
4. page number

The second line of each listing page contains the subtitle text specified in the last encountered .SBTTL directive.

### 5.5.1.3 .TITLE

The .TITLE directive is used to assign a name to the object module. The name is the first symbol following the directive and must be six Radix-50 characters or less (any characters beyond the first six are ignored. Non Radix-50 characters are not acceptable. For example:

```
.TITLE PROG TO PERFORM DAILY ACCOUNTING
```

causes the object module of the assembled program to be named PROG (this name is distinguished from the filename of the object module specified in the command string to the Assembler).

If there is no TITLE statement, the default name assigned to the first object module is

```
.MAIN.
```

The first tab or space following the .TITLE directive is not considered part of the object module name or header text, although subsequent tabs and spaces are significant.

If there is more than one .TITLE directive, the last .TITLE directive in the program conveys the name of the object module.

### 5.5.1.4 .SBTTL

The .SBTTL directive is used to provide the elements for a printed table of contents of the assembly listing. The text following the directive is printed as the second line of each of the following assembly listing pages until the next occurrence of a .SBTTL directive. For example:

```
.SBTTL CONDITIONAL ASSEMBLIES
```

The text

#### CONDITIONAL ASSEMBLIES

is printed as the second line of each of the following assembly listing pages.

During pass 1 of the assembly process, MACRO automatically prints a table of contents for the listing containing the line sequence number and text of each .SBTTL directive in the program. Such a table of contents is inhibited by specifying the .NLIST TOC directive within the source.

An example of the table of contents is shown in Figure 5-5. Note that the first word of the subtitle heading is not limited to six characters since it is not a module name.

.MAIN. RT-11 MACRO VM02-03 29-AUG-73  
TABLE OF CONTENTS

1-	15	RT-11 MACRO PARAMETER FILE
1-	25	COMMON PARAMETER FILE
2-	1	ASSEMBLY OPTIONS
3-	1	VARIABLE PARAMETERS
4-	1	GLOBALS
5-	1	SECTOR INITIALIZATION
7-	1	SUBROUTINE CALL DEFINITIONS
10-	1	MISCELLANEOUS MACRO DEFINITIONS
11-	2	MCIOCH - I/O CHANNEL ASSIGNMENTS
12-	2	****EXEC****
13-	1	PROGRAM START
14-	1	INIT OUTPUT FILES
15-	1	SWITCH HANDLERS
16-	1	END-OF-PASS ROUTINES
17-	1	SWITCH AND DATE DATA AREAS
18-	1	INIT OUTPUT FILES (CONTINUED)
19-	1	FINISH ASSEMBLY AND RESTART
20-	1	MEMORY MANAGEMENT
21-	1	GET PHYSICAL SOURCE LINE
22-	1	SYSTEM MACRO HANDLERS
23-	1	WRITE ROUTINES
24-	1	READ ROUTINE
25-	1	COMMON I/O ROUTINES
26-	1	MESSAGES
27-	1	I/O TABLES
29-	1	FINIS

Figure 5-5 Assembly Listing Table of Contents

Table of Contents text is taken from the text of each .SBTTL directive. The associated numbers are the page and line numbers of the .SBTTL directives.

#### 5.5.1.5 .IDENT

The .IDENT directive is a NOP.

#### 5.5.1.6 Page Ejection

There are several means of obtaining a page eject in a MACRO assembly listing:

1. After a line count of 58 lines, MACRO automatically performs a page eject to skip over page perforations on line printer paper and to formulate terminal output into pages.
2. A form feed character used as a line terminator (or as the only character on a line) causes a page eject. Used within a macro definition a form feed character causes a page eject. A page eject is not performed when the macro is invoked.
3. More commonly, the .PAGE directive is used within the source code to perform a page eject at that point. The format of this directive is

.PAGE

This directive takes no arguments and causes a skip to the top of the next page.

Used within a macro definition, the .PAGE is ignored, but the page eject is performed at each invocation of that macro.

#### 5.5.2 Functions: .ENABL and .DSABL Directives

Several functions are provided by MACRO through the .ENABL and .DSABL directives. These directives use three-character symbolic arguments to designate the desired function; and are of the forms:

.ENABL arg  
.DSABL arg

where: arg is one of the legal symbolic arguments defined below.

The following table describes the symbolic arguments and their associated functions in the MACRO language:

<u>Symbolic</u>	<u>Function</u>
AMA	Enabling of this function directs the assembly of all relative addresses (address mode 67) as absolute addresses (address mode 37). This switch is useful during the debugging phase of program development.
FPT	Enabling of this function (has no effect in ASEMBL) causes floating point truncation, rather than rounding, as is otherwise performed. .DSABL FPT returns to floating point rounding mode.
LC	Enabling of this function causes the Assembler to accept lower case ASCII input instead of converting it to upper case (has no effect in ASEMBL).
LSB	Enable or disable a local symbol block (has no effect in ASEMBL). While a local symbol block is normally entered by encountering a new symbolic label or .CSECT directive, .ENABL LSB forces a local symbol block which is not terminated until a label or .CSECT directive following the .DSABL LSB statement is encountered. The default case is .DSABL LSB.
PNC	The statement .DSABL PNC (has no effect in ASEMBL) inhibits binary output until an .ENABL PNC is encountered. The default case is .ENABL PNC.

An incorrect argument causes the directive containing it to be flagged as an error.

### 5.5.3 Data Storage Directives

A wide range of data and data types can be generated with the following directives and assembly characters:

```
.BYTE
.WORD
,
"
.ASCII
.ASCIIZ
.RAD50
↑B
↑D
↑O
```

These facilities are explained in the following sections.

### 5.5.3.1 .BYTE

The .BYTE directive is used to generate successive bytes of data. The directive is of the form:

```
.BYTE exp                ;WHICH STORES THE OCTAL
                        ;EQUIVALENT OF THE EXPRESSION
                        ;exp IN THE NEXT BYTE.

.BYTE exp1,exp2,...     ;WHICH STORES THE OCTAL
                        ;EQUIVALENTS OF THE LIST OF
                        ;EXPRESSION IN SUCCESSIVE BYTES.
```

where a legal expression must have an absolute value (or contain a reference to an external symbol) and must result in 8 bits or less of data. The 16-bit value of the expression must have a high-order byte (which is truncated) that is either all zeros or all ones. Each operand expression is stored in a byte of the object program. Multiple operands are separated by commas and stored in successive bytes. For example:

```
SAM=5
.=410
.BYTE ↑D48,SAM          ;060 (OCTAL EQUIVALENT OF 48
                        ;DECIMAL) IS STORED IN LOCATION
                        ;410, 005, IS STORED IN
                        ;LOCATION 411.
```

If the high-order byte of the expression equates to a value other than 0 or -1, it is truncated to the low-order 8 bits and flagged with a T error code. If the expression is relocatable, an A-type warning flag is given.

At link time it is likely that relocation will result in an expression of more than 8 bits, in which case, the Linker prints an error code. For example:

```
.BYTE 23                ;STORES OCTAL 23 IN NEXT BYTE.
A:
.BYTE A                  ;RELOCATABLE VALUE CAUSES AN "A"
                        ;ERROR FLAG.

.GLOBL X
X=3
.BYTE X                  ;STORES 3 IN NEXT BYTE.
```

In the case where X is defined in another program:

```
.GLOBL X
.BYTE X
```

If an operand following the .BYTE directive is null, it is interpreted as a zero. For example:

```
.=420
.BYTE ,, ;ZEROS ARE STORED IN BYTES 420, 421, AND 422.
```



### 5.5.3.2 .WORD

The .WORD directive is used to generate successive words of data. The directive is of the form:

```
.WORD exp                ;WHICH STORES THE OCTAL
                        ;EQUIVALENT OF THE EXPRESSION
                        ;exp IN THE NEXT WORD.

.WORD exp1,exp2,...     ;WHICH STORES THE OCTAL
                        ;EQUIVALENTS OF THE LIST OF
                        ;EXPRESSIONS IN SUCCESSIVE
                        ;WORDS.
```

where a legal expression must result in 16 bits or less of data. Each operand expression is stored in a word of the object program. Multiple operands are separated by commas and stored in successive words. For example:

```
SAL=0
.=500
.WORD 177535,..+4,SAL   ;STORES 177535, 506 AND 0 IN
                        ;WORDS 500, 502 AND 504.
```

If an expression equates to a value of more than 16 bits, it is truncated and flagged with a T error code.

If an operand following the .WORD directive is null, it is interpreted as zero. For example:

```
.=500
.WORD ,5,                ;STORES 0, 5, AND 0 IN LOCATIONS
                        ;500, 502, AND 504.
```

A blank operator field (any operator not recognized as a macro call, op-code, directive or semicolon) is interpreted as an implicit .WORD directive. Use of this convention is discouraged. The first term of the first expression in the operand field must not be an instruction mnemonic or assembler directive unless preceded by a + or - operator. For example:

```
.=440                    ;THE OP-CODE FOR MOV, WHICH
                        ;IS 010000, IS STORED IN
LABEL: +MOV,LABEL        ;LOCATION 440. 440 IS
                        ;STORED IN LOCATION 442.
```

Note that the default .WORD directive occurs whenever there is a leading arithmetic or logical operator, or whenever a leading symbol is encountered which is not recognized as a macro call, an instruction mnemonic or assembler directive. Therefore, if an instruction mnemonic, macro call or assembler directive is misspelled, the .WORD directive is assumed and errors will result. Assume that MOV is spelled incorrectly as MOR:

```
MOR A,B
```

Two error codes result: Q occurs because an expression operator is missing between MOR and A, and a U occurs if MOR is undefined. Two words are then generated; one for MOR A and one for B.

### 5.5.3.3 ASCII Conversion of One or Two Characters

The ' and " characters are used to generate text characters within the source text. A single apostrophe followed by a character results in a term in which the 7-bit ASCII representation of the character is placed in the low-order byte and zero is placed in the high-order byte. For example:

```
MOV #'A,R0
```

results in the following 16 bits being moved into R0:

```
0000000001000001
```

The ' character is never followed by a carriage return, null, RUBOUT, line feed or form feed. (For another use of the ' character, see Section 5.6.3.6.)

```
STMNT:
GETSYM
BEQ      4$
CMPB    @CHRPNT,#':    ;COLON DELIMITS LABEL FIELD.
BEQ      LABEL
CMPB    @CHRPNT,#'=-   ;EQUAL DELIMITS
BEQ      ASGMT         ;ASSIGNMENT PARAMETER.
```

A double quote followed by two characters results in a term in which the 7-bit ASCII representations of the two characters are placed. For example:

```
MOV #"AB,R0
```

results in the following word being moved into R0:

```
0100001001000001
```

The " character is never followed by a carriage return, null, rubout, line feed or form feed. For example:

```
;DEVICE NAME TABLE

DEVNAM:  .WORD      "RF      ;RF DISK
          .WORD      "RK      ;RK DISK
DEVNKB:  .WORD      "TT      ;TERMINAL KEYBOARD
          .WORD      "DT      ;DECTAPE
          .WORD      "LP      ;LINE PRINTER
          .WORD      "PR      ;PAPER TAPE READER
          .WORD      "PP      ;PAPER TAPE PUNCH
          .WORD      0        ;TABLE'S END
```

### 5.5.3.4 .ASCII

The .ASCII directive translates character strings into their 7-bit ASCII equivalents for use in the source program. The format of the .ASCII directive is:

.ASCII /character string/

where: character string is a string of any acceptable printing ASCII characters including spaces. The string may not include null characters, rubout, return, line feed, vertical tab, or form feed. Nonprinting characters can be expressed in digits of the current radix and delimited by angle brackets. (Any legal, defined expression is allowed between angle brackets.)

/ / these are delimiting characters and may be any printing characters other than ; < and = characters and any character within the string.

As an example:

```
A: .ASCII /HELLO/ ;STORES ASCII REPRESENTATION OF
;THE LETTERS H.E.L.L.O IN
;CONSECUTIVE BYTES.

.ASCII /ABC/<15><12>/DEF/
;STORES
;101,102,103,15,12,104,105,106 IN
;CONSECUTIVE BYTES.

.ASCII /<AB>/ ;STORES 74,101,102,76 IN
;CONSECUTIVE BYTES
```

The ; and = characters are not illegal delimiting characters, but are preempted by their significance as a comment indicator and assignment operator, respectively. For other than the first group, semicolons are treated as beginning a comment field. For example:

	<u>DIRECTIVE</u>	<u>RESULT</u>	<u>EXPLANATION</u>
.ASCII	;ABC;/DEF/	A B C D E F	Acceptable, but not recommended procedure.
.ASCII	/ABC/;DEF;	A B C	;DEF; is treated as a comment and ignored.
.ASCII	/ABC/=DEF=	A B C D E F	Acceptable, but not recommended procedure.
.ASCII	=DEF=		The assignment .ASCII=DEF is performed and a Q error generated upon encountering the second =.

### 5.5.3.5 .ASCIZ

The .ASCIZ directive is equivalent to the .ASCII directive with a zero byte automatically inserted as the final character of the string. For example:

```
When a list or text string has been created with a
.ASCIZ directive, a search for the null character
can determine the end of the list. For example:
CR=15
LF=12
.
.
.
MOV #HELLO,R1
MOV #LINBUF,R2
X:  MOVB (R1)+,(R2)+
    BNE X
.
.
.
HELLO: .ASCIZ <CR><LF>/MACRO-11 V001A/<CR><LF> ;INTRO MESSAGE
```

### 5.5.3.6 .RAD50

The .RAD50 directive allows the user the capability to handle symbols in Radix-50 coded form (this form is sometimes referred to as MOD40 and is used in PDP-11 system programs). Radix-50 form allows three characters to be packed into sixteen bits; therefore, any 6-character symbol can be held in two words. The form of the directive is:

```
.RAD50 /string/
```

where: / / delimiters can be any printing characters other than the =, <, and ; characters.

string is a list of the characters to be converted (three characters per word) and which may consist of the characters A through Z, 0 through 9, dollar (\$), dot (.) and space ( ). If there are fewer than three characters (or if the last set is fewer than three characters) they are considered to be left justified and trailing spaces are assumed. Illegal nonprinting characters are replaced with a ? character and cause an I error flag to be set. Illegal printing characters set the Q error flag.

The trailing delimiter may be a carriage return, semicolon, or matching delimiter. For example:

```
.RAD50 /ABC ;PACK ABC INTO ONE WORD.
.RAD50 /AB/ ;PACK AB (SPACE) INTO ONE WORD.
.RAD50 // ;PACK 3 SPACES INTO ONE WORD.
```

```
.RAD50 /ABCD/ ;PACK ABC INTO FIRST WORD AND
;D SPACE SPACE INTO SECOND WORD.
```

Each character is translated into its Radix-50 equivalent as indicated in the following table:

<u>Character</u>	<u>Radix-50 Equivalent (octal)</u>
(space)	0
A-Z	1-32
\$	33
.	34
0-9	36-47

Note that another character could be defined for code 35, which is currently unused.

The Radix-50 equivalents for three characters (C1,C2,C3) are combined in one 16-bit word as follows:

$$\text{Radix 50 value} = ((C1*50)+C2)*50+C3$$

For example:

$$\text{Radix-50 value of ABC is } ((1*50)+2)*50+3 \text{ or } 3223$$

See Appendix E for a table to quickly determine Radix-50 equivalents.

Use of angle brackets is encouraged in the .ASCII, .ASCIIZ, and .RAD50 statements whenever leaving the text string to insert special codes. For example:

```
.ASCII <101> ;EQUIVALENT TO .ASCII/A/
.RAD50 /AB/<35> ;STORES 3255 IN NEXT WORD

CHR1=1
CHR2=2
CHR3=3
.
.
.
.RAD50<CHR1><CHR2><CHR3> ;EQUIVALENT TO .RAD50/ABC/
```

#### 5.5.4 Radix Control

##### 5.5.4.1 .RADIX

Numbers used in a MACRO source program are initially considered to be octal numbers. However, the programmer has the option of declaring the following radices:

2, 4, 8, 10

This is done via the .RADIX directive, of the form:

```
.RADIX n
```

where: n is one of the acceptable radices.

The argument to the .RADIX directive is always interpreted in decimal radix. Following any radix directive, that radix is the assumed base for any number specified until the following .RADIX directive.

The default radix at the start of each program, and the argument assumed if none is specified, is 8 (octal). For example:

```
.RADIX 10          ;BEGINS SECTION OF CODE WITH  
                  ;DECIMAL RADIX  
.  
.  
.  
.RADIX           ;REVERTS TO OCTAL RADIX
```

In general it is recommended that macro definitions not contain or rely on radix settings from the .RADIX directive. The temporary radix control characters should be used within a macro definition. ( $\uparrow D$ ,  $\uparrow O$ , and  $\uparrow B$  are described in the following section.) A given radix is valid throughout a program until changed. Where a possible conflict exists within a macro definition or in possible future uses of that code module, it is suggested that the user specify values using the temporary radix controls.

#### 5.5.4.2 Temporary Radix Control: $\uparrow D$ , $\uparrow O$ , and $\uparrow B$

Once the user has specified a radix for a section of code, or has determined to use the default octal radix he may discover a number of cases where an alternate radix is more convenient (particularly within macro definitions). For example, the creation of a mask word might best be done in the binary radix.

MACRO has three unary operators to provide a single interpretation in a given radix within another radix as follows:

```
 $\uparrow Dx$  (x is treated as being in decimal radix)  
 $\uparrow Ox$  (x is treated as being in octal radix)  
 $\uparrow Bx$  (x is treated as being in binary radix)
```

For example:

```
 $\uparrow D123$   
 $\uparrow O 47$   
 $\uparrow B 00001101$   
 $\uparrow O <A+3>$ 
```

Notice that while the up arrow and radix specification characters may not be separated, the radix operator can be physically separated from the number by spaces or tabs for formatting purposes. Where a term or expression is to be interpreted in another radix, it should be enclosed in angle brackets.

These numeric quantities may be used any place where a numeric value is legal.

PAL-11R contains a feature, which is maintained for compatibility in MACRO, allowing a temporary radix change from octal to decimal by specifying a decimal radix number with a "decimal point". For example:

```
    100.    (144(8))
   1376.    (2540(8))
    128.    (200(8))
```

### 5.5.5 Location Counter Control

The four directives which control movement of the location counter are .EVEN and .ODD which move the counter a maximum of one byte, and .BLKB and .BLKW which allow the user to specify blocks of a given number of bytes or words to be skipped in the assembly.

#### 5.5.5.1 .EVEN

The .EVEN directive ensures that the assembly location counter contains an even memory address by adding one if the current address is odd. If the assembly location counter is even, no action is taken. Any operands following a .EVEN directive are ignored.

The .EVEN directive is used as follows:

```
    .ASCIZ /THIS IS A TEST/
    .EVEN                               ;ASSURES NEXT STATEMENT
                                         ;BEGINS ON A WORD BOUNDARY.
    .WORD XYZ
```

#### 5.5.5.2 .ODD

The .ODD directive ensures that the assembly location counter is odd by adding one if it is even. For example:

```
    ;CODE TO MOVE DATA FROM AN INPUT LINE
    ;TO A BUFFER

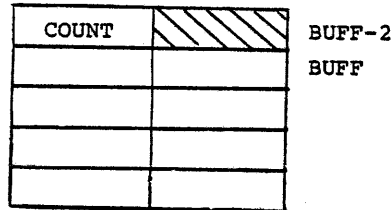
        N=5                               ;BUFFER HAS 5 WORDS
        .
        .
        .ODD
    BUFF: .BYTE N*2                       ;COUNT=2N BYTES
        .BLKW N                           ;RESERVE BUFFER OF N WORDS
        .
        .
        MOV #BUFF,R2                       ;ADDRESS OF EMPTY BUFFER IN R2
        MOV #LINE,R1                       ;ADDRESS OF INPUT LINE IS IN R1
```

```

AGAIN:   MOVB   -1(R2),R0      ;GET COUNT STORED IN BUFF-1 IN R0
        MOVB   (R1)+,(R2)+   ;MOVE BYTE FROM LINE INTO BUFFER
        BEQ    DONE          ;WAS NULL CHARACTER SEEN?
        DEC    R0            ;DECREMENT COUNT
        BNE    AGAIN         ;NOT = 0, GET NEXT CHARACTER.
        .
        .
        .
DONE:    CLRB   -(R2)         ;OUT OF ROOM IN BUFFER, CLEAR LAST
        .
        .
        .
LINE:    .ASCIZ /TEXT/

```

In this case, .ODD is used to place the buffer byte count in the byte preceding the buffer, as follows:



### 5.5.5.3 .BLKB and .BLKW

Blocks of storage can be reserved using the .BLKB and .BLKW directives. .BLKB is used to reserve byte blocks and .BLKW reserves word blocks. The two directives are of the form:

```

.BLKB   exp
.BLKW   exp

```

where: exp is the number of bytes or words to reserve. If no argument is present, 1 is the assumed default value. Any legal expression which is completely defined at assembly time and produces an absolute number is legal.

For example:

```

1          000000'          .CSECT  IMPURE
2
3 000000          PASS:   .BLKW
4
5 000002          SYMBOL: .BLKW  2          ;NEXT GROUP MUST STAY TOGETHER
6 000006          MODE:           ;SYMBOL ACCUMULATOR
7 000006          FLAGS:  .BLKB  1          ;FLAG BITS
8 000007          SECTOR: .BLKB  1          ;SYMBOL/EXPRESSION TYPE
9 000010          VALUE:  .BLKW  1          ;EXPRESSION VALUE
10 00012          RELLVL: .BLKW  1
11              .BLKW  2          ;END OF GROUPED DATA
12

```



```

13 00020          CLCNAM: .BLKW  2          ;CURRENT LOCATION COUNTER SYMBOL
14 00024          CLCFGs: .BLKB  1
15 00025          CLCSEC: .BLKB  1
16 00026          CLCLOC: .BLKW  1
17 00030          CLCMAX: .BLKW  1

```

The .BLKB directive has the same effect as

```

.=.+exp

```

but is easier to interpret in the context of source code.

### 5.5.6 Numeric Control

Several directives are available to provide software complements to the floating-point hardware on the PDP-11.

A floating-point number is represented by a string of decimal digits. The string (which can be a single digit in length) may optionally contain a decimal point, and may be followed by an optional exponent indicator; in the form of the letter E and a signed decimal exponent. The list of number representations below contains seven distinct, valid representations of the same floating-point number:

```

3
3.
3.0
3.0E0
3E0
.3E1
300E-2

```

As can be quickly inferred, the list could be extended indefinitely (e.g., 3000E-3, .03E2, etc.). A leading plus sign is ignored (e.g., +3.0 is considered to be 3.0). Leading minus signs complement the sign bit. No other operators are allowed (e.g., 3.0+N is illegal).

Floating-point number representations are only valid in the contexts described in the remainder of this Section.

Floating-point numbers are normally rounded. That is, when a floating-point number exceeds the limits of the field in which it is to be stored, the high-order excess bit is added to the low-order retained bit. For example, if the number were to be stored in a 2-word field, but more than 32 bits were needed for its value, the highest bit carried out of the field would be added to the least significant position. In order to enable floating-point truncation, the .ENABL FPT directive is used and .DSABL FPT is used to return to floating-point rounding (see Section 5.5.2).

#### 5.5.6.1 .FLT2 and .FLT4

Like the .WORD directive, the two floating-point storage directives cause their arguments to be stored in-line with the source program (have no effect in ASEMBL). These two directives are of the form:

```
.FLT2    arg1,arg2,...
.FLT4    arg1,arg2,...
```

where: arg1,arg2,... represents one or more floating point numbers separated by commas.

.FLT2 causes two words of storage to be generated for each argument while .FLT4 generates four words of storage.

The following code was assembled with the 4-word floating-point math package:

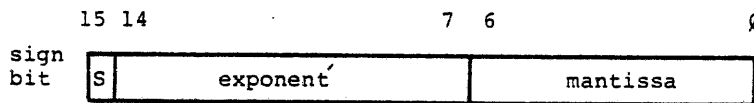
```
006010' 037314 146314 146314 ATOFTB: .FLT4 1.E-1 ;10-1
006016' 146315
006020' 036443 153412 036560 .FLT4 1.E-2 ;10-2
006026' 121727
006030' 034721 133427 054342 .FLT4 1.E-4 ;10-4
006036' 014545
006040' 031453 146167 010604 .FLT4 1.E-8 ;10-8
006046' 060717
006050' 022746 112624 137304 .FLT4 1.E-16 ;10-16
006056' 046741
006060' 005517 130436 126505 .FLT4 1.E-32 ;10-32
006066' 034625
```

#### 5.5.6.2 Temporary Numeric Control: †F and †C

Like the temporary radix control operators, operators are available to specify either a one-word floating-point number (†F) (not available in ASEMBL) or the one's complement of a one-word number (†C). For example:

```
FL3.7: †F3.7
```

creates a one-word floating-point number at location FL3.7 containing the value 3.7 as follows:



This one-word floating-point number is the first word of the 2- or 4-word floating-point number format shown in the PDP-11 Processor Handbook, and the statement:

```
CMP151: †C151
```

stores the one's complement of 151 in the current radix (assume current radix is octal) as follows:

177626
--------

Since these control operators are unary operators, their arguments may be integer constants or symbols, and the operators may be expressed recursively. For example:

↑F<1.2E3>  
 ↑C↑D25    or    ↑C31        or    177746

The term created by the unary operator and its argument is then a term which can be used by itself or in an expression. For example:

↑C2+6

is equivalent to:

<↑C2>+6    or    177775+6    or    000003

For this reason, the use of angle brackets is advised. Expressions used as terms, or arguments of a unary operator must be explicitly grouped.

An example of the importance of ordering with respect to unary operators is shown below:

↑F1.0	=	020400
↑F-1.0	=	120400
-↑F1.0	=	157400
-↑F-1.0	=	057400

The argument to the ↑F operator must not be an expression and should be of the same format as arguments to the .FLT2 and .FLT4 directives (see Section 5.5.6.1).

## 5.5.7 Terminating Directives

### 5.5.7.1 .END

The .END directive indicates the physical end of the source program. The .END directive is of the form:

.END exp

where:    exp        is an optional argument which, if present, indicates the program entry point, i.e., the transfer address.

When the load module is loaded, program execution begins at the transfer address indicated by the .END directive. In a runtime system

(the load module output of the Linker) a .END statement should terminate the first object module and .END statements should terminate any other object modules.

#### 5.5.7.2 .EOT

Under the RT-11 System, the .EOT directive is ignored. The physical end file allows several physically separate tapes to be assembled as one program.

#### 5.5.8 Program Boundaries Directive: .LIMIT

The .LIMIT directive reserves two words into which the Linker puts the low and high addresses of the load module's relocatable code. The low address (inserted into the first word) is the address of the first byte of code. The high address is the address of the first free byte following the relocated code. These addresses are always even since all relocatable sections are loaded at even addresses. (If a relocatable section consists of an odd number of bytes, the Linker adds one to the size to make it even.)

#### 5.5.9 Program Section Directives

The Assembler provides for 255(10) program sections: an absolute section declared by .ASECT, an unnamed relocatable program section declared by .CSECT, and 253(10) named relocatable program sections declared by .CSECT symbol, where symbol is any legal symbolic name, these directives allow the user to:

1. Create his program (object module) in sections:

The Assembler maintains separate location counters for each section. This allows the user to write statements which are not physically contiguous but will be loaded contiguously. The following examples will clarify this:

```

.CSECT                ;START THE UNNAMED RELOCATABLE SECTION
A: 0                  ;ASSEMBLED AT RELOCATABLE 0,
B: 0                  ;   RELOCATABLE 2 AND
C: 0                  ;   RELOCATABLE 4,
ST: CLR A             ;ASSEMBLE CODE AT
      CLR B           ;   RELOCATABLE ADDRESS
      CLR C           ;   6 THROUGH 21
.ASECT                ;START THE ABSOLUTE SECTION
.=4                   ;ASSEMBLE CODE AT
.WORD .+2,HALT       ;   ABSOLUTE 4 THROUGH 7,
.CSECT                ;RESUME THE UNNAMED RELOCATABLE
                      ;   SECTION
      INC A           ;ASSEMBLE CODE AT
      BR ST          ;   RELOCATABLE 22 THROUGH 27,
.END

```

The first appearance of .CSECT or .ASECT assumes the location counter is at relocatable or absolute zero, respectively. The scope of each directive extends until a directive to the contrary is given. Further occurrences of the same .CSECT or .ASECT resume assembling where the section was left off.

```

        .CSECT    COM1      ;DECLARE SECTION COM1
A:      0          ;ASSEMBLED AT RELOCATABLE 0.
B:      0          ;ASSEMBLED AT RELOCATABLE 2.
C:      0          ;ASSEMBLED AT RELOCATABLE 4.
        .CSECT    COM2      ;DECLARE SECTION COM2
X:      0          ;ASSEMBLED AT RELOCATABLE 0.
Y:      0          ;ASSEMBLED AT RELOCATABLE 2.
        .CSECT    COM1      ;RETURN TO COM1
D:      0          ;ASSEMBLED AT RELOCATABLE 6.
        .END

```

The Assembler automatically begins assembling at relocatable zero of the unnamed .CSECT if not instructed otherwise; that is, the first statement of an assembly is an implied .CSECT.

All labels in an absolute section are absolute; all labels in a relocatable section are relocatable. The location counter symbol, ".", is relocatable or absolute when referenced in a relocatable or absolute section, respectively. Undefined internal symbols are assigned the value of relocatable or absolute zero in a relocatable or absolute section, respectively. Any labels appearing on a .ASECT or .CSECT statement are assigned the value of the location counter before the .ASECT or .CSECT takes effect. Thus, if the first statement of a program is:

```
A: .ASECT
```

then A is assigned to relocatable zero and is associated with the unnamed relocatable section (because the Assembler implicitly begins assembly in the unnamed relocatable section).

Since it is not known at assembly time where the program sections are to be loaded, all references between sections in a single assembly are translated by the Assembler to references relative to the base of that section. The Assembler provides the Linker with the necessary information to resolve the linkage. Note that this is not necessary when making a reference to an absolute section (the Assembler knows all load addresses of an absolute section).

Examples:

```

        .ASECT
        .=1000
A:      CLR X          ;ASSEMBLED AS CLR BASE OF UNNAMED
                        ; RELOCATABLE SECTION + 10
        JMP Y          ;ASSEMBLED AS JMP BASE OF UNNAMED
                        ; RELOCATABLE SECTION + 6
        .CSECT
        MOV R0,R1
        JMP A          ;ASSEMBLED AS JMP 1000

```

```
Y:  HALT
X:   0
    .END
```

In the above example the references to X and Y were translated into references relative to the base of the unnamed relocatable section.

2. Share code and/or data between object modules (separate assemblies):

Named relocatable program sections operate as FORTRAN labeled COMMON; that is, sections of the same name from different assemblies are all loaded at the same location by Link. The unnamed relocatable section is the exception to this as all unnamed relocatable sections are loaded in unique areas by Link.

Note that there is no conflict between internal symbolic names and program section names; that is, it is legal to use the same symbolic name for both purposes. In fact, considering FORTRAN again, this is necessary to accommodate the FORTRAN statement:

```
COMMON  /X/A,B,C,X
```

where the symbol X represents the base of this program section and also the fourth element of this program section.

Program section names should not duplicate .GLOBL names. In FORTRAN language, COMMON block names and SUBROUTINE names should not be the same.

#### 5.5.9.1 .ASECT and .CSECT

The following program section directives are provided in MACRO to allow the user to specify an unnamed absolute or relative section. These directives are formatted as follows:

```
.ASECT
.CSECT
.CSECT      symbol
```

An unnamed absolute section can be declared with an

```
.ASECT
```

directive. No name can be associated with an absolute section specified by means of the .ASECT directive. The single unnamed relocatable program section can be declared with a

```
.CSECT
```

directive. All unnamed relocatable sections are loaded in unique areas by Link. Up to 253(10) named relocatable program sections can be declared with

.CSECT            symbol

directives, where symbol is any legal symbolic name.

The Assembler automatically begins assembling at relocatable zero of the unnamed .CSECT if not instructed otherwise; that is, the first statement of an assembly is an implied .CSECT.

#### 5.5.10 Symbol Control: .GLOBL

If a program is created in segments which are assembled separately, global symbols are used to allow reference to one symbol by the different segments.

A global symbol must be declared in a .GLOBL directive.

The form of the .GLOBL directive is:

```
.GLOBL    sym1,sym2,...
```

where:    sym1,sym2,... are legal symbolic names, separated by commas or spaces where more than one symbol is specified.

Symbols appearing in a .GLOBL directive are either defined within the current program or are external symbols, in which case they are defined in another program which is to be linked with the current program, by Link, prior to execution.

A .GLOBL directive line may contain a label in the label field and comments in the comment field.

At the end of assembly pass 1, MACRO has determined whether a given global symbol is defined within the program or is expected to be an external symbol.

```
      ;DEFINE A SUBROUTINE WITH 2 ENTRY POINTS WHICH CALLS AN  
      ;   EXTERNAL SUBROUTINE  
      .CSECT                   ;DECLARE THE CONTROL SECTION  
      .GLOBL    A,B,C         ;DECLARE A, B, C AS GLOBALS  
A:   MOV        @(R5)+,R0     ;ENTRY A IS DEFINED  
      MOV        #X,R1  
X:   JSR        PC,C         ;CALL EXTERNAL SUBROUTINE C  
      RTS       R5           ;EXIT  
B:   MOV        @(R5)+,R1     ;DEFINE ENTRY B  
      CLR       R1  
      BR        X
```

In the previous example, A and B are entry symbols (entry points), C is an external symbol and X is an internal symbol.

A global symbol is defined only when it appears in a .GLOBL directive. A symbol is not considered a global symbol if it is assigned the value of a global expression in a direct assignment statement.

References to external symbols can appear in the operand field of an instruction or assembler directive in the form of a direct reference, i.e.:

```

CLR      EXT
.WORD    EXT
CLR      @EXT

```

or a direct reference plus or minus a constant, i.e.:

```

A=6
CLR      EXT+A
.WORD    EXT-2
CLR      @EXT+A

```

An external symbol cannot be used in the evaluation of a direct assignment expression. A global symbol defined within the program can be used in the evaluation of a direct assignment statement.

#### 5.5.11 Conditional Assembly Directives

Conditional assembly directives provide the programmer with the capability to conditionally include or ignore blocks of source code in the assembly process. This technique is used extensively to allow several variations of a program to be generated from the source program.

The general form of a conditional block is as follows:

```

      .IF cond,argument(s)      ;START CONDITIONAL BLOCK
      .                          ;STATEMENTS IN RANGE OF
      .                          ;CONDITIONAL
      .                          ;BLOCK
      .ENDC                      ;END CONDITIONAL BLOCK

```

where *cond* is a condition which must be met if the block is to be included in the assembly. These conditions are defined below.

*argument(s)* are a function of the condition to be tested. If more than one argument is specified, they must be separated by commas.

*range* is the body of code which is included in the assembly or ignored depending upon whether the condition is met.

The following are the allowable conditions:

Conditions		ARGUMENTS	ASSEMBLE BLOCK IF
POSITIVE	COMPLEMENT		
EQ	NE	expression	expression=0 (or ≠0)
GT	LE	expression	expression>0 (or ≤0)
LT	GE	expression	expression<0 (or ≥0)



<u>POSITIVE</u>	<u>Conditions COMPLEMENT</u>	<u>ARGUMENTS</u>	<u>ASSEMBLE BLOCK IF</u>
DF	NDF	symbolic argument	symbol is defined (or undefined)
B	NB	macro-type argument	argument is blank (or nonblank)
IDN	DIF	two macro-type arguments separated by a comma	arguments identical (or different)
Z	NZ	expression	same as EQ/NE
G		expression	same as GT/LE
L		expression	same as LT/GE

If DIF and IF IDN are not available in ASEMBL.

NOTE

A macro-type argument is enclosed in angle brackets or within an up-arrow construction (as described in Section 5.6.3.1). For example:

```
<A,B,C>
↑/124/
```

For example:

```
.IF EQ    ALPHA+1    ;ASSEMBLE IF ALPHA+1=0
.
.
.
.ENDC
```

Within the conditions DF and NDF the following two operators are allowed to group symbolic arguments:

```
&        logical AND operator
!        logical inclusive OR operator
```

For example:

```
.IF DF SYM1 & SYM2
.
.
.
.ENDC
```

assembles if both SYM1 and SYM2 are defined.

### 5.5.11.1 Subconditionals

Subconditionals may be placed within conditional blocks to indicate:

1. assembly of an alternate body of code when the condition of the block indicates that the code within the block is not to be assembled,
2. assembly of a non-contiguous body of code within the conditional block depending upon the result of the conditional test to enter the block,
3. unconditional assembly of a body of code within a conditional block.

There are three subconditional directives, as follows:

<u>Subconditional</u>	<u>Function</u>
.IFF	The code following this statement up to the next subconditional or end of the conditional block is included in the program if the value of the condition tested upon entering the conditional block is false.
.IFT	The code following this statement up to the next subconditional or end of the conditional block is included in the program if the value of the condition tested upon entering the conditional block is true.
.IFTF	The code following this statement up to the next subconditional or the end of the conditional block is included in the program regardless of the value of the condition tested upon entering the conditional block.

The implied argument of the subconditionals is the value of the condition upon entering the conditional block. Subconditionals are used within outer level conditional blocks. Subconditionals are ignored within nested, unsatisfied conditional blocks.

For example:

```
.IF DF SYM ;ASSEMBLE BLOCK IF SYM IS DEFINED
.IFF
.
.
.
.IFT ;ASSEMBLE THE FOLLOWING CODE ONLY IF
. ;SYM IS DEFINED.
.
.
.IFTF ;ASSEMBLE THE FOLLOWING CODE
. ;UNCONDITIONALLY.
.
.
.ENDC

.IF DF X ;ASSEMBLY TESTS FALSE
.IF DF Y ;TESTS FALSE
```

```

.IFF                ;NESTED CONDITIONAL
.
.
.
.IFT                ;NOT SEEN
.
.
.ENDC
.ENDC

```

However,

```

.IF DF X            ;TESTS TRUE
.IF DF Y            ;TESTS FALSE
.IFF                ;IS ASSEMBLED
.
.
.
.IFT                ;NOT ASSEMBLED
.
.
.ENDC
.ENDC

```

#### 5.5.11.2 Immediate Conditionals

An immediate conditional directive is a means of writing a one-line conditional block. In this form, no .ENDC statement is required and the condition is completely expressed on the line containing the conditional directive. Immediate conditions are of the form:

```
.IIF cond, arg, statement
```

where: cond is one of the legal conditions defined for conditional blocks in Section 6.11.

arg is the argument associated with the conditional specified, that is, either an expression, symbol, or macro-type argument, as described in Section 5.5.11.

statement is the statement to be executed if the condition is met.

For example:

```
.IIF DF FOO, BEQ ALPHA
```

this statement generates the code

```
BEQ ALPHA
```

if the symbol FOO is defined.

A label must not be placed in the label field of the .IIF statement. Any necessary labels may be placed on the previous line:

```
LABEL:
.IIF DF   FPP,BEQ ALPHA
```

or included as part of the conditional statement:

```
.IIF DF   FOO,LABEL:   BEQ ALPHA
```

### 5.5.11.3 PAL-11R and PAL-11S Conditional Assembly Directives

In order to maintain compatibility with programs developed under PAL-11R and PAL-11S, the following conditionals remain permissible under MACRO. It is advisable that future programs be developed using the format for MACRO conditional assembly directives.

<u>Directive</u>	<u>Arguments</u>	<u>Assemble Block if</u>
.IFZ or .IFEQ	expression	expression=0
.IFNZ or .IFNE	expression	expression≠0
.IFL or .IFLT	expression	expression<0
.IFG or .IFGT	expression	expression>0
.IFGE	expression	expression≥0
.IFLE	expression	expression≤0
.IFDF	logical expression	expression is true (defined)
.IFNDF	logical expression	expression is false (undefined)

The rules governing the usage of these directives are now the same as for the MACRO conditional assembly directives previously described. Conditional assembly blocks must end with the .ENDC directive and are limited to a nesting depth of 16(10) levels (instead of the 127(10) levels allowed under PAL-11R).

## 5.6 MACRO DIRECTIVES

### 5.6.1 Macro Definition

It is often convenient in assembly language programming to generate a recurring coding sequence with a single statement. In order to do this, the desired coding sequence is first defined with dummy arguments as a macro. Once a macro has been defined, a single statement calling the macro by name with a list of real arguments (replacing the corresponding dummy arguments in the definition) generates the correct sequence or expansion.

#### 5.6.1.1 .MACRO

The first statement of a macro definition must be a `.MACRO` directive (not available in `ASEMBL`). The `.MACRO` directive is of the form:

```
.MACRO name, dummy argument list
```

where:

`name` is the name of the macro. This name is any legal symbol. The name chosen may be used as a label elsewhere in the program.

`,` represents any legal separator (generally a comma or space).

`dummy argument list` zero, one, or more legal symbols which may appear anywhere in the body of the macro definition, even as a label. These symbols can be used elsewhere in the user program with no conflicts of definition. Where more than one dummy argument is used, they are separated by any legal separator (generally a comma).

A comment may follow the dummy argument list in a statement containing a `.MACRO` directive. For example:

```
.MACRO ABS A,B ;DEFINE MACRO ABS WITH TWO ARGUMENTS
```

A label must not appear on a `.MACRO` statement. Labels are sometimes used on macro calls, but serve no function when attached to `.MACRO` statements.

#### 5.6.1.2 .ENDM

The final statement of every macro definition must be an `.ENDM` directive (not available in `ASEMBL`) of the form:

```
.ENDM name
```

where:

name is an optional argument, and is the name of the macro being terminated by the statement.

For example:

```
.ENDM (terminates the current macro definition)
.ENDM ABS (terminates the definition of the macro ABS)
```

If specified, the symbolic name in the .ENDM statement must correspond to that in the matching .MACRO statement. Otherwise the statement is flagged and processing continues. Specification of the macro name in the .ENDM statement permits the Assembler to detect missing .ENDM statements or improperly nested macro definitions.

The .ENDM statement may contain a comment field, but must not contain a label.

An example of a macro definition is shown below:

```
.MACRO TYPMSG MESSGE ;TYPE A MESSAGE
JSR R5,TYPMSG
.WORD MESSGE
.ENDM
```

### 5.6.1.3 .MEXIT

In order to implement alternate exit points from a macro (particularly nested macros), the .MEXIT directive is provided. .MEXIT (not available in ASEMBL) terminates the current macro as though an .ENDM directive were encountered. Use of .MEXIT bypasses the complications of conditional nesting and alternate paths. For example:

```
.MACRO ALTR N,A,B
.
.
.
. IF EQ,N ;START CONDITIONAL BLOCK
.
.
. MEXIT ;EXIT FROM MACRO DURING CONDITIONAL
;BLOCK
. ENDC ;END CONDITIONAL BLOCK
.
.
. ENDM ;NORMAL END OF MACRO
```

In an assembly where N=0, the .MEXIT directive terminates the macro expansion.

Where macros are nested, a .MEXIT causes an exit to the next higher level. A .MEXIT encountered outside a macro definition is flagged as an error.

#### 5.6.1.4 MACRO Definition Formatting

A form feed character used as a line terminator on MACRO source statement, (or as the only character on a line) causes a page eject. Used within a macro definition, a form feed character causes a page eject. A page eject is not performed when the macro is invoked.

Used within a macro definition, the .PAGE directive is ignored, but a page eject is performed at invocation of that macro.

#### 5.6.2 Macro Calls

A macro must be defined prior to its first reference. Macro calls are of the general form:

label: name, real arguments

where: label represents an optional statement label.

name represents the name of the macro specified in the .MACRO directive preceding the macro definition.

represents any legal separator (comma, space, or tab). No separator is necessary where there are no real arguments.

real arguments are those symbols, expressions, and values which replace the dummy arguments in the .MACRO statement. Where more than one argument is used, they are separated by any legal separator.

Where a macro name is the same as a user label, the appearance of the symbol in the operation field designates a macro call, and the occurrence of the symbol in the operand field designates a label reference. For example:

```
ABS: MOV @R0,R1          ;ABS IS USED AS LABEL
.
.
BR ABS                   ;ABS IS CONSIDERED A LABEL
.
.
ABS #4,ENT,LAR          ;CALL MACRO ABS WITH 3 ARGUMENTS
```

Arguments to the macro call are treated as character strings whose usage is determined by the macro definition.

#### 5.6.3 Arguments to Macro Calls and Definitions

Arguments within a macro definition or macro call are separated from other arguments by any of the separating characters described in Section 5.2.1.1.

For example:

```
.MACRO   REN A,B,C
.
.
.
REN      ALPHA,BETA,<C1,C2>
```

Arguments which contain separating characters are enclosed in paired angle brackets. An up-arrow construction is provided to allow angle brackets to be passed as arguments.

For example:

```
REN <MOV X,Y> ,#44,WEV
```

This call would cause the entire statement:

```
MOV X,Y
```

to replace all occurrences of the symbol A in the macro definition. Real arguments within a macro call are considered to be character strings and are treated as a single entity until their use in the macro expansion.

The up-arrow construction could have been used in the above macro call as follows:

```
REN †/MOV X,Y/,#44,WEV
```

which is equivalent to:

```
REN <MOV X,Y> ,#44,WEV
```

Since spaces are ignored preceding an argument, they can be used to increase legibility of bracketed constructions.

The form:

```
REN #44,WEV†/MOV X,Y/
```

however, contains three arguments which are "#44" "WEV†/MOV" and "X,Y/" (see section 5.2.1.1) because † is a unary operator.

#### 5.6.3.1 Macro Nesting

Macro nesting (nested macro calls), where the expansion of one macro includes a call to another macro, causes one set of angle brackets to be removed from an argument with each nesting level. The depth of nesting allowed is dependent upon the amount of core space used by the program. To pass an argument containing legal argument delimiters to nested macros, the argument should be enclosed in one set of angle brackets for each level of nesting, as shown below:

```
.MACRO   LEVEL1   DUM1,DUM2
LEVEL2   DUM1
LEVEL2   DUM2
```



```

.ENDM

.MACRO LEVEL2 DUM3
DUM3
ADD #10,R0
MOV R0,(R1)+
.ENDM

```

A call to the LEVEL1 macro:

```
LEVEL1 <<MOV X,R0>>,<<CLR R0>>
```

causes the following expansion:

```

→ MOV X,R0
  ADD #10,R0
  MOV R0,(R1)+
→ CLR R0
  ADD #10,R0
  MOV R0,(R1)+

```

where macro definitions are nested (that is, a macro definition is entirely contained within the definition of another macro) the inner definition is not defined as a callable macro until the outer macro has been called and expanded. For example:

```

.MACRO LV1 A,B
.
.
.
.MACRO LV2 A
.
.
.
.ENDM
.ENDM

```

The LV2 macro cannot be called by name until after the first call to the LV1 macro. Likewise, any macro defined within the LV2 macro definition cannot be referenced directly until LV2 has been called.

### 5.6.3.2 Special Characters

Arguments may include special characters without enclosing the argument in a bracket construction if that argument does not contain spaces, tabs, semi-colons, or commas. For example:

```

.MACRO PUSH ARG
MOV ARG,-(SP)
.ENDM
.
.
.
PUSH X+3(%2)

```

generates the following code:

```
MOV X+3(%2),-(SP)
```

### 5.6.3.3 Numeric Arguments Passed as Symbols

When passing macro arguments, a useful capability is to pass a symbol which can be treated by the macro as a numeric string. An argument preceded by the unary operator backslash (\) is treated as a number in the current radix. (\ is not available in ASEMBL.) The ASCII characters representing the number are inserted in the macro expansion; their function is defined in context. For example:

```
.MACRO CNT A,B
A'B: .WORD
.ENDM
B=0
.MACRO INC A,B
CNT A,\B
B=B+1
.ENDM
.
.
.
INC X,C
```

The macro call would expand to:

```
X0: .WORD
```

A subsequent identical call to the same macro would generate:

```
X1: .WORD
```

and so on for later calls. The two macros are necessary because the dummy value of B cannot be updated in the CNT macro. In the CNT macro, the number passed is treated as a string argument. (Where the value of the real argument is 0, a single 0 character is passed to the macro expansion.)

The number being passed can also be used to make source listings somewhat clearer. For example, versions of programs created through conditional assembly of a single source can identify themselves as follows:

```
.MACRO IDT SYM ;ASSUME THAT THE SYMBOL ID TAKES
.IDENT /SYM/ ;ON A UNIQUE 2 DIGIT VALUE FOR
.ENDM ;EACH POSSIBLE CONDITIONAL ASSEMBLY
.MACRO OUT ARG ;OF THE PROGRAM
IDT 005A'ARG .
.ENDM .
. .
. .
OUT \ID ;WHERE 005A IS THE UPDATE
;VERSION OF THE PROGRAM
;AND ARG INDICATES THE
;CONDITIONAL ASSEMBLY VERSION.
```

The above macro call expands to

```
.IDENT /005AXX/
```

where XX is the conditional value of ID.

Two macros are necessary since the text delimiting characters in the .IDENT statement would inhibit the concatenation of a dummy argument.

#### 5.6.3.4 Number of Arguments

If more arguments appear in the macro call than in the macro definition, the excess arguments are ignored. If fewer arguments appear in the macro call than in the definition, missing arguments are assumed to be null (consist of no characters). The conditional directives .IFB and .IFNB can be used within the macro to detect unnecessary arguments.

A macro can be defined with no arguments.

#### 5.6.3.5 Automatically Created Symbols

MACRO can be made to create symbols of the form n\$ where n is a decimal integer number such that  $64 \leq n \leq 127$ . Created symbols are always local symbols between 64\$ and 127\$. (For a description of local symbols, see Section 5.2.5.) Such local symbols are created by the Assembler in numerical order, i.e.:

```
64$
65$
.
.
.
126$
127$
```

Created symbols are particularly useful where a label is required in the expanded macro. Such a label must otherwise be explicitly stated as an argument with each macro call or the same label is generated with each expansion (resulting in a multiply-defined label). Unless a label is referenced from outside the macro, there is no reason for the programmer to be concerned with that label.

The range of these local symbols extends between two explicit labels. Each new explicit label causes a new local symbol block to be initialized.

The macro processor creates a local symbol on each call of a macro whose definition contains a dummy argument preceded by the ? character. For example:

```
        .MACRO  ALPHA, 3A,?B
        TST    A
        BEQ    B
        ADD    #5,A
B:
        .ENDM
```

Local symbols are generated only where the real argument of the macro call is either null or missing. If a real argument is specified in the macro call, the generation of a local symbol is inhibited and normal replacement is performed. Consider the following expansions of the macro ALPHA above.

Generate a local symbol for missing argument:

```

        ALPHA      %1
        TST        %1
        BEQ        64$
        ADD        #5,%1
64$:

```

Do not generate a local symbol:

```

        ALPHA      %2,XYZ
        TST        %2
        BEQ        XYZ
        ADD        #5,%2
XYZ:

```

These Assembler-generated symbols are restricted to the first sixteen (decimal) arguments of a macro definition.

#### 5.6.3.6 Concatenation

The apostrophe or single quote character (') operates as a legal separating character in macro definitions. An ' character which precedes and/or follows a dummy argument in a macro definition is removed and the substitution of the real argument occurs at that point. For example:

```

        .MACRO DEF A,B,C
A'B: .ASCIZ /C/
        .WORD 'A''B
        .ENDM

```

When this macro is called:

```
DEF X,Y,<MACRO-11>
```

it expands as follows:

```

XY: .ASCIZ /MACRO-11/
        .WORD 'X'Y

```

In the macro definition, the scan terminates upon finding the first ' character. Since A is a dummy argument, the ' is removed. The scan resumes with B, notes B as another dummy argument and concatenates the two dummy arguments. The third dummy argument is noted as going into the operand of the .ASCIZ directive. On the next line (this is not a useful example, but one for purely illustrative purposes) the argument to .WORD is seen as follows: The scan begins with a ' character. Since it is neither preceded nor followed by a dummy argument, the ' character remains in the macro definition. The scan then encounters the second ' character which is followed by a dummy argument and is

discarded. The scan of the argument A terminated upon encountering the second ' which is also discarded since it follows a dummy argument. The next ' character is neither preceded nor followed by a dummy argument and remains in the macro expansion. The last ' character is followed by another dummy argument and is discarded. (Note that the five ' characters were necessary to generate two ' characters in the macro expansion.)

Within nested macro definitions, multiple single quotes can be used, with one quote removed at each level of macro nesting.

#### 5.6.4 .NARG, .NCHR, and .NTYPE

These three directives allow the user to obtain the number of arguments in a macro call (.NARG), the number of characters in an argument (.NCHR), or the addressing mode of an argument (.NTYPE). (They are not available in ASEMBL.) Use of these directives permits selective modifications of a macro depending upon the nature of the arguments passed.

The .NARG directive enables the macro being expanded to determine the number of arguments supplied in the macro call and is of the form:

label: .NARG symbol

where: label is an optional statement label

symbol is any legal symbol whose value is equated to the number of arguments in the macro call currently being expanded. The symbol can be used by itself or in expressions.

This directive can occur only within a macro definition.

The .NCHR directive enables a program to determine the number of characters in a character string, and is of the form:

label: .NCHR symbol, <character string>

where: label is an optional statement label

symbol is any legal symbol which is equated to the number of characters in the specified character string. The symbol is separated from the character string argument by any legal separator.

<character string>

is a string of printing characters which should only be enclosed in angle brackets if it contains a legal separator. A semi-colon also terminates the character string.

This directive can occur anywhere in a MACRO program.

The `.NTYPE` directive enables the macro being expanded to determine the addressing mode of any argument, and is of the form:

```
label:      .NTYPE      symbol, arg
```

where: label is an optional statement label

symbol is any legal symbol, the low order 6-bits of which is equated to the 6-bit addressing mode of the argument. The symbol is separated from the argument by a legal separator. This symbol can be used by itself or in expressions.

arg is any legal macro argument (dummy argument) as defined in Section 5.6.3.

This directive can occur only within a macro definition. An example of `.NTYPE` usage in a macro definition is shown below:

```
.MACRO      SAVE      ARG
.NTYPE     SYM,ARG
.IF        EQ,SYM&70
MOV        ARG,TEMP      ;REGISTER MODE
.IFF
MOV        #ARG,TEMP      ;NON-REGISTER MODE
.ENDC
.ENDM
```

#### 5.6.5 `.ERROR` and `.PRINT`

The `.ERROR` directive (not available in `ASEMBL`) is used to output messages to the listing file during assembly pass 2. A common use is to provide diagnostic announcements of a rejected or erroneous macro call. The form of the `.ERROR` directive is as follows:

```
label:      .ERROR expr;text
```

where label is an optional statement label

expr is an optional legal expression whose value is output to the listing file when the `.ERROR` directive is encountered. Where `expr` is not specified, the text only is output to the listing file.

;

denotes the beginning of the text string to be output.

text is the string to be output to the listing file. The text string is terminated by a line terminator.

Upon encountering a `.ERROR` directive anywhere in a `MACRO` program, the Assembler outputs a single line containing:

1. the sequence number of the `.ERROR` directive line,

2. the current value of the location counter,
3. the value of the expression if one is specified, and,
4. the text string specified.

For example:

```
.ERROR      A;UNACCEPTABLE MACRO ARGUMENT
```

causes a line similar to the following to be output:

```
512 5642 000076      ;UNACCEPTABLE MACRO ARGUMENT
```

This message is being used to indicate an inability of the subject macro to cope with the argument A which is detected as being indexed deferred addressing mode (mode 70) with the stack pointer (%6) used as the index register.

