SOFTWARE REFERENCE MANUAL

OPERATING SYSTEM

Model HT11

for the

H11 Digital Computer System

Portions of this material have been adapted from Digital Equipment Corporation publications or documents. Heath Company assumes responsibility for the accuracy and content of this material.

Copyright © 1978
Heath Company
All Rights Reserved

HEATH COMPANY
BENTON HARBOR, MICHIGAN 49022

Part C
595-2225-01
Printed in the United States of America

CONTENTS

			Page
PREFACE			xiii
CHAPTER	1	**HT-11 OVERVIEW**	**1-1**
	1.1	PROGRAM DEVELOPMENT	1-2
	1.2	SYSTEM SOFTWARE COMPONENTS	1-2
CHAPTER	2	**SYSTEM COMMUNICATION**	**2-1**
	2.1	START PROCEDURE	2-1
	2.2	SYSTEM CONVENTIONS	2-1
	2.2.1	Data Formats	2-1
	2.2.2	Prompting Characters	2-2
	2.2.3	Physical Device Names	2-2
	2.2.4	File Names and Extensions	2-3
	2.2.5	Device Structures	2-4
	2.3	MONITOR SOFTWARE COMPONENTS	2-4
	2.3.1	Resident Monitor (RMON)	2-4
	2.3.2	Keyboard Monitor (KMON)	2-4
	2.3.3	User Service Routine (USR)	2-5
	2.3.4	Device Handlers	2-5
	2.4	GENERAL MEMORY LAYOUT	2-5
	2.5	ENTERING COMMAND INFORMATION	2-5
	2.6	KEYBOARD COMMUNICATION (KMON)	2-6
	2.6.1	Type-Ahead	2-7
	2.7	KEYBOARD COMMANDS	2-8
	2.7.1	Commands to Allocate System Resources	2-8
	2.7.1.1	DATE Command	2-8
	2.7.1.2	TIME Command	2-8
	2.7.1.3	INITIALIZE Command	2-9
	2.7.1.4	ASSIGN Command	2-9
	2.7.1.5	CLOSE Command	2-10
	2.7.1.6	LOAD Command	2-11
	2.7.1.7	UNLOAD Command	2-11
	2.7.1.8	SET Command	2-11
	2.7.2	Commands to Manipulate Memory Images	2-12
	2.7.2.1	GET Command	2-12
	2.7.2.2	Base Command	2-14
	2.7.2.3	Examine Command	2-15
	2.7.2.4	Deposit Command	2-15
	2.7.2.5	SAVE Command	2-16
	2.7.3	Commands to Start a Program	2-17
	2.7.3.1	RUN Command	2-17
	2.7.3.2	R Command	2-18
	2.7.3.3	START Command	2-18
	2.7.3.4	REENTER Command	2-19
	2.8	MONITOR ERROR MESSAGES	2-19
	2.8.1	Monitor HALTS	2-22
CHAPTER	3	**TEXT EDITOR**	**3-1**

CONTENTS (Cont.)