The line is flagged on the assembly listing with a P error code.

The .PRINT directive is identical to .ERROR except that it is not flagged with a P error code. (.PRINT is not available in ASEMBL.)

#### 5.6.6 Indefinite Repeat Block: .IRP and .IRPC

An indefinite repeat block (not available in ASEMBL) is a structure very similar to a macro definition. An indefinite repeat is essentially a macro definition which has only one dummy argument and is expanded once for every real argument supplied. An indefinite repeat block is coded in-line with its expansion rather than being referenced by name as a macro is referenced. An indefinite repeat block is of the form:

```
label:      .IRP arg,<real arguments>
            .
            .
            (range of the indefinite repeat)
            .
            .
            .ENDM
```

where: label is an optional statement label. A label may not appear on any .IRP statement within another macro definition, repeat range or indefinite repeat range, or on any .ENDM statement.

arg is a dummy argument which is successively replaced with the real arguments in the .IRP statement.

<real arguments>

is a list of arguments to be used in the expansion of the indefinite repeat range and enclosed in angle brackets. Each real argument is a string of zero or more characters or a list of real

arguments (enclosed in angle brackets). The real arguments are separated by commas.

range is the block of code to be repeated once for each real argument in the list. The range may contain macro definitions, repeat ranges, or other indefinite repeat ranges. Note that only created symbols should be used as labels within an indefinite repeat range.

An indefinite repeat block can occur either within or outside macro definitions, repeat ranges, or indefinite repeat ranges. The rules for creating an indefinite repeat block are the same as for the creation of a macro definition (for example, the .MEXIT statement is allowed in an indefinite repeat block). Indefinite repeat arguments follow the same rules as macro arguments.

A second type of indefinite repeat block is available which handles character substitution rather than argument substitution. The .IRPC directive is used as follows:

```
label:  .IRPC arg,string
        .
        .
        (range of indefinite repeat)
        .
        .
        .ENDM
```

On each iteration of the indefinite repeat range, the dummy argument (arg) assumes the value of each successive character in the string. Terminators for the string are: space, comma, tab, carriage return, line feed, and semicolon.

Figure 5-6 is an example of .IRP and .IRPC usage.



```

1          .TITLE      IRPTST
2          .LIST      MD,MC,ME
3          R0=%0
4
5 000000 012700      MOV      #TABLE,R0
           000056'
6
7          .IRP      X,<A,B,C,D,E,F>
8
9          MOV      X,(R0)+
10
11         .ENDM
           00004 016720      MOV      A,(R0)+
           000032
           00010 016720      MOV      B,(R0)+
           000030
           00014 016720      MOV      C,(R0)+
           000026
           00020 016720      MOV      D,(R0)+
           000024
           00024 016720      MOV      E,(R0)+
           000022
           00030 016720      MOV      F,(R0)+
           000020
12
13         .IRPC     X,ABCDEF
14
15         .ASCII    /X/
16
17         .ENDM
           00034   101      .ASCII    /A/
           00035   102      .ASCII    /B/
           00036   103      .ASCII    /C/
           00037   104      .ASCII    /D/
           00040   105      .ASCII    /E/
           00041   106      .ASCII    /F/
18
19
20 00042 041101 A:      .WORD      "AB
21 00044 041502 B:      .WORD      "BC
22 00046 042103 C:      .WORD      "CD
23 00050 042504 D:      .WORD      "DE
24 00052 043105 E:      .WORD      "EF
25 00054 043506 F:      .WORD      "FG
26 00056          TABLE: .BLKW      6
27
28          000001      .END

```

Figure 5-6 .IRP and .IRPC Example

### 5.6.7 Repeat Block: .REPT

Occasionally it is useful to duplicate a block of code a number of times in line with other source code. (.REPT is not available in ASEMBL.) This is performed by creating a repeat block of the form:

```
label:  .REPT expr
        .
        .
        .
        (range of repeat block)
        .
        .
        .
        .ENDM      ;OR .ENDR
```

where: label is an optional statement label. The .ENDR or .ENDM directive may not have a label. A .REPT statement occurring within another repeat block, indefinite repeat block, or macro definition may not have a label associated with it.

expr is any legal expression controlling the number of times the block of code is assembled. Where  $\text{expr} \leq 0$ , the range of the repeat block is not assembled.

range is the block of code to be repeated expr number of times. The range may contain macro definitions, indefinite repeat ranges, or other repeat ranges. Note that no statements within a repeat range can have a label.

The last statement in a repeat block can be an .ENDM or .ENDR statement. The .ENDR statement is provided for compatibility with previous assemblers.

The .MEXIT statement is also legal within the range of a repeat block.

### 5.6.8 Macro Libraries: .MCALL

All macro definitions must occur prior to their referencing within the user program. MACRO provides a selection mechanism for the programmer to indicate in advance those system macro definitions required by his program.

The .MCALL directive is used to specify the names of all system macro definitions not defined in the current program but required by the program (not available in ASEMBL). The .MCALL directive must appear before the first occurrence of a macro call for an externally defined macro. The .MCALL directive is of the form:

```
.MCALL arg1,arg2,...
```

where arg1,arg2,... are the names of the macro definitions required in the current program.

When this directive is encountered, MACRO searches the system library file SYSMAC.SML, to find the requested definition(s). Macro searches for SYSMAC.SML on the system device.

See Appendix G for a listing of the system macro file (SYSMAC.SML) stored on the system device.

## 5.7 OPERATING PROCEDURES

The MACRO Assembler assembles one or more ASCII source files containing MACRO statements into a single relocatable binary object file. The output of the Assembler consists of a binary object file and an assembly listing followed by the symbol table listing.

MACRO is executed with the RT-11 Monitor R command as follows:

```
.R MACRO
```

The Assembler responds by typing an "\*" character to indicate readiness to accept a command input string.

In response to the \* printed by the Assembler, the user types the output file specification(s), followed by an equal sign or left angle bracket, followed by the input file specification(s):

```
*object,listing=source1,source2,...,sourcen
```

where:	object	is the binary object file.
	listing	is the assembly listing file containing the assembly listing and symbol table.
	source1,source2 ...,sourcen	are the ASCII source files containing the MACRO source program(s). A maximum of six source files is allowed.

A null specification in any of the file fields signifies that the associated input or output file is not desired. Each file specification contains the following information and follows the standard RT-11 conventions for file specifications:

```
dev:filnam.ext
```

The default value for each file specification is noted below:

	<u>dev</u>	<u>filnam</u>	<u>ext</u>
object	DK:		.OBJ
listing	device used for object output		.LST

	<u>dev</u>	<u>filnam</u>	<u>ext</u>
source1	DK:	-	.MAC
source2	device used	-	.MAC
.	for last source		
.	file specified		
.			
sourceN			
system macro file	system device, sy:	SYSMAC	.SML

## 5.8 ERROR MESSAGES

MACRO error messages enclosed in question marks are output on the terminal. The single letter error codes are printed in the assembly listing.

In terminal mode these error codes are printed following a field of six asterisk characters and on the line preceding the source line containing the error. For example:

```

*****A
26 00236 000002' .WORD REL1+REL2

```

<u>Error Code</u>	<u>Meaning</u>
A	Addressing error. An address within the instruction is incorrect. Also may indicate a relocation error. The addition of two relocatable symbols is flagged as an A error.
B	Bounding error. Instructions or word data are being assembled at an odd address in memory. The location counter is updated by +1.
D	Doubly-defined symbol referenced. Reference was made to a symbol which is defined more than once.
E	End directive not found. (A .END is generated.)
I	Illegal character detected. Illegal characters which are also non-printing are replaced by a ? on the listing. The character is then ignored.
L	Line buffer overflow, i.e., input line greater than 132 characters. Extra characters on a line, (more than 72 (10)) are ignored in terminal mode.
M	Multiple definition of a label. A label was encountered which was equivalent (in the first six characters) to a previously encountered label.

<u>Error Code</u>	<u>Meaning</u>
N	Number containing 8 or 9 has decimal point missing.
O	Opcod error. Directive out of context.
P	Phase error. A label's definition of value varies from one pass to another.
Q	Questionable syntax. There are missing arguments or the instruction scan was not completed or a carriage return was not immediately followed by a line feed or form feed.
R	Register-type error. An invalid use of or reference to a register has been made.
T	Truncation error. A number generated more than 16 bits of significance or an expression generated more than 8 bits of significance during the use of the .BYTE directive.
U	Undefined symbol. An undefined symbol was encountered during the evaluation of an expression. Relative to the expression, the undefined symbol is assigned a value of zero.
Z	Instruction which is not compatible among all members of the PDP-11 family (11/5, 11/20, 11/40, 11/45).

<u>ERROR MESSAGE</u>	<u>EXPLANATION</u>
?BAD SWITCH?	The switch specified was not recognized by the program.
?INSUFFICIENT CORE?	There are too many symbols in the program being assembled. Try dividing program into separately-assembled subprograms.
?NO INPUT FILE?	No input file was specified and there must be at least one input file.
?OUTPUT DEVICE FULL?	No room to continue writing output. Try to compress device with PIP.
?TOO MANY OUTPUT FILES?	Too many output files were specified.



## CHAPTER 6

### LINKER

#### 6.1 INTRODUCTION

The RT-11 Linker converts the object modules produced by the RT-11 Assembler into a format suitable for loading and execution. This allows the user to separately assemble a main program and each of the subroutines without assigning an absolute load address at assembly time. The object modules of the main program and subroutines are processed by the Linker to:

Relocate each object module and assign absolute addresses.

Link the modules by correlating global symbols defined in one module and referenced in another module.

Create the initial core control block for the linked program.

Create an overlay structure if specified and include the necessary runtime overlay handlers and tables.

Optionally produce a load map showing the layout of the load module.

RT-11 Linker requires two passes over the input modules. During the first pass it constructs the global symbol table, including all control section names and global symbols in the input modules. On the second pass, it reads the object modules, performs most of the functions listed in the preceding paragraph and produces a load module.

The Linker requires at least 8K of core and any additional core is used to extend the symbol table. Input is accepted from any binary device on the system; there must be at least one random access device (disk or DECTape) for save image output.

#### 6.2 ABSOLUTE AND RELOCATABLE PROGRAM SECTIONS

A program assembled by the RT-11 Assembler can consist of an absolute program section, declared by the .ASECT assembler directive, and relocatable program sections declared by the .CSECT assembler directive. The Assembler assumes a .CSECT directive at the beginning of the source program. The instructions and data in relocatable sections are normally assigned locations beginning at 1000(8). The assignment of addresses can be influenced by command string options (see Section 6.5.1) and the size of the absolute section (.ASECT, if present). Each control section is assigned a core address, then the Linker appropriately modifies all instructions and/or data as necessary to account for the relocation of the control sections.

The RT-11 Linker handles the absolute section as well as the named and unnamed control sections. The unnamed control section is internal to each object module. That is, every object module can have an unnamed control section but the Linker treats each one independently. Each is assigned an absolute address such that it occupies an exclusive area of memory. Named control sections, on the other hand, are treated globally. That is, if different object modules have control sections with the same name, they are all assigned the same absolute load address and the size of the area reserved for loading of the section is the size of the largest. Thus, named control sections allow for the sharing of data and/or instructions among object modules. This is similar to the handling and function of COMMON in FORTRAN IV. The names assigned to control sections are global and can be referenced as any other global symbol.

### 6.3 GLOBAL SYMBOLS

Global symbols provide the links, or communication, between object modules. With the RT-11 Assembler these symbols would be created with the .GLOBL assembler directive. Symbols which are not global are called internal symbols. If the global symbol is defined (as a label or by direct assignment) in an object module, it is called an entry symbol and other object modules can reference it. If the global symbol is not defined in the object module, it is an external symbol and is assumed to be defined (as an entry symbol) in some other object module.

As the Linker reads the object modules it keeps track of all global symbol definitions and references. It then modifies the instructions and/or data which reference the global symbols.

### 6.4 INPUT AND OUTPUT

#### 6.4.1 Object Module

Object files, consisting of one or more object modules, are the input to the Linker. The object modules are created by the RT-11 Assembler. The Linker reads each object module twice; that is, it is a two pass processor. During the first pass, each object module is read to construct a global symbol table, and assign absolute values to the control section names and global symbols.

On the second pass, the Linker reads the object modules, links and relocates the modules and outputs the load module.

#### 6.4.2 Load Module

The primary output of the Linker is a load module which may be loaded and run under RT-11. The load module is a save image file arranged as follows:



Root segment	Overlay segments (optional)
--------------	-----------------------------

The first 256-word block of the root segment (main program) contains the core usage map and the locations used by the Linker to pass user program control parameters. The core usage map outlines the blocks of core memory used by the load module and is located in locations 360 to 377. The control parameters are located in locations 40-57 and contain the following information when linking is completed.

<u>Address</u>	<u>Information</u>
40:	Start address of program.
42:	Initial setting of R6 stack pointer.
44:	Job Status Word.
50:	Highest core location in user's program.

These locations (40-57 and 360-377) are reserved for program parameters. Words at locations 46, 52, 54, 56 are also in this block but are not set by the Linker. Words in locations 40, 42, 44, 46, 50, 52 and 56 can be modified by the user by including an appropriate ASECT in his program. Words in locations 54 and 360-377 cannot be successfully set in an ASECT. They must be set at run time by MOV instructions.

The load module can also be output in LDA format which can then be loaded with the Absolute Loader. Refer to section 6.6.2.1 for a description of the command switch which produces LDA format output. If the LDA switch is specified, the control parameters are not set.

#### 6.4.3 Load Map

If requested, a load map is produced following the completion of Pass 1 of the Linker. This map, shown in Figure 6-1, provides the layout of core memory for the load module.

Each CSECT included in the linking process is listed in the load map. The entry for a CSECT includes the name and low address of the section and its size (in bytes). The remaining columns contain the entry points (or globals) found in the section and their addresses.

The modules located in the root segment of the load module are listed first; then those modules which were assigned to overlays in order by their region number. Any undefined global symbols are listed next. The map ends with the transfer address (start address) and high limit of relocatable code.

RT-11 LINK V01-01 LOAD MAP

SECTION	ADDR	SIZE	ENTRY	ADDR	ENTRY	ADDR	ENTRY	ADDR
. ABS.	000000	000400	LIMIT	000002	PDL	000004	PD SIZE	000006
			ARRAYS	000010	COLUMN	000034	FAC1	000040
			FAC2	000042	T1	000056	T2	000060
			T3	000062	RND1	000064	RND2	000066
			SSTKSZ	000200	.EOL	000200	.RPAR	000244
			.COMMA	000250	.LPAR	000262		
			FNTBL	000400	USRARE	000402	GO	000404
			START	000412	SCRATC	000430	CLEAR	000440
			READY	000444	READY0	000444	READY2	000534
			IGNORE	000560	EXECUT	000564	ERRFIL	000670
			ERRBUF	000702	ERRGO	000710	ALLOC	000776
			BLOCK	001250	ERRFNO	001462	BLOCKE	001470
			BOMB	001520	BOMBDO	001572	BUFGET	001710
			CHKCHR	001744	CHKISE	002044	CHKOSE	002052
			ERCHAN	002120	FRESET	002126	DIVTEN	002132
			DNPACK	002170	TABLES	002364	TBLSEN	002436
			ERRSYN	002440	FILEA	002446	FLINE	002474
			FNDSTL	002526	FREEGE	002606	GETCHA	002670
			INT16	003030	LINGET	003112	LITEVA	003206
			MPYTEN	003342	MSGERR	003372	MSG	003400
			MSGODE	003424	NORM	003456	NUMOUT	003650
			NUMSGN	003662	SAVCHA	004514	PUTCHA	004540
			ERRWLO	004622	DEVERR	004630	SAVREG	004646
			SKIPEO	004720	ERRTRN	005004	VAL	005020
			VFBLK	005360				
			IFPMP	005436	ERRFPU	005440	SSBR	005440
			SADR	005444	ALOG	006200	AINT	006554
			SINTR	006572	SDVR	006672	EXP	007324
			SIR	007674	SMLR	007760	SOPR4	010332
			SOPRS	010332	SPOPR3	010344	SRI	010352
			COS	010474	SIN	010530	ATAN	011050
			SPOLSH	011534	SV20A	011534	SQRT	011540
			SERR	011676	SERRA	011706	SERVEC	011726
			EXECX	011734	ASSIGN	012424	ARGB	016344
			ERRARG	016374	EVAL	016402	ERRPDL	016722
			OPRATO	017116	SOPRAT	017616	ERRMIX	017710
			STPRO	020060	SINFN	020120	COSFN	020126
			SQRFN	020134	ATNFN	020146	EXPFN	020154
			LOGFN	020166	INTFN	020340	VFFN	021644
			FNDSTR	022010	GETVAR	022116	INT	022270
			PUSH	022542	POP	022554	PUSH1	022566
			MAKEST	022772	STOVAR	023152		
			EDIT	025672	EXECE	030072	RESTJ	031260
			CLOSAL	031642	CLRVAR	031660	INITSC	032406
			NAMSET	033106				
			ONCEON	040652				

TRANSFER ADDRESS = 000404  
HIGH LIMIT = 041670

Figure 6-1 Linker Load Map

## 6.5 USING OVERLAYS

The RT-11 program overlay facility enables the user to have virtually unlimited core space for an assembly language program. A program using the overlay facility can be much larger than would normally fit in the available core space, since portions of the program, called overlay segments, reside on the backup storage (that is, disk or DECTape). These overlays are a part of the SAV image file from which the user's program is run and are brought into a core overlay region as needed. The RT-11 overlay scheme is a strict multi-region arrangement (not a tree structure).

The overlay system which the user constructs from his completed program is composed of a root segment which is never overlaid, a core resident overlay region, and overlay segments stored on backup storage. The root segment is a required part of every overlay program. The transfer address of every overlaid program must be in the root segment and not in an overlay.

An overlay region corresponds to a runtime area of core that is shared by two or more subroutines. The area in core is called an overlay region, and there is a distinct core area for each overlay region specified to the Linker.

The overlay structure is specified to the Linker with the /O switch in the command string as discussed in paragraph 6.6.1. The Linker creates the overlay regions and edits the program to produce the desired overlays at run time. Save image files are the only files which can have an overlay structure.

The size of an overlay region is calculated by the Linker to be the size of the largest group of subroutines that can occupy that region at one time. The group of subroutines which occupy the region at the same time are called co-resident overlay routines.

Figure 6-2 shows a diagram of core for a program which has an overlay structure.

There is no special code or function call needed to use overlays but the following rules must be observed when referencing parts of the user program which might be overlaid.

1. Calls or branches to overlay segments must be made directly to entry points in the segment. Entry points are locations tagged with a global symbol.

For example:

If ENTER is a global tag in an overlay segment,

JMP ENTER+6	is illegal,
JMP ENTER	is legal.

2. Entries in overlay segments can be used only for transfer of control and not for referencing data within an overlay section. (e.g. MOV ENTER,R4 is illegal if ENTER is in an overlay segment, but MOV #ENTER,R7 is legal because it is used for transfer of control). Violations of this rule cannot be detected by the Assembler or Linker so no error is issued. However, it can cause the program to unexpectedly use incorrect data.

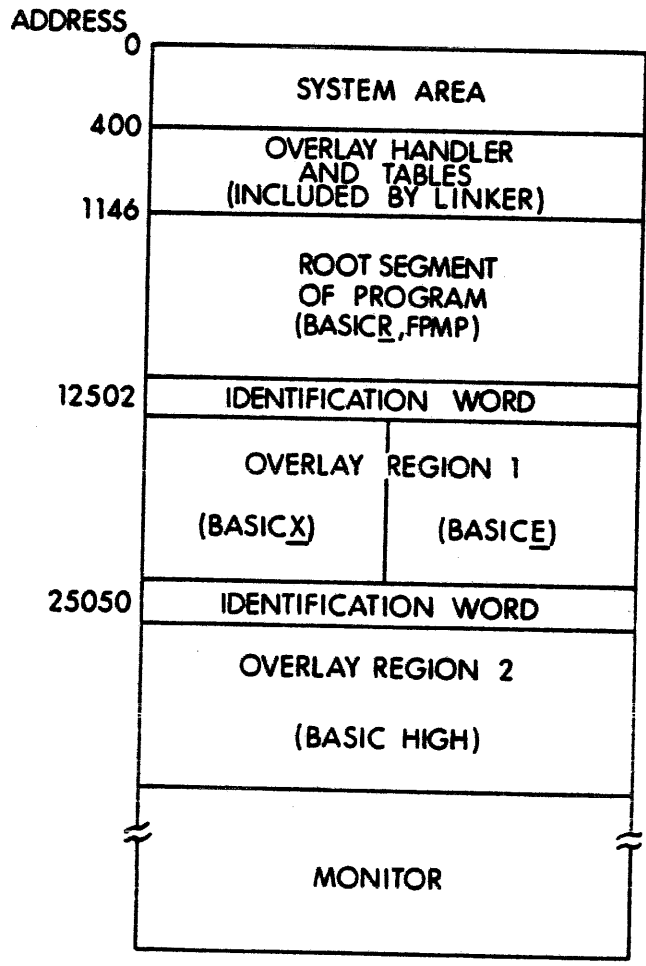


Figure 6-2 Core Diagram Showing Overlay Regions

3. When calls are made to overlays, the entire return path must be in core. This will happen if the following rules are followed:

Calls (with expected return) may be made from an overlay segment only to entries in the same segment, the root segment, or an overlay segment with a greater region number.

Calls to entries in the same region as the call must be entirely within the same segment, not another segment in the same region.

Jumps (with no expected return) can be made from an overlay segment to any entry in the program, except that jumps should not reference an overlay region whose number is lower than the region from which the last unreturned call was made (e.g., if a call was made from region 3, then no jumps should reference regions 1, 2 or 3 until the call has returned).

Subroutines in the root segment that are called from overlay segments may call entries in the same overlay segment from which they were called, the root segment, or an overlay segment with a greater region number. Such subroutines are considered part of the overlay segment which called them.

4. A .CSECT name cannot be used to pass control to an overlay. It will not cause the appropriate segment to be loaded into memory (e.g., JSR PC,OVSEC is illegal if OVSEC is used as a CSECT name in an overlay).
5. Channel 17(8) cannot be used by the user program because overlays are read on that channel.

The .ASECT never takes part in overlaying in any way; (i.e. if part of an ASECT is destroyed by overlay operations, it is not restored by the overlay handler.)

The above rules apply only to communications among the various modules that make up a program. Internally, each module must only observe standard programming rules for the PDP-11.

It should be noted that the condition codes set by a user program are not preserved across overlay segment boundaries.

The Linker provides overlay services by including a small resident overlay handler (Figure 6-3) in the same file with the user program to be used at program runtime. This overlay handler plus some tables will be inserted into the user's program beginning at the BOTTOM address computed by the Linker. The Linker moves the user's program up in core by an appropriate amount to make room for the overlay handler and tables, if necessary.

.SBTTL THE RUN-TIME OVERLAY HANDLER

; THE FOLLOWING CODE IS INCLUDED IN THE USER'S PROGRAM BY THE  
 ; LINKER WHENEVER OVERLAYS ARE REQUESTED BY THE USER.  
 ; 56.8 MICROSECONDS (APPROX) IS ADDED TO EACH REFERENCE OF  
 ; A RESIDENT OVERLAY SEGMENT.

; THE RUN-TIME OVERLAY HANDLER IS CALLED BY A DUMMY  
 ; SUBROUTINE OF THE FOLLOWING FORM:

```

;          JSR      R5,SOVRH      ;CALL TO COMMON CODE
;          .WORD   <OVERLAY #>   ;# OF DESIRED SEGMENT
;          .WORD   <ENTRY ADDR>   ;ACTUAL CORE ADDR
  
```

; ONE DUMMY ROUTINE OF THE ABOVE FORM IS STORED IN THE RESIDENT  
 ; PORTION OF THE USER'S PROGRAM FOR EACH ENTRY POINT TO  
 ; AN OVERLAY SEGMENT. ALL REFERENCES TO THE ENTRY POINT ARE  
 ; MODIFIED BY THE LINKER TO INSTEAD BE REFERENCES TO THE APPRO-  
 ; PRIATE DUMMY ROUTINE. EACH OVERLAY SEGMENT IS CALLED INTO  
 ; CORE AS A UNIT AND MUST BE CONTIGUOUS IN CORE. AN OVERLAY  
 ; SEGMENT MAY HAVE ANY NUMBER OF ENTRY POINTS, TO THE LIMITS  
 ; OF CORE MEMORY. ONLY ONE SEGMENT AT A TIME MAY OCCUPY AN  
 ; OVERLAY REGION.

; RESTRICTIONS:  
 ; SINCE REFERENCES TO OVERLAY SEGMENTS ARE AUTOMATICALLY TRANS-  
 ; LATED BY THE LINKER INTO REFERENCES TO DUMMY SUBROUTINES,  
 ; THE PROGRAMMER MUST NOT ATTEMPT TO REFERENCE DATA IN AN OVER-  
 ; LAY BY USING GLOBAL SYMBOLS.

```

SOVTAB=1000+SOVRHE-SOVRH
SOVRH:  MOV      R0,-(SP)
        MOV      R1,-(SP)
        MOV      R2,-(SP)
SOVRHB:
;      MOV      (R5)+,R0          ;PICK UP OVERLAY NUMBER
;      BR       $FIRST          ;FIRST CALL ONLY * * *
        MOV      R0,R1
SOVRHA: ADD      #SOVTAB-6,R1     ;CALC TABLE ADDR
        MOV      (R1)+,R2       ;GET CORE ADDR OF OVERLAY REGION
        CMP      R0,R2         ;IS OVERLAY ALREADY RESIDENT?
        BEQ      $ENTER        ;YES, BRANCH TO IT
        .READM  17,R2,(R1)+,(R1)+ ;READ FROM OVERLAY FILE
        BCS      $ERR
$ENTER: MOV      (SP)+,R2       ;RESTORE USER'S REGS
        MOV      (SP)+,R1
        MOV      (SP)+,R0
        MOV      @R5,R5        ;GET ENTRY ADDRESS
        RTS      R5           ;ENTER OVERLAY ROUTINE AND
;                                ;RESTORE USER'S R5

$FIRST: MOV      #12500,$OVRHB  ;RESTORE SWITCH INSTR
        MOV      (PC)+,R1       ;START ADDR FOR CLEAR OPERATION
$HROOT: .WORD   0              ;HIGH ADDR OF ROOT SEGMENT
        MOV      (PC)+,R2       ;COUNT
$HOVLY: .WORD   0              ;HIGH LIMIT OF OVERLAYS
1$:     CLR      (R1)+         ;CLEAR ALL OVERLAY REGIONS
        CMP      R1,R2
        BLO     1$
        BR      SOVRHB        ;AND RETURN TO CALL IN PROGRESS
$ERR:   IOT      10           ;SYSTEM ERROR 10 (OVERLAY I/O)
10
SOVRHE:
  
```

; OVERLAY SEGMENT TABLE FOLLOWS:  
 ; SOVTAB: .WORD <CORE ADDR>,<RELATIVE BLK>,<WORD COUNT>  
 ; THREE WORDS PER ENTRY, ONE ENTRY PER OVERLAY SEGMENT.

; ALSO, THERE IS ONE WORD PREFIXED TO EACH OVERLAY REGION  
 ; THAT IDENTIFIES THE SEGMENT CURRENTLY RESIDENT IN THAT REGION.  
 ; THIS WORD IS AN INDEX INTO THE SOVTAB TABLE.

Figure 6-3 The Run-Time Overlay Handler

## 6.6 OPERATING PROCEDURES

To call the Linker, use the command

R LINK

and the RETURN key in response to the Keyboard Monitor's dot. The Linker prints an asterisk and awaits a command string.

Type CTRL/C to halt the Linker at any time and return control to the Monitor. To restart the Linker type R LINK or the REENTER command in response to the Monitor's dot. The Linker performs an extra line feed operation when it is restarted with REENTER or after an error in the first command line. When the Linker is finished linking, control returns to the CSI automatically.

### 6.6.1 Command String

The first command string entered in response to the Linker's asterisk has the following format:

```
*dev:binout,dev:mapout<dev:obj1,dev:obj2,.../S1/S2/S3
```

where	dev:	is a random access device for the save image output file (binout) and any appropriate device in all other instances. If dev: is not specified, DK is assumed.
	binout	is the name to be assigned to the Linker's save image output file. This file is optional and if not specified, no output is produced.
	mapout	is the load map file and is optional.
	obj1,etc.	are files of one or more object modules to be input to the Linker.
	S1/S2/S3	are switches as explained in section 6.6.2.1.

If an output file is not specified, the Linker assumes that the associated output is not desired. For example if the load module and load map are not specified only the errors are printed by the Linker.

In the above command string, as in any command string, = can be substituted for <.

The default values for each specification are:

	<u>DEVICE</u>	<u>FILE NAME</u>	<u>EXTENSION</u>
Load Module	DK:	none	SAV
Map Output	Same as Load Module	none	MAP
Object Module	DK: or same as previous Object Module	none	OBJ

If a syntax error is made in a command string, an error message is printed. A new command string can then be typed following the asterisk (\*).

If a nonexistent file is specified (fatal error), the Linker returns to the \* and awaits a new command string.

If the /C switch is given, subsequent command lines may be entered as:

\*obj10,obj11,...,/S/S

The /C switch is necessary only if the command string will not fit on one line.

#### 6.6.1.1 Switches

The switches associated with the Linker are listed in Table 6-1. The letter representing each switch is always preceded by the slash character. Switches can be specified anywhere in a command string.

Table 6-1  
Linker Switches

Switch Name	Command Line	Meaning
/A	lst	Alphabetize the entries in the load map
/B	lst	Bottom address of program to be specified
/C	any	Continue input files on another command line
/L	lst	Produce output in LDA format.
/M	lst	Special purpose switch for linking the Monitor; do not use unless so directed by Digital documentation.
/O	any but the lst	Overlay-the program will be an overlay structure
/T	lst	Transfer address to be specified at terminal keyboard



## ALPHABETIZE SWITCH

The /A switch requests the Linker to list the linked modules in alphabetical order (by .CSECTs, then modules, and within modules in alphabetical order) on the load map.

The load map is normally arranged in order by module address as shown in Figure 6-1. Figure 6-4 is an example of an alphabetized load map.

## BOTTOM ADDRESS SWITCH

The /B switch specifies the lowest address to be used by the relocatable code in the load module. When /B is not specified, the Linker positions the load module so that the lowest address is location 1000 (octal). If the .ASECT length is greater than 1000, the length of .ASECT is used.

The form of the bottom switch is:

/B:n

Where n is a six digit unsigned octal number which defines the bottom address of the object program. An error message results if the bottom address (n) is not specified as part of the /B command.

If more than one B switch is specified during the creation of a load module, the first B switch specification is used.

### NOTE

The bottom value must be an unsigned even octal number. The Linker detects if the value is odd and outputs an error message.

Example:

\*OUTPUT,LP:=INPUT/B:500

Causes the file input to be linked starting at location 500 (octal).

RT-11 LINK V01-01 LOAD MAP

SECTION	ADDR	SIZE	ENTRY	ADDR	ENTRY	ADDR	ENTRY	ADDR		
. ABS.	000000	000400	ARRAYS	000010	COLUMN	000034	FAC1	000040		
			FAC2	000042	LIMIT	000002	PDL	000004		
			PDSIZE	000006	RND1	000064	RND2	000066		
			T1	000056	T2	000060	T3	000062		
			SSTKSZ	000200	.COMMA	000250	.EOL	000200		
			.LPAR	000262	.RPAR	000244				
			000400	005036	ALLOC	000776	BLOCK	001250	BLUCKE	001470
			BOMB	001520	BOMBDO	001572	BUFGET	001710		
			CHKCHR	001744	CHKISE	002044	CHKOSE	002052		
			CLEAR	000440	DEVERR	004630	DIVTEN	002132		
			DNPACK	002170	ERCHAN	002120	ERRBUF	000702		
			ERRFIL	000670	ERRFNO	001462	ERRGO	000710		
			ERRSYN	002440	ERRTRN	005004	ERRWLO	004622		
			EXECUT	000564	FILEA	002446	FLINE	002474		
			FNDSTL	002526	FNTBL	000400	FREEGE	002606		
			FRESET	002126	GETCHA	002670	GO	000404		
			IGNORE	000560	INT16	003030	LINGET	003112		
			LITEVA	003206	MPYTEN	003342	MSG	003400		
			MSGERR	003372	MSGODE	003424	NORM	003456		
			NUMOUT	003650	NUMSGN	003662	PUTCHA	004540		
			READY	000444	READY0	000444	READY2	000534		
			SAVCHA	004514	SAVREG	004646	SCRATC	000430		
			SKIPED	004720	START	000412	TABLES	002364		
			TBLSEN	002436	USRARE	000402	VAL	005020		
			VFBLK	005360						
			005436	004276	AINT	006554	ALOG	006200	ATAN	011050
			COS	010474	ERRFPU	005440	EXP	007324		
			IFPMP	005436	SIN	010530	SQRT	011540		
			SADR	005444	SDVR	006672	SERR	011676		
			SERRA	011706	SERVEC	011726	SINTR	006572		
			SIR	007674	SMLR	007760	SPOLSH	011534		
			SPOPR3	010344	SPOPR4	010332	SPOPRS	010332		
			SRI	010352	SSBR	005440	SV20A	011534		
			011734	012000	ARGB	016344	ASSIGN	012424	ATNFN	020146
			COSFN	020126	ERRARG	016374	ERRMIX	017710		
			ERRPDL	016722	EVAL	016402	EXECX	011734		
			EXPFN	020154	FNDSTR	022010	GETVAR	022116		
			INT	022270	INTFN	020340	LOGFN	020166		
			MAKEST	022772	UPRATO	017116	POP	022554		
			PUSH	022542	PUSH1	022566	SINFN	020120		
			SOPRAT	017616	SQRFN	020134	STUVAR	023152		
			STPRO	020060	VFFN	021644				
			023734	012344	CLOSL	031642	CLRVAR	031660	EDIT	025672
			EXECE	030072	INITSC	032406	NAMSET	033106		
			RESTJ	031260						
			036300	003370	ONCEON	040652				

TRANSFER ADDRESS = 000404  
HIGH LIMIT = 041670

Figure 6-4 Alphabetized Load Map

## CONTINUE SWITCH

The Continue switch (/C) is used to allow additional lines of command string input. The /C switch may be repeated on subsequent command lines as often as necessary to specify all input modules for which core is available. The last line of command string input is the only one that does not contain a /C switch. If core is exceeded, an error message is output.

The /C switch can also be repeated n times on a command line to cause the Linker to accept n more command lines.

### Examples:

```
*OUTPUT,LP:<INPUT/C      Input is to be continued on the next
*                          line, Linker prints an asterisk.

*OUTFIL,LP:<INFIL1/C/C
*INFIL2
*INFIL3                  Input is to be continued on the next
                          two lines, Linker prints the
                          asterisks.
```

## LDA FORMAT SWITCH

The LDA Format switch (/L) causes the output file to be in LDA format instead of save image format. The LDA output includes blocks of binary code preceded by the address at which they are to be stored. (Save image is a copy of core memory.) The LDA format file can be output to any device including non-file structured devices such as papertape or cassette (via PIPC) and is useful for files to be loaded with the Absolute Loader.

When the /L switch is specified, the .SAV extension of the first output file is changed to .LDA and the LDA format file is output to the specified device. No save image file is produced.

The /L switch cannot be used in conjunction with the overlay switch (/O).

### Example:

```
*PP:OUT,LP:<IN,IN2/L
```

Links disk files IN and IN2 and outputs an LDA format file OUT.LDA to the paper tape punch and a load map to the line printer.

## OVERLAY SWITCH

The Overlay switch (/O) is used to segment the load module so the entire program is not core resident at one time (overlay feature). This allows programs larger than the available core to be executed. The switch has the form

/O:n

Where n is a six digit unsigned octal number specifying the overlay region to which the module is assigned. (Refer to section 6.5.)

The /O switch must follow (on the same line) the specification of the object modules to which it applies, and only one overlay region can be specified on a command line. Overlay regions cannot be specified on the first command line (this is the root segment).

Co-resident overlay routines (a group of subroutines which occupy the overlay region at the same time) are specified as follows:

```
*OBJA,OBJB,OBJC/O:n/C
*OBJD,OBJE,...
.
.
.
```

All modules mentioned until the next "/O" switch will be co-resident overlay routines. If at a later time the "/O" switch is given with the same n value (same overlay region), then the corresponding overlay area is opened for a new group of subroutines. The new group of subroutines will occupy the same locations in core as the 1st group, but not at the same time. For example, if subroutines in object modules R and S are to be in core together, but are never needed at the same time as T, then the following commands to the Linker make R and S occupy the same core as T (but at different times):

```
*R,S/O:1/C
*T/O:1
```

The above could also be written as:

```
*R/O:1/C
*S/C
*T/O:1
```

Example:

```
*OUTPUT,LP:<INPUT/C
*OBJA/O:1/C
*OBJB/O:2
```

establishes two overlay regions.

#### TRANSFER ADDRESS SWITCH

The Transfer switch (/T) allows terminal keyboard specification of the start address of the load module to be executed. This switch has the form:

/T:n

where n is a six digit unsigned octal number which defines the transfer address, or

/T

which causes the Linker to print the message:

**TRANSFER ADDRESS:**

Specify the global symbol whose value is the transfer address of the load module followed by a carriage return. A number cannot be specified in answer to this message. When a nonexistent symbol is specified, an error message is printed and the transfer address is set to 1.

If the transfer address specified is odd, the program will not start after loading and control returns to the monitor.

Direct assignment (.ASECT) of the transfer address within the program takes precedence over assignment with the /T switch and the transfer address assigned with a /T has precedence over that assigned with a .END.

Example:

```
*PROG=PROG1,PROG2,ODT/T
TRANSFER ADDRESS:
O.ODT
```

## 6.7 ERROR HANDLING AND MESSAGES

The following error messages can be output by the Linker.

The messages enclosed in question marks are output to the terminal; the other messages are included in the load map. If a load map is not requested in the command string, all messages are output to the terminal.

When an I/O error occurs on a channel while overlays are being used, the Linker, while not outputting an immediate message, sets an error condition for output of a message at runtime.

<u>Message</u>	<u>Meaning</u>
ADDITIVE REF OF xxxxxx	Rule 1 of overlay rules explained in section 6.5 has been violated.
BAD OVERLAY	Check for a .ASECT in overlay.
BYTE RELOCATION ERROR AT xxxxxx	Linker attempted to relocate and link byte quantities but failed. Failure is defined as the high byte of the relocated value (or the linked value) not being all zero. In such a case, the value is truncated to 8 bits and the Linker continues processing.
INVALID OVERLAY REFERENCE	Reference of a global that is in an overlay violates one of the overlay rules.

<u>Message</u>	<u>Meaning</u>
MULT DEF OF xxxxxx	The symbol, xxxxxx, was defined more than once.
TRANSFER ADDRESS UNDEFINED OR IN OVERLAY	The transfer address was not defined or is in an overlay.
UNDEFINED GLOBALS	This condition also causes the warning message, UNDEF GLBLS, on the terminal.
xxxxxx	
xxxxxx	
.	
.	
.	
?/B NO VALUE?	The /B switch requires an octal number as an argument.
?/B ODD VALUE?	The argument to the /B switch was not an unsigned even octal number.
?BAD GSD?	There is an error in the global symbol directory (GSD). The file is probably not a legal object module. This error message occurs on pass 1 of the Linker.
?BAD RLD?	There is an invalid RLD command in the input file; the file is probably not a legal object module. The message occurs on pass 2 of the Linker.
?BAD SWITCH?	LINK did not recognize a switch specified on the first command line. On a subsequent command line, a bad switch causes this warning message but does not restart the Linker.
?CORE?	The command string is too complicated; not enough core to accommodate the command or the resultant load module.
?ERROR ERROR?	An error occurred while the Linker was producing an error message.
?ERROR IN FETCH?	The device is not available.
?HARD I/O ERROR?	Hardware error occurred. Try again.
?LDA FILE ERROR?	There was a hardware problem with the device specified for LDA output or the device is full.
?MAP FILE ERROR?	There was a hardware problem with the device specified for map output or the device is full.
?NO INPUT?	No input files were specified.

<u>Message</u>	<u>Meaning</u>
?OUTPUT FULL?	The output device is full.
?SYMBOL TABLE OVERFLOW?	There were too many global symbols used in the program.
?SAV FILE ERR?	The Linker had a problem writing the save image file; try again.
?TOO MANY OUTPUT FILES?	The Linker allows specification of two output files.





## CHAPTER 7

### ODT

RT-11 ODT (On-line Debugging Technique) is a system program which aids in debugging assembled and linked object programs. From the keyboard you interact with ODT and the object program to:

Print the contents of any location for examination or alteration.

Run all or any portion of your object program using the breakpoint feature.

Search the object program for specific bit patterns.

Search the object program for words which reference a specific word.

Calculate offsets for relative addresses.

Fill a block of words or bytes with a designated value.

The breakpoint is one of ODT's most useful features. When debugging a program, it is often desirable to allow the program to run normally up to a predetermined point, at which time the contents of various registers or locations can be examined and possibly modified. To accomplish this, ODT acts as a monitor to the user program.

The assembly listing of the program to be debugged should be readily available when ODT is being used. Minor corrections to the program may be made on-line during the debugging session. The program may then be run under control of ODT to verify any change made. Major corrections, however, such as a missing subroutine should be noted on the assembly listing and incorporated in a subsequent updated program assembly.

#### 7.1 RELOCATION

When the relocatable assembler produces a binary relocatable object module, the base address of the module is taken to be location 000000, and the addresses of all program locations as shown in the assembly listing are indicated relative to this base address. After the module is linked by the Linker, many values within the program, and all the addresses of locations in the program, will be incremented by a constant whose value is the actual absolute base address of the module after it has been relocated. This constant is called the relocation bias for the module. Since a linked program may contain several relocated modules, each with its own relocation bias, and since, in the process of debugging, these biases will have to be subtracted from absolute addresses continually in order to relate relocated code to assembly listings, RT-11 ODT provides an automatic relocation facility.

The basis of the relocation facility lies in 8 relocation registers, numbered 0 through 7, which may be set to the values of the relocation biases at different times during debugging. Relocation biases should be obtained by consulting the memory map produced by the Linker. Once set, a relocation register is used by ODT to relate relocatable code to relocated code. For more information on the exact nature of the relocation process, consult the Chapter on the RT-11 Linker.

## 7.2 RELOCATABLE EXPRESSIONS

The symbol *n* below stands for an integer in the range 0 to 7 inclusive.

The symbol *k* stands for an octal number up to six digits long, with a maximum value of 177777. If more than six digits are typed, ODT takes the last six digits, truncated to the low-order 16 bits. *k* may be preceded by a minus sign, in which case its value is the two's complement of the number typed. For example:

<u>k (number typed)</u>	<u>Values</u>
1	000001
-1	177777
400	000400
-177730	000050
1234567	034567

The symbol *r* is called a relocatable expression and is evaluated by ODT as a 16-bit (6 octal digit) number. It may be typed in any one of three forms:

<u>Form A</u>	<i>k</i>	The value <i>r</i> is simply the value of <i>k</i> .
<u>Form B</u>	<i>n,k</i>	The value of <i>r</i> is the value of <i>k</i> plus the contents of relocation register <i>n</i> . If the <i>n</i> part of this expression is greater than 7, ODT takes only the last octal digit of <i>n</i> .
<u>Form C</u>	<i>C</i> or <i>C,k</i> or <i>n,C</i> or <i>C,C</i>	Whenever the letter <i>C</i> is typed, ODT replaces <i>C</i> with the contents of a special register called the Constant Register. This value has the same role as the <i>k</i> or <i>n</i> that it replaces (i.e., when used in place of <i>n</i> it designates a relocation register). The Constant Register is designated by the symbol <i>\$C</i> and may be set to any value, as indicated below.

In the following examples, assume that relocation register 3 contains 003400 and that the constant register contains 000003.

<u>r</u>	<u>Value of r</u>
5	000005
-17	177761
3,0	003400
3,150	003550
3,-1	003377
C	000003
3,C	003403
C,0	003400
C,10	003410
C,C	003403

NOTE

For simplicity most examples in this section use Form A. All three forms of r are equally acceptable, however.

7.3 COMMAND SUMMARY

ODT's commands are composed of the characters and symbols shown in Table 7-1. They are often used in combination with the address upon which the operation is to occur. These commands will be discussed in detail in paragraph 7.4 but there are some terms which should be defined before the command summary is read.

An open location is one whose contents ODT has printed for examination, making those contents available for change. A closed location is one whose contents are no longer available for change.

In Table 7-1 unless indicated otherwise: r represents a relocatable expression as described in paragraph 7.2; n represents an octal number. In ODT commands a semicolon (;) separates commands from command arguments (used with alphabetic commands; or separates a relocation register specifier from an addend).

If the command entered cannot be handled by ODT, an ? is displayed on the terminal.

Table 7-1

ODT Commands

Format	Meaning
RETURN	Close open location and accept the next command.
LINE FEED	Close current location; open next sequential location.
↑	Open previous location. The circumflex, ^, appears on some keyboards and prints in place of the up-arrow.

Format	Meaning
+	Take contents of opened location, index by contents of PC, and open that location. The underline, <u>  </u> , appears on some keyboards and prints in place of the back-arrow.
@	Take contents of opened location as absolute address and open that location.
>	Take contents of opened location as relative branch instruction and open referenced location.
<	Return to sequence prior to last @, >, or + command and open succeeding location.
r/	Open the word at location r.
/	Reopen the last opened location.
r\ \	Open the byte at location r.
\	Reopen the last opened byte.
n!	After a word or byte has been opened, print the address of the opened location relative to relocation register n. If n is omitted, ODT selects the relocation register whose contents are closest, but less than or equal to the address of the opened location.
\$n/	Open general register n (0-7).
\$y/	Open special register y, where y may be one of the following letters:  S    Status register (saved by ODT after a breakpoint) M    Mask register B    First word of the breakpoint table P    Priority register C    Constant register R    First relocation register (register 0) F    Format register
r;nA	Print n bytes in their ASCII format, starting at location r; then allow n bytes to be typed in, starting at location r.
;B	Remove all Breakpoints.
r;B	Set Breakpoint at location r

Format	Meaning
r;nB	Set Breakpoint n at location r
;nB	remove nth Breakpoint.
r;C	Print the value of r and store it in the constant register.
r;E	Search for instructions that reference effective address r.
;F	Fill memory words with the contents of the constant register.
r;G	Go to location r and start program.
;I	Fill memory bytes with the contents of the low-order 8 bits of the constant register.
r;O	Calculate offset from currently open location to r.
;P	<p>Proceed with program execution from breakpoint; stop when next breakpoint is encountered or at end of program.</p> <p>In single-instruction mode only, execute next instruction.</p>
k;P	<p>Proceed with program execution from breakpoint; stop after encountering the breakpoint k times.</p> <p>In single-instruction mode only, execute next k instructions.</p>
;R	Set all relocation registers to -1 (highest address value).
;nR	Set relocation register n to -1.
r;nR	Set relocation register n to the value of r. If n is omitted, it is assumed to be 0.
nR	After a word has been opened, retype the contents of the word relative to relocation register n--i.e., subtract contents of relocation register n from the contents of the opened word and print the result. If n is omitted, ODT selects the relocation register whose contents are closest but less than or equal to the contents of the opened location.
;nS	Enable single-instruction mode (n can have any value and is not significant); disable breakpoints.

Format	Meaning
;S	Disable single-instruction mode; reenale breakpoints.
r;W	Search for words with bit patterns which match r.
X	Perform a Radix 50 unpack of the binary contents of the current opened word; then permit the storage of a new Radix 50 binary number in the same location.

#### 7.4 COMMANDS AND FUNCTIONS

When ODT is started as explained in paragraph 7.7 it indicates its readiness to accept commands by printing an asterisk on the left margin of the terminal paper. Most of the ODT commands can be issued in response to the asterisk; for example, a word can be examined and, if desired, changed, the object program can be run in its entirety or in segments, or core can be searched for certain words or references to certain words. The discussion below explains these features.

All commands to ODT are stated using the characters and symbols shown in sections 7.2 and 7.3. In the following examples, characters output by ODT are underlined to differentiate them from user input.

##### 7.4.1 Printout Formats

Normally, when ODT prints addresses (as with the commands  $\downarrow$ ,  $\uparrow$ ,  $+$ ,  $@$ ,  $<$ , and  $>$ ) it attempts to print them in relative form (Form B in section 7.2). ODT looks for the relocation register whose value is closest but less than or equal to the address to be printed, and then represents the address relative to the contents of the relocation register. However, if no relocation register fits the requirement, the address is printed in absolute form. Since the relocation registers are initialized to -1 (the highest number) the addresses are initially printed in absolute form. If any relocation register subsequently has its contents changed, it may then, depending on the command, qualify for relative form.

For example, suppose relocation registers 1 and 2 contained 1000 and 1004 respectively, and all other relocation registers contained numbers much higher. Then the following sequence might occur:

```

*774/000000  $\downarrow$ 
000776/000000  $\downarrow$ 
1,000000 /000000  $\downarrow$  (absolute location 1000)
1,000002 /000000  $\downarrow$  (absolute location 1002)
2,000000 /000000 (absolute location 1004)

```

The format is controlled by the format register, \$F. Normally this register contains 0, in which case ODT prints addresses relatively whenever possible. \$F may be opened and changed to a non-zero value, however, in which case all addresses will be printed in absolute form (see paragraph on Accessing Internal Registers).

#### 7.4.2 Opening, Changing and Closing Locations

The contents of an open location may be changed by typing the new contents followed by a single character command which requires no argument (i.e., +, ↑, RETURN, +, @, >, <). Any command typed to open a location when another location is already open, first causes the currently open location to be closed.

##### The Slash, /

One way to open a location is to type its address followed by a slash:

```
*1000/012746
```

Location 1000 is open for examination and is available for change.

If the contents of an open location are not to be changed, type the RETURN key and the location is closed; ODT prints another asterisk and waits for another command. However, to change the word, simply type the new contents before giving a command to close the location:

```
*1000/012746 012345  
*
```

In the example above, location 1000 now contains 012345 and is closed since the RETURN key was typed after entering the new contents, as indicated by ODT's second asterisk. Used alone, the slash reopens the last location opened:

```
*1000/012345 2340  
*/002340
```

In the example above, the open location was closed by typing the RETURN key. ODT changed the contents of location 1000 to 002340 and then closed the location before printing the \*. The single slash command directed ODT to reopen the last location opened. This allowed verification that the word 002340 was correctly stored in location 1000.

Note again, that opening a location while another is open automatically closes the currently open location before opening the new location.

Also note that if an odd numbered address is specified with a slash, ODT opens the location as a byte, and subsequently behaves as if a backslash had been typed (see the following paragraph).

##### The Backslash, \

In addition to operating on words, ODT operates on bytes. One way to open a byte is to type the address of the byte followed by a backslash. (On the LT33 terminal \ is printed by depressing the SHIFT key while typing the L key). This causes not only the printing of the byte value at the specified address but also the interpreting of the value as ASCII code, and the printing of the corresponding character (if possible) on the terminal.

\*1001\101=A

A backslash typed alone reopens the last open byte. If a word was previously open, the backslash reopens its even byte.

\*1002/000005\005=

The LINE FEED and up-arrow (or circumflex) keys operate on bytes if a byte is open when the command is given (see LINE FEED and up-arrow). For example:

\*1001\101=A+  
001002\004=+  
001003\102=B+  
\*

#### The LINE FEED Key, +

If the LINE FEED key is typed when a location is open, ODT closes the open location and opens the next sequential location:

\*1000/002340+           (+ denotes typing the LINE FEED key)  
001002/012740

In this example, the LINE FEED key caused ODT to print the address of the next location along with its contents, and to wait for further instructions. After the above operation, location 1000 is closed and 1002 is open. The open location may be modified by typing the new contents.

If a byte location was open, typing the LINE FEED key opens the next byte location.

#### The Up-Arrow, ↑

If the up-arrow (or circumflex) is typed when a location is open, ODT closes the open location and opens the previous location (as shown by continuing from the example above):

001002/012740↑           (↑ is printed by typing the SHIFT and N  
001000/002340           keys combined).

Now location 1002 is closed and 1000 is open. The open location may be modified by typing the new contents.

If the opened location was a byte, then up-arrow opens a byte as well.

#### The Back-Arrow, ←

If the back-arrow (or underline) is typed to an open word, ODT interprets the contents of the currently open word as an address indexed by the Program Counter (PC) and opens the location so addressed:

\*1006/000006←           (On the LT33 terminal ← is printed by  
001016/100405           typing the SHIFT and the O keys  
together).



Notice in this example that the open location, 1006, was indexed by the PC as if it were the operand of an instruction with address mode 67 as explained in the Assembler Chapter.

A modification to the opened location can be made before a LINE FEED, up-arrow, or back-arrow is typed. Also, the new contents of the location will be used for address calculations using the back-arrow command. Example:

```
*100/000222 4+ (modify to 4 and open next location)
000102/000111 6↑ (modify to 6 and open previous location)
000100/000004 100← (change to 100 and open location indexed
000202/123456 by PC)
```

#### Open the Addressed Location, @

The at symbol @ (SHIFT/P on the LT33 terminal) will optionally modify, then close an open word, and use its contents as the address of the location to open next.

```
*1006/001024 @ (open location 1024 next)
001024/000500
*1006/001024 2100 @ (modify to 2100 and open location 2100)
002100/177774
```

#### Relative Branch Offset, >

The right angle bracket, >, will optionally modify, then close an open word, and use its low-order byte as a relative branch offset to the next word to be opened.

```
*1032/000407 301> (modify to 301 and interpret as a
000636/000010 relative branch)
```

#### Return to Previous Sequence, <

The left-angle bracket, <, will optionally modify, then close an open location, and open the next location of the previous sequence interrupted by a back-arrow, @, or right-angle bracket command. Note that back-arrow, @, or right-angle bracket causes a sequence change to the word opened. If a sequence change has not occurred, the left-angle bracket simply opens the next location as a LINE FEED does. This command operates on both words and bytes.

```
*1032/000407 301> (> causes a sequence change)
000636/000010 < (< causes a return to original
sequence)
001034/001040 @ (@ causes a sequence change)
001040/000405\005=< (< now operates on byte)
001035\002=< (< acts like +)
001036\004=
```

### Accessing General Registers 0-7

The program's general registers 0-7 are opened with a command in the following format:

\*\$n/

where n is the integer representing the desired register (in the range 0 through 7). When opened, these registers can be examined or changed by typing in new data as with any addressable location. For example:

\*\$0/000033 (R0 was examined and closed)

\*

\*\$4/000474 464 (R4 was opened, changed, and closed)

\*

The example above can be verified by typing a slash in response to ODT's asterisk:

\*/000464

The LINE FEED, up-arrow, backarrow or @ commands may be used when a register is open.

### Accessing Internal Registers

The program's Status Register contains the condition codes of the most recent operational results and the interrupt priority level of the object program. It is opened using the following command:

\*\$S/000311

where \$S represents the address of the Status Register. In response to \$S in the example above, ODT printed the 16-bit word of which only the low-order 8 bits are meaningful: Bits 0-3 indicate whether a carry, overflow, zero, or negative (in that order) has resulted, and bits 5-7 indicate the interrupt priority level (in the range 0-7) of the object program. (Refer to the PDP-11 Processor Handbook for the Status Register format.)