		Page
3.1	CALLING AND USING EDIT	3-1
3.2	MODES OF OPERATION	3-1
3.3	SPECIAL KEY COMMANDS	3-2
3.4	COMMAND STRUCTURE	3-3
3.4.1	Arguments	3-4
3.4.2	Command Strings	3-4
3.4.3	The Current Location Pointer	3-5
3.4.4	Character- and Line-Oriented Command Properties	3-5
3.4.5	Command Repetition	3-6
3.5	MEMORY USAGE	3-8
3.6	EDITING COMMANDS	3-9
3.6.1	Input/Output Commands	3-9
3.6.1.1	Edit Read	3-9
3.6.1.2	Edit Write	3-9
3.6.1.3	Edit Backup	3-10
3.6.1.4	Read	3-10
3.6.1.5	Write	3-11
3.6.1.6	Next	3-11
3.6.1.7	List	3-12
3.6.1.8	Verify	3-12
3.6.1.9	End File	3-13
3.6.1.10	Exit	3-13
3.6.2	Pointer Relocation Commands	3-13
3.6.2.1	Beginning	3-13
3.6.2.2	Jump	3-14
3.6.2.3	Advance	3-14
3.6.3	Search Commands	3-15
3.6.3.1	Get	3-15
3.6.3.2	Find	3-16
3.6.3.3	Position	3-16
3.6.4	Text Modification Commands	3-17
3.6.4.1	Insert	3-17
3.6.4.2	Delete	3-17
3.6.4.3	Kill	3-18
3.6.4.4	Change	3-19
3.6.4.5	Exchange	3-20
3.6.5	Utility Commands	3-20
3.6.5.1	Save	3-20
3.6.5.2	Unsave	3-21
3.6.5.3	Macro	3-21
3.6.5.4	Execute Macro	3-22
3.6.5.5	Edit Version	3-22
3.6.5.6	Upper- and Lower-Case Commands	3-23
3.7	EDIT EXAMPLE	3-24
3.8	EDIT ERROR MESSAGES	3-25
CHAPTER 4	**PERIPHERAL INTERCHANGE PROGRAM (PIP)**	**4-1**
4.1	CALLING AND USING PIP	4-1
4.1.1	Using the "Wild Card" Construction	4-1

CONTENTS (Cont.)

			Page
	4.2	PIP SWITCHES	4-2
	4.2.1	Copy Operations	4-3
	4.2.2	Multiple Copy Operations	4-6
	4.2.3	The Extend and Delete Operations	4-7
	4.2.4	The Rename Operation	4-9
	4.2.5	Directory List Operations	4-10
	4.2.6	The Directory Initialization Operation	4-11
	4.2.7	The Compress Operation	4-12
	4.2.8	The Bootstrap Copy Operation	4-13
	4.2.9	The Boot Operation	4-13
	4.2.10	The Version Switch	4-14
	4.2.11	Bad Block Scan (/K)	4-14
	4.2.11.1	Recovery from Bad Blocks	4-14
	4.3	PIP ERROR MESSAGES	4-17
CHAPTER	**5**	**THE ASSEMBLY PROCESS**	**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	5-3
	5.1.1.4	Comment Field	5-3
	5.1.2	Format Control	5-4
	5.2	SYMBOLS AND EXPRESSIONS	5-4
	5.2.1	Character Set	5-4
	5.2.1.1	Separating Characters	5-5
	5.2.1.2	Illegal Characters	5-6
	5.2.1.3	Operator Characters	5-6
	5.2.2	Symbols	5-7
	5.2.2.1	Permanent Symbols	5-7
	5.2.2.2	User-Defined Symbols	5-7
	5.2.3	Direct Assignment	5-8
	5.2.4	Register Symbols	5-9
	5.2.5	Local Symbols	5-10
	5.2.6	Assembly Location Counter	5-11
	5.2.7	Numbers	5-13
	5.2.8	Terms	5-14
	5.2.9	Expressions	5-14
	5.3	RELOCATION AND LINKING	5-16
	5.4	ADDRESSING MODES	5-16
	5.4.1	Register Mode	5-17
	5.4.2	Register Deferred Mode	5-17
	5.4.3	Autoincrement Mode	5-17
	5.4.4	Autoincrement Deferred Mode	5-18
	5.4.5	Autodecrement Mode	5-18
	5.4.6	Autodecrement Deferred Mode	5-19
	5.4.7	Index Mode	5-19
	5.4.8	Index Deferred Mode	5-19
	5.4.9	Immediate Mode	5-19

CONTENTS (Cont.)