The \$ is used to open certain other internal locations:

- \$B location of the first word of the breakpoint table (see Section 7.4.3).
- \$M mask location for specifying which bits are to be examined during a bit pattern search (see paragraph 7.4.6).
- \$P location defining the operating priority of ODT (see paragraph 7.4.12).
- \$S location containing the condition codes (bits 0-3) and interrupt priority level (bits 5-7).
- \$C location of the Constant Register (see paragraph 7.4.7).

\$R location of Relocation Register 0, the base of the Relocation Register table (see paragraph 7.4.10).

\$F location of Format Register (see paragraph 7.4.1).

### Radix 50 Mode, X

The Radix 50 mode of packing certain ASCII characters three to a word is employed by many DEC-supplied PDP-11 system programs, and may be employed by any programmer via the Assembler's ".RAD50" directive.

ODT provides a method for examining and changing memory words packed in this way with the "X" command.

When a word is opened, and the X command is typed, ODT converts the contents of the opened word to its 3-character Radix 50 equivalent, and prints these characters on the terminal.

One of the following can then be typed:

<u>Type</u>	<u>Effect</u>
RETURN key	closes the currently open location
LINE FEED key	closes the location and opens the next one in sequence.
↑ key	closes the location and opens the previous one in sequence
Any three characters whose octal code is 040 (space) or greater.	converts the three specified characters into packed Radix 50 format.

Legal Radix 50 characters are:

\$ Space 0 through 9 A through Z

If any other characters are typed, the resulting binary number is unspecified. However, exactly three characters must be typed before ODT resumes its normal mode of operation.

After the third character is typed, the resulting binary number is available to be stored into the opened location by closing the location in any one of the usual ways (RETURN key, LINE FEED key, etc.). Example:

```
*1000/042431 X=KBI CBA  
*1000/011421 X=CBA
```

#### NOTE

After ODT has converted the three characters to binary, the binary number can be interpreted in

one of many different ways, depending on the command which follows. For example:

```
*1234/063337 X=PRO XIT/
```

Since the Radix 50 equivalent of XIT is 113574, the final slash in the example will cause ODT to open location 113574 if it is a legal address. (Refer to paragraph 7.5 for a discussion of command legality and detection of errors.)

### 7.4.3 Breakpoints

The breakpoint feature facilitates monitoring the progress of program execution. A breakpoint may be set at any instruction which is not referenced by the program for data. When a breakpoint is set, ODT replaces the contents of the breakpoint location with a trap instruction so that program execution is suspended when breakpoint is encountered. The original contents of the breakpoint location are restored, and ODT regains control.

With ODT, up to eight breakpoints, numbered 0 through 7, can be set at any one time. The r;B command sets the next available breakpoint. Specific breakpoints may be set or changed by the r;nB command where n is the number of the breakpoint. For example:

```
*1020;B      (sets breakpoint 0)
*1030;B    (sets breakpoint 1)
*1040;B      (sets breakpoint 2)
*1032;1B     (resets breakpoint 1)
*
```

The ;B command removes all breakpoints. Use the ;nB command to remove only one of the breakpoints, where n is the number of the breakpoint. For example:

```
*;2B        (removes the second breakpoint)
*
```

The \$B/ command opens the location containing the address of breakpoint 0. The next seven locations contain the addresses of the other breakpoints in order, and thus can be opened using the LINE FEED key. (The next location is for single-instruction mode, explained in Section 7.4.5). Example:

```
*$B/001020 +
nnnnnn/001032 +
nnnnnn/nnnnnn      (nnnnnn=address internal to ODT)
```

In this example, breakpoint 2 is not set. The contents printed is an address internal to ODT. Following the table of breakpoints is the table of Proceed command repeat counts, first for each breakpoint, and then for the single instruction mode (refer to section 7.4.5).

```

.
.
.
↓
nnnnnn/001036+ (breakpoint 7)
nnnnnn/nnnnnn+ (single-instruction address)
nnnnnn/000000 15+ (count for breakpoint 0)
nnnnnn/000000 (count for breakpoint 1)

```

It should be noted that a repeat count in a Proceed command refers only to the breakpoint that has most recently occurred. Execution of other breakpoints encountered is determined by their own repeat counts.

#### 7.4.4 Running the Program, r;G and k;P

Program execution is under control of ODT. There are two commands for running the program: r;G and k;P. The r;G command is used to start execution (Go) and k;P to continue (Proceed) execution after having halted at a breakpoint. For example:

```
*1000;G
```

starts execution at location 1000. The program runs until a breakpoint is encountered or until program completion, unless it gets caught in an infinite loop, where it must be either restarted or reentered as explained in paragraph 7.7.

When a breakpoint is encountered, execution stops and ODT prints Bn; (where n is the breakpoint number), followed by the address of the breakpoint. Locations can then be examined for expected data. For example:

```

*1010;3B (breakpoint 3 is set at location 1010)
*1000;G (execution started at location 1000)
B3;001010 (execution stopped at location 1010)
*

```

To continue program execution from the breakpoint, type ;P in response to ODT's last \*.

When a breakpoint is set in a loop, it may be desirable to allow the program to execute a certain number of times through the loop before recognizing the breakpoint. This can be done by typing the k;P command and specifying the number of times the breakpoint is to be encountered before program execution is suspended (on the kth encounter).

The count, k, is associated only with the numbered breakpoint which most recently occurred. A different proceed count may be associated with each numbered breakpoint, and applies to that breakpoint only.

Example:

```

B3;001010 (execution halted at breakpoint)
*1250;5B (set breakpoint 5 at location 1250)
*7;P (continue execution, loop through
B5;001250 breakpoint 3 six times and halt on seventh
* occurrence of the breakpoint)
-

```

The breakpoint repeat counts can be inspected by typing \$B/ and following that with the typing of nine LINE FEED's. The repeat count for breakpoint 0 is printed. The repeat counts for breakpoints 1 through 7, and the repeat count for the single instruction trap follow in sequence (see Section 7.4.3). Opening any one of these provides an alternative way of specifying the count. The location, being open, can have its contents modified in the usual manner by the typing of new contents and then the RETURN key.

Breakpoints are inserted when performing an r;G or k;P command. Upon execution of the r;G or k;P command, the general registers 0-6 are set to the values in the locations specified as \$0-\$6 and the processor status register is set to the value in the location specified as \$S.

#### 7.4.5 Single-Instruction Mode

With this mode the number of instructions to be executed before suspension of the program run can be specified. The Proceed command, instead of specifying a repeat count for a breakpoint encounter, specifies the number of succeeding instructions to be executed. Note that breakpoints are disabled when single-instruction mode is operative.

Commands for single-instruction mode are:

;nS	Enables single-instruction mode (n can have any non-zero value and serves only to distinguish this form from the form ;S). Breakpoints are disabled.
k;P	Proceeds with program run for next k instructions before reentering ODT (if k is missing, it is assumed to be 1). (Trap instructions and associated handlers can affect the Proceed repeat count. See paragraph 7.6.2).
;S	Disables single-instruction mode.

When the repeat count for single-instruction mode is exhausted and the program suspends execution, ODT prints:

B8:k  
\*

where k is the address of the next instruction to be executed. The \$B breakpoint table contains this address following that of breakpoint 7. However, unlike the table entries for breakpoints 0-7, direct modification has no effect.

Similarly, following the repeat count for breakpoint 7 is the repeat count for single-instruction mode. This table entry, however, can be directly modified, and thus is an alternative way of setting the single-instruction mode repeat count. In such a case, ;P implies the argument set in the \$B repeat count table rather than 1.

#### 7.4.6 Searches

With ODT all or any specified portion of core memory can be searched for any specific bit pattern or for references to a specific location.

##### Word Search, r;W

Before initiating a word search, the mask and search limits must be specified. The location represented by \$M is used to specify the mask of the search. \$M/ opens the mask register. The next two sequential locations (opened by LINE FEED's) contain the lower and upper limits of the search. Bits set to 1 in the mask are examined during the search; other bits are ignored. Then the search object and the initiating command are given using the r;W command where r is the search object. When a match is found, i.e., each bit set to 1 in the search object is set to 1 in the word being searched (over the mask range) the matching word is printed. For example:

```
*$M/000000 177400↓      (test high order eight bits)
nnnnnn/000000 1000↓      (set low address limit)
nnnnnn/000000 1040      (set high address limit)
*400;W                    (initiate word search)
001010/000770
001034/000404
*
```

In the above example, nnnnnn is an address internal to ODT.

In the search process an exclusive OR (XOR) is performed with the word currently being examined and the search object, and the result is ANDed to the mask. If this result is zero, a match has been found, and is reported on the terminal. Note that if the mask is zero, all locations within the limits are printed.

Typing CTRL/U during a search printout terminates the search.

##### Effective Address Search, r;E

ODT provides for a search for words which address a specified location. Open the mask register only to gain access to the low- and high-limit registers. After specifying the search limits (as explained for the word search), type the command r;E (where r is the effective address) to initiate the search.

Words which are either an absolute address (argument r itself), a relative address offset, or a relative branch to the effective address are printed after their addresses. For example:

```
*$M/177400 ↓      (open mask register only to gain
nnnnnn/001000 1010 ↓  access to search limits)
nnnnnn/001040 1060
*1034;E          (initiating search)
001016/001006    (relative branch)
001054/002767    (relative branch)
*1020;E          (initiating a new search)
001022/177774    (relative address offset)
001030/001020    (absolute address)
*
```

Particular attention should be given to the reported references to the effective address because a word may have the specified bit pattern of an effective address without actually being so used. ODT reports all possible references whether they are actually used as such or not.

Typing CTRL/U during a search printout terminates the search.

#### 7.4.7 The Constant Register, r;C

It is often desirable to convert a relocatable address into a relocated address or to convert a number into its two's complement, and then to store the converted value into one or more places in a program. The Constant Register provides a means of accomplishing this and other useful functions.

When r;C is typed, the relocatable expression r is evaluated to its six digit octal value and is both printed on the terminal and stored in the Constant Register. The contents of the Constant Register may be invoked in subsequent relocatable expressions by typing the letter C.

Examples:

<u>*-4432;C=173346</u>	(The two's complement of 4432 is placed in the Constant Register)
*1000/ <u>001000</u> C	(the contents of the Constant Register are stored in location 1000)
*1000;1R	(Relocation Register 1 is set to 1000)
<u>*1,4272;C=005272</u>	(Relative location 4272 is reprinted as an absolute location and stored in the Constant Register)

#### 7.4.8 Core Block Initialization, ;F and ;I

The Constant Register can be used in conjunction with the commands ;F and ;I to set a block of memory to a given value. While the most common value required is zero, other possibilities are plus one, minus one, ASCII space, etc.

When the command ;F is typed, ODT stores the contents of the Constant Register in successive memory words starting at the memory word address specified in the lower search limit, and ending with the address specified in the upper search limit.

When the command ;I is typed, the low-order 8 bits in the Constant Register are stored in successive bytes of memory starting at the byte address specified in the lower search limit and ending with the byte address specified in the upper search limit.

Example:

Assume relocation register 1 contains 1000, 2 contains 2000, and 3 contains 3000. The following sequence sets word locations 1000-1776 to zero, and byte locations 2000-2777 to ASCII spaces.



```

*$M/000000 +           (open mask register to gain access
                           to search limits)
nnnnnn/000000 1,0 +     (sets lower limit to 1000)
nnnnnn/000000 2,-2     (sets upper limit to 1776)
*0;C=000000           (Constant Register set to zero)
*;F                   (Locations 1000-1776 set to zero)
*$M/000000 +
nnnnnn/001000 2,0 +     (Sets lower limit to 2000)
nnnnnn/001776 3,-1     (Sets upper limit to 2777)
*40;C=000040         (Constant Register set to 40
                           (SPACE))
*;I                   (Byte locations 2000-2777 are set
                           to value in low order 8 bits of
                           Constant Register)
*
-

```

#### 7.4.9 Calculating Offsets, r;O

Relative addressing and branching involve the use of an offset - the number of words or bytes forward or backward from the current location to the effective address. During the debugging session it may be necessary to change a relative address or branch reference by replacing one instruction offset with another. ODT calculates the offsets in response to the r;O command.

The command r;O causes ODT to print the 16-bit and 8-bit offsets from the currently open location to address r. For example:

```

*346/000034 414;O 000044 022 22
*/000022

```

In the example, location 346 is opened and the offsets from that location to location 414 are calculated and printed. The contents of location 346 are then changed to 22 (the 8-bit offset) and verified on the next line.

The 8-bit offset is printed only if it is in the range -128(10) to 127(10) and the 16-bit offset is even, as was the case above. For example, the offset of a relative branch is calculated and modified as follows:

```

*1034/103421 1034;O 177776 377\021 377
*/103777

```

Note that the modified low-order byte 377 must be combined with the unmodified high-order byte.

#### 7.4.10 Relocation Register Commands, r;nR, ;nR, ;R

The use of the relocation registers has been defined in Section 7.1. At the beginning of a debugging session it is desirable to preset the registers to the relocation biases of those relocatable modules which will be receiving the most attention.

This can be done by typing the relocation bias, followed by a semicolon and the specification of relocation registers.

r;nR

r may be any relocatable expression and n is an integer from 0 to 7. If n is omitted it is assumed to be 0.

As an example:

```
*1000;5R      (puts 1000 into relocation register 5)
*5,100;5R     (effectively adds 100 to the contents of
               relocation register 5)
*
```

In certain uses, programs may be relocated to an address below that at which they were assembled. This could occur with PIC code which is moved without the use of the Linker.

In this case the appropriate relocation bias would be the 2's complement of the actual downward displacement. One method for easily evaluating the bias and putting it in the relocation register is illustrated in the following example:

Suppose the program was assembled at location 5000 and was moved to location 1000. Then the sequence:

```
*1000;1R
*1,-5000;1R
*
```

puts the 2's complement of 4000 in relocation register 1, as desired.

Relocation registers are initialized to -1, so that unwanted relocation registers never enter into the selection process when ODT searches for the most appropriate register.

To set a relocation register to -1, type ;nR. To set all relocation registers to -1, type ;R.

ODT maintains a table of relocation registers, beginning at the address specified by \$R. Opening \$R (\$R/) opens relocation register 0. Successively typing the LINE FEED key opens the other relocation registers in sequence. When a relocation register is opened in this way, it may be modified just as any other memory location.

#### 7.4.11 The Relocation Calculators, nR and n!

When a location has been opened, it is often desirable to relate the relocated address and the contents of the location back to their relocatable values. To calculate the relocatable address of the opened location relative to a particular relocation bias, type n!, where n specifies the relocation register. This calculator works with opened bytes and words. If n is omitted, the relocation register whose contents are closest but less than or equal to the opened location is selected automatically by ODT. In the following example,

assume that these conditions are fulfilled by relocation register 2, which contains 2000:

To find the most likely module that a given opened byte is in,

\*2500\011= !=2,000500

Typing nR after opening a word causes ODT to print the octal number which equals the value of the contents of the opened location minus the contents of relocation register n. If n is omitted, ODT selects the relocation register whose contents are closest but less than or equal to the contents of the opened location. For example, assume the relocation bias stored in relocation register 1 is 001234; then:

\*1,500/024550 1R=1,023314

The value 23314 is the content of 1,500, relative to the base 1234.

An example of the use of both:

If relocation register 1 contains 1000, and relocation register 2 contains 2000, then to calculate the relocatable addresses of location 3000 and its content, relative to 1000 and 2000, the following can be performed.

\*3000/005670 1!=1,002000 2!=2,001000 1R=1,004670 2R=2,003670

#### 7.4.12 ODT's Priority Level, \$P

\$P represents a location in ODT that contains the interrupt (or processor) priority level at which ODT operates. If \$P contains the value 377, ODT operates at the priority level of the processor at the time ODT is entered. Otherwise \$P may contain a value between 0 and 7 corresponding to the fixed priority at which ODT will operate.

To set ODT to the desired priority level, open \$P. ODT prints the present contents, which may then be changed:

\*\$P/000006 377  
\*

If \$P is not specified, its value will be seven.

Breakpoints may be set in routines which run at different priority levels. For example, a program running at a low priority may use a device service routine which operates at a higher priority level. If a breakpoint occurs from a low-priority routine, and ODT operates at a low priority, and an interrupt occurs from a high priority routine, then the breakpoints in the high priority routine will not be recognized since they have been removed when the low priority breakpoint occurred. That is, interrupts that occur at a priority higher than the one at which ODT is running will occur and any breakpoints will not be recognized. ODT removes all breakpoints from the program whenever it gains control. Breakpoints are only set when ;P and ;G commands are executed.

Example:

```
*$P/00007 5
*1000;B
*2000;B
*500;G
B0:1000
* (clock interrupt occurs and is serviced)
```

If a higher level interrupt occurs while ODT is waiting for input the interrupt will be serviced, and no breakpoints will be recognized.

#### 7.4.13 ASCII Input and Output, r;nA

ASCII text may be inspected and changed by the command

r;nA

where r is a relocatable expression, and n is a character count. If n is omitted it is assumed to be 1. ODT prints n characters starting at location r, followed by a carriage return/line feed. Type one of the following:

RETURN ODT outputs a carriage return/line feed and an asterisk and waits for another command.

LINE FEED ODT opens the byte following the last byte output.

up to n characters of text

ODT inserts the text into core, starting at location r.

If less than n characters are typed, terminate the command by typing CTRL/U, causing a carriage return/line feed/asterisk to be output. However, if exactly n characters are typed, ODT responds with a carriage return/line feed, the address of next available byte and a carriage return/line feed/asterisk.

ODT does not check the magnitude of n.

#### 7.5 ERROR DETECTION

ODT detects two types of error: illegal or unrecognizable command and bad breakpoint entry.

ODT does not check for the legality of an address when commanded to open a location for examination or modification.

Thus the command:

177774/

references nonexistent memory, thereby causing a trap through the vector at location 4. If this vector has not been properly initialized, unpredictable results occur.

Similarly, a command such as

\$20/

which references an address eight times the value represented by \$2, may cause an illegal (nonexistent) memory reference.

Typing something other than a legal command causes ODT to ignore the command, print

?  
\*

and wait for another command. Therefore, to cause ODT to ignore a command just typed, type any illegal character (such as 9 or RUBOUT) and the command will be treated as an error, i.e., ignored.

ODT suspends program execution whenever it encounters a breakpoint, i.e., traps to its breakpoint routine. If the breakpoint routine is entered and no known breakpoint caused the entry, ODT prints:

BE001542  
\*

and waits for another command. In the example above, BE001542 denotes Bad Entry from location 001542. A bad entry may be caused by an illegal trace trap instruction, setting the T-bit in the status register, or by a jump to the middle of ODT.

## 7.6 PROGRAMMING CONSIDERATIONS

Information in this section is not necessary for the efficient use of ODT. However, its content does provide a better understanding of how ODT performs some of its functions and in certain difficult debugging situations, this understanding is necessary.

### 7.6.1 Functional Organization

The internal organization of ODT is almost totally modularized into independent subroutines. The internal structure consists of three major functions: command decoding, command execution, and various utility routines.

The command decoder interprets the individual commands, checks for command errors, saves input parameters for use in command execution, and sends control to the appropriate command execution routine.

The command execution routines take parameters saved by the command decoder and use the utility routines to execute the specified command. Command execution routines exit either to the object program or back to the command decoder.

The utility routines are common routines such as SAVE-RESTORE and I/O. They are used by both the command decoder and the command executers.

Communication and data flow are illustrated in Figure 7-1.

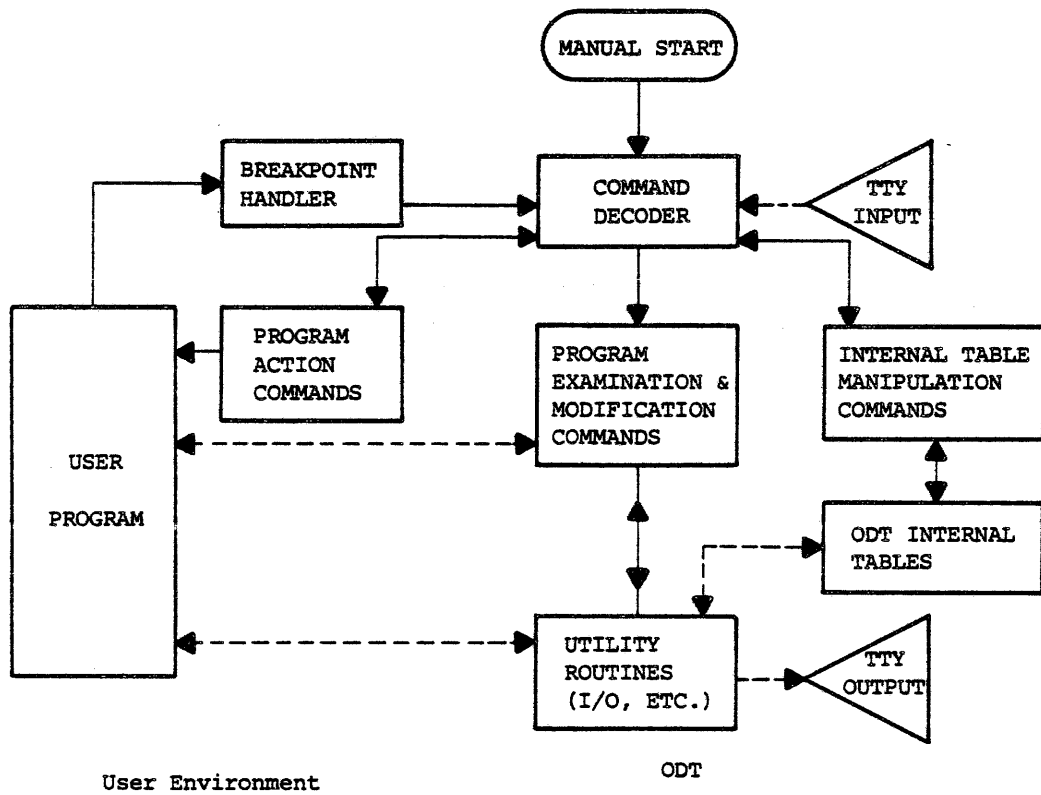


Figure 7-1 Communication and Data Flow

## 7.6.2 Breakpoints

The function of a breakpoint is to give control to ODT whenever the user program tries to execute the instruction at the selected address. Upon encountering a breakpoint, all of the ODT commands can be used to examine and modify the program.

When a breakpoint is executed, ODT removes all the breakpoint instructions from the user's code so that the locations may be examined and/or altered. ODT then types a message on the terminal of the form Bn;k where k is the breakpoint address (and n is the breakpoint number). The breakpoints are automatically restored when execution is resumed.

A major restriction in the use of breakpoints is that the word where a breakpoint has been set must not be referenced by the program in any way since ODT has altered the word. Also, no breakpoint should be set at the location of any instruction that clears the T-bit. For example:

```
MOV #240,177776      ;SET PRIORITY TO LEVEL 5
```

### NOTE

Instructions that cause or return from traps (e.g., EMT, RTI) are likely to clear the T-bit, since a new word from the trap vector or the stack is loaded into the Status Register.

A breakpoint occurs when a trace trap instruction (placed in the user program by ODT) is executed. When a breakpoint occurs, the following steps are taken:

1. Set processor priority to seven (automatically set by trap instruction).
2. Save registers and set up stack.
3. If internal T-bit trap flag is set, go to step 13.
4. Remove breakpoints.
5. Reset processor priority to ODT's priority or user's priority.
6. Make sure a breakpoint or single-instruction mode caused the interrupt.
7. If the breakpoint did not cause the interrupt, go to step 15.
8. Decrement repeat count.
9. Go to step 18 if non-zero; otherwise reset count to one.
10. Save Teletype status.
11. Type message about the breakpoint or single-instruction mode interrupt.



12. Go to command decoder.
13. Clear T-bit in stack and internal T-bit flag.
14. Jump to the Go processor.
15. Save Teletype status.
16. Type BE (Bad Entry) followed by the address.
17. Clear the T-bit, if set, in the user status and proceed to the command decoder.
18. Go to the Proceed processor, bypassing the TT restore routine.

Note that steps 1-5 inclusive take approximately 100 microseconds during which time interrupts are not permitted to occur (ODT is running at level 7).

When a proceed (;P) command is given, the following occurs:

1. The proceed is checked for legality.
2. The processor priority is set to seven.
3. The T-bit flags (internal and user status) are set.
4. The user registers, status, and Program Counter are restored.
5. Control is returned to the user.
6. When the T-bit trap occurs, steps 1, 2, 3, 13, and 14 of the breakpoint sequence are executed, breakpoints are restored, and program execution resumes normally.

When a breakpoint is placed on an IOT, EMT, TRAP, or any instruction causing a trap, the following occurs:

1. When the breakpoint occurs as described above, ODT is entered.
2. When ;P is typed, the T-bit is set and the IOT, EMT, TRAP, or other trapping instruction is executed.
3. This causes the current PC and status (with the T-bit included) to be pushed on the stack.
4. The new PC and status (no T-bit set) are obtained from the respective trap vector.
5. The whole trap service routine is executed without any breakpoints.
6. When an RTI is executed, the saved PC and PS (including the T-bit) are restored. The instruction following the trap-causing instruction is executed. If this instruction is not another trap-causing instruction, the T-bit trap occurs,

causing the breakpoints to be reinserted in the user program, or the single-instruction mode repeat count to be decremented. If the following instruction is a trap-causing instruction, this sequence is repeated starting at step 3.

#### NOTE

Exit from the trap handler must be via the RTI instruction. Otherwise, the T-bit is lost. ODT can not gain control again since the breakpoints have not been reinserted yet.

Note that the ;P command is illegal if a breakpoint has not occurred (ODT responds with ?); ;P is legal, however, after any trace trap entry.

The internal breakpoint status words have the following format:

1. The first eight words contain the breakpoint addresses for breakpoints 0-7. (The ninth word contains the address of the next instruction to be executed in single-instruction mode.)
2. The next eight words contain the respective repeat counts. The following word contains the repeat count for single-instruction mode.)

These words may be changed at will, either by using the breakpoint commands or by direct manipulation with \$B.

When program runaway occurs (that is, when the program is no longer under ODT control, perhaps executing an unexpected part of the program where a breakpoint has not been placed), ODT may be given control by pressing the HALT key to stop the computer and restarting ODT (see section 7.7). ODT prints \*, indicating that it is ready to accept a command.

If the program being debugged uses the teleprinter for input or output, the program may interact with ODT to cause an error since ODT uses the teleprinter as well. This interactive error will not occur when the program being debugged is run without ODT.

1. If the teleprinter interrupt is enabled upon entry to the ODT break routine, and no output interrupt is pending when ODT is entered, ODT generates an unexpected interrupt when returning control to the program.
2. If the interrupt of the teleprinter reader (the keyboard) is enabled upon entry to the ODT break routine, and the program is expecting to receive an interrupt to input a character, both the expected interrupt and the character are lost.
3. If the teleprinter reader (keyboard) has just read a character into the reader data buffer when the ODT break routine is entered, the expected character in the reader data buffer is lost.

### 7.6.3 Search

The word search allows the user to search for bit patterns in specified sections of memory. Using the \$M/ command, the user specifies a mask, a lower search limit (\$M+2), and an upper search limit (\$M+4). The search object is specified in the search command itself.

The word search compares selected bits (where ones appear in the mask) in the word and search object. If all of the selected bits are equal, the unmasked word is printed.

The search algorithm is:

1. Fetch a word at the current address.
2. XOR (exclusive OR) the word and search object.
3. AND the result of step 2 with the mask.
4. If the result of step 3 is zero, type the address of the unmasked word and its contents. Otherwise, proceed to step 5.
5. Add two to the current address. If the current address is greater than the upper limit, type \* and return to the command decoder, otherwise go the step 1.

Note that if the mask is zero, ODT prints every word between the limits, since a match occurs every time (i.e., the result of step 3 is always zero).

In the effective address search, ODT interprets every word in the search range as an instruction which is interrogated for a possible direct relationship to the search object. The mask register is opened only to gain access to the search limit registers.

The algorithm for the effective address search is (where (X) denotes contents of X, and K denotes the search object):

1. Fetch a word at the current address X.
2. If (X)=K [direct reference], print contents and go to step 5.
3. If (X)+X+2=K [indexed by PC], print contents and go the step 5.
4. If (X) is a relative branch to K, print contents.
5. Add two to the current address. If the current address is greater than the upper limit, perform a carriage return/line feed and return to the command decoder; otherwise, go to step 1.

#### 7.6.4 Terminal Interrupt

Upon entering the TT SAVE routine, the following occurs:

1. Save the LSR status register (TKS).
2. Clear interrupt enable and maintenance bits in the TKS.
3. Save the TT status register (TPS).
4. Clear interrupt enable and maintenance bits in the TPS.

To restore the TT:

1. Wait for completion of any I/O from ODT.
2. Restore the TKS.
3. Restore the TPS.

#### NOTE

If the TT printer interrupt is enabled upon entry to the ODT break routine, the following may occur:

1. If no output interrupt is pending when ODT is entered, an additional interrupt always occurs when ODT returns control to the user.
2. If an output interrupt is pending upon entry, the expected interrupt occurs when the user regains control.

If the TT reader (keyboard) is busy or done, the expected character in the reader data buffer is lost.

If the TT reader (keyboard) interrupt is enabled upon entry to the ODT break routine, and a character is pending, the interrupt (as well as the character) is lost.

#### 7.7 OPERATING PROCEDURES

ODT is supplied as a relocatable object module. It can be linked with the user program using the RT-11 Linker, for an absolute area in core and loaded with the user program.

Once loaded in core with the user program, ODT has three legal starting or restart addresses. The lowest (O.ODT) is used for normal entry, retaining the current breakpoints. The next (O.ODT+2) is a restart address which clears all breakpoints and re-initializes ODT saving the general registers, and clearing the relocation registers. The last address (O.ODT+4) is used to reenter ODT. A reenter saves the Processor Status and general registers and removes the breakpoint instructions from the user program. ODT prints the Bad Entry (BE) error message. Breakpoints which were set are reset on the next ;G command. (;P is illegal after a BE message).

The absolute address used is the address of the entry point O.ODT shown in the Linker load map. O.ODT is always the lowest address of ODT+172, i.e., O.ODT is relative location 172 in ODT.

NOTE

If linked with an overlay structured file, ODT should reside in the root segment so it is always in core. A breakpoint inserted in an overlay will be destroyed if it is overlaid during program execution.

Examples:

ODT linked with the user program-

.GET USER.SAV	User program previously linked to ODT is brought into core.
.START 1020	Value (1020) of entry point O.ODT from linker load map is used to start ODT.
ODT V01	

Loading ODT with the user program-

.GET USER.SAV	User program is loaded into core.
.GET ODT.SAV	ODT is loaded into core.
.START 1172	Assuming ODT has been linked for a bottom address of 1000, ODT starts.
ODT V01	

Restarting ODT clearing breakpoints-

.START 1174	Assuming ODT was originally linked for a bottom address of 1000 this command (O.ODT+2) re-initializes ODT and clears any previous breakpoints.
*	

Reentering ODT-

.START 1176	Assuming ODT was linked for a bottom address of 1000, the value of O.ODT 1172+4 is used as the start address.
BE001176	
*	

#### 7.7.1 Return to Monitor, CTRL/C

If ODT is awaiting a command, a CTRL/C from the keyboard calls the RT-11 Keyboard Monitor. The Monitor responds with a C on the terminal and awaits a Keyboard Monitor command.

#### 7.7.2 Terminate Search, CTRL/U

If typed during a search printout, a CTRL/U terminates the search and ODT prints an asterisk.

## CHAPTER 8

### PROGRAMMED REQUESTS

The Monitor provides a number of services for system programs which are available to user programs. These services include opening and closing files, data transfer, command string interpretation, and loading device handlers.

The user program calls for the services of the Monitor through programmed requests. Programmed requests are macro calls which are assembled into the user program and interpreted by the Monitor at execution time. (Refer to Chapter 5 and Appendix P for a description of Assembly Language programming.) A programmed request consists of a macro call followed, where appropriate, by one or more arguments.

For example:

```
.PRINT .MSG
```

is a programmed request called `.PRINT` followed by an argument `.MSG`. (It types the ASCII message which occurs at the specified address, on the terminal). The macro call is expanded at assembly time by the Macro Assembler to a sequence of instructions which trap to and pass the arguments to the appropriate Monitor service routine to carry out the specified function. The code used to expand the macro `.PRINT .MSG` is:

```
.MACRO .PRINT .MSG  
.IFNB .MSG  
MOV .MSG %↑00  
.ENDC  
EMT ↑0340+↑011  
.ENDM
```

The first argument of a programmed request is passed to the monitor in register 0; any additional arguments are pushed onto the stack. The programmed request macros are written so that the argument (if any) which goes into R0 is the last argument in the argument list. If the argument is omitted, R0 will not be affected. It is possible to build an argument in R0, and leave it intact throughout the programmed request by omitting the last argument. For example, suppose that in the example above, the address of the message had been built in R0 prior to the macro call. In that case, the macro could be called by

```
.PRINT
```

This saves code, as the macro expander recognizes that R0 already contains the desired argument, and therefore the `MOV .MSG, %↑00` is not included.

If an error occurs in the execution of the programmed request, the carry bit is set in the processor status and byte 52 contains the error code when execution of the programmed request is complete.

Because they will be used in the source fields of MOV instructions when the macros are expanded, arguments to programmed request macros must with one exception be legal MACRO assembler source operands. Continuing the above example

`.PRINT (R1)`

R1 contains the address of the message address

`.PRINT #ADDR`

ADDR is the address of the message

`.PRINT -(R2)`

R2 contains 2 more than the address of the message address.

The arguments specified will be used as source fields in the expanded instructions, which serve to place items in R0 or move them onto the stack. In the example, an argument of #ADDR is expanded into

`MOV #ADDR,%↑00.`

The one exception to the rule that arguments be MACRO assembler source operands is those macros which accept channel numbers. A channel number argument must be an octal number between 0 and 17. E.g.

`.READW .CHANNEL,BUFFER,WCOUNT,BLOCKN`

we might call this

`.READW 13,(R0)+,#400,BLOCKN`

All registers except R0 are preserved across programmed requests. With the exception of CSIGEN and CSISPC (calls to the command string interpreter), the position of the stack pointer is also preserved across a programmed request.

## 8.1 SYSTEM CONCEPTS

It is important to understand the basic operational characteristics of RT-11 described below to use the system effectively.

### 1. Channel

In many programmed requests, the term 'channel number' is used. A channel number is merely a logical identifier in the range (0 to 17(8)) used by the RT-11 Monitor. Thus, when a file is opened on a particular device, a logical number is assigned to that file. That identifying number is the channel number. To refer to data blocks within the specified file, it is merely necessary to refer to the appropriate channel I.D.

### 2. Device block (devblk)

A device block is defined as a .RAD50 (refer to the MACRO Chapter) string which specifies a physical device and file name for an RT-11 programmed request. For example, a devblk representing a file FILE.EXT on device DK: could be written as:



```
.RAD50 /DK /
.RAD50 /FIL/
.RAD50 /E /
.RAD50 /EXT/
```

Note that the RAD50 string must be filled out with spaces to 3 characters. This string could also be written as:

```
.RAD50 /DK FILE EXT/
```

Again, spaces are used to fill out each field.

Note that the period separator is not represented in the actual RAD50 string. The period is used by the Monitor keyboard interface to indicate where the extension field begins.

3. Program Parameter Words (Bytes 40-57)  
RT-11 uses bytes 40-57 as an area to hold information about the program currently executing, as well as certain information for Monitor use. The description of these bytes follows:

<u>Bytes</u>	<u>Meaning and Use</u>
40,41	Start address of job. When a file is LINKed into an RT-11 program, this word is set to the START address. The start address is set either with the Linker /T Switch, or as the argument in the .END statement in the program (refer to the MACRO and Linker chapters).
42,43	Initial value of stack pointer. This is set by LINKer. (Refer to the LINKer Chapter.)
44,45	Job Status Word This word is used as 16 flag bits for the Monitor. The bits currently in use and their meanings are:

<u>Bit #</u>	<u>Use</u>
15	Swapping bit. Set by Monitor. Programs which do not need to 'swap' the USR will have this bit =1. See 'Swapping Algorithm' for more detail.
14	Unused
13	Restart bit. If set to 1, the program is re-enterable (REENTER command from Keyboard Monitor). If the user program is restartable, this bit must be set by the program. The default case is 0, which means that the REENTER command will generate an error message.

<u>Bit #</u>	<u>Use</u>
12	1 indicates special keyboard mode of I/O. 0 indicates Normal Mode. Refer to the explanatin of the .TTYIN/.TTINR requests for the use of bit 12.
11,10	Unused
9	Overlay bit. 1 indicates the job uses the LINKer overlay structure. This bit is set up by the LINKer.
8	Unused
7	1 indicates halt on I/O Error. Set by user program. If it is desired to HALT whenever a hardware error occurs, this bit should be set to 1.
6-0	Unused

Since the currently unassigned bits may be used in the future, it is not recommended that user programs use this word for internal flags.

<u>Bytes</u>	<u>Meaning and Use</u>
46,47	USR load address. Normally 0, may be set to arbitrary word address by the user program. See 'Swapping Algorithm' for details of use.
50,51	Highcore address. The Monitor keeps track of the highest address the user program can use in this word. The LINKer does the initial setting. It can be modified via the .SETTOP (Set Top of Core) Monitor request.
52	EMT error code. If a Monitor request results in an error, the code number of the error is always returned in byte 52. Each Monitor call has its own set of possible errors. Note that the user program should check byte 52 with relative addressing, rather than absolute addressing. For example, <pre style="margin-left: 40px;">A = 52 TST A ;RELATIVE ADDRESSING TST @#A ;ABSOLUTE ADDRESSING</pre>
53	Currently unused
54,55	Address of the beginning of the Resident Monitor. Since RT-11 always loads the resident into the highest available core locations, a word is supplied which will always point to the first location of the resident. This word must NEVER be altered by the user, or RT-11 will malfunction.

<u>Byte</u>	<u>Meaning and Use</u>
56	Fill character (7-bit ASCII). Some high speed terminals will require filler (null) characters following certain output characters. Byte 56 will contain the 7-bit representation of the character requiring fillers.
57	Filler count. This byte, specifies the number of fill characters required. If bytes 56 and 57=0, no fillers are required.

The required fill characters are:

<u>Terminal</u>	<u>No. of fills after</u>	<u>Value of Word 56</u>
Serial LA30 @300 baud	9 after carriage return	5015
Serial LA30 @150 baud	4 after carriage return	2015
Serial LA30 @110 baud	2 after carriage return	1015
VT05 @ 2400 baud	4 after line feed	2012
VT05 @ 1200 baud	2 after line feed	1012
VT05 @ 600 baud	1 after line feed	412

#### 4. Swapping Algorithm

RT-11 may, for some programs, need to perform a 'swap' operation. This requires a portion of the user program to be temporarily saved in system scratch blocks, and the USR to be read in where that portion of code had been. Swapping occurs when the USR and the user program must occupy the same memory space. RT-11 determines if a swap is required by two basic items of information:

- a. The top of core, as defined by the .SETTOP operation. If the argument of a user performed .SETTOP causes the top of core to extend into the USR area, RT-11 performs swap operations when USR requests are made. Refer to the .SETTOP request for an example.
- b. The value of location 46. Loading Word 46 with a non-zero value causes the USR to be read into the area pointed to by 46. If 46=0, the USR is at it's normal location, below the resident Monitor. Thus, as an example:

```

UFLOAT=46           ;SYMBOLIC REFERENCE
MOV #3000,UFLOAT   ;IF SWAPPING IS REQUIRED, SWAP
                   ;USR INTO LOCATION 3000-6776
. ENTER A1,A2      ;DO A SWAPPING REQUEST
.
.
.

```

This sequence causes any USR swapping to occur at location 3000. If location 46 is cleared, the USR is again loaded at it's normal place, below RMON.

NOTE

1. Care should be exercised when using location 46 to specify the area to be used for the USR. The system is unprotected against reading over the resident. Thus, if location 46 is greater than the normal USR load point, the resident Monitor could be overwritten.
2. Locating normal load point of USR:  
It is sometimes desirable to know how much core is available in a particular configuration. This can be done by examining one of several words in the resident Monitor. To access these words, one must:
  - a. Pick up the contents of word 54, which is the address where RMON begins.
  - b. Add a suitable offset to that address. These offsets can be used as indexes off the contents of location 54 to obtain the information required.
  - c. Get the contents of the resultant address.

The offsets (for RT-11 release 1) from the start of RMON and the values they define are:

<u>OFFSET (bytes)</u>	<u>CONTENTS</u>
262	System date word
266	Start of normal USR area

To get as much core as possible without causing USR swapping to occur, do the following:

```
RMON=54                ;REFERENCE MONITOR WORDS
USRLD=266              ;SYMBOLICALLY
MOVY RMON,R0           ;START ADDRESS OF RMON
+                   ;OFFSET TO START OF USR
                       ;USED AS INDEX
MOV USRLD(R0),R0       ;FIRST LOCATION OF USR NOW IN
                       ;R0
TST -(R0)              ;LAST AVAILABLE WORD ADDRESS
                       ;IS 2 BEFORE FIRST LOC. OF
                       ;USR
                       ;TST INST. IS SPECIAL
                       ;TECHNIQUE TO SUBTRACT 2
                       ;FROM AN EVEN REGISTER.
                       ;USE R0 IN A .SETTOP
.SETTOP
```

This procedure allows maximum core space, while leaving the USR in core for speed of operation.

## File Structure

RT-11 uses a 'contiguous' file structure. This type of structure implies that every file on the device is made up of a contiguous group of physical blocks. Thus, a file that is 9 blocks long will occupy 9 contiguous blocks on the device. Using a contiguous structure gives rise to several classes of files in RT-11. They are:

1. Permanent files. This is a file which has been .CLOSEd on a device. Any files which appear in a PIP directory listing are permanent files.
2. Tentative files. A file which has been created via .ENTER, but not .CLOSEd, is a tentative file entry. When the .CLOSE is given, the tentative entry becomes a permanent file. If a permanent file with the same name existed previously, the old file is deleted.
3. Empty entry. When disk space is unused, or a permanent file is deleted, an empty entry is created. Empty entries appear in a PIP directory listing as <UNUSED> N, where N is the decimal length of the empty area.

Since a contiguous structure does not automatically reclaim unused disk space, as a linked structure does, the device may eventually become 'fragmented.' A device is fragmented when there are many empty entries which are scattered over the device. RT-11 PIP has an option which allows the user to quickly and easily collect all empty areas at the end of a device. Refer to the PIP chapter for details.

## Using the System Macro Library

User programs for RT-11 should always be written using the system macro library (SYSMAC.SML) which is supplied with RT-11. This ensures compatibility among all your programs and allows easy modification by redefining a macro.

### 8.2 TYPES OF PROGRAMMED REQUESTS

Services which the Monitor makes available to the user through programmed requests can be classified into three groups:

1. requests for file manipulation
2. requests for data transfer
3. requests for miscellaneous services

Table 8-1 summarizes the programmed requests available under the Monitor.

Table 8-1

Summary of Programmed Requests

Mnemonic	Purpose
<p>File Manipulation Requests</p> <p>.LOOKUP</p> <p>.ENTER</p> <p>.RENAME</p> <p>.REOPEN</p> <p>.CLOSE</p> <p>.SAVESTATUS</p>	<p>Opens an existing file for input and/or output via the specified channel.</p> <p>Creates a new file for output.</p> <p>Changes the name of the indicated file to a new name.</p> <p>Restores the parameters stored via SAVESTATUS request, and reopens the channel for I/O.</p> <p>Closes the specified channel.</p> <p>Saves status parameters of an open file in user core, and frees the channel for future use.</p>
<p>Data Transfer Requests</p> <p>.READW</p> <p>.READ</p> <p>.READC</p> <p>.WRITW</p> <p>.WRITE</p>	<p>Transfers data via the specified channel to a core buffer and returns control to the user program when the transfer is complete.</p> <p>Transfers data via the specified channel to a core buffer and returns control to the user program when the transfer request is entered in the I/O queue. No special action is taken upon completion of I/O.</p> <p>Transfers data via the specified channel to a core buffer and returns control to the user program when the transfer request is entered in the I/O queue. Upon completion of the read, control transfers to the routine specified in the .READC request.</p> <p>Transfers data via the specified channel to a device and returns control to the user program when transfer is complete.</p> <p>Transfers data via the specified channel to a device and returns control to the user program when the transfer request is entered in the I/O queue. No special action is taken upon completion of the I/O.</p>

(Continued on next page)

Table 8-1 (Cont)

## Summary of Programmed Requests

Mnemonic	Purpose
.WRITC	Transfers data via the specified channel to a device and returns control to the user program when the transfer request is entered in the I/O queue. Upon completion of the write, control transfers to the routine specified in the .WRITC request.
.TTYIN .TTINR	Transfers one character from the terminal buffer to R0.
.TTYOUT .TTOUTR	Transfers one character from R0 to the terminal buffer.
<b>Miscellaneous Services</b>	
.WAIT	Waits for completion of all I/O on a specified channel.
.FETCH	Loads device handlers into core.
.RELEASES	Removes device handlers from core.
.CSIGEN	Calls Command String Interpreter (CSI) in general mode.
.CSISPC	Calls CSI in special mode.
.LOCK	Makes the monitor User Service Routines (USR) permanently resident, until EXIT or UNLOCK is executed. The user program may be swapped out if necessary.
.UNLOCK	Releases USR if a LOCK was done. The user program will be swapped in if required.
.EXIT	Exits the user program and returns to the keyboard monitor.
.SRESET	Resets all channels and releases the device handlers from core.
.PRINT	Outputs the specified ASCII string to the terminal.
.DELETE	Deletes the file from the specified device.
.SETTOP	Specifies the highest core location to be used by the user program.

(Continued on next page)

Table 8-1 (Cont)

## Summary of Programmed Requests

Mnemonic	Purpose
.RCTRLO	Enables output to the terminal.
.QSET	Expands the size of the Monitor I/O queue.
.DSTATUS	Get the status of a particular device.
.HRESET	Does a .SRESET, executes a hardware RESET instruction.
.DATE	Moves the current date information into R0.

The programmed requests are separated into two classes according to whether or not they require the USR to be swapped into core. Any request which will require the USR in core may also require that a portion of the user program be swapped out to provide room for the USR. During normal operation, this swapping is invisible to the user, and he need not be concerned about it. However, it is possible to optimize programs so as to require as little swapping as possible. The swapping algorithm works by examining the high core limit set by the user program (.SETTOP) and the address specified for the loading of the USR (Refer to section on program parameters). If the top of the user's program has not gone beyond the address where the USR is to be loaded, no swapping will be required. If, however, the user program has set the upper limit beyond the USR load point, swapping will occur. For example, if RT-11 is running on a 16K PDP-11, this implies that the USR would normally be loaded at 63000. The user program executes:

```

USRADDR=46                ;REFERENCE SYMBOLICALLY
CLR USRADDR              ;USR AT NORMAL ADDRESS.
.SETTOP #60000           ;SET UPPER LIMIT TO 60000.
.FETCH #SPACE,#DEVBLK   ;NOW FETCH A HANDLER
BCS FERR
.
.
.

```

This sequence would not need to swap, because the top of core request was for an address below the USR load point. If, however, a

```
.SETTOP #66000
```

had been done, the .FETCH directive would have required the USR to be swapped in, and the user program restored when the FETCH was complete.

The programmed requests which may require swapping are:

```

.LOOKUP
.ENTER
.QSET

```



.RENAME  
.CLOSE  
.DSTATUS  
.FETCH  
.RELEASES  
.CSISGEN  
.CSISPC  
.SRESET  
.DELETE  
.HRESET

Those not requiring swapping are:

.SAVESTATUS  
.REOPEN  
.READW  
.READ  
.READC  
.WRITW  
.WRITE  
.WRITC  
.WAIT  
.TTYIN  
.TTINR  
.TTYOUT  
.TTOUTR  
.LOCK  
.UNLOCK } SPECIAL CASES. SEE DESCRIPTION  
.EXIT  
.SRESET  
.PRINT  
.SETTOP  
.RCTRLO

### 8.3 PROGRAMMED REQUEST USAGE

In the general format of programmed requests, the following definitions apply:

.channel is the octal channel number (in the range 0-17) as described in the System Concepts section.

.devblk is the four-word RAD50 file description of the file to be opened. Also described in System Concepts.

#### 8.3.1 .DATE

This request moves the current date information from the system date word into R0. The user program can then use this information to print out the current date where that operation is desired. The date word which is returned is in the following format:

Bit:           14           10 9           5 4           0

MONTH	DAY	YEAR-110(8)
-------	-----	-------------

Macro Call:            .DATE

Errors:

No errors are returned. A zero result indicates that no date value was entered.

### 8.3.2 .CLOSE

The .CLOSE request terminates activity on the specified channel. The tentative file is made permanent, and the channel is freed for use in another operation.

Macro Call:            .CLOSE .channel

A .CLOSE is required on any channel that was opened, either for input or output. A .CLOSE request specifying a channel that is not opened is ignored.

A .CLOSE performed on a file which was opened via .ENTER will cause the device directory to be updated. A file opened via .LOOKUP will not require any directory operations.

If the device associated with the specified channel already contains the new file name and extension, the old copy of the file with the same name is deleted when the new file is made permanent.

The length assigned to the file when it is .CLOSED is equal to the difference between the highest block written and the starting block of the file.

Errors:

Close does not return any errors. If the device handler for the operation is not in core, a fatal monitor error is generated.

Example:

The following example incorporates the .LOOKUP, .READW, and .CLOSE requests. The program opens the file SY: RT11.MAC which is on the system device, SY:, for input on channel 0. The first block is read and the file is then closed.

```
.FETCH #CORADD,#FPTR           ;FETCH DEVICE HANDLER.
.LOOKUP 0,#FPTR                ;OPEN SY:RT11.MAC ON CHANNEL 0
BCS LUKERR                     ;ERROR IN LOOKUP
.READW 0,#BUFF,#400,BLOCK      ;NOW READ 1 BLOCK
                               ;(400(8) WORDS) - INTO A BUFFER
                               ;LOCATION. BLOCK CONTAINS THE
                               ;RELATIVE BLOCK # TO READ
```

```

        BCS RERR                ;READ ERROR
        .CLOSE 0                ;TERMINATE I/O ON CHANNEL 0
        .                    ;AND FREE IT FOR USE.
        .
        .
        LUKERR:  HALT            ;.LOOKUP ERROR.  PROBABLY CAN'T
                                ;FIND THE FILE.  HALT IN THIS
                                ;SIMPLE EXAMPLE.
        RERR:    HALT            ;.READW ERROR.  CHECK BYTE 52
                                ;IN MOST NORMAL CASES.
                                ;HERE, JUST HALT.
        FPTR:    .RAD50 "SY RT11  MAC" ;LOOKUP FILE DESCRIPTOR
        BLOCK:  0                ;INITIALLY 0; MEANS READ
                                ;FIRST (0TH) BLOCK OF THE
                                ;FILE
        CORADD:  .BLKW 1000      ;CORE BUFFER
        BUFF=.

```

### 8.3.3 .CSIGEN

The .CSIGEN request calls the Command String Interpreter (CSI) in general mode to process a user command string and perform all file LOOKUP and ENTER's as well as handler FETCH's. The CSI accepts an ASCII string containing device, file name and switch specifications and interprets them as the input and output devices of the program. The area to be used for the device handlers must be specified in the .CSIGEN request.

```

Macro Call:      .CSIGEN .devspc,.defext,.cstring

```

where .devspc is the address of the core area where the device handlers are to be stored.

.defext is the address of the four-word block which contains the RAD50 default extensions. These extensions are used when a file is specified without an extension.

.cstring is the address of the input string or a #0 if input is to come from the system terminal handler.

The area specified for the device handlers must be large enough to hold all the necessary handlers simultaneously. If the device handlers exceed the area available, the user program could be overlaid (the system, however, is protected from being overlaid).

When the EMT is complete, register 0 points to the first available location above the handlers.

The four-word block for the default extension is arranged as follows:

```

Word 1:          default extension for all input channels
Words 2,3,and 4: default extensions for output channels 0,1,2
                  respectively

```

All extensions are expressed in Radix 50. The following code could be used to set up default extensions for a macro assembler:

```
DEFEXT:      .RAD50  "MAC"  
             .RAD50  "OBJ"  
             .RAD50  "LST"  
             .WORD   0
```

In the command string

```
*DT0:ALPHA,DT1:BETA=DT2:INPUT
```

the default extension for input is MAC; for output, OBJ and LST.

When called, the Command String Interpreter closes channels 0-10 (octal).

Switches and their associated values are returned on the stack (see the switch description in the CSISPC section).

Errors:

If CSI errors occur and input was from the console terminal, an error message describing the fault is output to the terminal. If the input was from a string, the C bit is set as usual, and byte 52 contains the error code: The errors are:

<u>Byte. 52=</u>	<u>Meaning</u>
0	Illegal command (Bad separators, illegal filename, etc.).
1	A device specified is not found in the system tables.
2	Unused.
3	An attempt to ENTER a file failed because of a full directory.
4	An input file was not found in a LOOKUP.

When control returns to the user program after a call to .CSIGEN, all the specified files will have been opened for input and/or output. The association is as follows: the three possible output files are assigned to channels 0, 1, and 2. The six input slots are assigned to channels 3 through 10. A null specification would cause the associated channel to remain inactive. For example, in the following string:

```
*,LP:=F1,F2
```

Channel 0 is inactive since the first slot was null. Channel 1 is associated with the line printer, and channel 2 is inactive. Channels 3 and 4 are associated with two files on DK:, while all other input channels are inactive.

A .WAIT directive will always return an error if a particular channel is not open. This allows the user program to issue a .WAIT on return from the .CSIGEN to determine if a given channel is open, or to determine if the user specified a file in a particular command string field.

Example:

This example uses the general mode of the CSI to transfer an input file to an output file. Command input to the CSI will be from the console terminal.

The following program will transfer an input file to an output file. The CSI is used in general mode to process the I/O specification.

```

      ERRWD=52                ;ADDRESS OF MONITOR ERROR LOCATION
START: .CSIGEN #DSPACE,#DEXT,#0 ;OBTAIN TRANSFER COMMAND
      ;FROM TERMINAL
      MOV R0,BUFFER          ;R0 POINTS TO FREE CORE AREA
      CLR INBLK              ;BLOCK TO READ
READ:  .READW 3,BUFFER,#400,INBLK ;INPUT STARTS WITH
      ;CHANNEL 3.
      BCC CONT              ;CARRY CLEAR=> NO ERROR
      TSTB 52                ;52=0=> END OF INPUT
      BEQ EOF
      HALT
CONT:  .WRITW 0,BUFFER,#400,INBLK ;PROBABLY HARDWARE ERROR
      BCC NOERR              ;NOW WRITE OUTPUT. FIRST
      HALT                   ;OUTPUT ALWAYS CHANNEL 0.
      ;OUTPUT ERRORS ALL HALT.
NOERR: INC INBLK            ;GET NEXT BLOCK.
      BR READ                ;NO EOF. READ NEXT BLOCK
EOF:   .CLOSE 0              ;MAKE THE OUTPUT FILE PERMANENT
      .CLOSE 3               ;FREE CHANNEL 3
      .SRESET                ;RELEASE HANDLERS FROM CORE
      BR START               ;DONE. RECALL CSI FOR NEXT
                                ;TRANSFER COMMAND
DEXT:  .WORD 0,0,0,0        ;NO DEFAULT EXTENSIONS.
BUFFER: 0                    ;CONTAINS ADDRESS OF BUFFER SPACE.
INBLK:  .WORD 0              ;RELATIVE BLOCK OF FILE.
DSPACE=.                      ;USED FOR HANDLERS.