		Page
5.4.10	Absolute Mode	5-20
5.4.11	Relative Mode	5-20
5.4.12	Relative Deferred Mode	5-21
5.4.13	Table of Mode Forms and Codes	5-21
5.4.14	Branch Instruction Addressing	5-22
5.4.15	EMT and TRAP Addressing	5-22
5.5	ASSEMBLER DIRECTIVES	5-22
5.5.1	Listing Control Directives	5-22
5.5.1.1	.LIST and .NLIST	5-22
5.5.1.2	Page Headings	5-27
5.5.1.3	.TITLE	5-27
5.5.1.4	.SBTTL	5-27
5.5.1.5	.IDENT	5-28
5.5.1.6	Page Ejection (.PAGE Directive)	5-29
5.5.2	Functions: .ENABL and .DSABL Directives	5-29
5.5.3	Data Storage Directives	5-30
5.5.3.1	.BYTE	5-30
5.5.3.2	.WORD	5-31
5.5.3.3	ASCII Conversion of One or Two Characters	5-32
5.5.3.4	.ASCII	5-33
5.5.3.5	.ASCIZ	5-34
5.5.3.6	.RAD50	5-35
5.5.4	Radix Control	5-36
5.5.4.1	.RADIX	5-36
5.5.4.2	Temporary Radix Control: ^D, ^O, and ^B	5-37
5.5.5	Location Counter Control	5-37
5.5.5.1	.EVEN	5-38
5.5.5.2	.ODD	5-38
5.5.5.3	.BLKB and .BLKW	5-39
5.5.6	Numeric Control	5-39
5.5.6.1	.FLT2 and .FLT4	5-40
5.5.6.2	Temporary Numeric Control: ^F and ^C	5-41
5.5.7	Terminating Directives	5-42
5.5.7.1	.END	5-42
5.5.7.2	.EOT	5-42
5.5.8	Program Boundaries Directive: .LIMIT	5-42
5.5.9	Program Section Directives	5-42
5.5.10	Symbol Control: .GLOBL	5-44
5.5.11	Conditional Assembly Directives	5-46
5.5.11.1	Subconditionals	5-48
5.5.11.2	Immediate Conditionals	5-49
5.5.11.3	PAL-11R and PAL-11S Conditional Assembly Directives	5-50
5.6	MACRO DIRECTIVES WITH THE EXPAND UTILITY PROGRAM	5-50
5.6.1	Macro Definition	5-50
5.6.1.1	.MACRO	5-50
5.6.1.2	.ENDM	5-51
5.6.1.3	MACRO Definition Formatting	5-51
5.6.2	Macro Calls	5-52
5.6.3	Arguments to Macro Calls and Definitions	5-52

CONTENTS (Cont.)

			Page
	5.6.3.1	Special Characters	5-53
	5.6.3.2	Number of Arguments	5-53
	5.6.3.3	Concatenation	5-53
	5.6.4	Macro Libraries: .MCALL	5-54
	5.7	CALLING AND USING EXPAND	5-54
	5.8	CALLING AND USING ASEMBL	5-58
	5.8.1	Switches	5-59
	5.8.1.1	Listing Control Switches	5-59
	5.8.1.2	Function Switches	5-60
	5.8.1.3	Cross Reference Table Generation (CREF)	5-61
	5.9	ERROR MESSAGES	5-64
	5.9.1	EXPAND Error Messages	5-64
	5.9.2	ASEMBL/CREF Error Messages	5-65
\PTER	**6**	**LINKER**	**6-1**
	6.1	INTRODUCTION	6-1
	6.2	CALLING AND USING THE LINKER	6-1
	6.2.1	Command String	6-1
	6.2.2	Switches	6-2
	6.3	ABSOLUTE AND RELOCATABLE PROGRAM SECTIONS	6-2
	6.4	GLOBAL SYMBOLS	6-3
	6.5	INPUT AND OUTPUT	6-4
	6.5.1	Object Modules	6-4
	6.5.2	Load Module	6-4
	6.5.3	Load Map	6-5
	6.5.4	Library Files	6-5
	6.6	USING OVERLAYS	6-5
	6.7	USING LIBRARIES	6-11
	6.7.1	User Library Searches	6-11
	6.8	SWITCH DESCRIPTION	6-13
	6.8.1	Alphabetize Switch	6-13
	6.8.2	Bottom Address Switch	6-13
	6.8.3	Continue Switch	6-15
	6.8.4	Default FORTRAN Library Switch	6-15
	6.8.5	Include Switch	6-15
	6.8.6	LDA Format Switch	6-15
	6.8.7	Modify Stack Address	6-16
	6.8.8	Overlay Switch	6-16
	6.8.9	Symbol Table Switch	6-18
	6.8.10	Transfer Address Switch	6-18
	6.9	LINKER ERROR HANDLING AND MESSAGES	6-18
IAPTER	**7**	**LIBRARIAN**	**7-1**
	7.1	CALLING AND USING LIBR	7-1
	7.2	USER SWITCH COMMANDS AND FUNCTIONS	7-1
	7.2.1	Command Syntax	7-1
	7.2.2	LIBR Switch Commands	7-2
	7.2.2.1	Command Continuation Switch	7-2
	7.2.2.2	Creating a Library File	7-3