```

#### NOTE

The .CLOSE on channel 3 is not absolutely necessary in this example, as the CSI is called again on completion. The CSI always .CLOSEs the first nine channels. If the CSI were not recalled, the .CLOSE would be necessary.

#### 8.3.4 .CSISPC

The .CSISPC request calls the Command String Interpreter (CSI) in special mode. In special mode the CSI does no LOOKUPs, ENTERs, or FETCHes but establishes a 39-word block for storage of file descriptors specified in command strings.

Macro Call:                    .CSISPC .outspc,.defext,.cstring

where .outspc            is the address of the 39-word block to contain the file descriptors. This area may overlay the space allocated to .cstring if desired.

.defext is the address of the four-word block which contains the RAD50 default extensions. These extensions are used when a file is specified without an extension.

.cstring is the address of the input string or a 0 if input is to come from the console terminal.

The 39-word file description consists of nine file descriptor blocks (five words for each of three possible output files; four words each for up to six possible input files) which correspond to the nine possible files (three output, six input).

The five-word blocks hold four words of RAD50 representing dev:file.ext, and 1 word representing the size specification given in the string. A size specification is a decimal number enclosed in square brackets [], following the output file descriptor. For example,

\*DT3:LIST.MAC[15]=PR:

Special mode CSI would return in the first five word slot:

```
16101 .RAD50 for DT3
46173 .RAD50 for LIS
76400 .RAD50 for T_ _
50553 .RAD50 for MAC
017 Octal value of size request
```

In the fourth slot, the CSI would return:

```
63320 .RAD50 for PR
0 No file name
0 Specified
0
```

Since this is an input file, only four words are returned.

Errors:

Errors are treated the same as in general mode. However, since LOOKUPs and ENTERs are not done, the error codes which are valid are:

<u>Code</u>	<u>Explanation</u>
0	Illegal command line
1	Illegal device

Example:

This example illustrates the use of the special mode of CSI. This example could be a program to read a file not in RT-11 format to a file under RT-11.

```
.CSISPC #OUTSPC,#DEFEXT,#CSTRNG ;INPUT IS FROM A
;STRING IN CORE.
BCS ERROR ;SOME ERROR IN THE LINE.
```

```

        .ENTER 0,#OUTSPC,#100      ;NOW ENTER THE DESIGNATED
        ;OUTPUT FILE UNDER RT-11.
        BCC NOERR                  ;NO ENTER FAILURE.
        ENTER FAIL.
NOERR:  JSR R5,INPUT              ;ROUTINE INPUT WILL USE
        .                           ;THE INFORMATION AT
        .                           ;#OUTSPC+36 TO READ INPUT
        .                           ;FROM THE NON-RT11 DEVICE.
        .                           ;INPUT IS PROCESSED AND WRITTEN
        .                           ;OUT VIA .WRITW REQUESTS
        .CLOSE 0                  ;CLOSE THE RT-11 FILE
        .EXIT                      ;AND GO TO THE KEYBOARD MONITOR.

CSRTNG: .ASCIZ "DT4:RTFIL.MAC=DT2:DOS.MAC"
        .EVEN                      ;ALWAYS FOLLOW .ASCIZ WITH .EVEN.
DEFEXT: .WORD 0,0,0,0            ;NO DEFAULT EXTENSIONS.
ERROR:  HALT                     ;CSI ERROR. HALT IN SIMPLE CASE.
OUTSPC=.                          ;I/O LIST GOES HERE.

```

Passing switch information:

In both general and special modes of the CSI, CSI switches and their associated values are returned on the stack. A CSI switch is defined by a slash (/) followed by any character. The CSI does not restrict the switch to printing characters; however, it is suggested that printing characters be used wherever possible. The switch can be followed by an optional value, which is indicated by a : separator, followed by an octal number. This number is the switch value. For example:

```
*DT5:FILE.OBJ/B:50
```

is a legal switch and value construction. Switches can be associated with files with the CSI. For instance,

```
*DK:FOO/A,DT4:FILE.OBJ/A:100
```

In this case, there are two A switches. One is associated with the file DK:FOO. The second with DT4:FILE.OBJ, and has a value of 100(8). The stack output of the CSI is as follows:

<u>Word #</u>	<u>Value</u>	<u>Meaning</u>
1 (top of stack)	N	Number of switches found in command string. If N=0, no switches occurred.
2	Switch value and file number	The even byte = 7-bit ASCII switch value. Bits 8-14 = Number (0-10) of the file the switch is associated with. Bit 15 = 1 If the switch had a value. = 0 If the switch had no value.

3            Switch value or next switch    If word 2 was <0, word 3 = switch value. If word 2 was >0, this word is the next switch value. (If it exists).

For example, the input to the CSI is:

\*FILE/B:20,FIL2/E=DT3:INPUT/X

on return, the stack would be:

Stack Pointer→

3
100102
20
505
1530

Three switches appeared.  
 Switch=B. Associated with file 0 and it has a value.  
 The value is 20.  
 Next switch=E; associated with file 1, no value  
 Last switch=X; with file 3, no value.

As an extended example, assume the following string was input for CSI in general mode:

\*FILE[8],LP:,,SY:FILE2[20]=PR:,DT1:IN1/B,DT2:IN2/M:5

Assume also that the default extension block is:

```
DEFEXT:  .RAD50  'MAC'      ;INPUT EXTENSION
         .RAD50  'OP1'     ;FIRST OUTPUT EXTENSION
         .RAD50  'OP2'     ;SECOND OUTPUT EXTENSION
         .RAD50  'OP3'     ;THIRD OUTPUT EXTENSION
```

The output of this CSI call would be:

1. A file FILE.OP1 is opened on Channel 0 on device DK:.  
 Channel 1 is opened for output to the device LP:.  
 A file FILE2.OP3 is opened on the system device.
2. Channel 3 is set for input from paper tape.  
 Channel 4 is open for input from a file IN1.MAC on device DT1.  
 Channel 5 is open for input from IN2.MAC on device DT2.
3. The stack contains switches and values as follows:

Explanation

2
2102
102515
7

2 switches found in string.  
 switch is B,  
 associated with Channel 4,  
 has no numeric value.  
 second switch is M,  
 associated with Channel 5,  
 negative => has numeric value.  
 numeric value = 7.

If the CSI were called in special mode, the output from the same call would be:



At location specified at .OUTSPC would be a table of descriptors.

```

.OUTSPC: 15270      ;.RAD50  'DK'
          23364      ;.RAD50  'FIL'
          17500      ;.RAD50  'E'
          60137      ;.RAD50  'OP1'
           10        ;LENGTH OF 8 BLOCKS
          46600      ;.RAD50  'LP'
           0         ;NO NAME OR LENGTH SPECIFIED
           0
           0
           0
          75250      ;.RAD50  'SY'
          23364      ;.RAD50  'FIL'
          22100      ;.RAD50  'E2'
          60141      ;.RAD50  'OP3'
           24        ;LENGTH OF 20 (10)
          63320      ;.RAD50  'PR'
           0
           0
           0
          16077      ;.RAD50  'DT1'
          35217      ;.RAD50  'IN1'
           0
          50553      ;.RAD50  'MAC'
          16100      ;.RAD50  'DT2'
          35220      ;.RAD50  'IN2'
           0
          50553      ;.RAD50  'MAC'

```

The stack would be the same as the call for general mode.

### 8.3.5 .DELETE

The .DELETE request deletes the named file from the specified device, and leaves the specified channel free for use. Note that the channel specified in the .DELETE must not be in use when the request is made, or an error will occur (i.e., do a CLOSE if the channel is in use). The file is deleted from the device, and an empty (UNUSED) entry of the same size is put in its place. A DELETE issued to a non-file structured device is ignored. DELETE requires that the handler to be used be resident at the time the .DELETE request is made.

Macro Call:                    .DELETE .channel,.devblk

#### Errors:

An error results if the channel specified is currently in use or the file was not found on the device.

<u>Code</u>	<u>Explanation</u>
0	Channel is active
1	File was not found in the device directory

Example:

This example uses the special mode of CSI to delete files on the system device.

```
START:  .CSISPC #OUTSPC,#DEFEXT,#0
                                         ;TTY DIALOG WAS *DT:FILE
                                         ;SINCE CSI CLOSES CHANNELS 0-10(8)
                                         ;ANY OF THOSE ARE AVAILABLE
      .DELETE 0,#INSPC                   ;DELETE FIRST INPUT FOUND
      BCC START                           ;NO ERRORS
      .PRINT #NOFILE                       ;NOT THERE.
      BR START                             ;RESTART CSI

NOFILE: .ASCIZ "FILE NOT FOUND"
        .EVEN

DEFEXT: .RAD50 "MAC"                     ;INPUT DEFAULT IS .MAC
        .WORD 0,0,0                       ;NO OUTPUT HANDLED.
OUTSPC=.
INSPC=+.36                               ;CSI BUILDS OUTPUT DESCRIPTORS HERE.
                                         ;FIRST INPUT SLOT IN I/O list.
```

INSPC is the pointer to the first input slot in the CSI output table.

### 8.3.6 .DSTATUS

This request is used to obtain information about a particular device.

Macro call:                            .DSTATUS .cblk, .devnam

Where    .cblk            is the 4-word space used to store the status  
                          information.  
          .devnam        is the pointer to the RAD50 device name.

.DSTATUS looks up the device specified by .cblock and, if found, returns four words of status starting at the address specified by .cblk. The four words returned are:

1. Status table entry
2. Size of handler in bytes.
3. Entry point of handler. 0 if non-resident
4. Size of device in 256-word blocks.  
0 if a non-file structured device.

The meaning of each of the four words is:

1. Status word. The status word is broken down into 2 bytes.

Even byte - physical device identifier. This is a number which identifies the device in question. The values now defined are:

```
0 = RK05 disk
1 = DECTape
```

- 2 = cassette
- 3 = Line printer
- 4 = console terminal (LT33, 35, LA30, VT05)
- 5,6 = unused
- 7 = High speed reader
- 10 = High speed punch.

Odd byte - A series of bits describing attributes of the device structure.

Bit 15: 1= File structured device (disk, DECTape)  
0= Non-file device

Bit 14: 1= Read only device

Bit 13: 1= Write only device

Bit 12: 1= Device whose directory is not an RT-11 directory.

2. Handler size. The size of the device handler, in bytes.
3. Entry point. Non-zero implies the handler is now in core. Zero implies it must be .FETCHed before it can be used.
4. Directory size. For file structured devices. The size of the device, in 256-word blocks.

Example:

```

.DSTATUS #CORE, #DEV      ;GET STATUS OF DT1
BCC 1$
HALT                      ;NOT LEGAL DEVICE.
1$: TST CORE              ;FILE STRUCTURED?
   BMI FILSTR             ;BRANCH IF YES
   .
   .
   .
DEV: .RAD50/DT1/          ;NOTE: .DSTATUS ONLY NEEDS THE
   .RAD50/FILE /         ;.RAD50 WORD CONTAINING THE DEVICE
                           ;NAME.
CORE: .BLKW 4            ;STATUS WORDS GO HERE.

```

Errors:

<u>Code</u>	<u>Explanation</u>
0	Device not found in tables.

### 8.3.7 .ENTER

The .ENTER request allocates space on the specified device and creates a tentative entry for the named file. The channel number specified is associated with the file until a .CLOSE request is executed.

Macro Call:                    .ENTER .channel,.devblk,.length

where    .length            is the number of blocks to be allocated to the file being opened. If length is 0, the system allocates half of the largest space currently available. If the length specified is N, a space of N blocks is allocated from the first space greater than or equal to N. If the length field is not specified in a .ENTER macro call, the macro expander uses the equivalent of #0. For example,

```
.ENTER 0,#DEVBLK,#0
and
.ENER 0,#DEVBLK
```

are equivalent.

The file created with an .ENTER is not a permanent file until the .CLOSE on that channel is given. Thus, the newly created file is not available to .LOOKUP or .SAVESTATUS requests. However, it is possible to go back and read data which has just been written into the file by referencing the appropriate block number. When the .CLOSE to the channel is given, any already existing permanent file of the same name is deleted and the new file becomes permanent. Although space is allocated to a file during the .ENTER operation, the actual length of the file is determined when .CLOSE is requested (Refer to .CLOSE).

There may be up to 16 files open on the system at any time. If required, all 16 may be opened for output with the .ENTER function. .ENTER requires that the device handler be in core when the request is made. Thus, a .FETCH is normally executed before a .ENTER can be done. On return, R0 contains the size of the area actually allocated for use.

Errors:

<u>Code</u>	<u>Explanation</u>
0	Channel is in use.
1	In a fixed length request, no space greater than or equal to N was found.

Example:

.ENTER may be used to open a file on a specified device, and then write data from core into that file as follows:

```
.SRESET                           ;MAKE SURE ALL CHANNELS ARE CLOSED.
.FETCH #CORSPC,#FPRT           ;FETCH DEVICE HANDLER
BCC A
HALT                               ;.FETCH ERROR. PROBABLY ILLEGAL
                                  ;DEVICE.
A:    .ENER 0,#FPRT,#0           ;OPEN A FILE ON THE DEVICE SPECIFIED
                                  ;LENGTH 0 WILL GIVE 1/2 OF THE
                                  ;LARGEST EMPTY SPACE NOW AVAILABLE.
BCC B
HALT                               ;FAILED. CHANNEL PROBABLY BUSY. HALT
                                  ;NOW.
```

```

B:      .WRITW 0, #BUFF, #END-BUFF/2, #0
        ;WRITE DATA FROM CORE.  THE SIZE IS
        ;THE NUMBER OF WORDS BETWEEN BUFF AND
        ;END.  WE START AT BLOCK 0.

        BCC C
        HALT
C:      .CLOSE 0
        .EXIT
FPRT:   .RAD50 /DK /
        .RAD50 /FILE EXT/
CORSPC: .BLKW 400
BUFF:   .BLKW 1000
END:

```

#### NOTE

When using the 0 length feature of .ENTER, it must always be kept in mind that only one half of the largest empty is allocated. This can have an important effect in transferring files between devices, particularly DECTape, which have a relatively small capacity. For example, if it is required to transfer a 200-block file to a DECTape on which the largest available empty space is 300<sub>10</sub> blocks, a 0 length transfer will NOT work. Since the enter allocates half the largest space, only 150<sub>10</sub> blocks are really allocated. Thus, an output error will occur during the transfer. If a length of 200 is specified, however, the transfer will proceed without error.

#### 8.3.8 .EXIT

The .EXIT request causes the user program to terminate and returns control to the Keyboard Monitor. Any pending I/O requests are allowed to complete. Any pieces of user code which were currently swapped out are read in again. If part of the user program overlaid KMON, that area is put into scratch blocks, and KMON gains control of the system. .EXIT uses an optional value in R0. If R0=0 when the .EXIT is done, all tentative files on the system are deleted, and all device handlers not part of the resident monitor are released. If R0≠0, all channels remain open, and a keyboard CLOSE command can be used to close any tentative files remaining open. (See CLOSE command in Chapter 2).

Macro Call:                   .EXIT

#### Errors:

.EXIT does not return any errors.

### 8.3.9 .FETCH

The .FETCH request loads device handlers into core from the system device.

Macro Call:                    .FETCH .coradd,.devname

      where .coradd            is the address where the device handler is to be loaded. (The address should never be odd, or the system will not operate properly).

                  .devname    is the pointer to the .RAD50 device name.

The storage address for the device handler is passed on the stack. When the .FETCH is complete, R0 points to the first available location above the handler. If the handler is already in core, R0 keeps the same value as was initially pushed onto the stack. If the argument on the stack is less than 400(8), it is assumed that a handler .RELEAS is being done. After a .RELEAS, a .FETCH must be done in order to use the device again.

Several of the requests require a device handler to be in core for successful operation. These are:

.CLOSE	.READC	.READ
.LOOKUP	.WRITC	.WRITE
.ENTER	.READW	.DELETE
.RENAME	.WRITW	

#### Errors:

<u>Code</u>	<u>Explanation</u>
0	The device name specified does not exist, or there is no handler for that device in the system.

#### Example:

In the following example, the PR and PP handlers are fetched into core in preparation for their use by a program. The program sets aside handler space out of its free core area.

```
.FETCH FREE, #PRNAME           ;FETCH PR HANDLER
BCS     FERR                   ;FETCH ERROR
MOV     R0,R2
.FETCH R2, #PPNAME             ;FETCH PP HANDLER
                              ;IMMEDIATELY FOLLOWING
                              ;PR HANDLER. R0 POINTS
                              ;TO THE TOP OF PR
                              ;HANDLER ON RETURN
                              ;FROM THAT CALL
BCS     FERR                   ;NO PP HANDLER
MOV     R0,FREE                ;UPDATE FREE CORE
                              ;POINTER TO POINT TO
```



Errors:

None

Example:

This example shows the usage of .LOCK, .UNLOCK, and their interaction with the system. It assumes that the configuration is a 16K PDP-11.

In this size system, the USR would normally reside upwards from 63000.

```
.SETTOP #65000                ;THIS CAUSES USR TO BECOME
                                ;NON-RESIDENT
                                ;BRING THE USR INTO CORE
                                ;LOOKUP A FILE ON CHANNEL 0.
.LOCK
.LOOKUP 0,#FILE1
BCC A
HALT                            ;.LOOKUP ERROR. FILE NOT FOUND.
A: .LOOKUP 1,#FILE2            ;ALSO A FILE ON CHANNEL 2.
   BCC B
   HALT                          ;FILE NOT FOUND.
B: .UNLOCK                      ;NOW PUT USER PROGRAM BACK IN CORE
   .
   .
FILE2: .RAD50 "DK "
        .RAD50 "FILE1 "
        .RAD50 "MAC"
FILE2: .RAD50 "DK "
        .RAD50 "FILE2 "
        .RAD50 "MAC"
```

If the .LOCK had not been done prior to the first .LOOKUP, the user program would have been restored after each lookup. With the .LOCK, the user program is initially swapped out, both lookups are performed, and the .UNLOCK causes the user program to be restored. Thus, using .LOCK and .UNLOCK can in certain cases conserve system device motion.

### 8.3.12 .LOOKUP

The .LOOKUP request associates the specified channel number and file for input and output operations. This channel number is in use until a .CLOSE .SRESET, .HRESET, or .SAVESTATUS is executed.

Macro Call:                    .LOOKUP .channel,.devblk

If a file name is not specified and the device is file structured, physical block 0 is assumed as the start of the file. This would allow read/write operations directly to any physical block of a device. File names specified for non-file devices are ignored. The .LOOKUP request assumes the necessary device handler is in core. If it is not, an error results.



Errors:

<u>Code</u>	<u>Explanation</u>
0	Channel already open.
1	File indicated was not found on the device.

Example:

In the following example, the file "DATA.001" is opened for input on device DT3. Channel 7 is used.

```

ERRWD=52
.FETCH #HSPACE,#DT3N           ;LOAD DT HANDLER
BCS FERR                       ;DT3 NOT AVAILABLE
.LOOKUP 7,#DT3N                ;LOOKUP THE FILE
BCC LDONE                      ;FILE SUCCESSFULLY FOUND
TSTB ERRWD                    ;LOOKUP ERROR-WHAT KIND
BNE NFD                        ;FILE NOT FOUND
.PRINT #CAMSG                  ;PRINT "CHANNEL ACTIVE"
HALT
NFD: .PRINT #NFMSG             ;PRINT "FILE NOT FOUND"
HALT
CAMSG: .ASCIZ /CHANNEL ACTIVE/
NFMSG: .ASCIZ /FILE NOT FOUND ;ERROR MESSAGES
DTMSG: .ASCIZ /DT3 NOT AVAILABLE/

FERR: .PRINT #DTMSG           ;FETCH FAILURE
HALT

LDONE: .                        ;PROGRAM CAN NOW
.                                     ;ISSUE READS AND
.                                     ;WRITES TO FILE
.                                     ;DATA.001 VIA
.                                     ;CHANNEL 7

DT3N: .RAD50 "DT3"             ;DEVICE
.RAD50 "DAT"                   ;FILENAME
.RAD50 "A "                    ;FILENAME
.RAD50 "001"                   ;EXTENSION

HSPACE: .=.+400                ;RESERVED SPACE FOR DT
;HANDLER

```

8.3.13 .PRINT

The .PRINT request outputs a message to the console terminal.

Macro Call: .PRINT .msgaddr

where .msgaddr is the address of the ASCIZ string to be printed.

PRINT automatically outputs a carriage return/line feed at the end of the message. Control returns to the user program after the message has been stored in the terminal output buffer.

It is always good practice to follow any group of .ASCIZ strings with a .EVEN directive to MACRO. This ensures that assembly will continue properly.

Errors:

.PRINT returns no errors.

Example:

```
        .PRINT #MSG
        .
        .
MSG:    .ASCIZ "TEXT STRING"
        .EVEN
```

#### 8.3.14 .QSET

The .QSET request is used to make the I/O queue for the Monitor larger, i.e. add available entries to the queue. All I/O under RT-11 is done on a queued basis. Thus, if the I/O traffic is heavy, and there are not sufficient queue entries available, program speed will depend on I/O rates. The general rule to follow is to have the queue contain one more entry than the number of devices which will be operating. (Note... this only applies when using the non-wait I/O modes. Thus the .READW/.WRITW modes do not require a .QSET request at all.)

Macro call:                    .QSET .qaddr,.qleng

Where .qaddr                is the address of the first entry of the new queue area.

.qleng                    is the number of entries to be added. Each queue entry is seven words long; hence the space set aside for the queue should be the appropriate multiple of seven words long.

Each time .QSET is called, a contiguous area of core is assigned to the queue. .QSET may be called as many times as required. The queue set up by multiple .QSET requests is a linked list. Thus, .QSET need not be called with strictly contiguous arguments.

Errors:

No errors are returned.

Example:

```
.QSET #Q1,#5 ;Add 5 elements to the queue
           ;starting at Q1
.QSET #Q3,#3 ;and 3 more at Q3.
.
.
.
Q1: .BLKW 35. ;first queue area (5 elements)
Q3: .BLKW 21. ;second queue area (3 elements)
```

Note that the queue areas for the two calls are not necessarily contiguous. The .QSET request links the specified area into the queue.

### 8.3.15 .RCTRLO

The .RCTRLO request ensures that the console terminal is able to print. A CTRL/O (↑O) struck while output is going to the console inhibits output until either another ↑O is struck, or the program resets the ↑O switch. Thus, a program which has a message which must appear at the console can override the print inhibiting ↑O struck at the keyboard.

Macro Call: .RCTRLO

Errors:

.RCTRLO does not return any errors.

Example:

In the following example, the user program calls the CSI in general mode, then processes the command. When finished, it returns to the CSI for another command line. To make certain that the prompting "" typed by the CSI is not inhibited by a CTRL/O in effect from the last operation, terminal output is re-enabled via a .RCTRLO command prior to the CSI call.

```
START:      .RCTRLO                ;MAKE SURE TT OUTPUT IS
           ;ENABLED
           .CSIGEN #DSPACE,#DEXT,#0 ;CALL CSI-IT WILL TYPE
           ;""
           .
           .
           .
           ;PROCESS COMMAND
           .
           .
           JMP START              ;GET NEXT COMMAND
```

```

DEXT:          0
               0           ;NO DEFAULT EXTENSIONS
               0
               0
DSPACE:        .+.400           ;HANDLER SPACE

```

### 8.3.16 .READ/.WRITE, .READC/.WRITC, .READW/.WRITW

RT-11 provides three modes of I/O (.READ, .READC and .READW). (Note: this discussion applies to .WRITE, .WRITC, and .WRITW as well). The differences are:

**.READW (.WRITW) - Wait mode I/O.**

In this mode, the I/O transfer is initiated and control does not return to the user program until the transfer is complete or an error is detected. On return from this call, the C bit set indicates an error of some type. If no error occurred, the data is in core at the specified address.

**.READ (.WRITE) - Queue transfer and return.**

In this mode, the transfer is queued and returns immediately to the user program to continue processing. No special action is taken when the transfer is done. When the user program needs to access the data read on this channel, a .WAIT request should be issued. This will insure that the data has been read completely. If an error occurred during the transfer, the .WAIT request indicates the error.

**.READC (.WRITC) - I/O with completion functions.**

This type of I/O uses the full flexibility of the PDP-11 and RT-11. The I/O transfer is queued and control returns to the user. When the transfer is completed, control passes to the routine specified when the transfer was first requested. The completion functions thus allow fully overlapped, asynchronous I/O.

The restrictions which must be observed when writing completion functions are:

1. Completion functions cannot issue a request which would cause the USR to be swapped in. They are primarily used for issuing READ/WRITE commands, not for opening or closing files, etc. A fatal monitor error is generated if the USR is called from a completion routine.
2. Completion routines should never reside in the core space which will be used for the USR, since the USR can be interrupted when I/O

terminates, and the completion routine is entered. If the USR has overlayed the routine, control passes to a random place in the USR, with a HALT or error trap the likely result.

3. The routine must be exited via an RTS PC, as it is called from the monitor via a JSR PC, ADDR, where ADDR is the user-supplied address.

### .READ

The .READ request transfers a specified number of words from the specified channel to core. Control returns to the user program immediately after the .READ is initiated.

Macro Call:                    .READ .channel,.buffer,.wcount,.blockn

where .buffer            is the address of the buffer to receive the data read.

.wcount                is the number of words to be read.

.blockn                is the block number to be read relative to the start of the file, not block 0 of the device. The Monitor translates the block supplied into an absolute device block number. The user program normally updates .blockn before it is used again.

### Errors:

<u>Code</u>	<u>Explanation</u>
0	End of file reached on input
1	Hard error occurred on channel
2	Channel is not open.

### .READC

The .READC request transfers a specified number of words from the specified channel to core. Control returns to the user program immediately after the read is initiated. Execution of the user program continues until the READ is complete then control passes to the routine specified in the request. When an RTS PC is executed in the completion routine, control returns to the user program.

Macro call:                    .READC .channel,.buffer,.wcount,.croutine,.blockn

where .buffer            is the address of the buffer to receive the data read.

.wcount is the number of words to be read.

.croutine is the address of the routine to be executed when the READ operation is complete.

.blockn is the block number relative to the start of the file, not block 0 of the device. The Monitor translates the block supplied into an absolute device block number. The user program normally updates .blockn before it is used again.

Error:

<u>Code</u>	<u>Explanation</u>
0	End of file reached on input
1	Hard error occurred on channel
2	Channel is not open.

.READW

The .READW request transfers a specified number of words from the specified channel to core. Control returns to the user program when the read is complete.

Macro Call: .READW .channel,.buffer,.wcount,.blockn

where .buffer is the address of the buffer to receive the data read.

.wcount is the number of words to be read. The number must be positive.

.blockn is the block number relative to the start of the file, not block 0 of the device. The Monitor translates the block supplied into an absolute device block number. The user program normally updates .blockn before it is used again.

Errors:

<u>Code</u>	<u>Explanation</u>
0	End of file reached on input
1	Hard error occurred on channel
2	Channel is not open.

Example:

The following routine illustrates the differences between the three types of read/write requests and is coded in three ways, each using a different mode of monitor I/O. The routine itself is a simple program to duplicate a paper tape.

In the first example, .READW and .WRITW are used. The I/O is completely synchronous, with each request retaining control until the buffer is filled (or emptied).

```

ERRWD=52                                ;ADDRESS OF MONITOR ERROR WORD
START:  .FETCH #HSPACE,#PRNAME           ;LOAD PR HANDLER
        BCS FERR                         ;PR NOT AVAILABLE
        MOV R0,R2
        .FETCH R2,#PPNAME                ;LOAD PP HANDLER IN CORE FOLLOWING
                                           ;PR HANDLER
        BCS FERR                         ;PP NOT AVAILABLE
        .ENTER 0,#PPNAME                 ;OPEN CHANNEL 0 FOR OUTPUT
        BCS ENERR                         ;ERROR
        .LOOKUP 1,#PRNAME                ;OPEN CHANNEL 1 FOR INPUT
        BCS LKERR                         ;LOOKUP ERROR
        CLR R1                            ;START AT BLOCK 0
LOOP:   .READW 1,#BUFF,#400,R1           ;READ A BUFFER LOAD FROM READER
        BCC NOERR                        ;READ WAS SUCCESSFUL
        TST ERRWD                         ;ERROR - WAS IT EOF?
        BEQ EOF                           ;YES - COPY DONE
        HALT                              ;I/O ERROR
NOERR:  .WRITW 0,#BUFF,#400,R1          ;PUNCH THIS BUFFER LOAD
        BCC NOERR1                       ;WRITE WAS SUCCESSFUL
        HALT                              ;I/O ERROR
NOERR1: INC R1                            ;INCREASE BLOCK NUMBER
        BR LOOP                           ;AND GET NEXT BUFFER LOAD
EOF:    .CLOSE 0                          .CLOSE PUNCH
        .CLOSE 1                          ;CLOSE READER
        .EXIT                              ;RETURN TO MONITOR

FERR:   HALT                              ;HANDLER LOAD ERROR
ENERR:  HALT                              ;ENTER ERROR
LKERR:  HALT                              ;LOOKUP ERROR
PRNAME: .RAD50 /PR/                       ;
        0
        0                                ;PR NEEDS NO FILE NAME
        0
PPNAME: .RAD50 /PP/
        0
        0
        0
BUFF:   .=.+1000                          ;256 (10) WORDS
HSPACE:                                ;HANDLER AREA

```

The same routine would be coded using .READ and .WRITE as follows. The .WAIT request is used to determine if the buffer is full or empty prior to its use.

```

START:   ERRWD=52                ;ADDRESS OF MONITOR ERROR WORD
        .
        .
        (same as above)
        .
        .
LOOP:    .READ 1,#BUFF,#400,R1   ;READ A BUFFER LOAD FROM READER
        BCC NOERR               ;RETURNS HERE IMMEDIATELY, TEST
                                       ;FOR ERROR
NOERR:   HALT                   ;ERROR-SHOULD NOT OCCUR IN PR
        .WAIT 1                 ;WAIT FOR BUFFER TO FILL
        BCC NOERR1              ;IT IS FULL-ANY ERRORS?
        TST ERRWD               ;YES-WHAT WAS ERROR?
        BEQ EOF                 ;END OF FILE-COPY DONE.
        HALT                     ;I/O ERROR
NOERR1:  .WRITE 0,#BUFF,#400,R1 ;PUNCH THIS BUFFER LOAD
        BCC NOERR2              ;RETURNS HERE IMMEDIATELY
        HALT                     ;IN THIS EXAMPLE, THERE SHOULD
                                       ;BE NO ERRORS
NOERR2:  .WAIT 0                 ;WAIT FOR BUFFER TO
                                       ;EMPTY
        BCC NOERR3              ;TEST FOR ERRORS ON WRITE
        HALT                     ;I/O ERROR
NOERR3:  INC R1                  ;INCREASE BLOCK NUMBER
        BR LOOP                  ;AND LOOP

EOF:     .
        .
        (same as above)

```

.READ and .WRITE are also often used for double buffered I/O. The basic double-buffering algorithm for input is:

		<u>Explanation</u>	
LOOP → <div style="display: inline-block; vertical-align: middle; border-left: 1px solid black; border-bottom: 1px solid black; width: 100px; height: 80px; margin-left: 10px;"></div>	READ	BUFFER 1	FILL BUFFER 1
	WAIT		WAIT FOR BUFFER 1 TO FILL
	READ	BUFFER 2	START FILLING BUFFER 2
	USE	BUFFER 1	PROCESS BUFFER 1 WHILE BUFFER 2 FILLS
	WAIT		WAIT FOR BUFFER 2 TO FILL
	READ	BUFFER 1	START FILLING BUFFER 1
	USE	BUFFER 2	PROCESS BUFFER 2 WHILE BUFFER 1 FILLS

Correspondingly, the basic double-buffering algorithm for output is:

		<u>Explanation</u>
LOOP → <div style="display: inline-block; vertical-align: middle; border-left: 1px solid black; border-bottom: 1px solid black; width: 100px; height: 100px; margin-left: 10px;"></div>	FILL BUFFER 1	PREPARE BUFFER 1 FOR OUTPUT
	WRITE BUFFER 1	START EMPTYING BUFFER 1
	FILL BUFFER 2	FILL BUFFER 2 WHILE BUFFER 1
		EMPTIES
	WAIT	WAIT FOR BUFFER 1 TO EMPTY
	WRITE BUFFER 2	START EMPTYING BUFFER 2
	FILL BUFFER 1	FILL BUFFER 1 WHILE BUFFER 2
		EMPTIES
	WAIT	WAIT FOR BUFFER 2 TO EMPTY



Finally, this example program can be coded using completion routines via .READC and .WRITC as follows. Once the initial read is performed, the remainder of the I/O is performed by the completion routine.

```

ERRWD=52                ;ADDRESS OF MONITOR ERROR WORD
START:
  .
  .
  .
  (same as previous
  examples)
  .
  .
LOOP:  .READC 1,#BUFF,#400,R1,#RDCOMP ;READ THE FIRST BUFFER LOAD
      BCC NOERR                ;THERE SHOULD BE NO ERROR
      HALT                    ;ERROR
NOERR: BR NOERR                ;IDLE HERE TILL DONE

RDCOMP: .WAIT 1                ;READ IS COMPLETE, CHECK FOR ERROR
       BCC NOERR1             ;NO READ ERROR
       TST ERRWD              ;ERROR-WAS IT EOF?
       BEQ EOF                ;YES-DONE
       HALT

NOERR1: .WRITE 0,#BUFF,#400,R1,#WRCOMP ;WRITE BUFFER
       RTS PC                 ;RETURN FROM COMPLETION ROUTINE

WRCOMP: .WAIT 0                ;WRITE COMPLETE-TEST FOR ERROR
       BCC NOERR2             ;TEST FOR ERROR
       HALT                    ;I/O ERROR
NOERR2: .READC 1,#BUFF,#400,R1,#RDCOMP ;READ NEXT BUFFER
       RTS PC                 ;RETURN
EOF:    .
       .
       .
       (same as above
       two examples)

```

In the above example the .WAIT requests at RDCOMP and WRCOMP return immediately; they are used to detect possible errors in the preceding .READC or .WRITC request, as described in the discussion of .WAIT.

### 8.3.17 .RELEASES

The .RELEASES request removes the handler for the specified device from core. Attempts to release a handler which is not in core, or a handler which is part of the resident monitor are ignored.

Macro Call:                    .RELEASES .devname

where .devname            is the pointer to the .RAD50 device name.

#### Errors:

<u>Code</u>	<u>Explanation</u>
0	Handler name was illegal.

Example:

In the following example, the DECTape handler (DT) is loaded into core, used, then released. For a DECTape system, both RELEAS and FETCH would be null operations.

```
START:          .FETCH #HSPACE,#DTNAME      ;LOAD DT HANDLER
                BCS FERR                    ;NOT AVAILABLE
                .
                .
                .
                Use handler
                .
                .
                .
                .RELEAS #DTNAME            ;MARK DT NO LONGER IN
                .                          ;CORE
                .
                .
                BR START
FERR:           HALT                        ;DT NOT AVAILABLE
DTNAME:         .RAD50 /DT/                ;NAME FOR DT HANDLER
HSPACE:         .                          ;BEGINNING OF HANDLER
                .                          ;AREA
```

### 8.3.18 .RENAME

The .RENAME request causes an immediate change of name of the file specified. The channel specified must not already be assigned, or an error occurs.

Macro Call:                                .RENAME .channel,.devblk

The argument DEVBLK consists of two consecutive .RAD50 strings. For example:

```
                .RENAME 7,#DBLK            ;USE CHANNEL 7 FOR THE
                .                          ;OPERATION.
                BCS RNMERR                  ;ERROR. ORIGINAL FILE DOESN'T
                .                          ;EXIST.
                .                          ;OR CHANNEL ACTIVE.
                .
                .
DBLK:           .RAD50 /DT3/
                .RAD50 /OLDFIL/
                .RAD50 /MAC/
                .RAD50 /DT3/
                .RAD50 /NEWFIL/
                .RAD50 /MAC/
```

The first string represents the file to be renamed and the device it is found on. The second represents the new file name. The second occurrence of the device name DT3 is necessary to proper operation, and should not be omitted.

Errors:

<u>Code</u>	<u>Explanation</u>
0	Channel open.
1	File not found.

Example:

In the following example the file DATA.TMP on DT0 is renamed to DATA.001:

```

START:      .FETCH #HSPACE,#NAMBLK          ;LOAD DT HANDLER
            BCS FERR                        ;DT NOT AVAILABLE
            .RENAME 0,#NAMBLK              ;RENAME FILE
            BCS RERR                        ;FILE NOT FOUND
            .EXIT
FERR:       HALT                            ;FETCH ERROR
RERR:       HALT                            ;FILE NOT FOUND ON RENAME
NAMBLK:     .RAD50 /DT0/
            .RAD50 /DAT/
            .RAD50 /A /
            .RAD50 /TMP/
            .RAD50 /DT0/
            .RAD50 /DAT/
            .RAD50 /A /
            .RAD50 /001/
HSPACE:

```

8.3.19 .REOPEN

The .REOPEN request reassociates the specified channel with a file on which a .SAVESTATUS was performed. The .SAVESTATUS/.REOPEN combination is useful when a large number of files must be operated on at one time. As many files as are needed can be opened with .LOOKUP, and their status preserved with .SAVESTATUS. When data is required from a file, a .REOPEN enables the program to read from the file.

Macro Call:                                    .REOPEN .channel,.cblock  
                   where .cblock                is the address of the core block where the  
   channel status information was stored.

Errors:

<u>Code</u>	<u>Explanation</u>
0	The specified channel is in use. The .REOPEN has not been done.

Example:

Refer to example for .SAVESTATUS

8.3.20 .SAVESTATUS

The .SAVESTATUS request stores five data words into a user specified area of core. These words contain all the information RT-11 requires to completely define a file. When a .SAVESTATUS is done, the data words are placed in core, and the specified channel is again available for use. When the saved channel data is required, the .REOPEN request is used.

.SAVESTATUS can only be used if a file has been opened with .LOOKUP. If .ENTER was used, .SAVESTATUS is illegal, and returns an error.

Macro Call:                    .SAVESTATUS .channel,.cblock  
      where .cblock            is the address of the user core block (5 words) where the channel status information is to be stored.

The five words stored are the five words normally contained in the channel status word table, \$CSW in RMON.

Errors:

<u>Code</u>	<u>Explanation</u>
0	The channel specified is not currently associated with any file, i.e. a previous .LOOKUP on the channel was never done.
1	The file was opened via .ENTER, and a .SAVESTATUS is illegal.

While the .SAVESTATUS/.REOPEN combination is very useful, care must be taken when using it. In particular, the following cases should be avoided:

1. Performing a .SAVESTATUS and then deleting the same file before reopening the file. When the file is deleted, it becomes available as an empty space which could be used by the .ENTER command. If this sequence occurred, the contents of the file supposedly saved would change.

2. The device handler for the required peripheral must be in core for execution of a .REOPEN. If the handler is not in core, when .REOPEN and a .READ/.WRITE is executed, a fatal error is generated.

Example:

One of the more common uses of .SAVESTATUS and .REOPEN is to consolidate all directory access motion and code at one place in the program. All files necessary are opened and saved, then are re-opened one-at-a-time as needed. USR swapping can be minimized by locking in the USR, doing .LOOKUPS as needed using .SAVESTATUS to save the data, and then .UNLOCKing the USR.

In the program segment below, three input files are specified in the command string, then processed one at a time.

```

START:   .CSIGEN #DSPACE,#DEXT,#0 ;GET INPUT FILES
         .SAVESTATUS 3, BLOCK1   ;SAVE FIRST INPUT FILE
         .SAVESTATUS 4, BLOCK2   ;SAVE SECOND FILE
         .SAVESTATUS 5, BLOCK3   ;SAVE THIRD FILE
         .
         .
         .
PROCESS:  MOV #BLOCK1,R0
         .REOPEN 0,R0           ;REOPEN FILE ON
         .                               ;CHANNEL 0
         .
         .
         .READ 0,BUFF,COUNT,BLOCK ;PROCESS FILE ON CHANNEL 0
         .
         .
DONE:    ADD #12,R0             ;POINT TO NEXT SAVESTATUS BLOCK
         CMP R0,#BLOCK3        ;LAST FILE PROCESSED?
         BLOS PROCESS          ;NO-DO NEXT
         .EXIT                 ;YES-FINISHED

BLOCK1:  .WORD 0,0,0,0,0
BLOCK2:  .WORD 0,0,0,0,0       ;CORE BLOCKS FOR SAVESTATUS
BLOCK3:  .WORD 0,0,0,0,0       ;INFORMATION

```

### 8.3.21 .SETTOP

.SETTOP allows the user program to tell RT-11 the highest core address required to do a particular job. With this information, RT-11 determines whether or not a core swap is necessary when the USR is required. For instance, if the program specified an upper limit below the start address of USR, no swapping is necessary, as the USR is not overlaid by the user program. If .SETTOP specifies a high limit greater than the address of the USR, a core swap is required. Section 8.1.4 gives the detail on determining where the USR is in core, and how to optimize the .SETTOP.

On return from .SETTOP, the word at location 50(octal) contains the highest core address allocated for use. If the initial specification was greater than the address of RMON, the address of RMON is returned in word 50.

Macro Call:                    .SETTOP .top  
                   where .top                    is the highest core location to be used by  
   the user program.

Errors:

.SETTOP does not return any errors.

Example:

This example illustrates the initialization code for a hypothetical user program. The program requires 16K of core; in 16K it sets its high location such that swapping will occur, while in configurations greater than 16K it initializes such that the USR/CSI remains in core for faster operation.

```

                HICORE=54                ;ADDRESS OF HI CORE WORD IN JOB
START:          MOV #START,SP            ;COMMAND AREA.
                MOV HICORE,R0           ;INITIALIZE STACK
                ;MOVE STARTING ADDRESS OF RESIDENT
                ;IN R0
                CMP R0,#60000           ;DOES MACHINE HAVE 16K?
                BLO NOCORE              ;NO-PRINT ERROR
                CMP R0,#100000          ;MACHINE HAS AT LEAST 16K
                ;DOES IT HAVE MORE?
                BLO C16K                ;NO-16K MACHINE. SET TOP
                SUB #10000,R0           ;TO JUST UNDER RESIDENT
C16K:          TST -(R0)                ;USR IS 10000(8) LOCATIONS
                ;LAST AVAILABLE LOCATION
                ;TO USER IS 2 LESS THAN
                ;FIRST LOCATION OF MONITOR
                .SETTOP                 ;INFORM MONITOR OF CORE REQUIREMENTS
                .                       ;HIGH LIMIT IS IN R0
                .
                .

```

### 8.3.22 .SRESET

The .SRESET (software reset) request resets certain areas of the system and performs the following functions:

1. Dismisses all non-resident device handlers from core. Any handlers not a part of the RT-11 resident monitor are made non-resident. A handler FETCH must be done before those handlers can be used again.
2. Deletes all tentative files from the system by zeroing file association tables in RMON.

Macro Call                    .SRESET

Errors:

None

Example:

In the example below, .SRESET is used prior to calling the CSI to insure that all handlers are removed from core and the CSI is started with a free handler area.

```
START:   .CSIGEN #DSPACE,#DEXT,#0 ;GET COMMAND STRING
        MOV R0,BUFFER           ;R0 POINTS TO FREE CORE
        .
        .
        .
        (process command)
        .
        .
        .
DONE:    .SRESET                ;RELEASE HANDLERS, DELETE
        .                       ;TENTATIVE FILES
        BR START                ;AND REPEAT PROGRAM

DEXT:    .WORD 0,0,0,0          ;NO DEFAULT EXTENSIONS
BUFFER:  0
DSPACE=  .                     ;START OF HANDLER AREA.
```

If the .SRESET had not been performed prior to the second calling of .CSIGEN, there would be danger that the second command string would result in a handler load at DSPACE, over a handler still resident from the previous command. One .SRESET request takes the place of several .RELEAS requests.

### 8.3.23 .TTYIN/TTINR

Both of these requests cause a character to be transferred from the console terminal into R0, right justified. The difference in the calls is that if a character is not available currently, .TTYIN will wait until one is available, while .TTINR does not automatically wait.

```
Macro call:   .TTYIN .char
              or
              .TTINR
```

where .char is the location where the character in R0 will be stored. If not specified, the character is left in R0.

If the carry bit is set when execution of the TTINR request is completed, it indicates that no character was available; the user has not yet typed a valid line.

There are two modes of doing console terminal input. This is governed by bit 12 of the job status word. If bit 12 = 0, normal I/O is performed. In this mode, the following conditions apply:

- a. Monitor echoes all characters typed.
- b. CTRL/U (^U) and RUBOUT perform line deletion and character deletion, respectively.
- c. A carriage return must be struck before characters on the current line are available to the program.

If bit 12 = 1, the console is in special mode. The effects are:

- a. The monitor does not echo characters typed except for CTRL/C and CTRL/O.
- b. CTRL/U and RUBOUT do not perform functions.
- c. Characters are immediately available to the program.

In special mode, the user program must echo the characters received. However, CTRL/C and CTRL/O are acted on by the monitor in the usual way. Bit 12 in the JSW must be set by the user program. This bit is cleared when control returns to RT-11.

Errors:

None

Example:

Refer to example for TTYOUT/TTOUTR.

#### 8.3.24 .TTYOUT/.TTOUTR

These requests cause a character to be transmitted from R0 to the console terminal. The difference, like the .TTYIN/.TTINR requests, is that if there is no room for the character in the Monitor's buffer, the .TTYOUT request waits for room before proceeding. The .TTOUTR does not wait for room, and the character in R0 is not output.

Macro call:                    .TTYOUT .char  
                                      or  
                                      .TTOUTR

where .char                    is the location containing the character to be loaded in R0 and printed. If not specified, the character in R0 is printed.

If the carry bit is set when execution of the .TTOUTR or .TTINR request is completed, it indicates that there is no room in the buffer and no character was output.

The .TTINR and .TTOUTR requests have been supplied as a help to those users who do not wish to suspend program execution until a console operation is complete. With these modes of I/O, if a no-character or no-room condition occurs, the user program can continue processing and try the operation again at a later time.

Error:

No errors are returned.



Example:

As an example of the various terminal requests, the following program is coded in two ways. The program itself accepts a line from the keyboard, then repeats it on the terminal.

The first example uses .TTYIN and .TTYOUT, which are synchronous. The Monitor retains control till both requests are satisfied, hence there is not time available for any other processing while waiting.

```
START:    MOV #BUFFER,R1          ;POINT R1 TO BUFFER
          CLR R2                  ;CLEAR CHARACTER COUNT
INLOOP:   .TTYIN (R1)+           ;READ CHAR INTO BUFFER
          INC R2                  ;BUMP COUNT
          CMPB -1(R1),#12        ;WAS LAST CHAR=LF?
          BNE INLOOP            ;NO-GET NEXT
          MOV #BUFFER,R1        ;YES-POINT R1 TO BUFFER
OUTLOOP:  .TTYOUT (R1)+         ;PRINT CHAR
          DEC R2                  ;DECREASE COUNT
          BEQ START             ;DONE IF COUNT=0
          BR OUTLOOP
```

Rather than wait for the user to type something at INLOOP or the output buffer to have available space at OUTLOOP, the routine can be recoded using .TTINR and .TOUTR as follows:

```
START:    MOV #BUFFER,R1          ;POINT R1 TO BUFFER
          CLR R2                  ;CLEAR COUNT
INLOOP:   .TTINR                 ;GET CHAR FROM TERMINAL
          BCS NOCHAR            ;NONE AVAILABLE
CHRIN:    MOV R0,(R1)+           ;PUT CHAR IN BUFFER
          INC R2                  ;INCREASE COUNT
          CMPB R0,#12           ;WAS LAST CHAR=LF?
          BNE INLOOP            ;NO-GET NEXT
          MOV #BUFFER,R1        ;YES-POINT R1 TO BUFFER
OUTLOOP:  MOV R0,(R1),R0        ;PUT CHAR IN R0
          .TOUTR                 ;TYPE IT
          BCS NOROOM            ;NO ROOM IN OUTPUT BUFFER
CHRROUT:  DEC R2                  ;DECREASE COUNT
          BEQ START             ;DONE IF COUNT=0
          INC R1                  ;BUMP BUFFER POINTER
          BR OUTLOOP            ;AND TYPE NEXT
NOCHAR:   .
          .
          .
          (code to be executed
           while waiting)
          .
          .
          .
          .TTINR                ;PERIODIC CHECK FOR
          .                       ;CHARACTER AVAILABILITY
          BCC CHRIN             ;GOT ONE
          .
          .
          .
```

```

NOROOM:      .
              .
              .
              (code to be executed
              while waiting)
              .
              .
              .
MOV B (R1),R0      ;PERIODIC ATTEMPT TO TYPE CHARACTER
.TTYOUTR
BCC CHROUT        ;SUCCESSFUL
              .
              .

```

### 8.3.25 .UNLOCK

The .UNLOCK request releases the User Service Routine from core. The USR must previously have been loaded with a .LOCK request. If the program does not require the USR to be swapped, this request is ignored.

Macro Call:                    .UNLOCK

Errors:

None

Example:

Refer to the .LOCK example.

### 8.3.26 .WAIT

The .WAIT request suspends further program processing until all input/output requests on the specified channel are completed. The .WAIT request combined with the .READ/.WRITE I/O modes makes double-buffering through RT-11 a very simple process.

In addition to waiting for I/O to complete, .WAIT conveys information through its error return. .WAIT gives an error when the channel specified is not associated with any file. Thus, .WAIT can be used to determine the active/inactive status of a channel. Also, if the last I/O operation encountered a hard error, the .WAIT error return is taken.

Macro Call:                    .WAIT .channel

Errors:

<u>Code</u>	<u>Explanation</u>
0	Channel specified is not open.
1	Hardware error found on specified channel during last I/O operation.

These error codes make the .WAIT request useful in checking channel status.

Example:

For an example of .WAIT used for I/O synchronization, see example for .READ.

An example of .WAIT used for error reporting is included in the example for .READC.

Another example of the use of .WAIT for error detection is its use in conjunction with .CSIGEN to determine which file fields in the command string have been specified. For example, a program such as MACRO might use the following code to determine if a listing file is desired.

```
      .
      .
      .CSIGEN #DSPACE,#DEXT,#0      ;PROCESS COMMAND STRING
      .WAIT 0                        ;CHECK FOR FILE IN FIRST FIELD
      BCS NOBINARY                  ;NO BINARY DESIRED
      .
      .
      .WAIT 1                        ;CHECK FOR LISTING SPECIFICATION
      BCS NOLISTING                 ;NO LISTING DESIRED
      .
      .
      .WAIT 3                        ;CHECK FOR INPUT FILE OPEN
      BCS ERROR                     ;NO INPUT FILE.
      .
      .
      .
```

8.3.27 .WRITC

The .WRITC request transfers a specified number of words from core to a specified channel. Control returns to the user program immediately after the request is queued. Execution of the user program continues until the .WRITE is complete then control passes to the routine specified in the request. When an RTS PC is encountered in the routine, control returns to the user program.

Macro Call:            .WRITC .channel,.buffer,.wcount,.croutine,.blockn

where .buffer        is the address of the core buffer to be used for output.

          .wcount     is the number of words to be written.

          .croutine   is the address of the routine to be executed when the WRITE operation is complete.

          .blockn     is the number relative to the start of the file, not block 0 of the device. The monitor translates the block supplied into an absolute device block number. The user program normally updates .blockn before it is used again.

Errors:

<u>Code</u>	<u>Explanation</u>
0	End of file on output. Tried to write outside limits of file.
1	Hardware error occurred.
2	Specified channel is not open.

Example:

Refer to .READ/.READC/.READW example.

8.3.28 .WRITE

The .WRITE request transfers a specified number of words from core to the specified channel. Control returns to the user program immediately after the request is queued.

Macro Call:            .WRITE .channel,.buffer,.wcount,.blockn

where .buffer        is the address of the core buffer to be used for output.

          .wcount     is the number of words to be written.

          .blockn     is the number of the block to be written.

Errors:

<u>Code</u>	<u>Explanation</u>
0	Attempted to write past end of file.
1	Hardware error.
2	Channel was not opened.

Example:

Refer to .READ/.READC/.READW example.

### 8.3.29 .WRITW

The .WRITW request transfers a specified number of words from core to the specified channel. Control returns to the user program when the write is complete.

Macro Call:                    .WRITW .channel,.buffer,.wcount,.blockn

      where .buffer            is the address of the buffer to be used for output.

              .wcount            is the number of words to be written. The number must be positive.

              .blockn            is the number of the block to be written.

#### Errors:

<u>Code</u>	<u>Explanation</u>
0	Attempted to write past EOF.
1	Hardware error.
2	Channel was not opened.

Example:

Refer to .READ/.READC/.READW example.



## CHAPTER 9

### EXPAND UTILITY PROGRAM

EXPAND is an RT-11 system program which processes the macro references in a MACRO source file. EXPAND accepts a subset of the complete macro language and produces an output file in which all legal macro references are expanded into macro-free source code. EXPAND is normally used with ASEMBL, the 8K assembler of RT-11. (Refer to Chapter 10.)

#### 9.1 LANGUAGE

EXPAND simply copies its input files to its output file unless it encounters any of the following directives (also see Chapter 5 MACRO Assembler for more information about these directives):

1. `.MCALL` directs EXPAND to search the file SY:SYSMAC.SML to find the macro names listed in the `.MCALL` directive. If the macro names are found, EXPAND stores their definitions into its internal tables from the file SYSMAC.
2. `.MACRO` directs EXPAND to copy a macro definition from the user's input file into the internal tables.
3. `.name` if `.name` is the name of a macro defined in either a `.MCALL` or `.MACRO` directive, then `.name` will be expanded according to the definition stored for it in the EXPAND internal tables.
4. `.ENDM` if encountered while storing a macro definition, the `.ENDM` directive terminates the definition. It is not recognized outside macro definitions.

#### 9.2 RESTRICTIONS

Unlike the full macro assembler (MACRO), EXPAND will only expand macros that observe the following restrictions:

1. The following directives may not be used:

- `.ERROR`
- `.IF DIF`
- `.IF IDN`
- `.IRP`
- `.IRPC`
- `.MEXIT`
- `.NARG`
- `.NCHR`
- `.NTYPE`
- `.PRINT`
- `.REPT`

1. Macros cannot be nested. Recursive macros that call themselves directly or indirectly are illegal and cause an error message.
2. Macros cannot be redefined. Once a name has been used for a macro name, it cannot be used again in the program for a macro or symbol name.
3. Macro names must begin with a dot (.). If the dot is missing, an error message is printed.
4. Dummy argument names must begin with a dot (.). Such names can only be used as dummy argument names in the macro but can be used for other purposes outside of the macro.
5. The backslash operator is not available.
6. Automatically created symbols are not available.
7. No more than 30 arguments may be used in any MACRO directive.

### 9.3 OPERATION

To run EXPAND, type:

```
.R EXPAND
```

in response to the RT-11 Monitor's dot. EXPAND responds with an asterisk (\*) indicating that it is ready to accept a command string. A command string must be of the following form:

```
*OFILE=IFILE1,IFILE2,...IFILE6
```

IFILE2 through IFILE6 are optional. Each file specification follows the general RT-11 command string syntax dev:filnam.ext. If dev: is not specified, then DK: is assumed. If an extension is not specified for the output file, .PAL is assumed; for any input file without a specified extension, .MAC is used. If dev: is not specified for IFILE2 through 6, it is the same as for the previous IFILE.

EXPAND copies the specified input file to the specified output file until a macro directive is encountered. EXPAND then changes the macro directive to a comment by inserting a semicolon so that it will not be seen later by the assembler (usually ASEMBL).