CONTENTS (Cont.)

Page

7.2.2.3 Inserting Modules Into a Library . . . 7-4
7.2.2.4 Replace Switch . . . 7-5
7.2.2.5 Delete Switch . . . 7-5
7.2.2.6 Delete Global Switch . . . 7-6
7.2.2.7 Update Switch . . . 7-7
7.2.2.8 Listing the Directory of a Library File . . . 7-8
7.2.2.9 Merging Library Files . . . 7-8
7.3 COMBINING LIBRARY SWITCH FUNCTIONS . . . 7-9
7.4 FORMAT OF LIBRARY FILES . . . 7-10
7.4.1 Library Header . . . 7-10
7.4.2 Entry Point Table (Library Directory) . . . 7-11
7.4.3 Object Modules . . . 7-11
7.4.4 Library End Trailer . . . 7-11
7.5 LIBR ERROR MESSAGES . . . 7-11

CHAPTER 8 ON-LINE DEBUGGING TECHNIQUE . . . 8-1
8.1 CALLING AND USING ODT . . . 8-1
8.1.1 Return to Monitor, CTRL C . . . 8-2
8.1.2 Terminate Search, CTRL U . . . 8-2
8.2 RELOCATION . . . 8-2
8.2.1 Relocatable Expressions . . . 8-3
8.3 COMMANDS AND FUNCTIONS . . . 8-4
8.3.1 Printout Formats . . . 8-4
8.3.2 Opening, Changing and Closing Locations . . . 8-4
8.3.3 Accessing General Registers 0-7 . . . 8-7
8.3.4 Accessing Internal Registers . . . 8-7
8.3.5 Radix 50 Mode, X . . . 8-7
8.3.6 Breakpoints . . . 8-9
8.3.7 Running the Program, r;G and r;P . . . 8-10
8.3.8 Single Instruction Mode . . . 8-11
8.3.9 Searches . . . 8-11
8.3.10 The Constant Register, r;C . . . 8-13
8.3.11 Memory Block Initialization, ;F and ;I . . . 8-13
8.3.12 Calculating Offsets, r;O . . . 8-13
8.3.13 Relocation Register Commands, r;nR, ;nR, ;R . . . 8-14
8.3.14 The Relocation Calculators, nR and n! . . . 8-15
8.3.15 ODT Priority Level, $P . . . 8-15
8.3.16 ASCII Input and Output, r;nA . . . 8-16
8.4 PROGRAMMING CONSIDERATIONS . . . 8-16
8.4.1 Functional Organization . . . 8-16
8.4.2 Breakpoints . . . 8-17
8.4.3 Searches . . . 8-19
8.4.4 Terminal Interrupt . . . 8-19
8.5 ODT ERROR DETECTION . . . 8-20

CHAPTER 9 PROGRAMMED REQUESTS . . . 9-1
9.1 FORMAT OF A PROGRAMMED REQUEST . . . 9-1
9.2 SYSTEM CONCEPTS . . . 9-3
9.2.1 Channel Number (chan) . . . 9-3

CONTENTS (Cont.)