If the directive is .MCALL, EXPAND searches the system macro library (SYSMAC.SML) for the requested macro definitions. The requested definitions are then included in the user's program in the order in which they are found in the library.

For the .MACRO directive, EXPAND reads each line following the directive up to the next .ENDM directive. Each line is stored in the internal definition table and then changed to a comment in the output file so that it is not processed later by the assembler. Also, any occurrence of a macro argument name within the definition is flagged internally so that it can be replaced by the real argument value whenever the macro is later referenced.



For macro references, EXPAND locates the stored macro definition in its internal tables, binds the actual argument values to the argument names, and changes the macro reference to a comment line. It then begins copying the stored definition to the output file. Whenever a macro argument name is encountered in the definition, it is replaced by the corresponding actual argument value.

Examples:

The following are examples of input and corresponding output of EXPAND:

<u>INPUT</u>	<u>OUTPUT</u>
<pre> R1=%1 SP=%6 .MACRO .CALL .SUBR JSR PC, .SUBR .ENDM A: MOV R1, -(SP) B: .CALL SQRT  .MCALL .LOOKUP, .READ  .CSECT MAIN START: MOV #STACK, SP       .LOOKUP 0, #INBLK  CLR R1 ;BLOCK NUMBER       .READ 0, #BUFR, #256., R1  HALT </pre>	<pre> R1=%1 SP=%6 ; .MACRO .CALL .SUBR ; JSR PC, .SUBR ; .ENDM A: MOV R1, -(SP) B;; .CALL SQRT    JSR PC, SQRT ; .MCALL .LOOKUP, .READ ; .MACRO .LOOKUP .CHANNEL, .DEVBLK ; .IF NB .DEVBLK ; MOV .DEVBLK, %↑00 ; .ENDC ; EMT↑ 020+↑0&lt;.CHANNEL&gt; ; .ENDM ; .MACRO .READ .CHANNEL, .BUFFER, ; .WCOUNT, .BLOCKN ; .IF NB .BLOCKN ; MOV .BLOCKN, %↑00 ; .ENDC ; MOV #↑01, -(%↑06) ; MOV .WCOUNT, -(%↑06) ; MOV .BUFFER, -(%↑06) ; EMT↑ 0200+↑0&lt;.CHANNEL&gt; ; .ENDM .CSECT MAIN START: MOV #STACK, SP ; .LOOKUP 0, #INBLK ; .IF NB #INBLK ; MOV #INBLK, %↑00 ; .ENDC EMT↑ 020+↑0&lt;0&gt; CLR R1 ;BLOCK NUMBER ; .READ 0, #BUFR, #256., R1 ; .IF NB R1 ; MOV R1, %↑00 ; .ENDC ; MOV #↑01, -(%↑06) ; MOV #256., -(%↑06) ; MOV #BUFR, -(%↑06) ; EMT↑ 0200+↑0&lt;0&gt; HALT </pre>

## 9.4 ERROR MESSAGES

The following messages are caused by fatal errors detected by EXPAND. They will print on the console terminal and cause EXPAND to restart:

<u>Message</u>	<u>Explanation</u>
?INPUT ERROR?	Hardware error in reading an input file.
?MISSING END IN MACRO?	End of input was encountered while storing a macro definition: probably missing an .ENDM.
?INSUFFICIENT CORE?	Not enough memory to store macro definitions.
?WRONG NUMBER OF OUTPUT FILES?	There must be exactly one output file.
?NO INPUT FILE?	There must be at least one input file.
?BAD SWITCH?	An unrecognized command string switch was specified.
?OUTPUT DEVICE FULL?	No room to continue writing output--try to compress the device with PIP.

The following errors are non-fatal but indicate that something is wrong in the input file(s). These errors appear in the output file as a line of the following form:

?\*\*\* ERROR \*\*\* message

After each run of EXPAND, the total number of non-fatal errors is printed on the console terminal.

<u>Message</u>	<u>Explanation</u>
LINE TOO LONG	A line has become longer than 132 characters.
MISSING DOT	A macro name or argument name doesn't begin with a dot.
SYNTAX	A macro directive is not constructed correctly.
NESTED MACROS	A macro is being defined or invoked within another macro.
NO NAME	A macro definition has no name.
MACRO ALREADY DEFINED	A macro was defined more than once.

<u>Message</u>	<u>Explanation</u>
MACRO(S) NOT FOUND	Macros listed in an .MCALL statement were not found in SYSMAC.SML (make sure SYSMAC.SML is present on system).
TOO MANY ARGS	A macro directive has more than 30 arguments.
MISSING COMMA IN MACRO ARG	Found spaces or tabs within a macro argument; try using brackets around the argument, e.g. <arg with spaces>.
BAD MACRO ARG	The macro argument is not formatted correctly.
NAME DOESN'T MATCH	Optional name given in .ENDM directive does not match name given in corresponding .MACRO directive.



## CHAPTER 10

### ASEMBL, THE 8K ASSEMBLER

ASEMBL is designed for use on an 8K RT-11 system (or larger systems where system table space is critical) and is a subset of the RT-11 MACRO assembler described in Chapter 5. ASEMBL has much the same features as MACRO but

MACRO directives (.MACRO, .MCALL, .ENDM, .IRP, etc.) are not recognized,

DATE is not printed in listings,

wide line printer output is not available,

there is no lower case mode,

there is no enable/disable punch directive,

there are no floating point directives, and

there are no local symbols or local symbol blocks.

Many of the macro features are supported by the EXPAND program (refer to Chapter 9).

#### 10.1 OPERATING PROCEDURES

ASEMBL is loaded with the RT-11 Monitor R Command as follows:

```
.R ASEMBL
```

followed by the RETURN key. ASEMBL responds with an asterisk and waits for specification of the output and input files in the standard RT-11 format as follows:

```
*object,listing=source1,...,source6
```

where: object is a binary object file output by ASEMBL

listing is the assembly listing file containing the assembly listing and symbol table.

```
source1,...,source6
```

are the ASCII source files containing the ASEMBL source program(s). A maximum of six source files is allowed.

A null specification in any of the file fields signifies that the associated input or output file is not desired. ASEMBL file specifications follow the standard RT-11 convention: dev:filnam.ext.

The default value for each file specification is noted below:

	<u>dev</u>	<u>filnam</u>	<u>ext</u>
object	DK	None	.OBJ
listing	device used for object output	None	.LST
source1, ..., sourceN	device used for last source file specified or DK.	None	.PAL

Table 10-1 lists the RT-11 macro directives which are not available in ASEMBL.

Table 10-1  
Directives not Available in ASEMBL

Directive	Explanation
.MACRO } .ENDM } .MEXIT } .MCALL }	Macros cannot be defined in ASEMBL.
.NCHR	The number of characters in an argument cannot be obtained with a macro.
.NARG	The number of arguments in a macro cannot be obtained with a macro.
\ (backslash)	Symbols used as macro arguments can not be passed as a numeric string.
.ERROR	Messages cannot be flagged with a P error code output as part of the assembly listing. Comment lines can be used to replace .ERROR.
.IF IDN } .IF DIF }	Strings cannot be compared.
.IRP } .IRPC }	Indefinite repeat blocks cannot be created.
.NTYPE	A macro cannot be modified based on the addressing mode of an argument.
.PRINT	Messages cannot be output as part of the assembly listing. Comment lines can be used to replace .PRINT.
.REPT	A block of code cannot be duplicated a number of times in line with other source code using a directive.
.LIST ME } .NLIST ME } .LIST MEB } .NLIST MEB } .LIST MD } .NLIST MD } .LIST MC } .NLIST MC }	These directives have no effect.
.LIST TTM } .NLIST TTM }	Teletype mode is standard and cannot be changed.
.ENABL LC } .DSABL LC }	All lower case ASCII input is converted to upper case.

(Continued on next page)

Table 10-1 (Cont.)  
 Directives not Available in ASEMBL

Directive	Explanation
.ENABL LSB .DSABL LSB	Local symbols and local symbol blocks are not available in ASEMBL.
.ENABLE PNC .DSABL PNC	
.ENABL FPT .DSABL FPT F .FLT2 .FLT4	
	Binary output is always enabled.
	Floating point directives are not available.

## 10.2 ERROR MESSAGES

The system error messages output for ASEMBL are in short form.

<u>Abbreviation</u>	<u>Explanation</u>
?BSW?	The switch specified was not recognized by the program.
?CORE?	There are too many symbols in the program being assembled. Try dividing program into separately-assembled subprograms.
?NIF?	No input file was specified and there must be at least one input file.
?ODF?	No room to continue writing output--try to compress device with PIP.
?TMO?	Too many output files were specified.



## APPENDIX A

### RT-11 SYSTEM BUILD

This appendix describes

1. System build procedure from each of the possible distribution media.
2. Customization of the system for special hardware (LA-30's, Line printers other than 80 columns, VT05B)
3. Optimizations which can be affected at system build time.
4. Assembly and linking instructions for system components.

#### NOTE

This Appendix assumes the user has read and is familiar with this entire manual; especially Chapter 2 (the Monitor), Chapter 4 (PIP), Appendix N (PIPC), and Chapter 7 (LINK). For step-by-step instructions for building your system the first time, refer to the RT-11 System Demonstration Package (DEC-11-ORCPA-A-D)

#### A.1 BUILDING RT-11 SYSTEMS

RT-11 is designed so that the monitor and device handlers which comprise the system are files on the systems device. These files (called system files) all have the extension .SYS, and can be manipulated between devices just as any other RT-11 core image file.

The running version of the monitor must be named MONITOR.SYS; other versions of the monitor may reside on the systems device, but they must be named something other than MONITOR.SYS. The handlers for the system must be named xx.SYS, where xx is the device mnemonic as used in command strings. For example, the high speed reader handler must be named PR.SYS. There may be many versions of a given handler on the systems device, but the one that is desired must be named as above.

Once copies of the system files have been obtained, the procedure for building an RT-11 system consists of the following basic steps:

1. Initializing the target device with an RT-11 directory.
2. Transferring the appropriate monitor file to the target device and giving it the name MONITOR.SYS.
3. Transferring the appropriate handler files to the target device.
4. Writing the appropriate bootstrap on the target device.
5. Transferring the rest of the system components (EDIT, LINK, etc.) to the target device.

After STEP 4 above, the target device may be bootstrapped, and the remainder of the build procedure may be carried out while executing the system off the new (and perhaps faster) systems device. Because the above build steps involve standard RT-11 file operations, system programs are used for the build procedure. When building from DECTape, PIP is used; from cassette, PIPC is used; and from paper tape, LINK is used.

#### A.1.2 DECTape from DECTape and Disk from DECTape

On DECTape, RT-11 is distributed as a "ready-to-run" DECTape system. When bootstrapped (as described in Chapter 2, Section 2.1), the DECTape system is running and may be used to build other DECTape and Disk systems. The following files on the distributed system DECTape (DEC-11-ORTSA-A-UC) have special significance.

MONITOR.SYS	This is a DECTape monitor, and is used as the monitor file for other DECTape systems.
RKMON.SYS	This is an RK11 monitor, and becomes the file MONITOR.SYS for disk systems after it has been transferred.
RK.SYS	This is an RK11 device handler, and allows RT-11 to read and write RK11 disks while running a DECTape system.
DT.SYS	This is a TC11 DECTape handler which allows RT-11 to read and write DECTapes while running disk systems.
PR.SYS	This is the high-speed reader (PC-11) handler.
PP.SYS	This is the high-speed punch (PC-11) handler.
TT.SYS	This is the general terminal handler.
LP.SYS	This is the 80 column line printer (LP-11) handler.
LP.132	This is the 132 column line printer (LP-11 or LS-11) handler. If your configuration contains a 132 column printer, this handler can be used in place of the 80 column handler by renaming it to LP.SYS.

To build another DECTape system (on Unit n) from a running DECTape system, the following set of commands (commands illustrated in this appendix are terminated with the carriage return key) to PIP can be used:

#### Explanation

.R PIP	
*DTn:/Z	Initialize the new
DTn/z are you sure?y	DECTape
*DTn:A=DT0:/S	Copy all files from DT0 to DTn.
*DTn:A=DT0:MONITR.SYS/U	Write the hardware bootstrap on DTn.

DTn now is a ready-to-run copy of the system DECTape.

In the above, the system files were copied to DTn: via the /S option in PIP; in actuality, any method of copying the files to the new device would have sufficed. For example; the command

```
*DTn:A=DT0:/S
```

could have been replaced by

```
*DTn:*.*=DT0:*/S/X
```

or

```
*DTn:MONITR.SYS=MONITR.SYS/Y
```

```
*DTn:LP.SYS=LP.SYS/Y
```

.

.

.

etc. until all desired files were transferred.

To build a disk system from a running DECTape system, use the following commands:

	<u>Explanation</u>
.R PIP	
*RK:/Z	Initialize the disk.
RK:/Z ARE YOU SURE?Y	
*RK:*.*=DT0:*/X/Y	Copy the DECTape files onto disk.
*RK:DTMON.SYS=RK:MONITR.SYS/Y/R	Rename the DECTape monitor on the disk to DTMON.SYS
*RK:MONITR.SYS=RK:RKMON.SYS/Y/R	Rename the disk monitor on the disk to MONITR.SYS
*RK:A=RK:MONITR.SYS/U	Write the system bootstrap on the disk.

The disk may now be bootstrapped.

Note that in the above examples, all files were copied from the running system to the system being built. Although this is the common practice, it is not necessary. What is required is that the following elements be transferred.

1. A monitor file
2. The handlers for the desired devices.
3. The hardware bootstrap
4. Those programs and files which will be used with the new system.

When building a system for a configuration which contains only LA30 and DECTape, it is wise to avoid transferring any of the unrequired handler files (DT is the system handler in the monitor) as they require space on the system DECTape, yet serve no purpose. Users of 8K machines may choose to build systems without the file MACRO.SAV, while EXPAND.SAV and ASEMBL.SAV could be eliminated from systems with 16K or more of memory.

### A.1.3 Disk from Cassette

On cassette, RT-11 is distributed as a series of RT-11 files on several cassettes, each cassette labeled DEC-11-ORTSA-A-TCn. The following files on the system cassettes have special significance.

**CBUILD.SYS**      CBUILD is the special system-build version of PIPC which is loaded via the cassette bootstrap. This program is used to initialize the disk and transfer the remaining files from cassette to disk.

**MONITR.SYS**      The RT-11 RK11 monitor file.

The remaining .SYS files are the handler files as described for the DECTape system in A.1.2.

To build an RT-11 system from cassette, perform the following operations:

1. Mount an RT-11 cassette which has the file CBUILD.SYS as the first file on unit 0.
2. Bootstrap cassette unit 0 as follows:
  - A. Load and start the system bootstrap loader (called CBOOT). This can be done in one of two ways:
    - a. If the system has a hardware bootstrap, enter 173300 in the Switch Register, press LOAD ADDR and START. (Step b may be ignored).
    - b. If no hardware bootstrap is available, CBOOT must be manually loaded and started by the user. Two versions of CBOOT are provided. The standard version is the version used in the hardware bootstrap and consists of the 28 words listed in Table A-1.

A shorter (20 word) version called QCBOOT may optionally be loaded by the user. This version does not provide some of the error checking and handling which the longer CBOOT does, but allows a faster means of manually booting the system. The binary instructions are listed in the following table:

Table A-1

CBOOT (QCBOOT) Instructions

	CBOOT	QCBOOT
Location	Contents	Contents
001000	012700	012700
001002	177500	177500
001004	005010	005010
001006	010701	010701
001010	062701	062701
001012	000052	000034

(Continued on the next page)

Table A-1 (Cont)

CBOOT (QCBOOT) Instructions

	CBOOT	QCBOOT
Location	Contents	Contents
001014	012702	112102
001016	000375	112110
001020	112103	032710
001022	112110	100240
001024	100413	001775
001026	130310	100001
001030	001776	005007
001032	105202	005202
001034	100772	100770
001036	116012	116012
001040	000002	000002
001042	120337	000766
001044	000000	017775
001046	001767	002415
001050	000000	
001052	000755	
001054	005710	
001056	100774	
001060	005007	
001062	017640	
001064	002415	
001066	112024	

After the bootstrap has been manually loaded (using the Switch Register, LOAD ADDR, and DEP keys), set 001000 in the switches, press LOAD ADDR and START.

At this point the RUN lamp should be lit and the System Cassette should begin to move.

3. CBUILD will print the following on the console terminal.

```
CBUILD Version number
*
```

CBUILD is a stand-alone version of PIPC. Use the following set of commands (or their equivalent) to build the disk system.

	<u>Explanation</u>
*RK:/Z ARE YOU SURE? Y	Initialize disk directory.
=CTn:MONITR.SYS/Y	Write monitor file on disk.
*RK:A=CTn:MONITR.SYS/U	Write bootstrap on disk.
*RK:PIPC.SAV=CTn:PIPC.SAV	Put copy of PIPC on disk.
*RK:/O	Boot the disk system.

RT-11 is now running off the disk and PIPC may be used in the normal fashion to copy the rest of the desired files from cassette to disk. Note that no devices other than disk or cassette can be used until their handler files have been added to the disk and the system has been rebooted.

#### A.1.4 Disk from Paper Tape

On paper tape, RT-11 is distributed as object modules on paper tape. Two of the tapes (PTBUILD Tape 1 and 2) are used to place the monitor and Linker on the disk. The disk system is then started, and the Linker is used to link PIP from paper tape onto the disk. Once linked, PIP is used to copy the remaining paper tapes onto the disk, where they can be linked to complete the system.

The following RT-11 paper tapes have special significance:

DEC-11-ORPBA-A-PB1  
PT BUILD Tape 1

These are the paper  
tape build tapes.

DEC-11-ORPBA-A-PB2  
PT BUILD Tape 2

To build an RT-11 system from paper tape, perform the following operations:

1. Load the Bootstrap Loader at 37744, then use it to load the absolute loader.
2. Using the Absolute Loader, load PT BUILD Tape 1 (DEC-11-ORPBA-A-PB1). It self-starts and prints:

PT BUILD Version number

3. There is a 10-15 second pause, after which PT BUILD prints  
PLACE SECOND TAPE IN READER.  
STRIKE ANY CHARACTER TO CONTINUE.
4. Place the tape PT BUILD Tape 2 (DEC-11-ORPBA-A-PB2) into the reader, then strike any character to start the tape.

There is a slight pause, after which the following is printed.

DISK BUILD COMPLETE.

RT-11 version number

.

During the build procedure, only a

WRITE FAILED

error message has exact meaning. If encountered, check to make sure the system disk is not write protected. Any other error indicates a hardware or disk problem.

5. Link PIP as follows:

```
.R LINK
*PIP=PR:
↑↑*
```

For each occurrence of the prompting "↑", place the tape PIP.OBJ (DEC-11-ORPPA-A-PR) in the reader, then strike a character to read the tape. Type CTRL/C when the second "\*" is printed.

6. Run PIP to copy the remaining tapes on the system disk. For example:

<u>COMMAND</u>	<u>USE TAPE</u>
.R PIP	
*PATCH.OBJ=PR:/B	(PATCH.OBJ DEC-11-ORPAA-A-PR)
↑*EDIT.OBJ=PR:/B	(EDIT.OBJ DEC-11-ORTEA-A-PR)
↑*ODT.OBJ=PR:/B	(ODT.OBJ DEC-11-ORODA-A-PR)
↑*TT.OBJ=PR:/B	(TT.OBJ DEC-11-ORTTA-A-PR)
↑*LP.OBJ=PR:/B	(LP.OBJ DEC-11-ORTLA-A-PR)
↑*PP.OBJ=PR:/B	(PP.OBJ DEC-11-ORTPA-A-PR)
↑*PREXEC.OBJ=PR:/B	(PREXEC.OBJ DEC-11-OREXA-A-PR1)
↑*PREPAS.OBJ=PR:/B	(PREPAS.OBJ DEC-11-OREXA-A-PR2)
↑*SMEXEC.OBJ=PR:/B	(SMEXEC.OBJ DEC-11-ORTAA-A-PR1)
↑*SMMAC.OBJ=PR:/B	(SMMAC.OBJ DEC-11-ORTAA-A-PR2)
↑*SMPST.OBJ=PR:/B	(SMPST.OBJ DEC-11-ORTAA-A-PR3)
↑*RTEXEC.OBJ=PR:/B	(RTEXEC.OBJ DEC-11-ORMAA-A-PR1)
↑*RTMAC.OBJ=PR:/B	(RTMAC.OBJ DEC-11-ORMAA-A-PR2)
↑*RTPST.OBJ=PR:/B	(RTPST.OBJ DEC-11-ORMAA-A-PR3)
↑*SYSMAC.SML=PR:/B	(SYSMAC.SML DEC-11-ORSYA-A-PA)

PREXEC.OBJ and PREPAS.OBJ are the object modules for EXPAND. SMEXEC.OBJ, SMMAC.OBJ and SMPST.OBJ are linked for ASEMBL, while RTEXEC.OBJ, RTMAC.OBJ and RTPST.OBJ are the MACRO object modules. The object modules with 2 character names are the handlers for the corresponding devices.

If a specific component is not required, it is not necessary to transfer the corresponding tapes.

7. Run Link to generate the handler .SYS files and program .SAV files as described later in this appendix.
8. Reboot the system (with PIP if desired). If desired, the .OBJ files (except for ODT) can be deleted as they are no longer required.

## A.2 CUSTOMIZATION FOR SPECIAL HARDWARE

### A.2.1 High Baud Rate Serial Console Devices

The serial LA30 requires that filler characters follow each carriage return; the 600, 1200 and 2400 baud VT05's require that filler characters follow each line feed. RT-11 has established a mechanism

by which any number of fills may follow any character. The byte at location 56(8) contains the character to be followed by fillers and the byte at location 57(8) contains the number of null fills to be used. These locations are initially set to zero which results in no fillers being generated. (Normal operation for LT-33 and LA-30P).

Depending on the terminal, modify the locations as follows:

	<u>Loc 56</u>	<u>Loc 57</u>	<u>Resulting Word (octal)</u>
LA30s 110 baud	015(8)	002(8)	1015
LA30s 150 baud	015(8)	004(8)	2015
LA30s 300 baud	015(8)	012(8)	5015
VT05 600 baud	012(8)	001(8)	412
VT05 1200 baud	012(8)	002(8)	1012
VT05 2400 baud	012(8)	004(8)	2012

The proper octal word can be changed permanently in the monitor by using PATCH to modify locations 56 and 57 in the monitor file. For example:

	<u>Explanation</u>
.R PATCH	
PATCH Version number	
FILE NAME--	
*MONITR.SYS/M	
*56 \ 0 15<LF>	Fill after CR
*57 \ 0 4	with 4 nulls.
*E	

Once the change has been made, all programs which use the monitor for console I/O will operate correctly.

#### A.2.2 Line Printer Column Length

RT-11 is distributed with two line printer handlers; LP.SYS (80 columns) and LP.132 (132 columns). The line printer handler may be modified for other column lengths by changing locations 1110(8) and 1164(8) in LP.SYS to the octal value of the line printer column length.

For example, to change to 96 columns:

```
.R PATCH
PATCH Version number
FILE NAME--
*LP.SYS
*1110/120 140 (96(10)=140(8))
*1164/120 140
*E
```



### A.3 SYSTEM OPTIMIZATION

When building RT-11 systems, performance can be optimized (especially on DECTape) by proper placement of .SYS files on the systems device.

Optimal file placement is:

MONITR.SYS

Most frequently used handler

:

:

Least frequently used handler

SYSMAC.SML (if you do a lot of assemblies)

Most frequently used program

:

:

Least frequently used program

Considerations for the above placements are:

1. By positioning all .SYS files at the beginning of the device, movement of .SYS files is avoided during /S operations with PIP, thus eliminating the necessity to reboot. This is the only meaningful optimization for disk users to make.
2. Placement of the monitor immediately following the directory optimizes tape motion during monitor swapping operations.
3. Positioning the handlers and programs in descending order related to frequency of use reduces the access times for those files.
4. In systems that will be used for frequent assembly operations, assembler performance can be improved by placing SYSMAC.SML near the beginning of the tape.

DECTape users can also conserve time and space by placing only those files needed on the system DECTape. Users of 8K DECTape systems need not place files such as MACRO.SAV and RK.SYS on the systems device, as these files cannot be used in 8K systems.

### A.4 ASSEMBLY AND LINKING INSTRUCTIONS

#### A.4.1 General Instructions

All RT-11 components, except MACRO, require 16K of core to be assembled. MACRO requires 20K. RT-11 MACRO is used as the assembler, and RT-11 Link is used as the linker in all cases. All assemblies (except ODT) and all links should be error free.

Throughout the following sections, the conventions used are:

1. Default extensions are not explicitly typed. For all the source files, the extensions are .MAC. The Assembler output is .OBJ and Linker output is .SAV.

2. The default device (DK) is used for all files in the example command strings. This is not a requirement.
3. Listings and link maps are not generated in the command strings shown. They may be generated by adding a second specification on the output side.

All RT-11 system assembling and linking operations are normal operations, and the command strings used can be altered to take full advantage of all RT-11 command features.

#### A.4.2 The Monitor

##### A.4.2.1 Assembling the Bootstrap

For an RK system:

```
.R MACRO
*RKBOOT=BSTRAP
```

For a DECTape system:

```
.R MACRO
*DTBOOT=TT:,DK:BSTRAP
↑$DTSYS=1
↑Z                               (CTRL/Z)
↑$DTSYS=1
↑Z
```

##### A.4.2.2 Assembling the Monitor

```
.R MACRO
*RT11=RT11
```

##### A.4.2.3 Assembling the Systems Handler

For RK disk:

```
.R MACRO
*RK=RK
```

For DECTape:

```
.R MACRO
*DT=DT
```

##### A.4.2.4 Linking the Monitor

For RK11 disk:

```
.R LINK
*RKMON.SYS=RKBOOT,RT11,RK/B:16000/M
```

For DEctape:

```
.R LINK
*DTMON.SYS=DTBOOT,RT11,DT/B:16000/M
```

#### A.4.3 The Handlers

##### A.4.3.1 Assembling the RK Handler

```
.R MACRO
*RK=RK
```

##### A.4.3.2 Assembling the DT Handler

```
.R MACRO
*DT=DT
```

##### A.4.3.3 Assembling the LP Handler

```
.R MACRO
*LP=LP
```

##### A.4.3.4 Assembling the PR Handler

```
.R MACRO
*PR=PR
```

##### A.4.3.5 Assembling the PP Handler

```
.R MACRO
*PP=PP
```

##### A.4.3.6 Assembling the TT Handler

```
.R MACRO
*TT=TT
```

##### A.4.3.7 Linking the Handlers

###### Disk

```
.R LINK
*RK.SYS=RK
```

###### DEctape

```
.R LINK
*DT.SYS=DT
```

### Line Printer

.R LINK  
\*LP.SYS=LP

### Papertape Reader

.R LINK  
\*PR.SYS=PR

### Papertape Punch

.R LINK  
\*PP.SYS=PP

### Terminal

.R LINK  
\*TT.SYS=TT

## A.4.4 EDIT

### A.4.4.1 Assembling EDIT

.R MACRO  
\*EDIT=EDIT

### A.4.4.2 Linking EDIT

.R LINK  
\*EDIT=EDIT

## A.4.5 MACRO

The MACRO source is divided into seven files: RTPAR.MAC, RPARAM.MAC, RCIOCH.MAC, RTEXEC.MAC, MACRO3.MAC, MACRO5.MAC and PST.MAC.

### A.4.5.1 Assembling MACRO

.R MACRO  
\*RTEXEC=RTPAR,RPARAM,RCIOCH,RTEXEC  
\*RTMAC=RTPAR,RPARAM,RCIOCH,MACRO3,MACRO5  
\*RTPST=RTPAR,PST

### A.4.5.2 Linking MACRO

.R LINK  
\*MACRO=RTEXEC,RTMAC,RTPST

#### A.4.6 EXPAND

The EXPAND source has five parts: PREPAR.MAC, PPARAM.MAC, PCIOCH.MAC, PREXEC.MAC, and PREPAS.MAC.

##### A.4.6.1 Assembling EXPAND

```
.R MACRO
*PREXEC=PREPAR,PPARAM,PCIOCH,PREXEC
*PREPAS=PREPAR,PPARAM,PCIOCH,PREPAS
```

##### A.4.6.2 Linking EXPAND

```
.R LINK
*EXPAND=PREXEC,PREPAS
```

#### A.4.7 ASEMBL

The ASEMBL source has seven components: SMPAR.MAC, RPARAM.MAC, RCIOCH.MAC, RTEXEC.MAC, MACRO3.MAC, MACRO5.MAC and PST.MAC.

##### A.4.7.1 Assembling ASEMBL

```
.R MACRO
*SMEXEC=SMPAR,RPARAM,RCIOCH,RTEXEC
*SMMAC=SMPAR,RTPARAM,RCIOCH,MACRO3,MACRO5
*SMPST=SMPAR,PST
```

##### A.4.7.2 Linking ASEMBL

```
.R LINK
*ASEMBL=SMEXEC,SMMAC,SMPST
```

#### A.4.8 LINK

##### A.4.8.1 Assembling LINK

```
.R MACRO
*LINK=LINK
```

##### A.4.8.2 Linking LINK

```
.R LINK
*LINK=LINK
```

#### A.4.9 PIP

##### A.4.9.1 Assembling PIP

```
.R MACRO
*PIP=PIP
```

#### A.4.9.2 Linking PIP

```
.R LINK  
*PIP=PIP
```

#### A.4.10 PIPC

##### A.4.10.1 Assembling PIPC

```
.R MACRO  
*PIPC=PIPC
```

##### A.4.10.2 Linking PIPC

```
.R LINK  
*PIPC=PIPC
```

#### A.4.11 PATCH

##### A.4.11.1 Assembling PATCH

```
.R MACRO  
*PATCH=PATCH
```

##### A.4.11.2 Linking PATCH

```
.R LINK  
*PATCH=PATCH
```

#### A.4.12 ODT

##### A.4.12.1 Assembling ODT

```
.R MACRO  
*ODT=ODT
```

ODT assembles with one error; a "Z" error which flags an ODT instruction which is machine dependent. The error is necessary and should be ignored.

APPENDIX B

PHYSICAL DEVICE NAMES

The following 2-character names are the permanent device names for RT-11.

<u>Name</u>	<u>I/O Device</u>
CTn	Cassette tape n where n is 0 or 1.
DK	The default storage device for all files, usually disk (or DT0 on a DECTape system).
DTn	DECTape n where n is a unit number (in the range 0 to 7).
LP	Line printer.
PP	High-speed paper tape punch.
PR	High-speed paper tape reader.
RKn	RK disk cartridge drive n, where n is in the range 0-7 inclusive.
SYn	System device, the device and unit from which the system is bootstrapped. n is an integer in the range 0-7.
TT	Terminal keyboard and printer.





APPENDIX C  
MONITOR SUMMARY

C.1 KEYBOARD COMMANDS

All commands are terminated by a carriage return.

<u>Command Format</u>	<u>Explanation</u>
ASsign dev udev	Assigns the 1-3 character user defined name (udev) as an alternative name for the device specified (dev). Deassigns synonyms if the standard device name is the only argument given. Deassigns all synonyms for all devices if no arguments are specified.
Base location	Sets the relocation base to be used in an Examine or Deposit operation.
CLose	Closes all currently open files.
DAte dd-mmm-yy	Enters the specified date as the current system date.
Deposit location=valuel,...,valuen	Deposits the specified value(s) starting at the location given.
Examine locationl-locationn	Prints the contents of the specified location(s).
GEt dev:filnam.ext	Loads the specified core image file.
INitialize	Resets all system tables, removes all user device assignments, sets all handlers non-resident, and clears the core control block.
REenter	Starts the program in core at its reentry address.
RUn dev:filnam.ext	Loads the specified core image file into core and starts execution.
R filnam.ext	Loads the specified core image file from the system device (SY) and starts execution.
SAve dev:filnam.ext parameters	Saves the data in the specified core locations on the specified device and assigns a name (filnam.ext). Transfers are handled in 256-word blocks.
STart address	Begins execution of the program in core at the address specified.

## C.2 SPECIAL FUNCTION KEYS

The following terminal keys are used to communicate with KMON.

<u>Key</u>	<u>Function</u>
CTRL/C	Interrupts execution of the user program and returns to Monitor command level if the program is waiting for terminal input. Otherwise CTRL/C must be typed twice. Monitor commands run to completion before a single CTRL/C takes effect. Two CTRL/C's echo two dots; cause the program being executed to be aborted and control returns to the Keyboard Monitor.
CTRL/O	Inhibits printing on the terminal until completion of current output or until another CTRL/O is typed.
CTRL/U	Deletes the current line from right to left up to the first CR/LF combination encountered to the left.
RUBOUT	Deletes the last character from the current line and echoes a backslash and the character deleted. Each succeeding RUBOUT typed deletes and echoes another character up to the last CR/LF typed.

APPENDIX D

RT-11 ERROR MESSAGES

The following messages are arranged in alphabetical order ignoring any special symbols preceding the text.

<u>Message</u>	<u>Program</u>	<u>Explanation</u>
?	ODT	The command specified cannot be handled by ODT
A	MACRO	Addressing error. An address within the instruction is incorrect. Also may indicate a relocation error.
ADDITIVE REF OF xxxxxx	LINK	Rule 1 of overlay rules explained in section 6.5 has been violated.
?ADDR?	Monitor	Address out of range in Examine or Deposit command.
?ADDR NOT IN SEG?	PATCH	The address is not in the specified segment.
B	MACRO	Boundary error. Instructions or word data are being assembled at an odd address in memory. The location counter is updated by +1.
?/B NO VALUE?	LINK	The /B switch requires an octal number as an argument.
?/B ODD VALUE?	LINK	The argument to the /B switch was not an unsigned even octal number.
?BAD BOOT?	PIP	Attempted to bootstrap a non-file structured device.
?BAD GSD?	LINK	There was an error in the Global Symbol Directory of the file being linked.
?***BAD MACRO ARG***?	EXPAND	The macro argument is not formatted correctly.
BAD OVERLAY	LINK	Check for a .ASECT in overlay.
?BAD RLD?	LINK	There is an invalid RLD command in the input file; the file is probably not a legal object module.
?BAD SWITCH?	MACRO LINK EXPAND PATCH	The switch specified was not recognized by the program.

<u>Message</u>	<u>Program</u>	<u>Explanation</u>
?BOOT COPY?	PIPC	The bootstrap file was not found on a /U copy operation.
?BOTTOM ADDR WRONG?		The bottom address specified or contained in location 42 of an overlay file is incorrect.
?BSW?	ASEMBL	The switch specified was not recognized by the program.
BYTE RELOCATION ERROR AT	xxxxxx LINK	Linker attempted to relocate and link byte quantities but failed. Failure is defined as the high byte of the relocated value (or the linked value) not being all zero. In such a case, the value is truncated to 8 bits and the Linker continues processing.
?CB FULL?	EDIT	Command string exceeds the space allowed in the command buffer.
?CHK SUM?	PIP	A checksum error occurred in a formatted binary transfer.
?CORE?	ASEMBL	There are too many symbols in the program being assembled. Try dividing program into separately assembled subprograms.
?CORE?	LINK	The command string is too complicated; not enough core to accommodate the command.
?COR OVR?	PIP PIPC	Core overflow. There are too many devices and/or file specifications (usually *. * operations) and/or there is no room for the buffers.
D	MACRO	Doubly-defined symbol referenced. Reference was made to a symbol which is defined more than once.
?DAT?	Monitor	DATE command format is incorrect.
?DEV FUL?	PIP PIPC	The device (not a cassette) specified does not have sufficient space to create a file of the desired size.
?*DIR FULL*?	EDIT	The directory of the device specified for output is full.
E	MACRO	End directive not found. (A listing is generated).

<u>Message</u>	<u>Program</u>	<u>Explanation</u>
?*EOF*?	EDIT	Attempted a Read, Next or file searching command and no data was available.
?ER WR DIR?	PIP	An unrecoverable error occurred while writing the directory.
? "<>" ERR?	EDIT	The nesting of iteration brackets is greater than 20 levels or brackets are unmatched or illegally used.
?ERROR ERROR?	LINK	An error occurred while the Linker was producing an error message.
?ERROR IN FETCH?	LINK	The device is not available.
?EXT NEG?	PIP	A /T command attempted to make the specified file smaller in size.
?FILE?	Monitor	A file was not specified where one was expected.
?*FILE FULL*?	EDIT	Available space for an output file is full.
?*FIL NOT FND*?	EDIT Monitor PIP PIPC	File specified in a command line was not found.
?HANDLR?	Monitor	A CLOSE command was issued on a device which has no handler in core.
?HARD I/O ERROR?	LINK	Hardware error occurred. Try again.
?*HDW ERR*?	EDIT	A hardware error occurred during I/O.
I	MACRO	Illegal character detected. Illegal characters which are also non-printing are replaced by a ? on the listing. The character is then ignored.
?ILL CMD?	Monitor	The command entered was illegal or the command line was too long.
?*ILL DEV*?	EDIT Monitor PIP PIPC	The device specified is illegal or nonexistent.

<u>Message</u>	<u>Program</u>	<u>Explanation</u>
?ILL MAC?	EDIT	Delimiters were improperly used, or an attempt was made to enter an M command during execution of a Macro, or an attempt was made to execute an EM command while an EM was in process.
?*ILL NAME*?	EDIT	File name specified in EB, EW, or ER is illegal.
?ILL SWT?	PIP	The switch specified in the command string is illegal.
?IN ER?	PIP PIPC	An unrecoverable error occurred while reading a file. This error is ignored during a /G operation.
?INCORRECT FILE SPEC?	PATCH	The response to the FILE NAME-- message was not of the correct form.
?INPUT ERROR?	EXPAND	Hardware error in reading an input file.
?INSUFFICIENT CORE?	EXPAND PATCH	Not enough memory available.
?INSUFFICIENT CORE?	MACRO	There are too many symbols in the program being assembled. Try dividing program into separately-assembled subprograms.
INVALID OVERLAY REFERENCE	LINK	Reference of a global that is in an overlay violates one of the overlay rules.
L	MACRO	Line buffer overflow, i.e., input line greater than 132 characters. Extra characters on a line, (more than 72) are ignored.
?LDA FILE ERROR?		There was a hardware problem with the device specified for LDA output or the device is full.
?***LINE TOO LONG***?	EXPAND	A line has become longer than 132 characters.
M	MACRO	Multiple definition of a label. A label was encountered which was equivalent (in the first six characters) to a previously encountered label.

<u>Message</u>	<u>Program</u>	<u>Explanation</u>
?M-DIR OVFL0?	Monitor	No more directory segments are available for expansion. Occurs when .ENTER causes directory expansion.
?M-DIR IO ERR?	Monitor	Error occurred while USR was reading/writing a device directory. Usually due to WRITE LOCKED device or a hardware problem.
?M-HAND LD FAIL?	Monitor	Failed reading a device handler from the system device. There may be a hardware problem.
?M-ILL HAND LD?	Monitor	Attempted to load a device handler over the USR.
?M-NO DEV?	Monitor	Requested an I/O operation on a channel but no device handler was in core.
?M-OVRL ERR?	Monitor	Attempted to read an overlay segment on channel 17 but was unsuccessful. (Can only occur if program was linked with the overlay feature requested.)
?M-USR ILL?	Monitor	USR was illegally called from an I/O completion routine.
?***MACRO ALREADY DEFINED***?	EXPAND	A macro was defined more than once.
?***MACRO(S) NOT FOUND***?	EXPAND	Macros listed in an .MCALL statement were not found in SYSMAC.SML (make sure SYSMAC.SML is present on system.
?MAP FILE ERROR?	LINK	There was a hardware problem with the device specified for map output or the device is full.
?***MISSING COMMA IN MACRO ARG***?	EXPAND	Found spaces or tabs within a macro argument; try using brackets around the argument, e.g. <arg with spaces>.
?***MISSING DOT***?	EXPAND	A macro name or argument name doesn't begin with a dot.
?MISSING END IN MACRO?	EXPAND	End of input was encountered while storing a macro definition: probably missing an ENDM.
MULT DEF OF xxxxxx	LINK	The symbol, xxxxxx, was defined more than once.

<u>Message</u>	<u>Program</u>	<u>Explanation</u>
N	MACRO	Number containing an 8 or 9 has a decimal point missing.
****NAME DOESN'T MATCH****?	EXPAND	Optional name given in .ENDM directive does not match name given in corresponding .MACRO directive.
****NESTED MACROS****?	EXPAND	A macro is being defined or invoked within another macro.
?NIF?	ASEMBL	No input file was specified and there must be at least one.
?*NO FILE*?	EDIT	Attempted to read or write when no file is open.
****NO NAME****?	EXPAND	A macro definition has no name.
?NO INPUT?	LINK	No input files were specified.
?NO INPUT FILE?	MACRO EXPAND	There must be at least one input file.
?*NO ROOM*?	EDIT	Attempted to Insert, Save, Unsave, Read, Next, Change or Exchange when there was not enough room in the appropriate buffer.
?NO SYS ACTION?	PIP	A warning message indicating that the /Y switch was not included with a command specified on a .SYS file. The .SYS file operations are not performed, but the remainder of the command is executed.
O	MACRO	Opcode error. Directive out of context.
?OFF LINE?	PIPC	No cassette in unit specified or the cassette is improperly mounted.
?ODF?	ASEMBL	No room to continue writing output--try to compress device with PIP.
?OUT ER?	PIP	An unrecoverable error occurred while writing a file. Perhaps a hardware or checksum error; try recopying the file. Also caused by an attempt to compress a larger device to a smaller one or not enough room on a command because only 1/2 the available area is used.
?OUT FIL?	PIP PIPC	The output file specification is illegal or the output file specified was not found.
?OUTPUT DEVICE FULL?	MACRO EXPAND	No room to continue writing output--try to compress the device with PIP.



<u>Message</u>	<u>Program</u>	<u>Explanation</u>
?OUTPUT FULL?	LINK	The output device is full.
?OVRFL?	PIPC	The file to be transferred will not fit on the cassette. The file is not transferred.
?OVR COR?	Monitor	The file specified in a GET or RUN command is too big.
P	MACRO	Phase error. A label's definition of value varies from one pass to another.
?PARAMS?	Monitor	The parameters specified to the SAVE command are bad.
Q	MACRO	Questionable syntax. There are missing arguments or the instruction scan was not completed or a carriage return was not immediately followed by a line feed or form feed.
R	MACRO	Register-type error. An invalid use of or reference to a register.
?REBOOT?	PIP PIPC	A warning message indicating that .SYS files have been transferred, renamed, compressed or deleted from the systems device. It may be necessary to reboot the system.
?ROOM?	PIP	The space following the file specified with a /T switch is insufficient to execute the extend operation.
?SAV FILE ERR?	LINK	The Linker had a problem reading the input file; try again.
?*SRCH FAIL*?	EDIT	The text string specified in a Get, Find, or Position command was not found in the available data.
?SV FIL I/O ER?	Monitor	An I/O error occurred on a .SAV file specified in a SAVE (output) or R, RUN or GET (input) command.
?SY I/O ER?	Monitor	An I/O error occurred on the system device (usually reading of writing scratch area).
?SYMBOL TABLE OVERFLOW?	LINK	There were too many global symbols used in the program.
?***SYNTAX***?	EXPAND	A macro directive is not constructed correctly.

<u>Message</u>	<u>Program</u>	<u>Explanation</u>
T	MACRO	Truncation error. A number generated more than 16 bits of significance or an expression generated more than 8 bits of significance during the use of the .BYTE directive.
?TIM ERROR?	PIPC	Timing error.
?TMO?	ASEMBL	Too many output files were specified.
?***TOO MANY ARGS***?	EXPAND	A macro directive has more than 30 arguments.
?TOO MANY OUTPUT FILES?	MACRO LINK	Too many output files were specified.
TRANSFER ADDRESS UNDEFINED OR IN OVERLAY	LINK	The transfer address was not defined or is in an overlay.
U	MACRO	Undefined symbol. An undefined symbol was encountered during the evaluation of an expression. Relative to the expression, the undefined symbol is assigned a value of zero.
UNDEFINED GLOBALS XXXXXX XXXXXX . . .	LINK	This condition also causes a warning message on the terminal.
?WRT LOCK?	PIPC	Cassette unit specified is write locked.
?WRONG NUMBER OF OUTPUT FILES?	EXPAND	There must be exactly one output file.
Z	MACRO	Instruction which is not compatible among all members of the PDP-11 family (11/15, 11/20, 11/45).

APPENDIX E

MACRO CHARACTER SETS

E.1 ASCII CHARACTER SET

<u>EVEN PARITY BIT</u>	<u>7-BIT OCTAL CODE</u>	<u>CHARACTER</u>	<u>REMARKS</u>
0	000	NUL	NULL, TAPE FEED, CONTROL/SHIFT/P.
1	001	SOH	START OF HEADING; ALSO SOM, START OF MESSAGE, CONTROL/A.
1	002	STX	START OF TEXT; ALSO EOA, END OF ADDRESS, CONTROL/B.
0	003	ETX	END OF TEXT; ALSO EOM, END OF MESSAGE, CONTROL/C.
1	004	EOT	END OF TRANSMISSION (END); SHUTS OFF TWX MACHINES, CONTROL/D.
0	005	ENQ	ENQUIRY (ENQRY); ALSO WRU, CONTROL/E.
0	006	ACK	ACKNOWLEDGE; ALSO RU, CONTROL/F.
1	007	BEL	RINGS THE BELL. CONTROL/G.
1	010	BS	BACKSPACE; ALSO FEO, FORMAT EFFECTOR. BACKSPACES SOME MACHINES, CONTROL/H.
0	011	HT	HORIZONTAL TAB. CONTROL/I.
0	012	LF	LINE FEED OR LINE SPACE (NEW LINE); ADVANCES PAPER TO NEXT LINE, DUPLICATED BY CONTROL/J.
1	013	VT	VERTICAL TAB (VTAB). CONTROL/K.
0	014	FF	FORM FEED TO TOP OF NEXT PAGE (PAGE). CONTROL/L.
1	015	CR	CARRIAGE RETURN TO BEGINNING OF LINE. DUPLICATED BY CONTROL/M.
1	016	SO	SHIFT OUT; CHANGES RIBBON COLOR TO RED. CONTROL/N.
0	017	SI	SHIFT IN; CHANGES RIBBON COLOR TO BLACK. CONTROL/O.
1	020	DLE	DATA LINK ESCAPE. CONTROL/B (DC0).
0	021	DC1	DEVICE CONTROL 1, TURNS TRANSMITTER (READER) ON, CONTROL/Q (X ON).
0	022	DC2	DEVICE CONTROL 2, TURNS PUNCH OR AUXILIARY ON. CONTROL/R (TAPE, AUX ON).
1	023	DC3	DEVICE CONTROL 3, TURNS TRANSMITTER (READER) OFF, CONTROL/S (X OFF).
0	024	DC4	DEVICE CONTROL 4, TURNS PUNCH OR AUXILIARY OFF. CONTROL/T (AUX OFF).
1	025	NAK	NEGATIVE ACKNOWLEDGE; ALSO ERR, ERROR. CONTROL/U.
1	026	SYN	SYNCHRONOUS FILE (SYNC). CONTROL/V.
0	027	ETB	END OF TRANSMISSION BLOCK; ALSO LEM, LOGICAL END OF MEDIUM. CONTROL/W.

<u>EVEN PARITY BIT</u>	<u>7-BIT OCTAL CODE</u>	<u>CHARACTER</u>	<u>REMARKS</u>
0	030	CAN	CANCEL (CANCL). CONTROL/X.
1	031	EM	END OF MEDIUM. CONTROL/Y.
1	032	SUB	SUBSTITUTE. CONTROL/Z.
0	033	ESC	ESCAPE. CONTROL/SHIFT/K.
1	034	FS	FILE SEPARATOR. CONTROL/SHIFT/L.
0	035	GS	GROUP SEPARATOR. CONTROL/SHIFT/M.
0	036	RS	RECORD SEPARATOR. CONTROL/SHIFT/N.
1	037	US	UNIT SEPARATOR. CONTROL/SHIFT/O.
1	040	SP	SPACE.
0	041	!	
0	042	"	
1	043	#	
0	044	\$	
1	045	%	
0	046	&	
0	047	'	ACCENT ACUTE OR APOSTROPHE.
0	050	(	
0	051	)	
1	052	*	
0	053	+	
1	054	,	
0	055	-	
0	056	.	
1	057	/	
0	060	0	
1	061	1	
1	062	2	
0	063	3	
1	064	4	
0	065	5	
0	066	6	
1	067	7	
1	070	8	
0	071	9	
0	072	:	
1	073	;	
0	074	<	
1	075	=	
1	076	>	
0	077	?	
1	100	@	
0	101	A	
0	102	B	
1	103	C	
0	104	D	
1	105	E	
1	106	F	
0	107	G	
0	110	H	
1	111	I	
1	112	J	
0	113	K	
1	114	L	
0	115	M	
0	116	N	

<u>EVEN PARITY BIT</u>	<u>7-BIT OCTAL CODE</u>	<u>CHARACTER</u>	<u>REMARKS</u>
1	117	O	
0	120	P	
1	121	Q	
1	122	R	
0	123	S	
1	124	T	
0	125	U	
0	126	V	
1	127	W	
1	130	X	
0	131	Y	
0	132	Z	
1	133	[	SHIFT/K.
0	134		SHIFT/L.
1	135	]	SHIFT/M.
1	136	↑	(APPEARS AS ^ ON SOME MACHINES)
0	137	+	(APPEARS AS _ (UNDERSCORE) ON SOME MACHINES)
0	140	`	ACCENT GRAVE.
1	141	a	
1	142	b	
0	143	c	
1	144	d	
0	145	e	
0	146	f	
1	147	g	
1	150	h	
0	151	i	
0	152	j	
1	153	k	
0	154	l	
1	155	m	
1	156	n	
0	157	o	
1	160	p	
0	161	q	
0	162	r	
1	163	s	
0	164	t	
1	165	u	
1	166	v	
0	167	w	
0	170	x	
1	171	y	
1	172	z	
0	173	{	
1	174		
0	175	}	THIS CODE GENERATED BY ALT MODE.
0	176	~	THIS CODE GENERATED BY PREFIIX KEY (IF PRESENT)
1	177	DEL	DELETE, RUB OUT.

E.2 RADIX-50 CHARACTER SET

<u>Character</u>	<u>ASCII Octal Equivalent</u>	<u>Radix-50 Equivalent</u>
space	40	0
A-Z	101-132	1-32
\$	44	33
.	56	34
unused		35
0-9	60-71	36-47

The maximum Radix-50 value is, thus,

$$47*50^2+47*50+47=174777$$

The following table provides a convenient means of translating between the ASCII character set and its Radix-50 equivalents. For example, given the ASCII string X2B, the Radix-50 equivalent is (arithmetic is performed in octal):

```

X=113000
2=002400
B=000002
X2B=115402

```

<u>Single Char.</u> or <u>First Char.</u>		<u>Second</u> <u>Character</u>		<u>Third</u> <u>Character</u>	
A	003100	A	000050	A	000001
B	006200	B	000120	B	000002
C	011300	C	000170	C	000003
D	014400	D	000240	D	000004
E	017500	E	000310	E	000005
F	022600	F	000360	F	000006
G	025700	G	000430	G	000007
H	031000	H	000500	H	000010
I	034100	I	000550	I	000011
J	037200	J	000620	J	000012
K	042300	K	000670	K	000013
L	045400	L	000740	L	000014
M	050500	M	001010	M	000015
N	053600	N	001060	N	000016
O	056700	O	001130	O	000017
P	062000	P	001200	P	000020
Q	065100	Q	001250	Q	000021
R	070200	R	001320	R	000022
S	073300	S	001370	S	000023
T	076400	T	001440	T	000024
U	101500	U	001510	U	000025
V	104600	V	001560	V	000026
W	107700	W	001630	W	000027
X	113000	X	001700	X	000030
Y	116100	Y	001750	Y	000031
Z	121200	Z	002020	Z	000032
\$	124300	\$	002070	\$	000033
.	127400	.	002140	.	000034
	132500		002210		000035
0	135600	0	002260	0	000036
1	140700	1	002330	1	000037
2	144000	2	002400	2	000040
3	147100	3	002450	3	000041
4	152200	4	002520	4	000042
5	155300	5	002570	5	000043
6	160400	6	002640	6	000044
7	163500	7	002710	7	000045
8	166600	8	002760	8	000046
9	171700	9	003030	9	000047





## APPENDIX F

### MACRO ASSEMBLY LANGUAGE AND ASSEMBLER

#### F.1 SPECIAL CHARACTERS

<u>Character</u>	<u>Function</u>
form feed	Source line terminator
line feed	Source line terminator
carriage return	Formatting character
vertical tab	Source line terminator
:	Label terminator
=	Direct assignment indicator
%	Register term indicator
tab	Item terminator
	Field terminator
space	Item terminator
	Field terminator
#	Immediate expression indicator
@	Deferred addressing indicator
(	Initial register indicator
)	Terminal register indicator
,	Operand field separator
, (comma)	Comment field indicator
;	Comment field indicator
+	Arithmetic addition operator or auto increment indicator
-	Arithmetic subtraction operator or auto decrement indicator
*	Arithmetic multiplication operator
/	Arithmetic division operator
&	Logical AND operator
!	Logical OR operator
"	Double ASCII character indicator
' (apostrophe)	Single ASCII character indicator
.	Assembly location counter
<	Initial argument indicator
>	Terminal argument indicator
↑	Universal unary operator
↑	Argument indicator
\	MACRO numeric argument indicator

## F.2 ADDRESS MODE SYNTAX

n is an integer between 0 and 7 representing a register. R is a register expression, E is an expression, ER is either a register expression or an expression in the range 0 to 7.

<u>Format</u>	<u>Address Mode Name</u>	<u>Address Mode Number</u>	<u>Meaning</u>
R	Register	0n	Register R contains the operand. R is a register expression.
@R or (ER)	Deferred Register	1n	Register R contains the operand address.
(ER)+	Autoincrement	2n	The contents of the register specified by ER are incremented after being used as the address of the operand.
@(ER)+	Deferred Auto-increment	3n	ER contains the pointer to the address of the operand. ER is incremented after use.
-(ER)	Autodecrement	4n	The contents of register ER are decremented before being used as the address of the operand.
@-(ER)	Deferred Auto-decrement	5n	The contents of register ER are decremented before being used as the pointer to the address of the operand.
E(ER)	Index	6n	E plus the contents of the register specified, ER, is the address of the operand.
#E	Immediate	27	E is the operand
@#E	Absolute	37	E is the address of the operand.
E	Relative	67	E is the address of the operand.
@E	Deferred Relative	77	E is the pointer to the address of the operand.

## F.3 INSTRUCTIONS

The instructions which follow are grouped according to the operands they take and the bit patterns of their op-codes.

In the instruction type format specification, the following symbols are used:

OP	Instruction mnemonic
R	Register expression
E	Expression
ER	Register expression or expression $0 \leq ER \leq 7$
AC	Floating point register expression
A	General address specification

In the representation of op-codes, the following symbols are used:

SS	Source operand specified by a 6-bit address mode.
DD	Destination operand specified by a 6-bit address mode.
XX	8-bit offset to a location (branch instructions).
R	Integer between 0 and 7 representing a general register.

Symbols used in the description of instruction operations are:

SE	Source Effective address
FSE	Floating Source Effective address
DE	Destination Effective Address
FDE	Floating Destination Effective Address
	Absolute value of
()	Contents of
→	Becomes

The condition codes in the processor status word (PS) are affected by the instructions. These condition codes are represented as follows:

N	<u>N</u> egative bit:	set if the result is negative
Z	Zero bit:	set if the result is zero
V	<u>O</u> verflow bit:	set if the operation caused an overflow
C	<u>C</u> arry bit:	set if the operation caused a carry

In the representation of the instruction's effect on the condition codes, the following symbols are used:

*	Conditionally set
-	Not affected
0	Cleared
1	Set

To set conditionally means to use the instruction's result to determine the state of the code (see the PDP-11 Processor Handbook).

Logical operations are represented by the following symbols:

	Inclusive OR
⊕	Exclusive OR
&	AND
-	(used over a symbol) NOT (i.e., 1's complement)

### F.3.1 Double-Operand Instructions

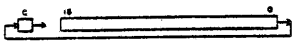
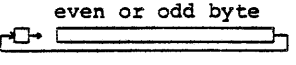
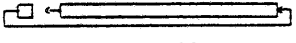
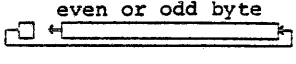
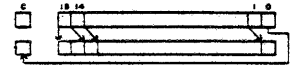
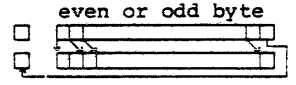
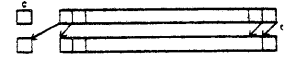
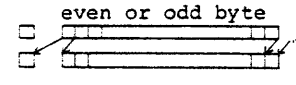
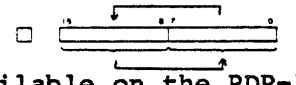
Instruction type format: Op A,A

<u>Op-Code</u>	<u>Mnemonic</u>	<u>Stands for</u>	<u>Operations</u>	<u>Status Word Condition Codes</u> N Z V C
01SSDD 11SSDD	MOV MOVB	MOVE MOVE Byte	(SE) → DE	* * 0 -
02SSDD 12SSDD	CMP CMPB	CoMPare CoMPare Byte	(SE) - (DE)	* * * *
03SSDD 13SSDD	BIT BITB	BIt Test BIt Test Byte	(SE) & (DE)	* * 0 -
04SSDD 14SSDD	BIC BICB	BIt Clear BIt Clear Byte	(SE) & (DE) → DE	* * 0 -
05SSDD 15SSDD	BIS BISB	BIt Set BIt Set Byte	(SE) ! (DE) → DE	* * 0 -
06SSDD 16SSDD	ADD SUB	ADD SUBtract	(SE) + (DE) → DE (DE) - (SE) → E	* * * * * * * *

### F.3.2 Single-Operand Instructions

Instruction type format: Op A

<u>Op-Code</u>	<u>Mnemonic</u>	<u>Stands for</u>	<u>Operation</u>	<u>Status Word Condition Codes</u> N Z V C
0050DD 1050DD	CLR CLRB	CLear CLear Byte	0 → DE	0 1 0 0
0051DD 1051DD	COM COMB	COMplement COMplement Byte	( $\overline{DE}$ ) → DE	* * 0 1
0052DD 1052DD	INC INDB	INCrement INCrement Byte	( $\overline{DE}$ ) + 1 → DE	* * * -
0053DD 1053DD	DEC DECB	DECrement DECrement Byte	(DE) - 1 → DE	* * * -
0054DD 1054DD	NEG NEGB	NEGate NEGate Byte	( $\overline{DE}$ ) + 1 → DE	* * * *
0055DD 1055DD	ADC ADCB	ADd Carry ADd Carry Byte	( $\overline{DE}$ ) + (C) → DE	* * * *
0056DD 1056DD	SBC SBCB	SuBtract Carry SuBtract Carry Byte	(DE) - (C) → DE	* * * *

Op-Code	Mnemonic	Stands for	Operation	Status Word Condition Codes			
				N	Z	V	C
0057DD	TST	TeST	(DE) - 0 → DE	*	*	0	0
1057DD	TSTB	TeST Byte		*	*	*	*
0060DD	ROR	ROtate Right		*	*	*	*
1060DD	RORB	ROtate Right Byte		*	*	*	*
0061DD	ROL	ROtate Left		*	*	*	*
1061DD	ROLB	ROtate Left Byte		*	*	*	*
0062DD	ASR	Arithmetic Shift Right		*	*	*	*
1062DD	ASRB	Arithmetic Shift Right Byte		*	*	*	*
0063DD	ASL	Arithmetic Shift Left		*	*	*	*
1063DD	ASLB	Arithmetic Shift Left Byte		*	*	*	*
0001DD	JMP	JuMP	DE → PC	-	-	-	-
0003DD	SWAB	SWAp Bytes		*	*	0	0
The following instructions are available on the PDP-11/45 only:							
0065DD	MFPI	Move From Previous Instruction space	} See Chapter 6 in <u>PDP-11/45 Processor Handbook</u>	*	*	0	-
1065DD	MFPC	Move from Previous Data space		*	*	0	-
0066DD	MTPI	Move To Previous Instruction space		*	*	0	-
1066DD	MTPD	Move To Previous Data space		*	*	0	-
1701DD	LDFPS	Load FPP Program Status	(DE) → FPS	-	-	-	-
0067DD	SXT	Sign eXTend	0 → DE if N bit clear -1 → DE if N bit is set	-	*	-	-
0707DD	NEGD	NEGate Double	-(FDE) → FDE	F	N	F	Z

<u>Op-Code</u>	<u>Mnemonic</u>	<u>Stands for</u>	<u>Operations</u>	<u>Status Word Condition Codes</u>			
				<u>F</u>	<u>N</u>	<u>Z</u>	<u>V</u>
0707DD	NEGF	NEGate Floating	-(FDE)→FDE	*	*	0	0
1702DD	STFPS	STore Floating Point processor program Status	See Chapter 7 in <u>PDP-11/45 Processor Handbook</u>	-	-	-	-
1703DD	STST	STore floating point processor Status		-	-	-	-
1704DD	CLRD	CLear Double	0→FDE	0	1	0	0
1704DD	CLRF	CLear Floating	0→FDE	0	1	0	0
1705DD	TSTD	TeST Double	(FDE)-0→FDE	*	*	0	0
1705DD	TSTF	TeST Floating	(FDE)-0→FDE	*	*	0	0
1706DD	ABSD	make ABSolute Double	FDE →FDE	0	*	0	0
1706DD	ABSF	make ABSolute Floating	FDE →FDE	0	*	0	0

### F.3.3 Operate Instructions

Instruction type format: Op

<u>Op-Code</u>	<u>Mnemonic</u>	<u>Stands for</u>	<u>Operation</u>	<u>N</u>	<u>Z</u>	<u>V</u>	<u>C</u>
000000	HALT	HALT	The computer stops all functions.	-	-	-	-
0000001	WAIT	WAIT	The computer stops and waits for an interrupt.	-	-	-	-
0000002	RTI	ReTurn from interrupt	The PC and PS are popped off the SP stack: ((SP))→PC (SP)+2→SP ((SP))→PS (SP)+2→SP  RTI is also used to return from a trap.	*	*	*	*
000005	RESET	RESET	Returns all I/O devices to power-on status.	-	-	-	-

000241	CLC	CLear Carry bit	0 → C	- - - 0
000261	SEC	SEt Carry bit	1 → C	- - - 1
000242	CLV	CLear oVerflow	0 → V	- - 0 -
000262	SEV	SEt oVerflow bit	1 → V	- - 1 -
000244	CLZ	CLear Zero bit	0 → Z	- 0 - -
000264	SEZ	SEt Zero bit	1 → Z	- 1 - -
000250	CLN	CLear Negative bit	0 → N	0 - - -
000270	SEN	SEt Negative bit	1 → N	1 - - -
000243		CLear OVerflow and Carry bits	0 → V 0 C	- - 0 0
000254	CNZ	Clear Negative and Zero bits	0 → N 0 Z	0 0 - -
000257	CCC	Clear all Condition Codes	0 → N 0 → Z 0 → V 0 → C	0 0 0 0
000277	SCC	Set all Condition Codes	1 → N 1 → Z 1 → V 1 → C	1 1 1 1
000240	NOP	No Operation		- - - -

The following instructions are available on the PDP-11/45 only:

<u>Op-Code</u>	<u>Mnemonic</u>	<u>Stands for</u>	<u>Operation</u>	<u>N Z V C</u>
170000	CFCC	Copy Floating Condition	Copy FPP condition codes into CPU condition codes.	* * * *
000006	RTT	ReTurn from inTerrupt	Same as RTI instruction but inhibits trace trap	* * * *
170011	SETD	SET Double floating mode	FPP set to double precision	- - - -
170001	SETF	SET Floating mode	FPP set to single precision mode	- - - -
170002	SETI	SET Integer mode	FPP set for integer data (16 bits)	- - - -

170012      SETL            SET Long            FPP set for      - - - -  
                                  integer mode            long integer  
    data (32 bits)

### F.3.4 Trap Instructions

Instruction type format: Op or Op E where  $0 \leq E \leq 377(8)$   
                                  \*OP (only)

<u>Op-Code</u>	<u>Mnemonic</u>	<u>Stands for</u>	<u>Operation</u>	<u>Status Word</u>			
				<u>Condition codes</u>	<u>N</u>	<u>Z</u>	<u>V</u>
*000003	BPT	BreakPoint Trap	Trap to location 14. This is used to call ODT.	*	*	*	*
*000004	IOT	Input/Output Trap	Trap to location 20. This is used to call IOX.	*	*	*	*
104000- 104377	EMT	EMulator Trap	Trap to locations 30. This is used to call system programs.	*	*	*	*
104400- 104777	TRAP	TRAP	Trap to location 34. This is used to call any routine desired by the programmer.	*	*	*	*

### F.3.5 Branch Instructions

Instruction type format: Op E where  $-128(10) < (E-.2)/2 < 127(10)$

<u>Op-Code</u>	<u>Mnemonic</u>	<u>Stands for</u>	<u>Condition to be met if branch is to occur</u>
0004XX	BR	BRanch always	
0010XX	BNE	Branch if Not Equal (to zero)	Z=0
0014XX	BEQ	Branch if Equal (to zero)	Z=1
0020XX	BGE	Branch if Greater than or Equal (to zero)	N ⊕ V=0
0024XX	BLT	Branch if Less Than (zero)	N ⊕ V=1



<u>Op-Code</u>	<u>Mnemonic</u>	<u>Stands for</u>	<u>Condition to be met if branch is to occur</u>
0030XX	BGT	Branch if Greater Than (zero)	Z!(NⓈ)V)=0
0034XX	BLE	Branch if Less than or Equal (to zero)	Z!+(NⓈ)V)=1
1000XX	BPL	Branch if Plus	N=0
1004XX	BMI	Branch if Minus	N=1
1010XX	BHI	Branch if Higher	C ! Z=0
1014XX	BLOS	Branch if Lower or Same	C ! Z=1
1020XX	BVC	Branch if overflow Clear	V=0
1024XX	BVS	Branch if overflow Set	V=1
1030XX	BCC (or BHIS)	Branch if Carry Clear (or Branch if Higher or Same)	C=0
1034XX	BCS (or BLO)	Branch if Carry Set (or Branch if Lower)	C=1

### F.3.6 Register Destination

Instruction type format: OP ER,A

<u>Op-Code</u>	<u>Mnemonic</u>	<u>Stands for</u>	<u>Operation</u>	<u>Status Word Condition Codes</u> N Z V C
004RDD	JSR	Jump to SubRoutine	Push register on the SP stack, put the PC in the register:  DE→TEMP (TEMP= temporary storage register internal to processor.)  (SP)-2→SP (REG)→(SP) (PC)→REG (TEMP)→PC	- - - -

The following instruction is available only on the PDP-11/45:

074RDD XOR eXclusive OR (R) Ⓢ DE DE \* \* 0 -

### F.3.7 Register-Offset

Instruction type format: OP R,E

<u>Op-Code</u>	<u>Mnemonic</u>	<u>Stands for</u>	<u>Operation</u>	<u>N</u>	<u>Z</u>	<u>V</u>	<u>C</u>
077RDD	SOB	Subtract One and Branch	(R)-1 R PC-(2*DE)→PC	-	-	-	-

### F.3.8 Subroutine Return

Instruction type format: Op ER

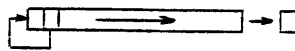
<u>Op-Code</u>	<u>Mnemonic</u>	<u>Stands for</u>	<u>Operation</u>	<u>N</u>	<u>Z</u>	<u>V</u>	<u>C</u>
00020R	RTS	ReTurn from Subroutine	Put register in PC and pop old contents from SP stack into register.	-	-	-	-

### F.3.9 Source-Register

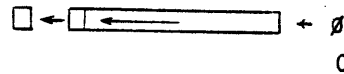
The following instructions are available on the PDP-11/45 only:

Instruction type format: Op A,R

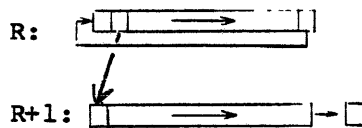
<u>Op-Code</u>	<u>Mnemonic</u>	<u>Stands for</u>	<u>Operation</u>	<u>Status Word Condition Codes</u>			
				<u>N</u>	<u>Z</u>	<u>V</u>	<u>C</u>
071RSS	DIV	DIVide	R,Rvl/(SRC) → R,RVL	*	*	*	*
070RSS	MUL	MULtiple	R*(SRC) → R,Rvl	*	*	0	*
072RSS	ASH	Arithmetic SHift	R is shifted according to low-order 6-bits of source	*	*	*	*



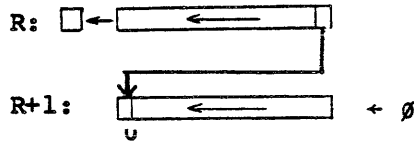
or



073RSS	ASHC	Arithmetic SHift Combined	R,Rvl are shifted according to low-order 6 bits of source	*	*	*	*
--------	------	---------------------------	---	---	---	---	---



or



### F.3.10 Floating-Point Source Double Register

The following instructions are available on the PDP-11/45 only:

Instruction type format: Op A,AC

Op-Code	Mnemonic	Stands for	Operation	Status Word Floating Condition Codes			
				FN	FZ	FV	FC
172 (AC) SS	ADDD	ADD Double	(FSE) +AC→AC	*	*	*	0
172 (AC) SS	ADDF	ADD Floating	(FSE) +AC→AC	*	*	*	0
173 (AC+4) SS	CMPD	CoMPare Double	(FSE) -AC	*	*	0	0
173 (AC+4) SS	CMPF	CoMPare Floating	(FSE) -AC	*	*	0	0
174 (AC+4) SS	DIVD	DIVide Double	AC/(FSE) →AC	*	*	*	0
174 (AC+4) SS	DIVF	DIVide Floating	AC/(FSE) →AC	*	*	*	0
177 (AC+4) SS	LDCDF	LoaD and Con-vert from Double to Floating	(FSE) →AC	*	*	*	0
177 (AC+4) SS	LDCFD	LoaD and Con-vert from Floating to Double	(FSE) →AC	*	*	*	0
172 (AC+4) SS	LDD	LoaD Double	(FSE) →AC	*	*	0	0
172 (AC+4) SS	LDF	LoaD Floating	(FSE) →AC	*	*	0	0
171 (AC+4) SS	MODD	Multiply and integerize double	AC*(FSE) →AC, AC1	*	*	*	0
171 (AC+4) SS	MODF	Multiply and integerize floating-point	AC*(FSE) →AC	*	*	*	0
171 (AC) SS	MULD	MULTiply Double	AC*(FSE) →AC	*	*	*	0
171 (AC) SS	MULF	MULTiply Floating	AC*(FSE) →AC	*	*	*	0
173 (AC) SS	SUBD	SUBtract Double	(FSE) -AC→AC	*	*	*	0



<u>Op-Code</u>	<u>Mnemonic</u>	<u>Stands for</u>	<u>Operation</u>	Status Word			
				FN	FZ	FV	FC
175 (AC+4) DD	STCDL <sup>1</sup>	STore, Con- vert from Double to Long integer	AC FDE	*	*	0	*
175 (AC+4) DD	STCFI <sup>1</sup>	STore, Con- vert from Floating to Integer	AC FDE	*	*	0	*
175 (AC+4) DD	STCFL <sup>1</sup>	STore, Con- vert from Floating to Long integer	AC FDE	*	*	0	*
174 (AC) DD	STD	STore Double	AC FDE	-	-	-	-
174 (AC) DD	STF	STore Floating	AC FDE	-	-	-	-
175 (AC) DD	STEXP <sup>1</sup>	STore EXPonent	AC EXP-200 DE	*	*	0	0

### F.3.13 Number

The following instruction is available on the PDP-11/45 only:

<u>Op-Code</u>	<u>Mnemonic</u>	<u>Stands for</u>	<u>Operation</u>	Status Word			
				N	Z	V	C
0064NN	MARK	MARK	Stack cleanup on return from sub- routine.	-	-	-	-

<sup>1</sup> These instructions set both the floating-point and processor condition codes as indicated.

### F.3.14 Priority

The following instruction is available on the PDP-11/45 only:

<u>Op-Code</u>	<u>Mnemonic</u>	<u>Stands for</u>	<u>Operations</u>	<u>Status Word Condition Codes</u>			
				<u>N</u>	<u>Z</u>	<u>V</u>	<u>C</u>
00023N	SPL	Set Priority Level	N→PC (bits 7-5)	-	-	-	-

### F.4 ASSEMBLER DIRECTIVES

<u>Form</u>	<u>Described in Manual Section</u>	<u>Operation</u>
'	5.5.3.3	A single quote character (apostrophe) followed by one ASCII character generates a word containing the 7-bit ASCII representation of the character in the low-order byte and zero in the high-order byte.
"	5.5.3.3	A double quote character followed by two ASCII characters generates a word containing the 7-bit ASCII representation of the two characters.
\	5.6.3.3	A backslash preceding an argument causes the number to be treated in the current radix.
↑Bn	5.5.4.2	Temporary radix control; causes the number n to be treated as a binary number.
↑Cn	5.5.6.2	Creates a word containing the one's complement of n.
↑Dn	5.5.4.2	Temporary radix control; causes the number n to be treated as a decimal number.
↑Fn	5.5.6.2	Creates a one-word floating point quantity to represent n.
↑On	5.5.4.2	Temporary radix control; causes the number n to be treated as an octal number.
.ASCII string	5.5.3.4	Generates a block of data containing the ASCII equivalent of the character string (enclosed in delimiting characters) one character per byte.

<u>Form</u>	<u>Described in Manual Section</u>	<u>Operation</u>
.ASCIZ string	5.5.3.5	Generates a block of data containing the ASCII equivalent of the character string (enclosed in delimiting characters) one character per byte with a zero byte following the specified string.
.ASECT	5.5.9	Begin or resume absolute section.
.BLKB exp	5.5.5.3	Reserves a block of storage space exp bytes long.
.BLKW exp	5.5.5.3	Reserves a block of storage space exp words long.
.BYTE exp1,exp2,..	5.5.3.1	Generates successive bytes of data containing the octal equivalent of the expression(s) specified.
.CSECT symbol	5.5.9	Begin or resume named or unnamed relocatable section.
.DSABL arg	5.5.2	Disables the assembler function specified by the argument.
.ENABL arg	5.5.2	Provides the assembler function specified by the argument.
.END .END exp	5.5.7.1	Indicates the physical end of the source program. An optional argument specifies the transfer address.
.ENDC	5.5.11	Indicates the end of a condition block.
.ENDM .ENDM symbol	5.6.1.2	Indicates the end of the current repeat block, indefinite repeat block, or macro. The optional symbol, if used, must be identical to the macro name.
.EOT	5.5.7.2	Ignored. Indicates End-of-Tape which is detected automatically by the hardware.
.ERROR exp,string	5.6.5	Causes a text string to be output to the command device containing the optional expression specified and the indicated text string.
.EVEN	5.5.5.1	Ensures that the assembly location counter contains an even address by adding 1 if it is odd.
.FLT2 arg1,arg2,..	5.5.6.1	Generates successive two-word floating-point equivalents for the floating-point numbers specified as arguments.
.FLT4 arg1,arg2,..	5.5.6.1	Generates successive four-word floating-point equivalents for the

<u>Form</u>	<u>Described in Manual Section</u>	<u>Operation</u>
		floating-point numbers specified as arguments.
.GLOBL sym1,sym2,..	5.5.10	Defines the symbol(s) specified as global symbol(s).
.IDENT symbol	5.5.1.5	IDENT is currently a NOP.
.IF cond,arguments	5.5.11	Begins a conditional block of source code which is included in the assembly only if the stated condition is met with respect to the argument(s) specified.
.IFF	5.5.11.1	Appears only within a conditional block and indicates the beginning of a section of code to be assembled if the condition tested false.
.IFT	5.5.11.1	Appears only within a conditional block and indicates the beginning of a section of code to be assembled if the condition tested true.
.IFTF	5.5.11.1	Appears only within a conditional block and indicates the beginning of a section of code to be unconditionally assembled.
.IIF cond,arg, statement	5.5.11.2	Acts as a one-line conditional block where the condition is tested for the argument specified. The statement is assembled only if the condition tests true.
.IRP sym,<arg1,arg2,...>	5.6.6	Indicates the beginning of an indefinite repeat block in which the symbol specified is replaced with successive elements of the real argument list (which is enclosed in angle brackets).
.IRPC sym,string	5.6.6	Indicates the beginning of an indefinite repeat block in which the symbol specified takes on the value of successive characters in the character string.
.LIMIT	5.5.8	Reserves two words into which the Linker inserts the low and high addresses of the relocated code.
.LIST .LIST arg	5.5.1.1	Without an argument, .LIST increments the listing level count by 1. With an argument, .LIST does not alter the listing level count but formats the assembly listing according to the argument specified.



<u>Form</u>	<u>Described in Manual Section</u>	<u>Operation</u>
.MACRO sym,arg1,arg2,...	5.6.1.1	Indicates the start of a macro with the specified name containing the dummy arguments specified.
.MEXIT	5.6.1.3	Causes an exit from the current macro or indefinite repeat block.
.NARG symbol	5.6.4	Appears only within a macro definition and equates the specified symbol to the number of arguments in the macro call currently being expanded.
.NCHR sym<string>	5.6.4	Can appear anywhere in a source program; equates the symbol specified to the number of characters in the string (enclosed in delimiting characters).
.NLIST .NLIST arg	5.5.1.1	Without an argument, .NLIST decrements the listing level count by 1. With an argument, .NLIST deletes the portion of the listing indicated by the argument.
.NTYPE symbol,arg	5.6.4	Appears only in a macro definition and equates the low-order six bits of the symbol specified to the six-bit addressing mode of the argument.
.ODD	5.5.5.2	Ensures that the assembly location counter contains an odd address by adding 1 if it is even.
.PAGE	5.5.1.6	Causes the assembly listing to skip to the top of the next page.
.PRINT exp,string	5.6.5	Causes a text string to be output to the command device containing the optional expression specified and the indicated text string.
.RADIX n	5.5.4.1	Alters the current program radix to n, where n can be 2, 4, 8, or 10.
.RAD50 string	5.5.3.6	Generates a block of data containing the Radix-50 equivalent of the character string (enclosed in delimiting characters).
.REPT exp	5.6.7	Begins a repeat block. Causes the section of code up to the next .ENDM or .ENDR to be repeated exp times.
.SBTTL string	5.5.1.4	Causes the string to be printed as part of the assembly listing page header. The string part of each .SBTTL directive is collected into a table of contents at the beginning of the assembly listing.

<u>Form</u>	<u>Described in Manual Section</u>	<u>Operation</u>
.TITLE string	5.5.1.3	Assigns the first symbolic name in the string to the object module and causes the string to appear on each page of the assembly listing. One .TITLE directive should be issued per program.
.WORD exp1,exp2,...	5.5.3.2	Generates successive words of data containing the octal equivalent of the expression(s) specified.

APPENDIX G  
SYSTEM MACRO FILE

```
.TITLE SYSMAC V01 5/29/73  
;  
RT=11 SYSTEM MACROS  
;  
DEC=11-ORSYA=A-LA  
;  
COPYRIGHT 1973, DIGITAL EQUIPMENT CORPORATION,  
MAYNARD, MASSACHUSETTS 01754  
;  
DEC ASSUMES NO RESPONSIBILITY FOR THE  
USE OR RELIABILITY OF ITS SOFTWARE ON  
EQUIPMENT WHICH IS NOT SUPPLIED BY DEC.  
;
```

```
.MACRO .LOOKUP .CHANNEL, .DEVBLK  
.IF NB .DEVBLK  
MOV .DEVBLK, X^00  
.ENDC  
EMT ^020+^0<.CHANNEL>  
.ENDM
```

```
.MACRO .ENTER .CHANNEL, .DEVBLK, .LENGTH  
MOV .DEVBLK, X^00  
.IF B .LENGTH  
CLR -(X^06)  
.IFF  
MOV .LENGTH, -(X^06)  
.ENDC  
EMT ^040+^0<.CHANNEL>  
.ENDM
```

```
.MACRO .QSET .QADDR, .QLENG  
.IF NB .QLENG  
MOV .QLENG, X^00  
.ENDC  
MOV .QADDR, -(X^06)  
EMT ^0340+^013  
.ENDM
```

```
.MACRO .RENAME .CHANNEL, .DEVBLK  
.IF NB .DEVBLK  
MOV .DEVBLK, X^00  
.ENDC  
EMT ^0100+^0<.CHANNEL>  
.ENDM
```

```
.MACRO .SAVESTATUS .CHANNEL, .CBLOCK  
.IF NB .CBLOCK  
MOV .CBLOCK, X^00  
.ENDC  
EMT ^0120+^0<.CHANNEL>  
.ENDM
```

```

.MACRO .REOPEN .CHANNEL, .CBLOCK
.IF NB .CBLOCK
MOV .CBLOCK, X^00
.ENDC
EMT ^0140+^0< .CHANNEL>
.ENDM

.MACRO .CLOSE .CHANNEL
EMT ^0160+^0< .CHANNEL>
.ENDM

.MACRO .READW .CHANNEL, .BUFFER, .WCOUNT, .BLOCKN
.IF NB .BLOCKN
MOV .BLOCKN, X^00
.ENDC
CLR -(X^06)
MOV .WCOUNT, -(X^06)
MOV .BUFFER, -(X^06)
EMT ^0200+^0< .CHANNEL>
.ENDM

.MACRO .READ .CHANNEL, .BUFFER, .WCOUNT, .BLOCKN
.IF NB .BLOCKN
MOV .BLOCKN, X^00
.ENDC
MOV #^01, -(X^06)
MOV .WCOUNT, -(X^06)
MOV .BUFFER, -(X^06)
EMT ^0200+^0< .CHANNEL>
.ENDM

.MACRO .READC .CHANNEL, .BUFFER, .WCOUNT, .CROUTNE, .BLOCKN
.IF NB .BLOCKN
MOV .BLOCKN, X^00
.ENDC
MOV .CROUTNE, -(X^06)
MOV .WCOUNT, -(X^06)
MOV .BUFFER, -(X^06)
EMT ^0200+^0< .CHANNEL>
.ENDM

.MACRO .WRITW .CHANNEL, .BUFFER, .WCOUNT, .BLOCKN
.IF NB .BLOCKN
MOV .BLOCKN, X^00
.ENDC
CLR -(X^06)
MOV .WCOUNT, -(X^06)
MOV .BUFFER, -(X^06)
EMT ^0220+^0< .CHANNEL>
.ENDM

.MACRO .WRITE .CHANNEL, .BUFFER, .WCOUNT, .BLOCKN
.IF NB .BLOCKN
MOV .BLOCKN, X^00
.ENDC
MOV #^01, -(X^06)
MOV .WCOUNT, -(X^06)
MOV .BUFFER, -(X^06)
EMT ^0220+^0< .CHANNEL>
.ENDM

```

```

.MACRO .WRITC .CHANNEL, .BUFFER, .WCOUNT, .CROUTNE, .BLOCKN
  .IF NB .BLOCKN
  MOV .BLOCKN, X^00
  .ENDC
  MOV .CROUTNE, -(X^06)
  MOV .WCOUNT, -(X^06)
  MOV .BUFFER, -(X^06)
  EMT ^0220+^0<.CHANNEL>
  .ENDM

```

```

.MACRO .WAIT .CHANNEL
  EMT ^0240+^0<.CHANNEL>
  .ENDM

```

```

.MACRO .TTYIN .CHAR
  EMT ^0340
  BCS .-2
  .IF NB .CHAR
  MOVB X^00, .CHAR-
  .ENDC
  .ENDM

```

```

.MACRO .TTINR
  EMT ^0340
  .ENDM

```

```

.MACRO .TTYOUT .CHAR
  .IF NB .CHAR
  MOVB .CHAR, X^00
  .ENDC
  EMT ^0340+^01
  BCS .-2
  .ENDM

```

```

.MACRO .TTOUTR
  EMT ^0340+^01
  .ENDM

```

```

.MACRO .DSTATUS .RETSPC, .DEVNAME
  .IF NB .DEVNAME
  MOV .DEVNAME, X^00
  .ENDC
  MOV .RETSPC, -(X^06)
  EMT ^0340+^02
  .ENDM

```

```

.MACRO .FETCH .CORADD, .DEVNAME
  .IF NB .DEVNAME
  MOV .DEVNAME, X^00
  .ENDC
  MOV .CORADD, -(X^06)
  EMT ^0340+^03
  .ENDM

```

```

.MACRO .RELEAS .DEVNAME
  .IF NB .DEVNAME
  MOV .DEVNAME, X^00
  .ENDC
  CLR -(X^06)
  EMT ^0340+^03
  .ENDM

```

```

.MACRO .CSIGEN .DEVSPC, .DEFEXT, .CSTRING
MOV .DEVSPC,=(X^06)
MOV .DEFEXT,=(X^06)
MOV .CSTRING,=(X^06)
EMT ^0340+^04
.ENDM

.MACRO .CSISPC .OUTSPC, .DEFEXT, .CSTRING
MOV .OUTSPC,=(X^06)
MOV .DEFEXT,=(X^06)
MOV .CSTRING,=(X^06)
EMT ^0340+^05
.ENDM

.MACRO .LOCK
EMT ^0340+^06
.ENDM

.MACRO .UNLOCK
EMT ^0340+^07
.ENDM

.MACRO .EXIT
EMT ^0340+^010
.ENDM

.MACRO .SRESET
EMT ^0340+^012
.ENDM

.MACRO .PRINT .MSGADDR
.IF NB .MSGADDR
MOV .MSGADDR,X^00
.ENDC
EMT ^0340+^011
.ENDM

.MACRO .DELETE .CHANNEL, .DEVBLK
.IF NB .DEVBLK
MOV .DEVBLK,X^00
.ENDC
EMT ^0<,CHANNEL>
.ENDM

.MACRO .SETTOP .TOP
.IF NB .TOP
MOV .TOP,X^00
.ENDC
EMT ^0340+^014
.ENDM

.MACRO .RCTRLO
EMT ^0340+^015
.ENDM

.MACRO .HRESET
EMT ^0340+^017
.ENDM

.MACRO .DATE
MOV #^054,X^00
MOV ^0262(X^00),X^00
.ENDM

```

APPENDIX H

SUMMARY OF MONITOR PROGRAMMED REQUESTS

<u>Mnemonic</u>	<u>Function</u>	<u>MACRO Call</u> (see note 1)	<u>Assembly Language Expansion</u> (see note 2)	<u>Error Codes</u>
.CLOSE	Terminates activities on specified channel	.CLOSE .channel	EMT ↑0160+↑0<.channel>	None
.CSIGEN	Calls CSI in general mode	.CSIGEN .devspc, .defext, .cstring	MOV .devspc,-(↑06) MOV .defext,-(↑06) MOV .cstring,-(↑06) EMT ↑0340+↑04	Byte 52= 0-illegal command 1-device not found 2-unused 3-full directory 4-input file not found
.CSISPC	Calls CSI in special mode	.CSISPC .outspc, .defext, .cstring	MOV .outspc,-(↑06) MOV .defext,-(↑06) MOV .cstring,-(↑06) EMT ↑0340+↑05	0-illegal command 1-illegal device
.DATE	Moves current date into R0	.DATE	MOV @#↑054,↑00 MOV ↑0262 (↑00),↑00	None
.DELETE	Deletes named file	.DELETE .channel, .devblk	.IF NB .devblk MOV .devblk,↑00 .ENDC EMT ↑0<.channel> .ENDM	0-open channel 1-file not found
.DSTATUS	Provides data about device	.DSTATUS .cblock, .devname	.IF NB .devname MOV .devname,↑00 .ENDC MOV .retspc,-(↑06) EMT ↑0340+↑02	0-device not found

<u>Mnemonic</u>	<u>Function</u>	<u>MACRO Call</u> (see note 1)	<u>Assembly Language Expansion</u> (see note 2)	<u>Error Codes</u>
.ENTER	Allocates space on device and creates a tentative entry	.ENTER .channel, .devblk, .length	MOV .devblk,%↑00 .IF B .length CLR -(%↑06) .IFF MOV .length,-(%↑06) .ENDC EMT ↑40+↑0<.channel>	0-Open channel 1-no space <= .length available
.EXIT	Terminates user program and returns to KMON.	.EXIT	EMT ↑0340+↑010	None
.FETCH	Loads device handlers	.FETCH .coradd, .devname	.IF NB .devname MOV .devname,%↑00 .ENDC MOV .coradd,-(%↑06) EMT ↑0340+↑03	0-nonexistent device name or no handler for that device
.HRESET	Does SRESET and RESET instruction.	.HRESET	EMT ↑0340+↑017	None
.LOCK	Locks the USR in core	.LOCK	EMT ↑0340+↑06	None
.LOOKUP	Associates channel # and file	.LOOKUP .channel, .devblk	.IF NB .devblk MOV .devblk, %↑00 .ENDC EMT ↑020+↑0<.channel>	0-channel open 1-file not found



<u>Mnemonic</u>	<u>Function</u>	<u>MACRO Call</u> (see note 1)	<u>Assembly Language Expansion</u> (see note 2)	<u>Error Code</u>
.PRINT	Outputs message on terminal	.PRINT msgaddr	.IF NB .msgaddr MOV .msgaddr, %t00 .ENDC EMT t0340+t011	None
.QSET	Enlarges I/O queue for monitor.	.QSET .qaddr, .qleng	.IF NB .qleng MOV .qleng, %t00 .ENDC MOV .qaddr, -(%t06) EMT t0340+t013	None
.RCTRL0	Enables terminal printing	.RCTRL0	EMT t0340+t015	None
.READ	Transfers words from channel to core	.READ .channel, .buffer, .wcount, .blockn	.IF NB .blockn MOV .blockn, %t00 .ENDC MOV #t01, -(%t06) MOV .wcount, -(%t06) MOV .buffer, -(%t06) EMT t0200+t0<.channel>	0-end of input file reached 1-error on channel 2-channel not open
.READC	Transfers words from channel to core. When read is complete, control passes to specified routine.	.READC .channel, .buffer, .wcount, .croutine, .blockn	.IF NB .blockn MOV .blockn, %t00 .ENDC MOV .croutine, -(%t06) MOV .wcount, -(%t06) MOV .buffer, -(%t06) EMT t0200+t0 .channel	0-end of input file reached 1-error on channel 2-channel not open.

<u>Mnemonic</u>	<u>Function</u>	<u>MACRO Call</u> (see note 1)	<u>Assembly Language Expansion</u> (see note 2)	<u>Error Code</u>
.READW	Transfers words from channel to core. When read is complete, control returns to user program	.READW .channel, .buffer, .wcount, .blockn	.IF NB .blockn MOV .blockn, %†00 .ENDC CLR -(%†06) MOV .wcount, -(%†06) MOV .buffer, -(%†06) EMT †0200+†0<.channel>	0-end of input file reached 1-hard error occurred on channel 2-channel not open.
.RELEAS	Removes handler from core	.RELEAS .devname	.IF NB .devname MOV .devname, %†00 .ENDC CLR -(%†06) EMT 0340+†03	0-illegal handlername
.RENAME	Changes file name	.RENAME .channel, .devblk	.IF NB .devblk MOV .devblk, %†00 .ENDC EMT 0100+†0<.channel>	0-channel open 1-file not found
.REOPEN	Reassociates channel with SAVESTATUSed file.	.REOPEN .channel, .cblock	.IF NB .cblock MOV .cblock, %†00 .ENDC EMT †0140+†0<.channel>	0-channel in use
.SAVESTATUS	Stores five file definition data words in core	.SAVESTATUS .channel, .cblock	.IF NB .cblock MOV .cblock, %†00 .ENDC EMT †0120+†0<.channel>	0-channel not open 1-SAVESTATUS illegal

<u>Mnemonic</u>	<u>Function</u>	<u>MACRO Call</u> (see note 1)	<u>Assembly Language</u> <u>Expansion</u> (see note 2)	<u>Error Code</u>
.SETTOP	Specifies core required for program	.SETTOP .top	.IF NB .top MOV .top, %↑00 .ENDC EMT ↑0340+↑014	None
.SRESET	Resets data base of system	.SRESET	EMT ↑0340+012	None
.TTYIN	Inputs character from terminal waits till done	.TTYIN .char	EMT ↑0340 BCS.-2 .IF NB .char MOVB %↑00, .char	None  .ENDC
.TTINR	Inputs character from terminal	.TTINR	EMT ↑0340	None
.TTYOUT	Outputs character to terminal, waits till done	.TTYOUT .char	.IF NB .char MOVB .char,%↑00 .ENDC EMT ↑0340+↑01 BCS .-2	None
.TTOUTR	Outputs character to terminal	.TTOUTR	.EMT ↑0340+↑01	None
.UNLOCK	Release USR from core	.UNLOCK	.EMT ↑0340+↑07	None
.WAIT	Suspends processing until channel I/O is complete	.WAIT .channel	EMT ↑240+↑0<.channel>	0-channel not open 1-channel hardware error

<u>Mnemonic</u>	<u>Function</u>	<u>MACRO Call</u> (see note 1)	<u>Assembly Language</u> <u>Expansion</u> (see note 2)	<u>Error Code</u>
.WRITC	Transfers words from core to channel	.WRITC .channel,.buffer,.wcount,.croutine,.blockn	.IF NB .blockn MOV .blockn, %↑00 .ENDC MOV .croutine, -(%↑06) MOV .wcount, -(%↑06) MOV .buffer, -(%↑06) EMT ↑0220+↑0<.channel> .ENDM	0-end of output file reached 1-hardware error 2-channel not open
.WRITE	Transfers words from core to channel when write begins	.WRITE .channel,.buffer,.wcount,.blockn	.IF NB .blockn MOV .blockn, %↑00 .ENDC MOV #↑01, -(%↑06) MOV .wcount, -(%↑06) MOV .buffer, -(%↑06) EMT ↑0220+↑0<.channel> .ENDM	0-attempted to write past end of file 1-hardware error 2-channel not open
.WRITW		.WRITW .channel,.buffer,.wcount,.blockn	.IF NB .blockn MOV .blockn, %↑00 .ENDC CLR -(%↑06) MOV .wcount, -(%↑06) MOV .buffer, -(%↑06) EMT ↑0220+↑0<.channel> .ENDM	0-attempted to write past end of file 1-hardware error 2-channel not open

#### NOTE 1

.blockn	Block number to be read relative to the start of the file.
.buffer	Address of the buffer to receive data.
.cblock	Address of core block where channel status is stored.
.channel	The octal channel number (0-7).
.char	Storage location of character
.coradd	Address for storage of device handler.
.croutine	Address of routine to be executed when read is complete.
.cstring	Address of input string or 0.
.defext	Address of 4-word block containing RAD50 default extensions.
.devblk	Four-word RAD50 file description of the file to be opened.
.devname	Pointer to RAD50 device name.
.devspc	Address of core area for storage of device handlers.
.length	# of blocks allocated to file being opened.
.msgaddr	Address of ASCIZ string to be printed.
.outspc	Address of 39-word block containing file descriptions.
.qaddr	Address of first entry of new queue area.
.qleng	Number of entries to be added.
.wcount	Number of words to be read.

#### NOTE 2

The lines preceded by a dot will not be assembled. The code they enclose may or may not be assembled depending on the conditionals.



APPENDIX I  
EDITOR SUMMARY

I.1 EDIT KEY COMMANDS

<u>Key</u>	<u>Explanation</u>
ALTMODE	A single ALTMODE terminates a text string. A double ALTMODE executes the command string.
CTRL/C	Terminates execution of EDIT commands and returns to Monitor command mode. A double CTRL/C is necessary when I/O is in progress.
CTRL/O	Terminates printing on the terminal until completion of the current output or a second CTRL/O.
CTRL/U	Deletes all characters on current terminal input line.
CTRL/X	Causes the entire command string to be ignored and EDIT types another *.
RUBOUT	Deletes a character from the current line and echoes a \ and the character deleted.
TAB	Spaces to next tab stop.

I.2 EDIT COMMANDS

<u>Command</u>	<u>Format</u>	<u>Explanation</u>
ADVANCE	nA	Advance pointer n lines to beginning of the n+1 line.
	0A	Advance to beginning of current line.
	/A	Advance pointer to end of text buffer.
BEGINNING	B	Move the current location pointer to the beginning of the text buffer.
CHANGE	nCtext\$	Replace n characters with specified text.
	0Ctext\$	Replace characters from beginning of line to pointer with specified text.
	/Ctext\$	Replace characters from the pointer to the end of the buffer with the specified text.

<u>Command</u>	<u>Format</u>	<u>Explanation</u>
	=Ctext\$	Replace n characters to the left of the pointer with the text where n equals the length of the last text argument used.
DELETE	nD	Delete n characters to the right of the pointer.
	0D	Delete from pointer to beginning of current line.
	/D	Delete from pointer to end of text buffer.
	=D	Delete n characters to the left of the pointer, where n equals the length of the last text argument used.
EDIT BACKUP	EBdev:filnam.ext [n] \$	Open a file for editing without changing the file name.
EDIT READ	ERdev:filnam.ext\$	Open a file for input.
EDIT VERSION	EV	Display the version number of the Editor on the console terminal.
EDIT WRITE	EWdev:filnam.ext [n] \$	Create a new file for output.
END FILE	EF	Close the current output file.
EXCHANGE	nXtext\$	Replace n lines with specified text.
	0Xtext\$	Replace the current line from the beginning to the pointer with the specified text.
	/Xtext\$	Replace the lines from the pointer to the end of the buffer with the specified text.
EXECUTE MACRO	nEM	Execute the command string specified in the last macro command.
EXIT	EX	Terminate editing and return control to Monitor after outputting remainder of input file.
FIND	nFtext\$	Search entire input file starting at pointer for nth occurrence of text string. Write contents of buffer to output file after unsuccessful search.
GET	nGtext\$	Search current text buffer starting at pointer for nth occurrence of text string.



<u>Command</u>	<u>Format</u>	<u>Explanation</u>
INSERT	Itext\$	Place specified text in text buffer.
JUMP	nJ	Move pointer n characters.
	0J	Move pointer to beginning of current line.
	/J	Move pointer to end of text buffer.
	=J	Move pointer backward n characters, where n equals the length of the last text argument used.
KILL	nK	Delete n lines from text buffer.
	0K	Delete from start of current line to pointer.
	/K	Delete from pointer to end of text buffer.
LIST	nL	Print n lines on terminal beginning at pointer.
	0L	Print from beginning of current line to pointer.
	/L	Print from pointer to end of buffer.
MACRO	M/command string/	Insert a command string into the EDIT Macro Buffer. / represents the delimiter character.
	0M	Clear MACRO buffer and reclaim MACRO buffer for text.
NEXT	nN	Write contents of current text buffer to output file and read next page of input file.
POSITION	nPtext\$	Search input file for nth occurrence of text string. Delete contents of buffer after search.
READ	R	Move the next page of text into the text buffer.
SAVE	nS	Copy the specified number of lines starting at the pointer into the Save Buffer.
VERIFY	V	Print current text line on terminal.
WRITE	nW	Write n lines of text to the output file.
	0W	Write text from beginning of current line to pointer.
	/W	Write the text from the pointer to the end of the buffer.

<u>Command</u>	<u>Format</u>	<u>Explanation</u>
UNSAVE	U	Insert the contents of the Save Buffer into the text buffer at the pointer location.
	OU	Clear the Save Buffer and reclaim the area for text.
VERIFY	V	Print the current text line on the terminal.

APPENDIX J  
LINKER SUMMARY

This appendix summarizes the Linker operating procedures and command switches; for more detailed information refer to Chapter 6.

J.1 OPERATING PROCEDURES

Call the Linker with the command

R LINK

in response to the keyboard Monitor's dot. LINK replies with

\*

and waits for a command string of the following format:

dev:binout,dev:mapout<dev:obj1,dev:obj2,.../S1/S2/S3

which specifies the random access device (dev:) for the save image output file (binout); the load map file (mapout); the input modules (obj1, obj2, etc.) and the command switches (S1/S2...etc).

The switches and the command line on which they must appear are:

<u>Switch</u>	<u>Command Line</u>	<u>Explanation</u>
/A	lst	Alphabetize switch - lists the linked modules in alphabetical order on the load map.
/B:n	lst	Bottom switch - specifies the lowest address to be used by the program.
/C	any	Continue switch - allows additional lines of command string input.
/L	lst	LDA format switch - Outputs file in LDA format instead of save image format.
/O:n	any but the lst	Overlay switch - segments the program so all of it is not core resident at one time.
/T:n	lst	Transfer switch - specifies the starting address of the program. If the argument is missing, LINK prints:

TRANSFER ADDRESS:

and waits for specification of the global symbol whose value is the transfer address.

When the command string is entered, LINK links the specified modules and outputs a load module and a load map if requested.



APPENDIX K

ODT SUMMARY

In the command format shown below r represents a relocatable expression and n represents an octal number.

<u>Command</u>	<u>Format</u>	<u>Explanation</u>
RETURN		Close open location and accept the next command.
LINE FEED		Close current location; open next sequential location.
↑ or	↑ or	Open previous location.
+or	+or	Index the contents of the opened location by the contents of the PC and open the resulting location.
>	>	Take contents of opened location as relative branch instruction and open referenced location.
<	<	Return to sequence prior to last @, >, or + command and open succeeding location.
@	@	Take contents of opened location as absolute address and open that location.
/	/	Reopen the last opened location.
	r/	Open the word at location r.
\	\	Reopen the last opened byte (SHIFT/L).
	r\	Open the byte at location r.
!	!	Print address of opened location relative to relocation register whose contents are closest.
	n!	Print address of opened location relative to relocation register n.
\$	\$n/	Open general register n (0-7).
	\$B/	Open first word of the breakpoint table.
	\$C/	Open Constant register.

<u>Command</u>	<u>Format</u>	<u>Explanation</u>
	\$F/	Open Format register.
	\$P/	Open Priority register.
	\$R/	Open first relocation register (register 0).
	\$S/	Open Status register.
A	r;nA	Starting at location r, print n bytes in their ASCII format; then input n bytes from the terminal starting at location r.
B	;B	Remove all Breakpoints.
	r;B	Set Breakpoint at location r.
	r;nB	Set Breakpoint n at location r.
	;nB	Remove the nth Breakpoint.
C	r;C	Print the value of r and store it in the Constant register.
E	r;E	Search for instructions that reference effective address r.
F	;F	Fill memory words with contents of the Constant register.
G	r;G	Go to location r and start program.
I	;I	Fill memory bytes with the low-order 8 bits of the Constant register.
O	r;O	Calculate offset from currently open location to r.
P	;P	Proceed with program execution from breakpoint. In single instruction mode only, execute next instruction.
	k;P	Proceed with program execution from breakpoint; stop after encountering the breakpoint k times. In single instruction mode only, execute next k instructions.
R	;R	Set all relocation registers to -1 (highest address value).
	R	Select relocation register whose contents are closest to but less than or equal to contents of the opened location. Subtract contents of register from contents of opened word and print result.

<u>Command</u>	<u>Format</u>	<u>Explanation</u>
	nR	Subtract contents of relocation register n from contents of opened word and print result.
S	;S	Disable single-instruction mode; reenable breakpoints.
	;nS	Enable single-instruction mode (n can have any value and is not significant); disable breakpoints.
W	r;W	Search for words with bit patterns which match r.
X	X	Perform a Radix 50 unpack of the binary contents of the current opened word; then permit the storage of a new Radix 50 binary number in the same location.





APPENDIX L  
BASIC/RT11 SUMMARY

L.1 BASIC/RT11 STATEMENTS

BASIC/RT11 is available to use with the RT-11 system and may be purchased separately.

The following summary of BASIC statements defines the general format for the statement and gives a brief explanation of its use.

CALL "function name" (argument list)	Used to reference assembly language user functions from a BASIC program.
CHAIN "dev:filnam.ext" LINE number	Terminates execution of user program, loads and executes the specified program starting at the line number if included.
CLOSE { VFn #n }	Closes the logical file specified. If no file is specified closes all files which are open.
DATA data list	Used in conjunction with READ to input data into an executing program.
DEF function (argument)=expression	Defines a user function to be used in the program.
DIM variable(n), variable(n,m), variable\$(n), variable\$(n,m)	Reserves space for lists and tables according to subscripts specified after variable name.
END	Placed at the physical end of the program to terminate program execution.
FOR variable = expression1 TO expression2 STEP expression3	Sets up a loop to be executed the specified number of times.
GOSUB line number	Used to transfer control to the first line of a subroutine.
GOTO line number	Used to unconditionally transfer control to other than the next sequential line in the program.
IF expression relation expression2 THEN line number GOTO	Used to conditionally transfer control to the specified line of the program.
IF END #n { THEN GOTO } line number	Used to test for end of file on sequential input file #n:

INPUT list	Used to input data from the terminal keyboard or papertape reader.
INPUT #expression: list	Inputs from a particular file.
LET variable = expression	Used to assign a value to the specified variable(s).
LET VFn(i) = expression	Used to set value of a virtual memory file element.
NEXT variable	Placed at the end of a FOR loop to return control to the FOR statement.
OPEN file FOR $\left\{ \begin{array}{l} \text{INPUT} \\ \text{OUTPUT} \end{array} \right\}$ AS FILE #n [DOUBLEBUF]	Opens a sequential file for input or output as specified. File may be of the form "dev:filename.ext" or a scalar string variable, e.g. A\$
OPEN file FOR $\left\{ \begin{array}{l} \text{INPUT} \\ \text{OUTPUT} \end{array} \right\}$ AS FILE VFn x [dimension] [=string length]	Opens a virtual memory file for input or output. File may be of the form "dev:filename.ext" or a scalar string variable. x represents the type of file floating point (blank), integer (%), contains character strings (\$).
OVERLAY "file description"	Used to overlay or merge program in core with specified file.
PRINT list	Used to output data to the terminal. The list can contain expressions or text strings.
PRINT "text"	Used to print a message or a string of characters.
PRINT #expression: expression list	Outputs to a particular file.
PRINT TAB(x)	Used to space to the specified column.
RANDOMIZE	Causes the random number generator to calculate different random numbers every time the program is run.
READ variable list	Used to assign the values listed in a DATA statement to the specified variables.
REM comment	Used to insert explanatory comments into a BASIC program.
RESTORE	Used to reset data block pointer so the same data can be used again.
RESTORE #n	Rewinds the specified input sequential file.

RETURN Used to return program control to the statement following the original GOSUB statement.

STOP Used at the logical end of the program to terminate execution.

## L.2 Commands

The following key commands halt program execution, erase characters or delete lines.

<u>Key</u>	<u>Explanation</u>
ALTMODE	Deletes the entire current line. Echoes DELETED message (same as CTRL/U). On some terminals the ESC key must be used.
CTRL/C	Terminates program execution. BASIC returns to the RT-11 monitor.
CTRL/O	Stops output to terminal and returns BASIC to READY message when program or command execution is completed.
CTRL/U	Deletes the entire current line. Echoes DELETED message (same as ALTMODE).
←	(SHIFT/O) Deletes the last character typed and echoes a backarrow (same as RUBOUT). On VT05 or LA30 use the underscore (-) key.
RUBOUT	Deletes the last character typed and echoes a backarrow (same as ←).

The following commands list, punch, erase, execute and save the program currently in core.

<u>Command</u>	<u>Explanation</u>
CLEAR	Sets the array and string buffers to nulls and zeroes.
LIST	Prints the user program currently in core on the terminal.
LIST [line number] [ [END line number] ]	
LISTNH [line number] [ [END line number] ]	
	Lists the lines associated with the specified numbers but does not print a header.
NEW ["filnam"]	Does a SCRatch and sets the current program name to the one specified.
OLD ["dev:filnam.ext"]	Does a SCRatch and inputs the program from the specified file.

<u>Command</u>	<u>Explanation</u>
RENAME ["filnam"]	Changes the current program name to the one specified.
REPLACE "dev:filnam.ext"	Destroys the specified file, writing the program in core to the file.
RUN	Executes the program in the buffer area.
RUNNH	Executes the program in the buffer area but does not print a header line.
SAVE ["dev:filnam.ext"]	Outputs the program in core to the specified device or file.
SCRatch	Erases the entire storage area keeping the same program name.

### L.3 Functions

The following functions perform standard mathematical operations in BASIC.

<u>Name</u>	<u>Explanation</u>
ABS(x)	Returns the absolute value of x.
ATN(x)	Returns the arctangent of x as an angle in radians in the range + or - pi/2.
BIN(x)	Computes the integer value of a string of 1's and 0's.
COS(x)	Returns the cosine of x radians.
EXP(x)	Returns the value of e <sup>x</sup> where e=2.71828.
INT(x)	Returns the greatest integer less than or equal to x.
LOG(x)	Returns the natural logarithm of x.
OCT(x)	Computes an integer value from a string of blanks and digits from 0 to 7.
RND(x)	Returns a random number between 0 and 1.
SGN(x)	Returns a value indicating the sign of x.
SIN(x)	Returns the sine of x radians.
SQR(x)	Returns the square root of x.
TAB(x)	Causes the terminal type head to move to column number x.

The string functions are:

<u>Name</u>	<u>Explanation</u>
ASC(x\$)	Returns as a decimal number the seven-bit internal code for the one-character string (x\$).
CHR\$(x)	Generates a one-character string having the ASCII value of x.
DAT\$	Returns a string containing the current date in the format 5-MAY-73.
LEN(x\$)	Returns the number of characters in the string (x\$).
POS(x\$,y\$,z)	Searches for and returns the position of the first occurrence of y\$ in x\$ starting at the zth position.
SEG\$(x\$,y\$,z)	Returns the string of characters in positions y through z in x\$.
STR\$(x)	Returns the string which represents the numeric value of x.
TRM\$(x\$)	Returns the string x\$ with trailing blanks removed.
VAL(x\$)	Returns the number represented by the string (x\$).

#### L.4 BASIC ERROR MESSAGES

<u>Abbrevia- tion</u>	<u>Message</u>	<u>Explanation</u>
?ARG	ARGUMENT ERROR AT LINE xxxxx	Arguments in a function call do not match, in number or in type, the arguments defined for the function.
?ATL	ARRAYS TOO LARGE AT LINE xxxxx	There is not enough room in the core available for the arrays specified in the DIM statements.
?BDR	BAD DATA READ AT LINE xxxxx	Item input from DATA statement list by READ statement is bad.
?BRT	BAD DATA-RETYPE FROM ERROR	Item entered to input statement is bad.
?BSO	BUFFER STORAGE OVERFLOW AT LINE xxxxx	Not enough room available in file buffers.
?DCE	DEVICE CHANNEL ERROR AT LINE xxxxx	The device channel number specified for a sequential or virtual memory file is out of range (1-7), or tried to open a virtual memory file on a non-file structured device.

<u>Abbrevia- tion</u>	<u>Message</u>	<u>Explanation</u>
?DNR	DEVICE NOT READY	An OLD command read a file which did not have any BASIC statements.
?DRF	DIRECTORY FULL	The device directory is full and cannot accomodate another file.
?DV0	DIVISION BY 0 AT LINE xxxxxx	Program attempted to divide some quantity by 0.
?ETC	EXPRESSION TOO COMPLEX AT LINE xxxxxx	The expression being evaluated caused the stack to overflow usually because the parentheses are nested too deeply.  The degree of complexity that produces this error varies according to the amount of space available in the stack at the time. Breaking the statement up into several simpler ones eliminates the error.
?FDE	FILE DATA ERROR AT LINE xxxxxx	Tried to write an element on an integer virtual memory file outside the range (x) <32,768.
?FIO	FILE I/O ERROR AT LINE xxxxxx	A I/O error occurred. All files are automatically closed.
?FNF	FILE NOT FOUND AT LINE xxxxxx	The file requested was not found on the specified device.
?FNO	FILE NOT OPEN AT LINE xxxxxx	The sequential or virtual memory file referenced is not open.
?FTS	FILE TOO SHORT AT LINE xxxxxx	The sequential file space allocated to an output file is inadequate.
?FWN	FOR WITHOUT NEXT AT LINE xxxxxx	The program contains a FOR statement without a corresponding NEXT statement to terminate the loop.
?GND	GOSUBS NESTED TOO DEEPLY AT LINE xxxxxx	Program GOSUBS nested to more than 20 levels.
?IDF	ILLEGAL DEF AT LINE xxxxxx	The DEF statement contains an error.
?IDM	ILLEGAL DIM AT LINE xxxxxx	Syntax error in a dimension statement.
?ILN	ILLEGAL NOW	An attempt was made to execute an INPUT statement in immediate mode.

<u>Abbrevia- tion</u>	<u>Message</u>	<u>Explanation</u>
?ILR	ILLEGAL READ AT LINE xxxxxx	Tried to open a write-only device for input or tried to read on a sequential file open for output.
?LTL	LINE TOO LONG	The line being typed is longer than 120 characters; the line buffer overflows.
?NBF	NEXT BEFORE FOR AT LINE xxxxxx	The NEXT statement corresponding to a FOR statement precedes the FOR statement.
?NER	NOT ENOUGH ROOM AT LINE xxxxxx	There is not enough room on the selected device for the specified number of output blocks.
?NPR	NO PROGRAM	The RUN command has been specified, but no program has been typed in.
?NSM	NUMBERS AND STRINGS MIXED AT LINE xxxxxx	String and numeric variables may not appear in the same expression, nor may they be set equal to each other as A\$=2.
?OOD	OUT OF DATA AT LINE xxxxxx	The data list was exhausted and a READ requested additional data.
?OVF	OVERFLOW AT LINE xxxxxx	The result of a computation is too large for the computer to handle.
?PTB	PROGRAM TOO BIG	The line just entered caused the program to exceed the user code area.
?RBG	RETURN BEFORE GOSUB AT LINE xxxxxx	A RETURN was encountered before execution of a GOSUB statement.
?SOB	SUBSCRIPT OUT OF BOUNDS AT LINE xxxxxx	The subscript computed is greater than 32,767 or is outside the bounds defined in the DIM statement.
?SSO	STRING STORAGE OVERFLOW AT LINE xxxxxx	There is not enough core available to store all the strings used in the program.
?STL	STRING TOO LONG AT LINE xxxxxx	The maximum length of a string in a BASIC statement is 255 characters.
?SYN	SYNTAX ERROR AT LINE xxxxxx	The program has encountered an unrecognizable statement. Common examples of syntax errors are misspelled commands and unmatched parentheses, and other typographical errors.

<u>Abbrevia- tion</u>	<u>Message</u>	<u>Explanation</u>
?TTLT	LINE TOO LONG TO TRANSLATE	Lines are translated as entered and the line just entered exceeds the area available for translation.
?UFN	UNDEFINED FUNCTION AT LINE xxxxx	The function called was not defined by the program or was not loaded with BASIC.
?ULN	UNDEFINED LINE NUMBER AT LINE xxxxx	The line number specified in an IF, GOTO or GOSUB statement does not exist anywhere in the program.
?WLO	WRITE LOCKOUT AT LINE xxxxx	Tried to open a read-only device for output, or tried to write on a sequential or virtual file opened for input only.
?↑ER	↑ERROR AT LINE xxxxx	The program tried to compute the value $A+B$ , where A is less than 0 and B is not an integer. This produces a complex number which is not represented in BASIC.

#### Function Errors

The following errors can occur when a function is called improperly.

?ARG	The argument used is the wrong type. For example, the argument was numeric and the function expected a string expression.
?SYN	The wrong number of arguments was used in a function, or the wrong character was used to separate them. For example, PRINT SIN(X,Y) produces a syntax error.

In addition, the functions give the errors listed below.

FNa(...)	?UFN	The function a has not been defined (function cannot be defined by an immediate mode statement).
RND or RND(X)		No errors
SIN(X)		No errors
COS(X)		No errors
SQR(X)	?ARG	X is negative
ATN(X)		No errors
EXP(X)	?↑ER	X is greater than 87
LOG(X)	?ARG	X is negative or 0



ABS (X)		No errors
INT (X)		No errors
SGN (X)		No errors
TAB (X)	?ARG	X is not in the range $0 \leq x < 256$
LEN (A\$)		No errors
ASC (A\$)	?ARG	A\$ is not a string of length 1
CHR\$ (X)	?ARG	X is not in the range $0 < x < 256$
POS (A\$, B\$, N)		No errors
SEG\$ (A\$, N1, N2)		No errors
VAL (A\$)	?ARG	A\$ is not a valid numeric expression
STR\$ (X)		No errors
BIN (A\$)	?ARG	Characters other than 0, 1 or blank are included in A\$, or the number exceeds the range $-2^{15} < n < 2^{15}$ .
DAT\$		No errors
OCT (A\$)	?ARG	Characters other than 0 through 7 or blank are included in A\$, or the number exceeds the range $-2^{15} < n < 2^{15}$ .
TRM\$ (X\$)		No errors



APPENDIX M

MACRO PERMANENT SYMBOL TABLE

PST V01 PERMANENT SYMBO RT-11 MACRO VM01-01 31-AUG-73 PAGE 1

```

1      ;      RT-11 MACRO ASSEMBLER   VM01-01
2      ;      DEC-11-ORMAA-A-LA
3      ;      MAY 10, 1973
4      ;      BOB BOWERING
5      ;      ERIC PETERS
6
7      ;      COPYRIGHT 1973
8      ;      DIGITAL EQUIPMENT CORPORATION
9      ;      MAYNARD, MASSACHUSETTS 01754
10
11     ; DEC ASSUMES NO RESPONSIBILITY FOR THE
12     ; USE OR RELIABILITY OF ITS SOFTWARE ON
13     ; EQUIPMENT WHICH IS NOT SUPPLIED BY DEC.
14
15     .SBTTL  RT-11 MACRO PARAMETER FILE
16
17     000000 RT11= 0
18     000000 XBAW= 0
19     000000 XCHEF= 0
20     000000 XEDABS= 0
21     000000 XEJPIC= 0
22     000000 XEJCDR= 0
23     000000 XS=IT= 0
24
25     .TITLE  PST   V01           PERMANENT SYMBOL TABLE
26
27     .IDENT  /V01/
28
29     ;      COPYRIGHT 1972  DIGITAL EQUIPMENT CORPORATION
30     ;      15 NOV 72
31
32     .GLOBL  PSTBAS, PSTTOP           ;LIMITS
33     .GLOBL  WRDSYM                   ;POINTER TO ,WORD
34
35     000200 DR1= 200                   ;DESTRUCTIVE REFERENCE IN FIRST
36     000100 DR2= 100                   ;DESTRUCTIVE REFERENCE IN SECOND
37
38     .GLOBL  DFLGEV, DFLGBM, DFLCND, DFLMAC, DFLSMC
39
40     000020 DFLGEV= 020                 ;DIRECTIVE REQUIRES EVEN LOCATIO
41     000010 DFLGBM= 010                 ;DIRECTIVE USES BYTE MODE
42     000004 DFLCND= 004                 ;CONDITIONAL DIRECTIVE
43     000002 DFLMAC= 002                 ;MACRO DIRECTIVE
44     000001 DFLSMC= 001                 ;MCALL
45
46     .IF OF  PAL11R                     ;PAL11R SUBSET
47     XMACRO= 0
48     X40= 0
49     X45= 0
50
51     .ENDC
52
53     .IIF OF X40&X45,                   XFLTG= 0
54     .IIF OF XMACRO, XSML= 0
55
56     .MACRO  UPCDEF  NAME,  CLASS,  VALUE,  FLAGS,  COND
57     .IF NB  <COND>
58     .IF OF  COND
59     .MEXIT

```

```
59 .ENDC
60 .ENDC
61 .RAD50 /NAME/
62 .BYTE FLAGS+0
63 .GLOBL OPCL'CLASS
64 .BYTE 200+OPCL'CLASS
65 .WORD VALUE
66 .ENDM
67
68 .MACRO DIRDEF NAME, FLAGS, COND
69 .RAD50 /.'NAME/
70 .BYTE FLAGS+0, 0
71 .IF NB <COND>
72 .IF OF COND
73 .GLOBL OPCERR
74 .WORD OPCERR
75 .MEXIT
76 .ENDC
77 .ENDC
78 .GLOBL NAME
79 .WORD NAME
80 .ENDM
81
82 00000 PSTBAS: ;BASE
```

1	000000	OPCDEF	<ABSU >	01,	170600,	DR1,	X45
2	000010	UPCDEF	<ABSF >	01,	170600,	DR1,	X45
3	000020	OPCDEF	<ADC >	01,	005500,	DR1	
4	000030	UPCDEF	<ADCB >	01,	105500,	DR1	
5	000040	UPCDEF	<ADD >	02,	060000,	DR2	
6	000050	UPCDEF	<ADDD >	11,	172000,	DR2,	X45
7	000060	UPCDEF	<ADDF >	11,	172000,	DR2,	X45
8	000070	OPCDEF	<ASH >	09,	072000,	DR2,	X40&X45
9	000100	OPCDEF	<ASMC >	09,	073000,	DR2,	X40&X45
10	00110	UPCDEF	<ASL >	01,	006300,	DR1	
11	00120	UPCDEF	<ASLB >	01,	106300,	DR1	
12	00130	UPCDEF	<ASR >	01,	006200,	DR1	
13	00140	UPCDEF	<ASRB >	01,	106200,	DR1	
14	00150	OPCDEF	<BCC >	04,	103000,		
15	00160	OPCDEF	<BCS >	04,	103400,		
16	00170	UPCDEF	<BEQ >	04,	001400,		
17	00200	UPCDEF	<BGE >	04,	002000,		
18	00210	UPCDEF	<BGT >	04,	003000,		
19	00220	UPCDEF	<BHI >	04,	101000,		
20	00230	OPCDEF	<BHIS >	04,	103000,		
21	00240	UPCDEF	<BIC >	02,	040000,	DR2	
22	00250	UPCDEF	<BICB >	02,	140000,	DR2	
23	00260	UPCDEF	<BIS >	02,	050000,	DR2	
24	00270	UPCDEF	<BISB >	02,	150000,	DR2	
25	00300	UPCDEF	<BIT >	02,	030000,		
26	00310	UPCDEF	<BITB >	02,	130000,		
27	00320	OPCDEF	<BLE >	04,	003400,		
28	00330	UPCDEF	<BLO >	04,	103400,		
29	00340	UPCDEF	<BLOS >	04,	101400,		
30	00350	UPCDEF	<BLT >	04,	002400,		
31	00360	UPCDEF	<BMI >	04,	100400,		
32	00370	UPCDEF	<BNE >	04,	001000,		
33	00400	UPCDEF	<BPL >	04,	100000,		
34	00410	UPCDEF	<BPT >	00,	000003,	,	X45
35	00420	OPCDEF	 	04,	000400,		
36	00430	UPCDEF	<BVC >	04,	102000,		
37	00440	UPCDEF	<BVS >	04,	102400,		
38	00450	UPCDEF	<CCC >	00,	000257,		
39	00460	OPCDEF	<CFCC >	00,	170000,	,	X45
40	00470	UPCDEF	<CLC >	00,	000241,		
41	00500	UPCDEF	<CLN >	00,	000250,		
42	00510	UPCDEF	<CLR >	01,	005000,	DR1	
43	00520	OPCDEF	<CLRB >	01,	105000,	DR1	
44	00530	UPCDEF	<CLRD >	01,	170400,	DR1,	X45
45	00540	UPCDEF	<CLRF >	01,	170400,	DR1,	X45
46	00550	UPCDEF	<CLV >	00,	000242,		
47	00560	UPCDEF	<CLZ >	00,	000244,		

1	000570	OPCDEF	<CMP	>	02,	020000,		
2	000580	OPCDEF	<CMPB	>	02,	120000,		
3	000610	OPCDEF	<CMPD	>	11,	173400,		X45
4	000620	OPCDEF	<CMPF	>	11,	173400,		X45
5	000630	OPCDEF	<CNZ	>	00,	000254,		
6	000640	OPCDEF	<COM	>	01,	005100,	DR1	
7	000650	OPCDEF	<COMB	>	01,	105100,	DR1	
8	000660	OPCDEF	<DEC	>	01,	005300,	DR1	
9	000670	OPCDEF	<DECB	>	01,	105300,	DR1	
10	00700	OPCDEF	<DIV	>	07,	071000,	DR2,	X40&X45
11	00710	OPCDEF	<DIVD	>	11,	174400,	DR2,	X45
12	00720	OPCDEF	<DIVF	>	11,	174400,	DR2,	X45
13	00730	OPCDEF	<EMT	>	06,	104000,		
14	00740	OPCDEF	<FADD	>	03,	075000,	DR1,	X40
15	00750	OPCDEF	<FDIV	>	03,	075030,	DR1,	X40
16	00760	OPCDEF	<FMUL	>	03,	075020,	DR1,	X40
17	00770	OPCDEF	<FSUB	>	03,	075010,	DR1,	X40
18	01000	OPCDEF	<HALT	>	00,	000000,		
19	01010	OPCDEF	<INC	>	01,	005200,	DR1	
20	01020	OPCDEF	<INCB	>	01,	105200,	DR1	
21	01030	OPCDEF	<IOT	>	00,	000004,		
22	01040	OPCDEF	<JMP	>	01,	000100,		
23	01050	OPCDEF	<JSR	>	05,	004000,	DR1	
24	01060	OPCDEF	<LDCDF	>	11,	177400,	DR2,	X45
25	01070	OPCDEF	<LDCFD	>	11,	177400,	DR2,	X45
26	01100	OPCDEF	<LDCID	>	14,	177000,	DR2,	X45
27	01110	OPCDEF	<LDCIF	>	14,	177000,	DR2,	X45
28	01120	OPCDEF	<LDCLO	>	14,	177000,	DR2,	X45
29	01130	OPCDEF	<LDCLF	>	14,	177000,	DR2,	X45
30	01140	OPCDEF	<LDD	>	11,	172400,	DR2,	X45
31	01150	OPCDEF	<LDEXP	>	14,	176400,	DR2,	X45
32	01160	OPCDEF	<LOF	>	11,	172400,	DR2,	X45
33	01170	OPCDEF	<LDFPS	>	01,	170100,		X45
34	01200	OPCDEF	<LUSC	>	00,	170004,		X45
35	01210	OPCDEF	<LDUB	>	00,	170003,		X45
36	01220	OPCDEF	<MARK	>	10,	006400,		X45
37	01230	OPCDEF	<MFPU	>	01,	106500,		X45
38	01240	OPCDEF	<MFPI	>	01,	006500,		X45
39	01250	OPCDEF	<MOOD	>	11,	171400,	DR2,	X45
40	01260	OPCDEF	<MOOF	>	11,	171400,	DR2,	X45
41	01270	OPCDEF	<MOV	>	02,	010000,	DR2	
42	01300	OPCDEF	<MOVB	>	02,	110000,	DR2	
43	01310	OPCDEF	<MTPD	>	01,	106600,	DR1,	X45
44	01320	OPCDEF	<MTPI	>	01,	006600,	DR1,	X45
45	01330	OPCDEF	<MUL	>	07,	070000,	DR2,	X40&X45
46	01340	OPCDEF	<MULO	>	11,	171000,	DR2,	X45
47	01350	OPCDEF	<MULF	>	11,	171000,	DR2,	X45
48	01360	OPCDEF	<NEG	>	01,	005400,	DR1	
49	01370	OPCDEF	<NEGB	>	01,	105400,	DR1	
50	01400	OPCDEF	<NEGD	>	01,	170700,	DR1,	X45
51	01410	OPCDEF	<NEGF	>	01,	170700,	DR1,	X45
52	01420	OPCDEF	<NOP	>	00,	000240,		
53	01430	OPCDEF	<RESET	>	00,	000005,		

1	001440	OPCDEF	<ROL >	01,	006100,	DR1	
2	001450	OPCDEF	<ROLB >	01,	106100,	DR1	
3	001460	OPCDEF	<ROR >	01,	006000,	DR1	
4	001470	OPCDEF	<RORB >	01,	106000,	DR1	
5	001500	OPCDEF	<RTI >	00,	000002,		
6	001510	OPCDEF	<RTS >	03,	000200,	DR1	
7	001520	OPCDEF	<RTT >	00,	000006,		X45
8	001530	OPCDEF	<SBC >	01,	005600,	DR1	
9	001540	OPCDEF	<SBCB >	01,	105600,	DR1	
10	01550	OPCDEF	<SCC >	00,	000277,		
11	01560	OPCDEF	<SEC >	00,	000261,		
12	01570	OPCDEF	<SEN >	00,	000270,		
13	01600	OPCDEF	<SETD >	00,	170011,		X45
14	01610	OPCDEF	<SETF >	00,	170001,		X45
15	01620	OPCDEF	<SETI >	00,	170002,		X45
16	01630	OPCDEF	<SETL >	00,	170012,		X45
17	01640	OPCDEF	<SEV >	00,	000262,		
18	01650	OPCDEF	<SEZ >	00,	000264,		
19	01660	OPCDEF	<SOB >	00,	077000,	DR1,	X45
20	01670	OPCDEF	<SPL >	13,	000230,		X45
21	01700	OPCDEF	<STA0 >	00,	170005,		X45
22	01710	OPCDEF	<STB0 >	00,	170006,		X45
23	01720	OPCDEF	<STCDF >	12,	176000,	DR2,	X45
24	01730	OPCDEF	<STCDI >	12,	175400,	DR2,	X45
25	01740	OPCDEF	<STCDL >	12,	175400,	DR2,	X45
26	01750	OPCDEF	<STCFD >	12,	176000,	DR2,	X45
27	01760	OPCDEF	<STCFI >	12,	175400,	DR2,	X45
28	01770	OPCDEF	<STCFL >	12,	175400,	DR2,	X45
29	02000	OPCDEF	<STD >	12,	174000,	DR2,	X45
30	02010	OPCDEF	<STEXP >	12,	175000,	DR2,	X45
31	02020	OPCDEF	<STF >	12,	174000,	DR2,	X45
32	02030	OPCDEF	<STFPS >	01,	170200,	DR1,	X45
33	02040	OPCDEF	<STQ0 >	00,	170007,		X45
34	02050	OPCDEF	<STST >	01,	170300,	DR1,	X45
35	02060	OPCDEF	<SUB >	02,	160000,	DR2	
36	02070	OPCDEF	<SUB0 >	11,	173000,	DR2,	X45
37	02100	OPCDEF	<SUBF >	11,	173000,	DR2,	X45
38	02110	OPCDEF	<SWAB >	01,	000300,	DR1	
39	02120	OPCDEF	<SXT >	01,	006700,	DR1,	X45
40	02130	OPCDEF	<TRAP >	06,	104400,		
41	02140	OPCDEF	<TST >	01,	005700,		
42	02150	OPCDEF	<TSTB >	01,	105700,		
43	02160	OPCDEF	<TSTU >	01,	170500,		X45
44	02170	OPCDEF	<TSTF >	01,	170500,		X45
45	02200	OPCDEF	<WAIT >	00,	000001,		
46	02210	OPCDEF	<XOR >	05,	074000,	DR2,	X45

1	002220	DIRDEF	<ASCII>	DFLGBM	
2	002230	DIRDEF	<ASCIZ>	DFLGBM	
3	002240	DIRDEF	<ASECT>	,	XREL
4	002250	DIRDEF	<BLKB >		
5	002260	DIRDEF	<BLKW >	DFLGEV	
6	002270	DIRDEF	<BYTE >	DFLGBM	
7	002300	DIRDEF	<CSECT>	,	XREL
8		.IF OF	YPHASE		
9		DIRDEF	<DEPHA>		
10		.ENDC			
11	02310	DIRDEF	<OSABL>		
12	02320	DIRDEF	<ENABL>		
13	02330	DIRDEF	<END >		
14	02340	DIRDEF	<ENOC >	DFLCND	
15	02350	DIRDEF	<ENDM >	DFLMAC, XMACRO	
16	02360	DIRDEF	<ENDR >	DFLMAC, XMACRO	
17	02370	DIRDEF	<EOT >		
18	02400	DIRDEF	<ERROR>		
19	02410	DIRDEF	<EVEN >		
20	02420	DIRDEF	<FLT2 >	DFLGEV, XFLTG	
21	02430	DIRDEF	<FLT4 >	DFLGEV, XFLTG	
22	02440	DIRDEF	<GLOBL>	,	XREL
23	02450	DIRDEF	<IDENT>		
24	02460	DIRDEF	<IF >	DFLCND	
25	02470	DIRDEF	<IFDF >	DFLCND	
26	02500	DIRDEF	<IFEQ >	DFLCND	
27	02510	DIRDEF	<IFF >	DFLCND	
28	02520	DIRDEF	<IFG >	DFLCND	
29	02530	DIRDEF	<IFGE >	DFLCND	
30	02540	DIRDEF	<IFGI >	DFLCND	
31	02550	DIRDEF	<IFL >	DFLCND	
32	02560	DIRDEF	<IFLE >	DFLCND	
33	02570	DIRDEF	<IFLT >	DFLCND	
34	02600	DIRDEF	<IFNUF>	DFLCND	
35	02610	DIRDEF	<IFNE >	DFLCND	
36	02620	DIRDEF	<IFNZ >	DFLCND	
37	02630	DIRDEF	<IFT >	DFLCND	
38	02640	DIRDEF	<IFTF >	DFLCND	
39	02650	DIRDEF	<IFZ >	DFLCND	
40	02660	DIRDEF	<IIF >		
41	02670	DIRDEF	<IRP >	DFLMAC, XMACRO	
42	02700	DIRDEF	<IRPC >	DFLMAC, XMACRO	
43	02710	DIRDEF	<LIMIT>	DFLGEV, XREL	
44	02720	DIRDEF	<LIST >		



```

1 002730      DIRDEF <MACR >,      DFLMAC, XMACRO
2 002740      DIRDEF <MACRO>,     DFLMAC, XMACRO
3 002750      DIRDEF <MCALL>,     DFLSMC, XSML
4 002760      DIRDEF <MEXIT> ,    ,      XMACRO
5 002770      DIRDEF <NARG > ,    ,      XMACRO
6 003000      DIRDEF <NCHR > ,    ,      XMACRO
7 003010      DIRDEF <NLIST>      ,      XMACRO
8 003020      DIRDEF <NTYPE> ,    ,      XMACRO
9 003030      DIRDEF <OOD >      ,      XMACRO
10 003040     DIRDEF <PAGE >
11           .IF OF YPHASE
12           DIRDEF <PHASE>
13           .ENDC
14 003050     DIRDEF <PRINT>
15           .IF OF RSX11D
16           DIRDEF <PSECT>
17           .ENDC
18 003060     DIRDEF <RADIX>
19 003070     DIRDEF <RAD50>,     DFLGEV
20 003100     DIRDEF <REM >
21 003110     DIRDEF <REPT >,    DFLMAC, XMACRO
22 003120     DIRDEF <SBTTL>
23 003130     DIRDEF <TITLE>
24 003140     WRDSYM:
25 003140     DIRDEF <WORD >,    DFLGEV
26
27
28 003150     PSTTOP:           JTOP LIMIT
29
30 000001'    .END
  
```

PST V01  
SYMBOL TABLE

PERMANENT SYMBO RT-11 MACRO VM01-01 31-AUG-73 PAGE 6+

ASCII	=	*****	G	ASCIZ	=	*****	G	ASECT	=	*****	G
BLKB	=	*****	G	BLKW	=	*****	G	BYTE	=	*****	G
CSECT	=	*****	G	DFLCND	=	000004	G	DFLG8M	=	000010	G
DFLGEV	=	000020	G	DFLMAC	=	000002	G	DFLSMC	=	000001	G
DR1	=	000200		DR2	=	000100		DSABL	=	*****	G
ENABL	=	*****	G	END	=	*****	G	ENDC	=	*****	G
ENDM	=	*****	G	ENDR	=	*****	G	EOT	=	*****	G
ERROR	=	*****	G	EVEN	=	*****	G	FLT2	=	*****	G
FLT4	=	*****	G	GLOBL	=	*****	G	IDENT	=	*****	G
IF	=	*****	G	IFDF	=	*****	G	IFEQ	=	*****	G
IFF	=	*****	G	IFG	=	*****	G	IFGE	=	*****	G
IFGT	=	*****	G	IFL	=	*****	G	IFLE	=	*****	G
IFLT	=	*****	G	IFNDF	=	*****	G	IFNE	=	*****	G
IFNZ	=	*****	G	IFT	=	*****	G	IFTF	=	*****	G
IFZ	=	*****	G	IIF	=	*****	G	IRP	=	*****	G
IRPC	=	*****	G	LIMIT	=	*****	G	LIST	=	*****	G
MACR	=	*****	G	MACRO	=	*****	G	MCALL	=	*****	G
MEXIT	=	*****	G	NARG	=	*****	G	NCHR	=	*****	G
NLIST	=	*****	G	NTYPE	=	*****	G	ODD	=	*****	G
OPCL00	=	*****	G	OPCL01	=	*****	G	OPCL02	=	*****	G
OPCL03	=	*****	G	OPCL04	=	*****	G	OPCL05	=	*****	G
OPCL06	=	*****	G	OPCL07	=	*****	G	OPCL08	=	*****	G
OPCL09	=	*****	G	OPCL10	=	*****	G	OPCL11	=	*****	G
OPCL12	=	*****	G	OPCL13	=	*****	G	OPCL14	=	*****	G
PAGE	=	*****	G	PRINT	=	*****	G	PSTBAS	=	000000RG	
PSTTUP	=	003150RG		RADIX	=	*****	G	RAD50	=	*****	G
REM	=	*****	G	REPT	=	*****	G	RT11	=	000000	
S8TTL	=	*****	G	TITLE	=	*****	G	WORD	=	*****	G
WROSYM	=	003140RG		XBAW	=	000000		XCREP	=	000000	
XEDABS	=	000000		XEDCOR	=	000000		XEDPIC	=	000000	
XSWIT	=	000000									
. ABS.	=	000000	000								
		003150	001								
ERRORS DETECTED:	=	0									
FREE CORE:	=	6756.	WORDS								

## APPENDIX N

### PERIPHERAL INTERCHANGE PROGRAM--CASSETTE (PIPC)

#### N.1 INTRODUCTION

PIPC is an RT-11 program that is used to transfer files between standard cassettes and other RT-11 system devices, delete cassette files, and transfer cassette directories. PIPC allows the RT-11 user to read or write any standard cassette file. Cassette to cassette transfers are not allowed. In particular, PIPC can read or write any file created by or to be used by the RT-11 system (using any RT-11 device handler). PIPC is designed to allow cassettes to serve as a backup device.

PIPC may be run on any RT-11 system equipped with at least 8K of memory and cassette drives. PIPC supports any RT-11 system device.

#### N.2 CASSETTE DESCRIPTION

A cassette is a magnetic tape device much like that used in a cassette tape recorder. The tape itself and the reels it is wound on are enclosed inside a rectangular plastic case (see Figure N-1), making handling, storage, and care of the cassette convenient for the user.

On either end of one side of the cassette are two flexible plastic tabs called write-protect tabs (see A in Figure N-1). There is one tab for each end of the tape; since data should only be written in one direction, the user needs to be concerned only with the tab specifically marked on the cassette label. Depending upon the position of this tab the tape is protected against accidental writing and destruction of data. When the tab is pulled in toward the middle of the cassette so that the hole is uncovered, the tape is write-locked; data cannot be written on it and any attempt to do so results in an error message. When the tab is pushed toward the outside of the cassette so that the hole is covered, the tape is write-enabled and data can be written onto it. Data can be read from the cassette with the tab in either position.

The bottom of the cassette (B in Figure N-1) provides an opening where the magnetic tape is exposed. The cassette is locked into position on a TU60 cassette unit drive so that the tape comes in contact with the read/write head through this opening.

Both ends of the magnetic tape in a cassette consist of clear plastic leader/trailer tape; this section of the tape is not used for information storage purposes, but as a safeguard in handling and storing the cassette itself. Since magnetic tape is susceptible to dust and fingerprints, a cassette should always be rewound so that the leader/trailer tape is the only part of the tape exposed whenever the cassette is not on a drive.

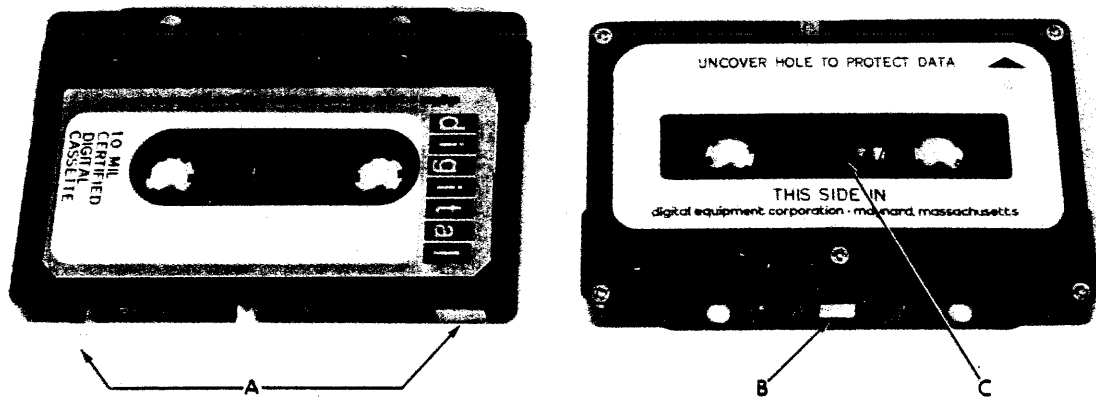


Figure N-1 RT-11 Cassette

#### N.2.1 Cassette Format

A cassette is formatted so that it consists of a sequence of one or more files. Files on cassette are sequential, and each file is preceded and followed by a file gap. (A gap in this sense is a fixed length of blank tape.) All cassettes must start with a file gap; information preceding the initial file gap is unreliable.

A file consists of a sequence of one or more data records separated from one another by a record gap. The records of any given file must follow one another in succession, as there is no provision for record linking. The first record of a file is called the header record and contains information concerning the name of the file, its type, length, and so on. (The Cassette Standard may be referenced in Appendix Q.) A data record contains 128 (decimal) bytes of information; there are approximately 600 records per cassette tape. RT-11 recognizes an end-of-file by the presence of either a file gap or clear leader following a data record.

Data records consist of 128 (decimal) cassette bytes; a byte in turn consists of eight bits each representing a binary zero or one. Characters and numbers are stored in bytes using the standard ASCII character codes and binary notation.

The number of records of information on a cassette tape may be estimated. On the outside of the cassette case is a clear plastic window (C in Figure N-1). Along the bottom of this window is a series of marks; each mark represents about 50 inches of magnetic tape. Knowing that approximately 2 records fit on an inch of tape, a reasonable guess can be made as to the length of tape and number of records available for use. By simply glancing at the width of the tape reel showing in the window, the user can tell quickly if he is

very close to the end. Since no advance warning of a full tape condition is given, the user must visually keep track of the length of tape available. Should the tape become full before a file transfer has completed, another cassette may be substituted and the transfer may be restarted or continued depending on the mode of transfer. (see Table N-1).

### N.2.2 The Sentinel File

The last file on a cassette tape is called the sentinel file. This file consists of only a 32 (decimal) byte header record and represents the logical end-of-tape (PIPC also recognizes clear leader as logical end-of-tape). A sentinel file is identified by a null character (ASCII=000) as the first name character in the header record. A zeroed or blank cassette tape is one consisting of only the sentinel file.

### N.3 MOUNTING AND DISMOUNTING A CASSETTE

To mount a tape on a drive, hold the tape so that the open part of the cassette is to the left and the full reel is at the top. Set the top write-protect tab to the desired position depending upon whether data is to be written on the tape.

Open the locking bar on the cassette drive by pushing it to the right away from the drive (see A in Figure N-2). Next hold the tape up to the cassette drive at approximately a 45-degree angle and insert the tape into the drive by applying a leftward pressure while simultaneously pushing the cassette onto the drive sprockets. This brings the tape into position against the read/write head. When the cassette is properly mounted, the locking bar automatically closes over the cassette back edge. Figure N-2 illustrates this procedure.

Press the rewind button on the cassette unit (see B in Figure N-2; there is a rewind button for each drive). This causes the cassette to rewind to the beginning of its leader/trailer tape. (Pressing the rewind button a second time causes the cassette to rewind to the end of the leader/trailer tape and to the physical end-of-tape. The cassette unit will click; this sound is almost inaudible and the user may not hear it unless he is listening carefully. Normal usage requires that the rewind button be pressed only once whenever a cassette is to be rewound). Even though tapes which are not actively being used on a drive should already be positioned at the beginning, it is a good habit to automatically rewind a cassette.

When the tape has finished winding, the cassette will stop moving. The cassette is now in place and ready for transfer operations.

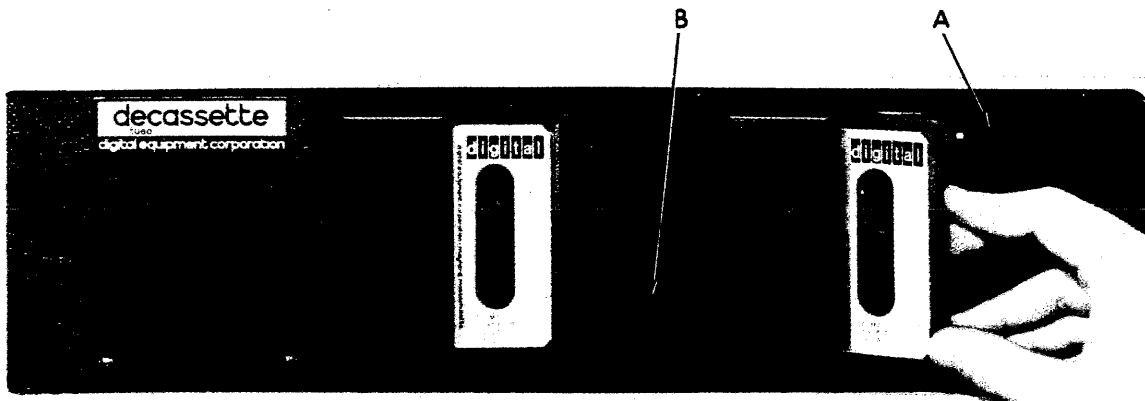


Figure N-2

#### Mounting a Cassette

Since PIPC rewinds the cassette after every operation it is not necessary to press the rewind button before removing the cassette from a drive. To remove a cassette from the cassette drive, open the locking bar and the cassette will pop out. When cassettes are not being actively used on a cassette drive, they can be stored in the small plastic boxes provided for this purpose by the manufacturer.

#### NOTE

Before using a new cassette, or prior to using a cassette that has just been shipped or accidentally dropped, mount the cassette on a drive so that the Digital label faces the inside of the unit and perform a rewind operation. Remove the cassette, turn it over, and perform another rewind operation. This packs the tape neatly in the cassette and places the full tape reel at the proper tension.

#### N.4 CALLING AND USING PIPC

To call PIPC from the RT-11 system device, the user types:

```
.R PIPC
```

in response to the dot printed by the Keyboard Monitor. The Command String Interpreter then prints an asterisk at the left margin of the terminal and waits to receive a line of I/O files and command switches. PIPC accepts up to six input files and three output files. The contents of the input file are transferred to the output file in image mode. In response to the asterisk, type an I/O specification in the standard RT-11 format.

PIPC supports only one cassette unit with the permanent device name CT0 and CT1. Permanent device names for other RT-11 devices are listed in the PIP chapter. These device names are used in the I/O specification, along with any file name that is necessary. For example, to transfer a file named DATA01 to the user's disk, the user types:

```
*DK:DATA01<CT1:DATA01/I
```

PIPC allows the use of a wild card character (\*) to represent a file name or extension. The asterisk may be used to replace a file name and/or extension as follows:

<u>Specification to be replaced</u>	<u>Form</u>
file name	*.ext
extension	filnam.*
file name and extension	*.*

Use of the asterisk causes PIPC to consider any name in that field as a match.

System files (.SYS extension) are ignored when the wild card character is used unless the /Y switch is specified.

Examples:

```
*CT0:*.SAV/D  Deletes all files with the extension .SAV from
                cassette unit 0.

*CT1:F1.* /D   Deletes all files (except .SYS or .BAD files)
                named F1 from cassette unit 1 regardless of
                extension.

*CT0:*.* *.*   Transfers all files (except .SYS or .BAD files)
                from disk to cassette unit 0 regardless of file
                name or extension.
```

Since PIPC performs file transfers for all file types, there are no assumed extensions assigned by PIPC to file names for either input or output files. All extensions, where present, must be explicitly specified, except when the \* option is used.

When a file is added to a cassette, PIPC checks for the existence of a file of the same name. If the file of the same name is found, it is renamed \*EMPTY. Then PIPC searches for the sentinel file, writes the new file over it and creates a new sentinel file. The space where the \*EMPTY file resides is not reclaimed until the cassette is zeroed.

If a transfer is attempted to a full cassette, the message

```
MNT CAS
```

is printed. To continue the transfer, mount a new cassette and type the RETURN key. Type N and the RETURN key to halt the transfer. Any portion of the file already on the full cassette will be deleted.

Following completion of a PIPC operation, the Command String Interpreter again prints an asterisk at the left margin and waits for another PIPC I/O specification line. Type CTRL/C to return to the Keyboard Monitor.

#### N.5 PIPC COMMANDS

The various switches allowed on a PIPC I/O specification line are detailed in Table N-1. If no command switch is specified, PIPC assumes the operation is a file transfer in image mode; the files are not concatenated.

Table N-1  
PIPC Options

Option	Meaning
/I or none	Transfer files in image mode. For example <pre>*CT0:FA,FB,FC&lt;FD,FE,FF/I</pre> transfers files FD, FE and FF from device DK to cassette unit 0 as files FA, FB, FC.
/A	Transfer files in ASCII mode (nulls are ignored).
/U	Transfer the system bootstrap to the specified device. For example: <pre>*DK:BFILE&lt;CT1:MONITR.SYS/U</pre> transfers the bootstrap in the file, MONITR.SYS from cassette unit 1 to block 0 and 2 of the device DK. The file name BFILE is a dummy specification.
/D	Delete the file specified from the output cassette. The /D option is only valid if the output device is a cassette. For example: <pre>*CT1:OFILE/D</pre> will delete OFILE from the cassette on drive 1.



Table N-1 (Cont.)

## PIPC Options

Option	Meaning
/F	<p>Print the short directory of specified cassette (file names only). For example,</p> <p style="text-align: center;">*LP:&lt;CT1/F</p> <p>prints the directory (file names only) of cassette unit 1 on the line printer.</p>
/G	<p>Copy a file(s) and ignore input errors. When copying from a cassette to another device, input errors in data and the header are ignored.</p>
/L	<p>Read the input cassette directory and write it on the output device. The directory includes file names and dates for each entry. Notice that in this case the input file itself is not transferred, only the directory. The /L option applies only if the input device is a cassette.</p>
/M	<p>Read or write multi-volume cassette file(s).</p>
/N	<p>Used with /Z to specify the number of directory blocks to allocate on the disk or DECTape.</p>
/O	<p>Boot the RT-11 system from the specified disk or DECTape unit 0. For example,</p> <p style="text-align: center;">*RK:/0</p>
/Q	<p>Read after Write; writes a block then reads that block and checks for write errors.</p>
/Y	<p>Allows system files to be operated on by other command switches. Attempted transfers or deletions on files with the SYS extension without specifying the /Y are null operations.</p>
/E	<p>List the entire directory including unused spaces called *EMPTY. For example,</p> <p style="text-align: center;">*CT0:/E</p> <p>lists the directory of the cassette mounted on unit 0 including *EMPTY entries for unused areas.</p>
/V	<p>Display the version number of PIPC. For example,</p> <p style="text-align: center;">CT0:/V</p> <p>causes a message similar to</p> <p style="text-align: center;">PIPC V01-06</p>

## PIPC Options

Option	Meaning
/B	Transfers files in formatted binary mode.
/Z	<p>Clear the directory of the specified device. The message</p> <p style="text-align: center;">ARE YOU SURE?</p> <p>is displayed. Enter Y and a carriage return to zero the directory; any other response causes PIPC to ignore the command. /Z allows specification of directory size when used to zero a disk or DEctape. For example,</p> <p style="text-align: center;">DT1:/Z:2</p> <p>provides two extra words per directory entry. A value given to the /Z when used to zero a cassette has no effect.</p>

The /M switch allows creation or transfer of a multi-volume cassette file. For example, the command

```
*CT0:*. * <*. */M
```

starts transferring files from device DK onto cassette unit 0. When the end of tape (EOT) is encountered, the message

```
filenam MNT CAS
```

is displayed. Mount a new cassette and type the Y and carriage return keys to continue the transfer.

When /M is used to create a file, the cassette is zeroed by writing over any previously existing files; therefore if a file is continued onto another cassette, the continued file is the first file on the new cassette. As an example of transferring a multi-volume file from cassette to another device,

```
**.* <CT0:*. */M
```

starts transferring all files from cassette unit 0 to device DK. When the EOT is encountered, the message

```
MNT CAS
```

is displayed. Mount the cassette which contains the continued file and type the carriage return key. PIPC checks the file header of the second tape and if it does not match the continued file or the sequence number is not correct, the message

```
ARE YOU SURE?
```

is displayed. If Y (for yes) and a carriage return is entered, the continued file is closed and transfer continues with this file from the new cassette. If N or any other character is entered, the continued file is deleted from the output device and PIPC outputs an \* and awaits a new command.

A transfer without the specification of /M will not split files across cassettes. For example, in a transfer such as

```
CT0:*. *<*. *
```

the message

```
filnam MNT CAS
```

is printed when the EOT is encountered (where filnam is the file which would not fit on the cassette). PIPC backs up the cassette tape and writes a sentinel file after the last complete file on the cassette. To continue the transfer, mount a new cassette and type the /Y and RETURN keys. Otherwise type N and the RETURN key and PIPC will print \*.

If the file being transferred to cassette without the /M switch is too large for a single cassette, the message

```
filename ?OVRFL?
```

is printed. The partial file is deleted from the cassette and the transfer continues with the next file, if any.

The size of a file to be opened on a device can be specified as part of the command line by enclosing the number of blocks to be assigned in square brackets ([]) after the specification of the file to be opened. For example,

```
F1[20] <CT0:F2
```

opens a 20 block file called F1 on device DK and transfers file F2 from cassette unit 0.

## N.6 PIPC ERROR MESSAGES

Error messages which appear while PIPC is running are shown in Table N-2. If an output file is specified on a cassette and a file by that name already exists, the file on the output cassette is deleted before any transfer is performed. If PIPC detects an error while a cassette output file is open, it tries to close the output file by writing a sentinel file on the output cassette.

Table N-2

## PIPC Error Messages

Message	Meaning
?ILL DEV?	Error in the fetch operation; input and output both request cassette operations; or illegal cassette unit number.
?OFF LINE?	No cassette in unit specified or the cassette is improperly mounted.
?WRT LOCK?	Cassette unit specified is write locked.
?TIM ERROR?	Timing error - probably caused by a cassette hardware failure.
?IN ERR?	Input or read error.
?DEV FULL?	The specified device (not cassette) is full.
?OUT FIL?	Attempt to write file name with a * as name or extension.
?BOOT COPY?	The bootstrap file was not found on a /U copy operation.
?COR OVR?	Not enough core for buffers.
?FIL NOT FND?	File not found on specified cassette or device.
?ER RD DIR?	Directory read in error from specified device.
?OVRFL?	File will not fit on the cassette; the file is not transferred.
The following warning messages are output by PIPC.	
?REBOOT?	A file with extension .SYS, has just been transferred to the specified device and this message is a reminder to reboot the system.
?NO SYS ACTION?	The /Y switch was not included with a command specified on a .SYS file. The .SYS file operations are not performed but the remainder of the command is executed.

## APPENDIX O

### PATCH PROGRAM

The PATCH utility program may be used to make code modifications to absolute SAV files, including overlay-structured and monitor files. PATCH like ODT can be used to interrogate, and then change words or bytes in the file. PATCH provides eight relocation registers. Before changing a program with PATCH, copy the old file to a backup file with PIP, as the old file is destroyed.

To run PATCH type the monitor command:

```
.R PATCH
```

followed by the RETURN key. PATCH prints a version number message of the form:

```
PATCH V 01-00
```

and then prints the message

```
FILE NAME --  
*
```

There may be a noticeable delay on DECTape systems before the \* is printed.

Type in the name of the file which is to be modified in the format:

```
[DEV:]FILNAM[.EXT] [/O] [/M]
```

the elements enclosed in [] are optional. If not specified, DK: and .SAV are the default device and extension, respectively. The /O switch indicates that the file is an overlay-structured file. This flag must be specified to correctly patch an overlay file. The /M switch indicates that the file is an RT-11 monitor file; it must be specified to correctly patch a monitor file. At this point, PATCH prints an asterisk (\*) prompt character, indicating that it is waiting for a command.

The FILE NAME message may also appear after the printing of an error message.

Table O-1 summarizes the PATCH commands. Note that the /O and /M switches must be specified when the file name is typed.

Table O-1

## Summary of PATCH Commands

Command	Action
/O	Indicates overlay-structured file.
/M	Indicates Monitor file.
Vr;rR	Set relocation register r to value Vr.
b;B	Set bottom address of overlay file to b.
[s:]r.o/	Open word location Vr + o in segment s.
[s:]r,o	Open byte location Vr + o in segment s.
<carriage return>	Close currently-opened word/byte.
<line feed>	Close currently-open word/byte, and open the next one.
↑	Close currently-open word/byte, and open the previous one.
@	Close the currently-open word and open the word addressed by it.
F	Begin patching a new file.
E	Exit to RT-11 Monitor.

## 0.1. PATCH a new file

The command F causes PATCH to close the file being patched, and request that a new file name be typed in.

## 0.2. Exit from PATCH

The command E causes PATCH to close the file being patched, and return to the RT-11 monitor.

## 0.3. Examine, Change locations in the file

For a non-overlay file, a word address may be opened, as with ODT, by typing

```
[<relocation register>,]offset/
```

At this point, PATCH will type out the contents of the location and wait for the user to type in a new location contents in octal or another command.

For example,

In an overlay file, the format is

```
[<segment number>:][<relocation register>,<offset>/
```

Where <segment number> is the overlay segment number as it is printed on the link map for the file. If it is omitted, the root segment is assumed.

Similarly, to open a byte address in file, the format is

```
[<relocation register>,<offset>\
```

for non overlay files, or

```
[<segment number>:][<relocation register>,<offset>\
```

for overlay files.

Once a location has been opened, the user may optionally type in the new contents, in the format:

```
[<relocation register>,<value>
```

followed by one of these control characters:

<carriage return>	close the current location by changing its contents if a new contents was specified, and await more control input.
<line feed>	close the current location, and open the next word/byte
↑	close the current location, and open the previous word/byte
@	close the current word location, and open the word addressed by it (in the same segment if an overlay file).

#### 0.4 Set Bottom Address

To patch an overlay file, PATCH must know the bottom address at which the program was linked, if it is different from the initial stack pointer. This will be true if the program sets location 42 in an .ASECT. To set the bottom address, type:

```
<bottom address>;B
```

#### 0.5 Set Relocation Registers

The relocation registers 0-7 are set, as with ODT, by the R command. It has the format:

```
<relocation value>;<relocation register>R
```

Once one of the eight relocation registers has been set, the expression

<relocation register>,<octal number>  
typed as part of a command will have the value  
<relocation value> + <octal number>

## 0.6 PATCH ERROR MESSAGES

?ADDR NOT IN SEG?	The address is not in the specified segment.
?BAD SWITCH?	Typed a switch other than /O or /M.
?BOTTOM ADDR WRONG?	The bottom address specified or contained in location 42 of an overlay file is incorrect. Specify the correct one using the b;B command.
?INCORRECT FILE SPEC?	The response to the "FILE NAME --" message was not of the correct form. Try again.
?INSUFFICIENT CORE?	PATCH did not have enough core to hold the file's device handler plus the internal "segment table." This message should not occur.
?INVALID RELOC REG?	Tried to reference a relocation register outside of the range 0-7.
?INVALID SEG NO?	The segment number S: does not exist.
?MUST OPEN WORD?	The @ command was typed when a byte location was open.
?MUST SPECIFY SEG?	The address referenced is not in the root section; a segment number S: must be used.
?NO ADDR OPEN?	The <line feed>, ↑ or @ command was typed when no location was open.
?NOT IN PROGR BOUNDS?	Tried to open a location beyond the end of the file.
?ODD ADDRESS?	Tried to open a word address which was odd. (Use "\".)
?ODD BOTTOM ADDR?	The bottom address specified or contained in location 42 of an overlay file is odd.
?PROGR HAS NOT SEGS?	The file specified as an overlay file is not.
?READ ERROR?	File I/O error in reading.
?WRITE ERROR?	File I/O error in writing.



## APPENDIX P

### FUNDAMENTALS OF PROGRAMMING THE PDP-11

This appendix presents some fundamental software concepts essential to efficient assembly language programming of the PDP-11 computer. A description of the hardware components of the PDP-11 family can be found in the two DEC paperback handbooks:

PDP-11 Processor Handbook (11/20 or 11/45 edition)  
PDP-11 Peripherals and Interfacing Handbook

No attempt is made in this document to describe the PDP-11 hardware or the function of the various PDP-11 instructions. The reader is advised to become familiar with this material before proceeding.

#### P.1 MODULAR PROGRAMMING

The PDP-11 family of computers lend themselves most easily to a modular system of programming. In such a system the programmer must envision the entire program and break it down into constituent subroutines. This will provide for the best use of the PDP-11 hardware (as will become clearer later in this chapter), as well as resulting in programs which are more easily modified than those coded with straight-line coding techniques.

To this end, flowcharting of the entire system is best performed prior to coding rather than during or after the coding effort. The programmer is then able to attack small bits of the program at any one time. Subroutines of approximately one or two pages are considered desirable.

Modular programming practices maximize the usefulness of an installation's resources. Programmed modules can be used in other programs or systems where similar or identical functions are required without the overhead of redundant development. Software modules developed as functional entities are more likely to be free of serious logical errors as a result of the original programming effort. Confidence in such modules allows for easy creation of later systems incorporating proven pieces.

Modular development provides for ease of use and modification rather than simplifying the original development. Some pains must be taken to ensure correct modular system development, but the benefits of standardization to the generations of maintenance programmers which deal with a given system are many. (See also the notes under Commenting Assembly Language Programs.)

Modular development forces an awareness of the final system. Ideally, this should cause all components of the system to be considered from the very beginning of the development effort rather than patched into a partially-developed system.

It is assumed that the human mind can best work with limited pieces of information at any one time, combining the results of the individual functions to encompass the entire program in steps. PDP-11 assembly

language programming best follows a tree-like structure with the top of the tree being the final results and the base being the smallest component functions. (The Assembler itself is a tree structure and is briefly described in Figure P-1.)

### P.1.1 Commenting PDP-11 Assembly Language Programs

When programming in a modular fashion, it is desirable to heavily comment the beginning of each subroutine, telling what that routine does; its inputs, outputs, and register usage.

Since subroutines are short and encompass only one operation it is not necessary to tell how the subroutine functions, but only

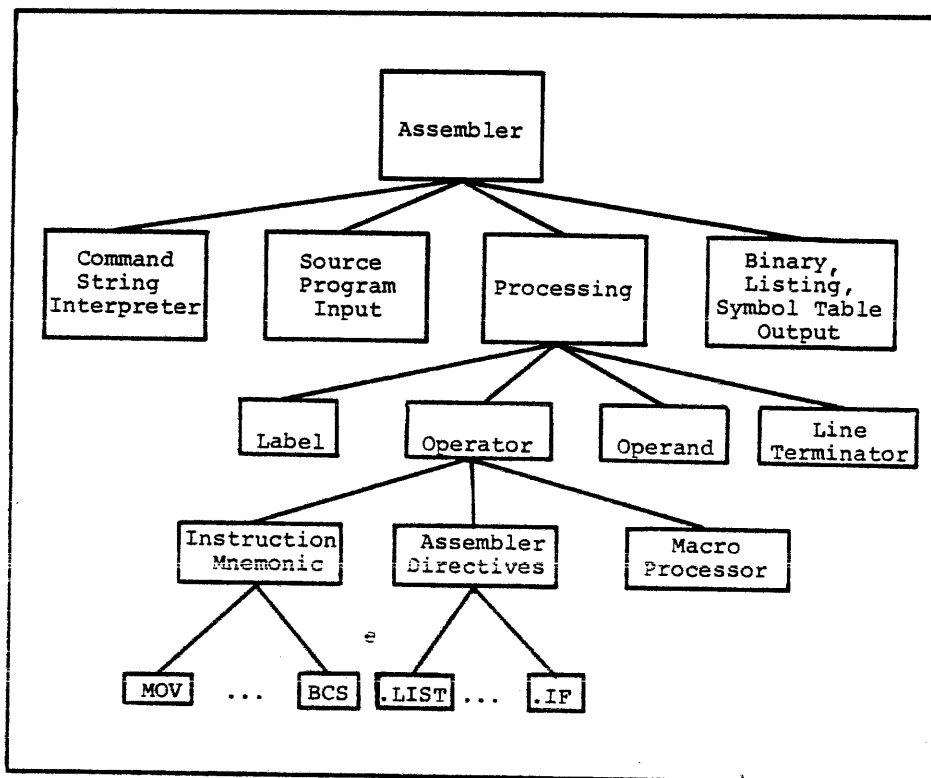


Figure P-1 Problem Oriented Tree-Structure

what it does. The subroutine should be fully documented only when the procedure is not obvious to the reader. This enables any later inspection of the subroutine to disclose the maximum amount of useful information to the reader.

### P.1.2 Localized Register Usage

A useful technique in writing subroutines is to save all registers upon entering a subroutine and restore them prior to leaving the subroutine. This allows the programmer unrestricted use of the PDP-11 registers, including the program stack, during a subroutine.

Use of registers avoids two- and three-word addressing instructions. The code in Figure 8-2 compares the use of registers with symbolic addressing. Register use is faster and requires less storage space than symbolic addressing.

### P.1.3 Conditional Assemblies

Conditional assemblies are valuable in macro definitions. The expansion of a macro can be altered during assembly as a result of specific arguments passed and their use in conditionals. For example, a macro can be written to handle a given data item differently, depending upon the value of the item. Only a single algorithm need be expanded with each macro call. (Conditionals are described in detail in section 5.5.11.)

Conditional assemblies can also be used to generate versions of a program from a single source. This is usually done as a result of one or more symbols being either defined or undefined. Conditional assemblies are preferred to the creation of a multiplicity of sources. This principle is followed in the creation of PDP-11 system programs for the following reasons:

1. Maintenance of a single source program is easier, and guarantees that a change in one version of the program, which may affect other versions, is reflected automatically in all possible versions.
2. Distribution of a single source program allows a customer or individual user to tailor a system to his configuration and needs, and continue to update the system as the hardware environment or programming requirements change.
3. As in the case of maintenance, the debugging and checkout phase of a single program (even one containing many separate modules) is easier than the testing of several distinct versions of the same basic program.

```

1
2 002060      10$:  .TFT          20$
3 002064 003375  CALL          20$          ;MOVE A CHARACTER
4 002066 001432  BGT           10$          ;LOOP IF GT ZERO
5 002070 114200  BFQ           19$          ;END IF ZERO
6 002072 020027  MOVR          -(R2),R0     ;TERMINATOR,BACK UP POINTER
   177603      CMP           R0,#MT,MAX  ;END OF TYPE:
7 002076 101453  BLOS          22$          ;YES
8 002100 010146  MOV           R1,-(SP)    ;REMEMBER READ POINTER
9 002102 016701  MOV           MSBARG,R1
   002034'
10 02106 005721  TST           (R1)+
11 02110 010203  MOV           R2,R3       ; AND WRITE POINTER
12 02112 005400  NEG           R0          ;ASSUME MACRO
13 02114 026727  CMP           MSBTYP,#MT,MAC ;TRUE?
   002026'
   177603
14 02122 001402  BFQ           12$          ; YES, USE IT
15 02124 016700  MOV           MSBCNT,R0   ;GET ARG NUMBER
   002036'
16 02130 010302  12$:  MOV           R3,R2   ;RESET WRITE POINTER
17 02132      13$:  CALL          20$          ;MOVE A BYTE
18 02136 003375  BGT           13$          ;LOOP IF PZN
19 02140 002402  BLT           14$          ;END IF LESS THAN ZERO
20 02142 005300  DEC           R0          ;ARE WE THERE YET?
21 02144 003371  BGT           12$          ; NO
22 02146 105742  14$:  TSTR          -(R2)    ;YES, BACK UP POINTER
23 02150 012601  MOV           (SP)+,R1    ;RESET READ POINTER
24 02152 000742  BR            10$          ;END OF ARGUMENT SUBSTITUTION
25
26 02154 010167  19$:  MOV           R1,MSRMRP ;END OF LINE, SAVE POINTER
   002042'
27 02160 052767  BTS           #LC,ME,LCFLAG ;FLAG AS MACRO EXPANSION
   000400
   000010'
28 02166 000726  BR            9$
29
30 02170 032701  20$:  BIT           #RPMB-1,R1 ;MACRO, END OF BLOCK?
   000017
31 02174 001003  BNE           21$          ; NO
32 02176 016101  MOV           -RPMB(R1),R1 ;YES, POINT TO NEXT BLOCK
   177760
33 02202 005721  TST           (R1)+       ;MOVE PAST LINK
34 02204 020227  21$:  CMP           R2,#LTNBUF+SRCLFN ;OVERFLOW?
   001744'
35 02210 101404  BLOS          23$          ; NO
36 02212  ERROR    L            ;YES, FLAG ERROR
37 02220 105742  TSTR          -(R2)       ; AND MOVE POINTER BACK
38 02222 112122  23$:  MOVR          (R1)+,(R2)+ ;MOVE CHAR INTO LINE BUFFER
39 02224  RETURN
40
41 02226  22$:  CALL          ENDMAC    ;CLOSE MACRO
42 02232 000167  JMP           1$
   177326
43      .ENDC
44
45

```

Figure P-2 Segment of PDP-11 Code  
Showing 1-, 2-, and 3-word Instructions

## P.2 POSITION INDEPENDENT CODE (PIC)

The output of MACRO-11 (and ASEMBL) assemblies is a relocatable object module. This module, is linked (with LINK) to a specified address prior to being executed.

Once linked, a program can generally be loaded and executed only at the address specified at link time. This is because the Linker has had to make adjustments in some lines to reflect the absolute area of core (locations) in which the program is to run.

It is possible to write a source program that can be loaded and run in any section of core. Such a program is said to consist of position independent code. The construction of position independent code is dependent upon the correct usage of PDP-11 addressing modes. (Addressing modes are described in detail in section 5.4. The remainder of this section assumes the reader is familiar with the various addressing modes.)

All addressing modes involving only register references are position independent. These modes are as follows:

R	register mode
@R	deferred register mode
(R)+	autoincrement mode
@(R)+	deferred autoincrement mode
-(R)	autodecrement mode
@-(R)	deferred autodecrement mode

When using these addressing modes, position independence is guaranteed providing the contents of the registers have been supplied such that they are not dependent upon a particular core location.

The relative addressing modes are generally position independent. These modes are as follows:

A	relative mode
@A	relative deferred mode

Relative modes are not position independent when A is an absolute address (that is, a non-relocatable address) and is referenced from a relocatable module.

Index modes can be either position independent or nonposition independent, according to their usage in the program. These modes are:

X(R)	index mode
@X(R)	index deferred mode

Where the base, X, is position independent, the reference is also position independent. For example:

MOV 2(SP),R0	;POSITION INDEPENDENT
N=4	
MOV N(SP),R0	;POSITION INDEPENDENT
CLR ADDR(R1)	;NONPOSITION INDEPENDENT

Caution must be exercised in the use of index modes in position independent code.

Immediate mode can be either position independent or not, according to its usage. Immediate mode references are formatted as follows:

```
#N          immediate mode
```

Where an absolute number or a symbol defined by an absolute direct assignment replaces N, the code is position independent. Where a label replaces N, the code is nonposition independent. (That is, immediate mode references are position independent only where N is an absolute value.)

Such a reference is position independent if A is an absolute address.

Position independent code is used in writing programs such as device drivers and utility routines which are most useful when they can be brought into any available core space. Figure P-3 and Figure P-4 show pieces of device driver code; one of which is position independent and the other is not.

```
;DVRINT -- ADDRESS OF DEVICE DRIVER INTERRUPT SERVICE
;VECTOR -- ABSOLUTE ADDRESS OF DEVICE INTERRUPT VECTOR
;DRIVE -- START ADDRESS OF DEVICE DRIVER
MOV #DVRINT,VECTOR          ;SET INTERRUPT ADDRESS
MOVB DRIVER+6,VECTOR+2     ;SET PRIORITY
CLRB VECTOR+3              ;CLEAR UPPER STATUS BYTE
```

Figure P-3 Non Position Independent Code

```
MOV PC,R1                  ;GET DRIVER START
ADD #DRIVER-,R1
MOV #VECTOR,R2             ;...& VECTOR ADDRESSES
CLR @R2                    ;SET INTERRUPT ADDRESS
MOVB 5(R1),@R2             ;...AS START ADDRESS+OFFSET
ADD R1,(R2)+
CLR @R2                    ;SET PRIORITY
MOVB 6(R1),@R2
```

Figure P-4 Position Independent Code

In both examples it is assumed that the program calling the device driver has correctly initialized its interrupt vector (VECTOR) within absolute memory locations 0-377. The interrupt entry point offset is in byte DRIVER+5. (The contents of the Driver Table shows at DRIVER+5: .BYTE DVRINT-DRIVER.) The priority level is at byte DRIVER+6.

In the first example, the interrupt address is directly inserted into the absolute address of VECTOR. Neither of these addressing modes are position independent.

The instruction to initialize the driver priority level uses an offset from the beginning of the driver code to the priority value and places that value into the absolute address VECTOR+2 (which is not position independent). The final operation clearing the absolute address

VECTOR+3 is also not position independent.

In the position independent code, operations are performed in registers wherever possible. The process of initializing registers is carefully planned to be position independent. For example, the first two instructions obtain the starting address of the driver. The current PC value is loaded into R1, and the offset from the start of the driver to the current location is added to that value. Each of these operations is position independent. The immediate mode value of VECTOR is loaded into R2; which places the absolute address of the transfer vector into a register for later use. The transfer vector is then cleared, and the offset for the driver starting address is loaded into the vector. The starting address of the driver is then added into the vector, giving the desired entry point to the driver. (This is equivalent to the first statement in Figure P-3.) Since R2 has been updated to point to VECTOR+2, that location is then cleared and the priority level inserted into the appropriate byte.

The position independent code demonstrates a principle of PDP-11 coding practice, which was discussed earlier; that is, the programmer is advised to work primarily with register addressing modes wherever possible, relying on the setup mechanism to determine position independence.

The MACRO-11 Assembler provides the user with a way of checking the position independence of the code. In an assembly listing, MACRO-11 inserts a ' character following the contents of any word which requires the Linker to perform an operation. In some cases this character indicates a nonposition independent instruction, in other cases, it merely draws the user's attention to the use of a symbol which may or may not be position independent. The cases which cause a ' character in the assembly listing are as follows:

1. Absolute mode symbolic references are flagged with an ' character when the reference is not position independent. References are not flagged when they are position independent (i.e., absolute). For example:

```
MOV  @#ADDR,R1          ;PIC ONLY IF ADDR IS ABSOLUTE.
```

2. Index mode and index deferred mode references are flagged with an ' character when the base is a symbolic label address (relocatable rather than an absolute value). For example:

```
MOV  ADDR(R1),R5        ;NON-PIC IF ADDR IS RELOCATABLE.  
MOV  @ADDR(R1),R5       ;NON-PIC IF ADDR IS RELOCATABLE.
```

3. Relative mode and relative deferred mode are flagged with an ' character when the address specified is a global symbol. For example,

```
MOV  GLB1,R1           ;PIC WHEN GLB1 IS A GLOBAL SYMBOL.  
MOV  @GLB1,R1          ;PIC WHEN GLB1 IS A GLOBAL SYMBOL.
```

If the symbol is absolute, the reference is flagged and is not position independent.

4. Immediate mode references to symbolic labels are always flagged with an ' character.

```

MOV #3,R0          ;ALWAYS POSITION INDEPENDENT.
MOV #ADDR.R1       ;NON-PIC WHEN ADDR IS RELOCATABLE.

```

### P.3 REENTRANT CODE

Both the interrupt handling hardware and the subroutine call instructions (JSR, RTS, EMT, and TRAP) facilitate writing reentrant code for the PDP-11. Reentrant code allows a single copy of a given subroutine or program to be shared by more than one process or task. This reduces the amount of core needed for multi-task applications such as the concurrent servicing of peripheral devices.

On the PDP-11, reentrant code depends upon the stack for storage of temporary data values and current processing status. Presence of information in the stack is not affected by the changing of operational control from one task to another. Control is always able to return to complete an operation which was begun earlier but not completed.

### P.4 PREFERRED ADDRESSING MODES

Addressing modes are described in detail in section 5.4. Basically, the PDP-11 programmer has eight types of register addressing and four types of addressing through the PC register. Those operations involving general register addressing take one word of core storage, while symbolic addressing can cost up to three words.

For example:

```

MOV A,B           ;THREE WORDS OF STORAGE
MOV R0,R1         ;ONE WORD OF STORAGE

```

The user is advised to perform as many operations as possible with register addressing modes, and use the remaining addressing modes to preset the registers for an operation. This technique saves space and time over the course of a program.

### P.5 PARAMETER ASSIGNMENTS

Parameter assignments should be used to enable a program to be easily followed. For example:

```

SYM=42
.
.
.
MOV #SYM,R0

```

Another standard PDP-11 convention is to name the general registers as follows:

```

R0 = %0
R1 = %1

```



R2 = %2  
R3 = %3  
R4 = %4  
R5 = %5  
SP = %6 (processor stack pointer)  
PC = %7 (program counter)

The PDP-11/45 floating-point accumulators are named by convention as follows:

AC0 = %0  
AC1 = %1  
AC2 = %2  
AC3 = %3  
AC4 = %4  
AC5 = %5

Use of these standard symbols makes examination of another programmer's code much easier than the use of random symbolic names or constants.

#### NOTE

Where a register reference is made in a 2-bit field within a floating-point instruction, AC0 through AC3 may be referenced. In such instructions the 6-bit source or destination field can be filled with addressing modes 1 through 7 which reference the processor registers R0 through R7 or addressing mode 0 which references floating-point registers AC0 through AC5.

## P.6 SPACE VS. TIMING TRADEOFFS

On the PDP-11, as on all computers, some techniques lead to savings in storage space and others lead to decreased execution time. Only the individual user can determine which is the best combination of the two for his application. It is the purpose of this section to describe several means of conserving core storage and/or saving time.

### P.6.1 Trap Handler

The use of the trap handler and a dispatch table conserve core requirements in subroutine calling, but can lead to a decrease in execution speed due to indirect transfer of control. To illustrate, a subroutine call can be made in either of the following ways:

1. A JSR instruction which generally requires two PDP-11 words:

JSR R5,SUBA

but is direct and fast.

2. A TRAP instruction which requires one PDP-11 word:

TRAP N

but is indirect and slower. The TRAP handler must use N to index through a dispatch table of subroutine addresses and then JMP to the Nth subroutine in the table.

#### P.6.2 Register Increment

The operation:

CMPB (R0)+,(R0)+

is preferable to:

TST (R0)+

to increment R0 by 2. The TST instruction should not be used where the initial contents of R0 may be odd.

#### P.7 CONDITIONAL BRANCH INSTRUCTION

When using the PDP-11 conditional branch instructions, it is imperative that the correct choice be made between the signed and the unsigned branches.

<u>SIGNED</u>	<u>UNSIGNED</u>
BGE	BHIS (BCC)
BLT	BLO
BGT	BHI
BLE	BLOS (BCS)

A common pitfall is to use a signed branch (e.g., BGT) when comparing two memory addresses. A problem occurs when the two addresses have opposite signs; that is, one address goes across the 16K (100000(8)) boundary. This type of coding error usually appears as a result of re-linking at different addresses and/or a change in size of the program.

## APPENDIX Q

### CASSETTE STANDARDS

Extracts from the preliminary INTERCHANGE CASSETTE FILE STRUCTURE STANDARD are included here for reference purposes.

The following describes the format and labeling conventions for files and records written under RT-11 PIPC. This standard must be followed when reading and writing cassettes intended for interchange between systems; it is recommended for other cassettes.

The header records contain information on file, name, data format, record length, volume number for multi-volume files, and creation date.

PIPC provides header record labels, fixed-length, 128-byte records, and date, and supports multi-volume files. There is no continuation of the header record; the second record in a file must be a data record.

#### Q.1 DEFINITIONS

A cassette consists of a sequence of one or more files, separated from each other by a single file gap. The first file on the cassette must be preceded by a file gap; the last file must be followed by a file gap and a Sentinel File (refer to paragraph Q.3), or by clear trailer.

Each file consists of a sequence of a header record plus zero or more data records, separated from each other by record gaps. The first record of a file is called the file header record, or file label.

A record consists of a sequence of one up to  $2^{16}-1$  cassette bytes followed by a two-byte cyclic redundancy check. (This is a logical limit; there is no physical limit, except for the length of the tape.)

A cassette byte is eight bits. A bit is a binary zero (0) or one (1).

A character is a byte interpreted via the ASCII character codes. Parity is not required or written.

## Q.2 THE HEADER RECORD

### Q.2.1 The File Name

Each file must begin with a 32(decimal) byte file header record. Figure Q-1 illustrates the format of the header record. The name and the date are in seven-bit ASCII.

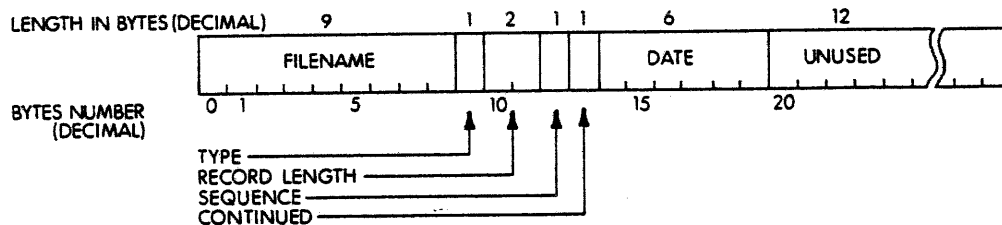


Figure Q-1 File Header Record Format

The first nine bytes of a header record contain the file's name. File names are divided into a six-character "name" and a three character "extension". File names and extensions may consist of letters, numerals and blanks. The first character may not be blank; there can be no imbedded blanks within name or extension; name or extension may be padded on the right with blanks.

#### NOTE

When a file is deleted, the current systems change the name to begin with an asterisk (\*), in addition to setting the type bit (refer to section Q.2.2) to 14.

### Q.2.2 The File Type

Byte nine in the header record contains the "File Type". The File Type defines the mode in which data was recorded in that file. RT-11 PIPC sets the file type to zero which is undefined.

### Q.2.3 File Record Length

Bytes 10 and 11 of the File Header Record contain the file record length which is fixed at 128 bytes per record.

### Q.2.4 File Sequence Number

Byte 12 contains the sequence number for multi-volume files. It is normally zero, otherwise. It is used for information that is split up among files of the same name. Successive continuation files on different cassettes should be numbered 1, 2, 3, ... etc. in this field.

#### Q.2.5 Header Continuation Byte

Byte 13, is set to zero, to indicate that data begins immediately with the next record.

#### Q.2.6 File Creation Date

The file creation date is contained in the six bytes starting at byte 14. When specified, this date shall consist of six seven-bit ASCII digits specifying the day number (01-31), the month number (01-12), and the last two digits of the year number, in the order ddmmyy. If not used, the first byte should be zero (null), or blank (ASCII=40).

#### Q.2.7 Unused Bytes

The twelve bytes starting at byte 20 are not currently specified.

#### Q.3 LOGICAL END OF TAPE

Logical end of tape is signified by clear trailer or a Sentinel File. The Sentinel File consists of a single header record whose file name begins with a zero (null).

#### Q.4 MULTI-VOLUME FILES

Multi-volumn files are supported by the "fall off the tape" method. Whenever the end of a tape is reached before a file have been closed, the system types a message to that effect and allows the user to mount another tape, if necessary.

- On READ, the system:
  1. Types out the message:
  2. If the user indicates that the end of file has been reached, the system returns to PIPC.
  3. If the user indicates that end of file has not been reached, the system allows the user to mount another tape.
  4. The system should verify that there is a file on the tape with the same name and the next higher volume number as the previous file;

5. If that is the case, the system continues processing the file.
- On WRITE, the system:
    1. Erases any partially-written record by backspacing two and forward-spacing one, and writing an EOF to the end of the tape. (When using 128-byte records, the hardware will never mistake the inter-record gap plus the erased tape for a file gap. This is possible when using larger records. Systems using such records should consider the second method for supporting multi-volume files.)
    2. Types out the message;
    3. Allows the user to mount a new tape.
    4. The system may either assume a blank tape, or space to logical end of tape; then write a file gap followed by a header record(s) with the proper name and volume number;
    5. Continue processing.

## APPENDIX R

### PIP COMMAND SUMMARY

This Appendix summarizes the commands described in Chapter 4.

<u>Switch</u>	<u>Explanation</u>
/A	Copies file(s) in ASCII mode (ignores nulls; converts to 7-bit ASCII).
/B	Copies files in formatted binary.
/D	Deletes file(s) from specified device.
/E	Lists the entire directory including unused spaces and their sizes.
/F	Prints the short directory (file names only) of the specified device.
/G	Ignores any input errors which occur during a file transfer and continues copying.
/I or no switch	Copies file(s) in image mode (byte by byte).
/L	Lists the entire directory of the specified device.
/N	Used with /Z to specify the number of directory blocks to allocate to the directory.
/O	Bootstraps the specified device (DT0 or RK0 only).
/R	Renames the specified file.
/S	Compresses the file on the specified directory device so all free area is combined.
/T	Extends number of blocks allocated for a file.
/U	Copies the specified bootstrap file into absolute blocks 0 and 2 of the specified device.
/V	Outputs the version number of the PIP program being used to the terminal.
/X	Copies files individually (without concatenation).
/Y	Causes system files to be operated on by the command specified. Attempted modifications or deletions of .SYS files without /Y are null operations and cause the message ?NO SYS ACTION? to be printed.
/Z	Zeroes (initializes) the directory of the specified device and allows specification of directory size and directory entry size when used with /N.





## GLOSSARY

- absolute address**  
A binary number that is assigned as the address of a fixed core storage location.
- absolute blocks**  
Any blocks which use block 0 of a physical device as a base. Relative blocks use the start of a file as a base.
- address**  
A label, name or number which designates a location where information is stored.
- array**  
A list or table of elements usually variables or data.
- ASCII file**  
A file whose data elements are characters coded in ASCII.
- assembler directives**  
The mnemonics used in the assembly language programs to control or direct the assembly process.
- backup file**  
A copy of a file created for protection in case the primary file is inadvertently destroyed.
- binary file**  
A file whose data elements are binary numbers.
- block replaceable**  
An I/O device on which data is organized in 256-word units (blocks) that can be read or written in a random access manner.
- Bootstrap**  
A technique or device designed to bring a program into memory from an input device.
- breakpoint**  
A preset point in a user program where execution halts to permit program debugging.
- byte**  
A group of 8 binary digits usually operated on as a unit.
- channel number**  
A channel number is an identifier for RT-11. When a file is opened for I/O with the various programmed requests, a channel argument is supplied in the form of an octal number between 0 and 17. When the request is completed, the file can be referenced by the channel number specified.
- command buffer**  
The core area where commands are stored prior to execution.
- command mode**  
The mode of the editor program which interprets an input string as a command.
- command string**  
A series of characters which specify the input/output devices, files, and switches.
- Command String Interpreter**  
The portion of the RT-11 system software which accepts an ASCII string input in response to the \* prompt character and interprets the string as input and output files and switches.
- concatenate**  
To link together.
- Constant Register**  
A logical register in ODT or PATCH which is used to store an oft-used constant.
- core control block**  
Bytes 360-377. Used to indicate which areas of core are to be loaded during GET, RUN, or R commands.
- core image**  
The format executable programs are stored in. A 'picture' of core. The core control block selects the areas to be loaded.
- core resident**  
Denotes code which is always in memory.
- CSI**  
See Command String Interpreter.
- data base**  
An area of core set aside to contain commonly used information.
- data file (BASIC)**  
Any RT-11 file which is read or written by a BASIC program. The default extension of such a file is .DAT, unless otherwise specified.
- debugging**  
The act of detecting, locating and correcting mistakes in a program.
- delimiter**  
A character that separates, terminates and organizes elements of a statement or program.
- device handlers**  
Subroutines designed to transfer data to and from peripheral devices.

directives  
See assembler directives.  
drivers  
See device handlers.

EMT  
A PDP-11 machine instruction.  
EMT is most often used for  
monitor communication.  
EOF  
End-of-File  
EOT  
End-of-Tape  
error flag  
Condition indicating an error has  
occurred. If the C bit is set on  
return from an RT-11 call, an  
error occurred. Thus, the C bit  
set is the RT-11 error flag.  
expression  
A combination of variables,  
constants, and operators (as  
in a mathematical expression).

file  
A collection of related data  
treated as a unit.  
file specification  
That information necessary to  
uniquely specify a file. The  
file specification includes  
device name, file name, extension,  
and length (if needed).  
filler characters  
Null characters output to a device  
to give it time to perform un-  
usually long operations (such as  
return the carriage) without  
explicitly waiting.

general registers  
A set of eight general purpose  
registers available for use as  
accumulators, as auto index  
registers or as pointers. General  
registers 6 and 7 serve as the  
hardware stack pointer and pro-  
gram counter respectively.

.Global  
A value defined in one program  
module and used in others.  
Globals are often referred to  
as "entry" points" in the module  
which they are defined and  
"externals" in the other modules  
which use them.

global symbols  
See .Global.

handlers  
See device handlers.  
hardware bootstrap  
See bootstrap.

I/O queue  
A first-in, first-out list of I/O  
requests which are waiting for  
processing.  
internal registers  
See general registers.  
iteration brackets  
Symbols (<>) used to cause repeti-  
tion of an Editor command or  
commands.

JSW  
Job Status Word. A word in the  
RT-11 communications region con-  
taining bit flags indicating the  
status of the program currently  
in core.

label  
One or more characters used to  
identify a location in a program.

load  
To place data or programs into  
core storage.

load module  
A program which is ready to be  
loaded and executed.

location  
A place in storage or memory where  
a unit of data or instruction may  
be stored.

loop  
A sequence of instructions that is  
executed repeatedly until a termi-  
nal condition prevails.

macro  
An instruction in a source language  
that is equivalent to a specified  
sequence of machine instructions  
or a command in a command language  
that is equivalent to a specified  
sequence of commands.

memory  
The alterable storage in a com-  
puter.

Monitor  
The master control program that  
observes, supervises, controls or  
verifies the operation of a system.

nesting  
Including a program loop inside a loop.

NOP  
An instruction that specifically does nothing (control proceeds to the next instruction in sequence.)

object program  
The relocatable binary program which is output after translation of a source language program.

operand  
1. A quantity which is affected, manipulated or operated upon.  
2. The contents of the field following the operator of an assembler instruction.

operator  
The symbol or code which indicates an action (or operation) to be performed.

overlay  
A method of moving program sections into memory from the system device during processing so several routines can occupy the same core area at different times.

overlay segment  
A section of code handled as a unit. This segment of code can overlay code already in core or be overlaid by other overlay segments.

parameter list  
A set of arguments which may be given different values.

parity error  
An I/O error where the number of bits written does not equal the number of bits read. Reflects a hardware problem.

patch  
To modify a routine.

programmed request  
Machine language instruction which is used to invoke a monitor service for the issuing program.

pseudo-op  
See pseudo operation and assembler directives.

pseudo-operation  
An instruction to the assembler; an operation code that is not part of the computer's hardware command repertoire.

RAD5Ø  
A method of packing three 7-bit ASCII characters into one 16-bit computer word.

relocation base value  
A fixed number added to addresses to create a new address. The original address specified is "relocated" by the fixed amount.

relocation bias  
See relocation base value.

relocatable object module  
A set of instructions which are written so that the module can be loaded and executed in any core area.

root segment  
That portion of an overlaid program which is always in core (resident).

SAVE file  
A core image file produced by the Linker or the .SAVE command. An RT-11 load module.

source file  
A file to be used as input to a translating program such as MACRO or BASIC.

stack  
An area of memory set aside by the programmer for temporary storage or subroutine interrupt service linkage. The stack uses the "last-in, first-out" concept. The stack starts at the highest location reserved for it and expands linearly downward as items are added to the stack.

stack pointer  
A general register used to keep track of the last locations where data is entered into the stack.

symbols  
Names which can be assigned values or can be used to indicate specific locations in a program (see labels).

SYSMAC.SML  
The source file containing the macro definitions for the RT-11 system macros.

system file  
Files which contain vital parts of the monitor. Typically, MONITR.SYS, and any others with the extension .SYS.

system I/O tables  
Tables internal to the monitor which translate user I/O requests into physical device requests.

system macro library  
See SYSMAC.SML.

tentative output file  
The output of .ENTER. The tentative file will become a permanent entry when a .CLOSE is given.

terms  
The basic elements of expressions.

text mode  
That mode of EDIT operation where the editor interprets characters as text as opposed to commands.

text string  
A connected sequence of ASCII text.

trap  
A trap is an automatic transfer of control to a prespecified routine that can be caused by software or hardware. The trap instruction is an example of a hardware implementation.

trap handler  
The code transferred to when a TRAP instruction is executed. The trap handler address should be stored in location 34.

truncate  
Reduce by dropping one or more of the least significant digits.

type ahead feature  
The ability to type information at the console terminal and have it remembered by the system for later use.

unary operator  
An operator that applies to only one operand. e.g., in the expression -A, the minus sign is a unary operator.

User Service Routine (USR)  
A system routine which loads device handlers, searches file directories, creates and closes output files and contains the Command String Interpreter.

user stack area  
The area of core the user wishes to use for his stack storage space. Specified in the Linker or in the arguments to SAVE.

USR  
See User Service Routines.

utility program  
Any program which performs useful functions, i.e., PIP.

word  
In the PDP-11 a 16-bit unit of data which may be stored in an addressable location.

- Absolute,
  - Addresses, ODT, 7-6
  - Base Address, ODT, 7-1
  - Program sections, Linker, 6-1
- Accessing,
  - General registers, 7-10
  - Internal registers, 7-10
- Address
  - Assignment, Linker, 6-1
  - Mode Syntax, Assembler, F-2
  - Search, 7-15, 7-27
- Addresses or Numbers,
  - ODT Conversion of, 7-16
- Addresses, ODT,
  - Absolute, 7-6
  - Relative, 7-6
- Addressing modes, 5-20, P-7
- Advance command (EDIT),
  - 3-16
- Allocation of System
  - resources, 2-4
- Alphabetize Switch,
  - Linker, 6-11
- Altmode key, 3-2
- Angle bracket, 2-12, 3-8,
  - 5-44, 7-9
- Apostrophe character, MACRO,
  - 5-19
- Arguments,
  - Programmed Request, 8-1
- Arguments, Number of, 5-65
- ASCII,
  - Character set, E-1
  - Directive, 5-40
  - Input and Output, 7-20
- .ASCIIZ directive, 5-42
- .ASECT directive, 5-52, 6-1
- ASEMBL, 5-1
  - Differences from MACRO, 10-1
  - Directives not available, 10-4
  - Error messages, 10-5
  - Operating procedures, 10-1
  - 8K Assembler, 10-1
- Assembler,
  - Address mode syntax, F-2
  - ASEMBL, 8K, 10-1
  - Branch instructions, F-8
  - Directives, 5-26, F-12, F-14
  - Double operand instructions, F-3
  - Double register destination
    - instruction, F-12
  - Floating-point source double
    - register instruction, F-11
  - Instructions, F-2
  - MACRO, 5-1
  - Operate Instructions, F-6
  - Priority Instruction, F-12, F-14
  - Register destination, F-9
  - Register offset instruction, F-9
  - Single operand instructions, F-4
  - Source double register
    - instruction, F-12
  - Source register instruction, F-10
  - Subroutine return instruction, F-10
  - Trap instructions, F-8
- Assemblies,
  - Conditional, P-3
  - System, A-7
- Assembly language programs,
  - Commenting, P-2
- ASSIGN Command, Monitor, 2-5
- Asterisk character,
  - EDIT, 3-5
  - Linker, 6-9
  - Monitor, 2-2
  - ODT, 7-6
  - PIP, 4-1
- Automatic Relocation
  - Facility, ODT, 7-1
- Automatically Created
  - Symbols, 5-65
- Back Arrow or Underline
  - Key, ODT, 7-8
- Backslash, ODT, 7-7
- Bad entry,
  - ODT Breakpoints, 7-20
- Base command, Monitor, 2-8
- BASIC/RT11,
  - Commands, L-3
  - Error messages, L-4, L-5
  - Functions, L-4
  - Statements, L-1
- Binary Relocatable Object
  - Module, 7-1
- .BLKB directive, 5-46
- .BLKW directive, 5-46
- Boot command (PIP), 4-13
- Bootstrap,
  - BM792-YB, 2-1
  - MR11-DB, 2-1
- Bottom Switch, Linker, 6-11
- Branch instruction,
  - Addressing, 5-25
  - Assembler, F-8
  - Conditional, P-9
- Branching, 7-17
- Breakpoints, 7-12, 7-24
  - Bad entry, 7-20
  - Different priority levels, 7-19
  - ODT, 7-1
  - Repeat counts, 7-14
  - Set in a loop, 7-13
- .BYTE directive, 5-38

Calling and Using,  
     EDIT, 3-1  
     PIP, 4-1  
     PIPC, N-4  
 Cassette,  
     Description, PIPC, N-1  
     Format, PIPC, N-2  
     Mounting and dismounting, PIPC, N-3  
     Standards, Q-1  
 CBOOT instructions, A-4  
 Change of file name, 8-36  
 Changing Locations, ODT, 7-7  
 Channel number, 8-2  
 Character deletion, 2-3, 3-2  
 Character set,  
     ASCII, E-1  
     MACRO, 5-5, E-1  
     RADIX-50, E-4  
 Character transfer, 8-41,  
     8-42  
 Characters, Radix-50, 7-11  
 CLOSE command, Monitor, 2-6  
 .CLOSE request, 8-12  
 Closing locations, ODT, 7-3, 7-7  
 Co-resident Overlay  
     Routines, Linker, 6-5  
 Code, reentrant, P-7  
 Command,  
     Format, EDIT, 3-4  
     Input Buffer (EDIT), 3-10  
     Mode, EDIT, 3-3  
     Repetition (EDIT), 3-8  
     String, Linker, 6-9  
 Command Summary,  
     EDIT, I-1  
     ODT, 7-3  
     PATCH, O-2  
 Commands,  
     BASIC/RT11, L-3  
     EDIT, 3-2  
     Keyboard Monitor, 2-3  
     ODT, 7-3, 7-6  
     PIP, 4-2  
     PIPC, N-6  
 Commands to Move Location  
     Pointer (EDIT), 3-15  
 Comment field, 5-4  
 Commenting Assembly  
     language programs, P-2  
 Compress command (PIP),  
     4-12  
 Concatenation, MACRO, 5-66  
 Conditional Assemblies, P-3  
 Conditional Assembly  
     Directives, 5-54  
     PAL-11R, 5-58  
     PAL-11S, 5-58  
 Conditional branch  
     instruction, P-9  
 Condition codes, 7-10  
 Constant register, 7-2, 7-16  
 Contiguous File structure,  
     8-7  
 Continue Switch, Linker, 6-13  
 Control Keys, Keyboard  
     Monitor, 2-3  
 Conversion of Addresses or  
     Numbers, 7-16  
 Copy commands (PIP), 4-4  
 Core,  
     Block Initialization, 7-16  
     Image File, Loading, 2-6  
     Programmed requests swapping, 8-10  
     Requirement, Linker, 6-1  
     Usage, EDIT, 3-10  
     Usage Map, Linker, 6-3  
 .CSECT directive, 5-52, 6-1  
 CSI,  
     General mode, 8-13  
     Special mode, 8-15  
 .CSIGEN request, 8-13  
 .CSISPC request, 8-15  
 CTRL commands, EDIT, 3-2  
 CTRL keys, 2-3  
 CTRL/C ODT return to  
     Monitor, 7-30  
 CTRL/U ODT Search  
     termination, 7-30  
 Customization, system, A-7  
  
 Data storage directives,  
     5-37  
 Data Transfer Requests, 8-8  
 DATE Command, Monitor, 2-4  
 .DATE request, 8-11  
 Double register instruction,  
     Assembler source, F-12  
     Floating-point source, F-11  
 Decimal point, MACRO, 5-45  
 Delete,  
     Character or line, 2-3, 3-2  
     Command, (EDIT), 3-20, (PIP), 4-7  
     .DELETE request, 8-19  
 Delimiter characters, 3-25, 5-6, 5-7  
 Deposit Command, Monitor, 2-7  
 Destination,  
     Assembler register, F-9  
     Assembler double register, F-12  
 Device Block,  
     Programmed Request, 8-2  
 Device Assignment, Monitor,  
     2-5  
 Device handler loading, 8-24  
 Device names, 2-12  
     Logical, 2-13  
     Permanent, 2-13  
     Physical, 2-12, B-1  
 Direct assignment  
     statement, MACRO, 5-11

- Directive,
  - Program Boundaries, 5-30
  - Symbol Control, 5-53
- Directives,
  - Assembler, F-12, F-14
  - Conditional Assembly, 5-54
  - EXPAND, 9-1
  - PAL-11R Conditional Assembly, 5-58
  - PAL-11S Conditional Assembly, 5-58
  - Program Section, 5-30
  - Subconditional, 5-55
  - Terminating, 5-49
- Directory initialization command (PIP), 4-11
- Directory list commands (PIP), 4-9
- Dismounting a Cassette, N-3
- Dot Character, Monitor, 2-2
- Dot usage, EXPAND, 9-2
- Double operand instructions, Assembler, F-3
- .DSABL directive, 5-36
- .DSTATUS request, 8-20
- EDIT,
  - Calling and Using, 3-1
  - Commands, 3-2
  - Command summary, I-1
  - Error messages, 3-27
  - Example, 3-29
  - Key commands, 3-2
  - Mode of operation, 3-3
- Empty entry, 8-7
- EMT trap instruction, 7-25
- .ENABL directive, 5-36
- .END directive, 5-49
- .ENDM directive, 5-59
- .ENTER request, 8-21
- Entry point, MACRO, 5-10
- Equal Sign, 2-12
- Error Detection,
  - ODT, 7-20
- .ERROR directive, 5-68
- Error message summary, D-1
- Error Messages,
  - ASEMBL, 10-5
  - BASIC/RT11, L-4, L-5
  - EDIT, 3-27
  - EXPAND, 9-4
  - Linker, 6-15
  - MACRO, 5-74
  - Monitor, 2-14
  - PATCH, 0-4
  - PIP, 4-13
  - PIPC, N-10
- .EVEN directive, 5-45
- Example, EDIT, 3-29
- Example command, Monitor, 2-7
- .EXIT request, 8-23
- EXPAND, 5-1
  - Error messages, 9-4
  - Language, 9-1
  - Operation of, 9-2
  - Restrictions, 9-1
  - Utility Program, 9-1
- Expressions, 5-18
  - Symbols and, 5-5
- Extend command (PIP), 4-9
- Extensions,
  - File Names and, 2-13
- External Symbol, Linker, 6-2
- .FETCH request, 8-24
- File,
  - Allocation (PIP), 4-5
  - Closing, Monitor, 2-6
  - Deletion (PIP), 4-7
  - Descriptor blocks, 8-16
  - Manipulation Requests, 8-8
  - Structure, contiguous, 8-7
- File name, 2-12
  - Change of, 8-36
  - Extensions, 2-13
- File Specification,
  - Monitor, 2-12
  - EXPAND, 9-2
- Files,
  - Permanent, 8-7
  - Tentative, 8-7
- Floating-point number,
  - MACRO, 5-47
- Floating-point source double register instruction, F-11
- .FLT2 directive, 5-47
- .FLT4 directive, 5-47
- Form feed character, 5-1, 5-61
- Format control, 5-4
- Function,
  - Key, Special, 2-3
- Functional organization, 7-21
- Functions,
  - BASIC/RT11, L-4
  - ODT Commands and, 7-6
- General mode call,
  - CSI, 8-13
- General registers,
  - Accessing, 7-10
- GET Command, Monitor, 2-6
- Global symbols, 5-10, 6-2
- .GLOBL assembler directive, 5-53, 6-2

Handler, Trap, P-8  
 Handler release, 8-35  
 Hardware Requirements, 1-2  
 Hardware, Linker, 6-1  
 .HRESET request, 8-25  
 I/O,  
   Modes of, 8-30  
   Monitor que, 8-28  
   .IDENT directive, 5-36  
 Illegal characters, MACRO,  
   5-7  
 Increment register, P-9  
 Indefinite Repeat Block,  
   MACRO, 5-69  
 Inhibit Printer, 2-3  
 INITIALIZE command, Monitor,  
   2-4  
 Input,  
   Terminal keyboard, 2-3  
 Input and Output,  
   Linker, 6-2  
   ODT ASCII, 7-20  
 Input/Output commands,  
   EDIT, 3-10  
 Instruction,  
   Assembler double register  
     destination, F-12  
   Assembler floating-point  
     source double register, F-11  
   Assembler priority, F-12, F-14  
   Assembler register offset,  
     F-9  
   Assembler source register,  
     F-10  
   Assembler Subroutine  
     return, F-10  
 Instructions,  
   Assembler, F-2  
   Assembler branch, F-8  
   Assembler double operand,  
     F-3  
   Assembler operate, F-6  
   Assembler single operand,  
     F-4  
   Assembler Trap, F-8  
 Internal registers,  
   Accessing, 7-10  
 Internal structure, 7-21  
 Internal Symbol, Linker,  
   6-2  
 Interrupt Priority level,  
   7-10, 7-19  
 IOT trap instruction, 7-25  
 .IRP directive, 5-69  
 .IRPC directive, 5-69  
 Key commands,  
   EDIT, 3-2  
 Key, Special Function, 2-3  
 Keyboard commands,  
   Monitor, 2-3  
   Summary, C-1  
 Keyboard Communication,  
   Monitor, 2-3  
 Keyboard input,  
   Terminal, 2-3  
 Keyboard Monitor,  
   Control Keys, 2-3  
 Keys, CTRL, 2-3  
 Label field, 5-2  
 Language, EXPAND, 9-1  
 LDA Format Switch,  
   Linker, 6-13  
 Left angle bracket, 2-12, 7-9  
 .LIMIT directive, 5-30  
 Line,  
   Deletion, 2-3, 3-2  
   Feed character, 5-1, 7-8  
   Printer, Off-line, 2-3  
   Terminator, 3-4  
 Linker,  
   Error Messages, 6-15  
   Global Symbols, 6-2  
   Input/Output, 6-2  
   Load Map, 6-3  
   Load Module, 6-2  
   Object Module, 6-2  
   Operating Procedures, 6-9  
   Overlay Facility, 6-5  
   Pass 1, 6-3  
   Summary, J-1  
   Transfer address switch, 6-14  
 Linking,  
   Relocation and, 5-19  
   System, A-7  
 .LIST directive, 5-27  
 Listing control directives,  
   MACRO, 5-27  
 Load Map,  
   Linker, 6-3  
 Load Module,  
   Linker, 6-2  
 Loading,  
   Core Image File, 2-6  
   Device handler, 8-24  
   USR, 8-5  
 Local symbols, MACRO, 5-13  
 Location 46, 8-5  
   Counter control, 5-45  
   Counter, MACRO, 5-14  
   Pointer (EDIT),  
     Commands to Move, 3-15



Locations, ODT,  
     Changing, 7-7  
     Closing, 7-3, 7-7  
     Opening, 7-3, 7-7  
 .LOCK request, 8-25  
 Logical Device names, 2-13  
 .LOOKUP request, 8-26  
  
 MACRO,  
     Assembler directives, 5-26  
 MACRO Assembler, 5-1  
 Macro,  
     Buffer (EDIT), 3-10  
     Calls, 8-1  
     Character set, E-1  
     Command (EDIT), 3-25  
     Definition, 5-59  
     Delimiter character  
         (EDIT), 3-25  
     Directive, 5-59  
     Libraries, 5-72  
     Nesting, 5-62  
     Restrictions, (EXPAND), 9-2  
     Source program format, 5-1  
     Symbols, 5-9  
 .MCALL directive, 5-72  
 .MEXIT Directive, 5-60  
 Mode of operation, EDIT, 3-3  
 Modes of I/O, 8-30  
 Modular programming, P-1  
 Monitor,  
     ASSIGN Command, 2-5  
     Control Keys, 2-3  
     Error messages, 2-14  
     File specification, 2-12  
     I/O que, 8-28  
     Keyboard Communication, 2-3  
     Start Procedure, 2-1  
 Mounting and Dismounting a  
     Cassette, N-3  
 Multiple copy commands  
     (PIP), 4-5  
  
 .NARG directive, 5-67  
 .NCHR directive, 5-67  
 .NLIST directive, 5-27  
 Non-numeric arguments  
     (EDIT), 3-8  
 .NTYPE directive, 5-67  
 Numbers, 5-16, 7-16  
 Numeric,  
     Arguments, 3-6  
     Control, 5-47, 5-48  
  
 Object Module,  
     Linker, 6-2  
  
 .ODD directive, 5-45  
 ODT,  
     Commands and Functions,  
         7-3, 7-6  
     Error Detection, 7-20  
     Operating Procedures, 7-28  
     Programming  
         Consideration, 7-21  
         Radix-50 mode, 7-11  
     Relocatable Expressions, 7-2  
     Relocation, 7-1  
     Return to Monitor, 7-30  
     Search termination, 7-30  
     Summary, K-1  
 Off-line,  
     Line Printer, 2-3  
 Offset calculation, 7-17  
 Open addressed location, 7-9  
 Opening Locations, ODT, 7-7  
 Operand field, 5-4  
 Operate Instructions,  
     Assembler, F-6  
 Operating procedures,  
     ASEMBL, 10-1  
     Linker, 6-9  
     MACRO, 5-73  
     ODT, 7-28  
 Operation of EXPAND, 9-2  
 Operator characters, MACRO,  
     5-8  
 Operator field, 5-3  
 Options, PIPC, N-6  
 Overlay,  
     Facility, Linker, 6-5  
     Routines, Linker, Co-  
         resident, 6-5  
     Rules, Linker, 6-6  
     Switch, Linker, 6-13  
  
 Page ejection, 5-36, 5-61  
 Page headings, 5-33  
 PAL-11R Conditional  
     Assembly Directives, 5-58  
 PAL-11S Conditional  
     Assembly Directives, 5-58  
 Paper Tape Reader Start Up,  
     2-2  
 Parameter assignments, P-7  
 Parameter words, 8-3  
 Pass 1, Linker, 6-3  
 PATCH,  
     Command summary, O-2  
     Error messages, O-4  
     Program, O-1  
 PDP-11 Programming  
     fundamentals, P-1

Percent character MACRO, 5-12  
 Period usage, 8-2  
 Peripheral Interchange  
     Program, PIP, 4-1  
 Peripheral Interchange  
     program, Cassette, N-1  
 Permanent Device Names, 2-13  
 Permanent Files, 8-7  
 Permanent symbols, MACRO,  
     5-9  
 Physical device names, 2-12, B-1  
 PIC (Position Independent  
     Code), 7-18, P-3  
 PIP, 4-1  
     Calling and Using, 4-1  
     Commands, 4-2  
     Error messages, 4-13  
     Summary, R-1  
 PIPC,  
     Calling and Using, N-4  
     Cassette description, N-1  
     Cassette format, N-2  
     Commands, N-6  
     Error messages, N-10  
     Mounting and dis-  
         mounting a Cassette, N-3  
     Options, N-6  
     Sentinel file, N-3  
 Pointer, current character  
     (EDIT), 3-6  
 Position command (EDIT), 3-19  
 Position Independent Code  
     (PIC), P-3  
 .PRINT directive, 5-68  
 .PRINT request, 8-27  
 Printing Inhibited, 2-3  
 Printout Formats, ODT, 7-6  
 Priority Instruction,  
     Assembler, F-12, F-14  
 Priority level, 7-19  
     Interrupt, 7-10  
 Processor priority level,  
     7-19  
 Program Boundaries  
     Directive, 5-30  
 Program counter, 5-20  
 Program execution, 7-13  
 Program Section Directives,  
     5-30  
 Program sections, Linker,  
     Absolute, 6-1  
     Relocatable, 6-1  
 Programmed Request  
     Arguments, 8-1  
 Programmed Request  
     Device Block, 8-2  
 Programmed requests, 8-1, 8-7  
     Core swapping, 8-10  
         Summary, 8-8, H-1  
 Programming Consideration,  
     ODT, 7-21  
 Programming fundamentals,  
     PDP-11, P-1  
 Q error flag, 5-18  
 .QSET request, 8-28  
 R Command, Monitor, 2-10  
 .RAD50 directive, 5-42,  
     7-11  
 Radix-50 mode, 7-11  
 Radix control, 5-43  
     Temporary, 5-44  
 .RADIX directive, 5-43  
 Radix-50 Characters, 7-11, E-4  
 .RCTRLO request, 8-29  
 .READ request, 8-31  
 .READC request, 8-31  
 .READW request, 8-32  
 Reenter command, Monitor,  
     2-11  
 Reentrant Code, P-7  
 Register,  
     Constant (ODT), 7-16  
     Destination (Assembler), F-9  
     Increment, P-9  
     Instruction, F-11, F-12  
     Offset instruction (Assembler), F-9  
     Relocation, 7-2  
     Status, 7-10  
     Symbols, MACRO, 5-12  
     Usage, P-2  
 Relative,  
     Addresses (ODT), 7-6  
     Addressing, 7-17  
     Branch offset, 7-9  
 .RELEAS request, 8-35  
 Release from core, 8-44  
 Relocatable,  
     Assembler, ODT, 7-1  
     Expressions, ODT, 7-2  
     Program sections, Linker, 6-1  
 Relocation and Linking,  
     5-19  
 Relocation  
     Bias, 7-1, 7-18  
     Calculators, 7-18  
     Automatic facility (ODT), 7-1  
     Register commands, 7-17  
     Registers, 7-2, 7-18  
 Rename command (PIP), 4-8  
 .RENAME request, 8-36  
 .REOPEN request, 8-37  
 Repeat Block, MACRO, 5-72  
     Indefinite, 5-69  
 .REPT directive, 5-72  
 Requests,

- Data Transfer, 8-8
- File Manipulation, 8-8
- Resident Monitor, RMON, 1-2
- Restrictions,
  - EXPAND, 9-1
  - RETURN Key, ODT, 7-7
  - Return to previous sequence, 7-9
- RMON,
  - Resident Monitor, 1-2
  - Start address, 8-6
- Root Segment, Linker, 6-3, 6-5
- RT-11,
  - Error message summary, D-1
  - Programmed requests Summary, 8-8
  - System concepts, 8-2
  - Types of Programmed requests, 8-7
- RUBOUT key, 3-2
- Run Command, Monitor, 2-10
- Runaway program, 7-26
- Running the program, 7-13
- Save Buffer (EDIT), 3-10
- Save command, Monitor, 2-8
- .SAVE status request, 8-38
- .SBTTL directive, 5-33
- Search commands (EDIT), 3-17
- Search termination, 7-16
  - ODT, 7-30
- Sentinel file, PIPC, N-3
- Separating characters,
  - MACRO, 5-6
- .SETTOP request, 8-39
- Single Instruction mode, 7-14
- Single Operand Instructions,
  - Assembler, F-4
- Slash, ODT, 7-7
- Software components, 1-2
- Software reset, 8-40
- Source double register
  - instruction, F-11, F-12
- Source program format,
  - MACRO, 5-1
- Space,
  - Timing vs., P-8
- Special Character, MACRO, 5-63
- Special mode, CSI, 8-15
- .SRESET request, 8-40
- Stack Pointer, Monitor, 2-11
- Start Address, 2-11, 7-28, 8-6
- Start command, Monitor, 2-11
- Start Procedure, Monitor, 2-1
- Statements,
  - BASIC/RT11, L-1
- Status of device, 8-20
- Status Register, 7-10
- Subconditional Directives, 5-55
- Subroutine Return
  - Instruction, Assembler, F-10
- Swapping algorithms, 8-5
- Switches,
  - PIP, 4-3
  - Linker, 6-10
  - RT-11 Utility Program, 2-12
- Symbol Control Directive, 5-53
- Symbols,
  - Automatically Created, 5-65
- Symbols and Expressions, 5-5
- Syntax,
  - Assembler Address mode, F-2
- System,
  - Assembly, A-7
  - Build, A-1
  - Concepts, 8-2
  - Customization, A-7
  - Linking, A-7
  - Macro file, G-1
  - Resources, Monitor allocation, 2-4
- TAll Cassette, 1-2
- Tab key, 3-2
- Temporary Numeric Control, 5-48
- Temporary Radix control, 5-44
- Tentative Files, 8-7
- Terminal interrupt, 7-28
- Terminal keyboard input, 2-3
- Terminating directives, 5-49
- Terminator, line, 3-4
- Terms, MACRO, 5-17
- Text Buffer (EDIT), 3-10
- Text Editor Program,
  - EDIT, 3-1
- Text mode, EDIT, 3-3
- Timing vs. Space, P-8
- .TITLE directive, 5-33
- Trace trap instruction, 7-24
- Transfer Address Switch,
  - Linker, 6-14
- Trap Handler, P-8

Trap Instructions,  
  Assembler, F-8  
TRAP instruction, 7-25  
.TTINR request, 8-41  
.TTOUTR Request, 8-42  
.TTYIN request, 8-41  
.TTYOUT request, 8-42  
Types of Programmed  
  Requests, 8-7  
  
.UNLOCK request, 8-44  
Unsave command (EDIT), 3-25  
Up arrow character, 2-2  
  5-44, 5-48, 7-8  
Up arrow key, ODT, 7-8  
User Service Routine  
  Release from core, 8-44  
User-defined symbols, 5-9  
Using PIPC,  
  Calling and, N-4  
USR Loading, 8-5  
  
Utility commands (EDIT),  
  3-23  
Utility Program,  
  EXPAND, 9-1  
Utility Program Switches,  
  2-12  
  
Version command (PIP), 4-13  
  
.WAIT request, 8-44  
Wild card (PIP), 4-1  
.WORD directive, 5-39  
Word searches, 7-15, 7-27  
Word transfer, 8-31, 8-32,  
  8-45, 8-46  
.WRITC request, 8-45  
.WRITE request, 8-46  
.WRITW request, 8-47  
  
Z error code, 5-21

## HOW TO OBTAIN SOFTWARE INFORMATION

Announcements for new and revised software, as well as programming notes, software problems, and documentation corrections, are published by Software Information Service in the following newsletters.

DIGITAL Software News for the PDP-8 and PDP-12  
DIGITAL Software News for the PDP-11  
DIGITAL Software News for 18-bit Computers

These newsletters contain information applicable to software available from DIGITAL'S Software Distribution Center. Articles in DIGITAL Software News update the cumulative Software Performance Summary which is included in each basic kit of system software for new computers. To assure that the monthly DIGITAL Software News is sent to the appropriate software contact at your installation, please check with the Software Specialist or Sales Engineer at your nearest DIGITAL office.

Questions or problems concerning DIGITAL'S software should be reported to the Software Specialist. If no Software Specialist is available, please send a Software Performance Report form with details of the problems to:

Digital Equipment Corporation  
Software Information Service  
Software Engineering and Services  
Maynard, Massachusetts 01754

These forms, which are provided in the software kit, should be fully completed and accompanied by terminal output as well as listings or tapes of the user program to facilitate a complete investigation. An answer will be sent to the individual, and appropriate topics of general interest will be printed in the newsletter.

Orders for new and revised software manuals, additional Software Performance Report forms, and software price lists should be directed to the nearest DIGITAL field office or representative. USA customers may order directly from the Software Distribution Center in Maynard. When ordering, include the code number and a brief description of the software requested.

Digital Equipment Computer Users Society (DECUS) maintains a user library and publishes a catalog of programs as well as the DECUSCOPE magazine for its members and non-members who request it. For further information, please write to:

Digital Equipment Corporation  
DECUS  
Software Engineering and Services  
Maynard, Massachusetts 01754

READER'S COMMENTS

Digital Equipment Corporation maintains a continuous effort to improve the quality and usefulness of its publications. To do this effectively we need user feedback--your critical evaluation of this document.

Did you find errors in this document? If so, please specify by page.

---

---

---

---

---

How can this document be improved?

---

---

---

---

---

How does this document compare with other technical documents you have read?

---

---

---

---

---

Job Title \_\_\_\_\_ Date: \_\_\_\_\_

Name: \_\_\_\_\_ Organization: \_\_\_\_\_

Street: \_\_\_\_\_ Department: \_\_\_\_\_

City: \_\_\_\_\_ State: \_\_\_\_\_ Zip or Country \_\_\_\_\_

Fold Here

Do Not Tear - Fold Here and Staple

FIRST CLASS  
PERMIT NO. 33  
MAYNARD, MASS.

BUSINESS REPLY MAIL  
NO POSTAGE STAMP NECESSARY IF MAILED IN THE UNITED STATES

Postage will be paid by:

**digital**

Digital Equipment Corporation  
Software Information Service  
Software Engineering and Services  
Maynard, Massachusetts 01754