		Page
9.2.2	Device Block (dblk)	9-4
9.2.3	EMT Argument Blocks	9-4
9.2.4	Important Memory Areas	9-4
9.2.4.1	Vector Addresses (0-37, 60-477)	9-4
9.2.4.2	Resident Monitor	9-5
9.2.4.3	System Communication Area	9-5
9.2.5	Swapping Algorithm	9-7
9.2.6	Offset Words	9-8
9.2.7	File Structure	9-9
9.2.8	Completion Routines	9-10
9.2.9	Using The System Macro Library	9-10
9.3	TYPES OF PROGRAMMED REQUESTS	9-10
9.3.1	System Macros	9-10
9.3.1.1	.DATE	9-15
9.3.1.2	.INTEN	9-15
9.3.1.3	.REGDEF	9-16
9.3.1.4	.SYNCH	9-16
9.4	PROGRAMMED REQUEST USAGE	9-18
9.4.1	.CDFN	9-18
9.4.2	.CHAIN	9-19
9.4.3	.CLOSE	9-21
9.4.4	.CSIGEN	9-21
9.4.5	.CSISPC	9-24
9.4.5.1	Passing Switch Information	9-26
9.4.6	.DELETE	9-29
9.4.7	.DSTATUS	9-30
9.4.8	.ENTER	9-32
9.4.9	.EXIT	9-34
9.4.10	.FETCH	9-34
9.4.11	.GTIM	9-35
9.4.12	.GTJB	9-36
9.4.13	.HERR/.SERR	9-37
9.4.14	.HRESET	9-39
9.4.15	.LOCK/.UNLOCK	9-39
9.4.16	.LOOKUP	9-41
9.4.17	.PRINT	9-42
9.4.18	.PROTECT	9-43
9.4.19	.PURGE	9-44
9.4.20	.QSET	9-45
9.4.21	.RCTRLO	9-46
9.4.22	.READ/.READC/.READW	9-46
9.4.23	.RELEAS	9-49
9.4.24	.RENAME	9-50
9.4.25	.REOPEN	9-52
9.4.26	.SAVESTATUS	9-52
9.4.27	.SETTOP	9-54
9.4.28	.SFPA	9-56
9.4.29	.SRESET	9-57
9.4.30	.TRPSET	9-58

CONTENTS (Cont.)

			Page
	9.4.31	.TTYIN/.TTINR	9-59
	9.4.32	.TTYOUT/.TTOUTR	9-60
	9.4.33	.WAIT	9-62
	9.4.34	.WRITE/.WRITC/.WRITW	9-64
APPENDIX	**A**	**COMMAND AND SWITCH SUMMARIES**	**A-1**
	A.1	KEYBOARD MONITOR	A-1
	A.1.1	Command Summary	A-1
	A.1.2	Special Function Keys	A-2
	A.2	EDITOR	A-3
	A.2.1	Command Arguments	A-3
	A.2.2	Input and Output Commands	A-3
	A.2.3	Pointer Relocation Commands	A-4
	A.2.4	Search Commands	A-4
	A.2.5	Text Modification Commands	A-4
	A.2.6	Utility Commands	A-5
	A.2.7	Key Commands	A-5
	A.3	PIP	A-6
	A.3.1	Switch Summary	A-6
	A.4	ASEMBL/CREF	A-7
	A.5	LINKER	A-7
	A.5.1	Switch Summary	A-7
	A.6	LIBRARIAN	A-8
	A.6.1	Switch Summary	A-8
	A.7	ODT	A-8
	A.7.1	Command Summary	A-8
	A.8	PROGRAMMED REQUESTS	A-10
	A.9	DUMP	A-10
	A.9.1	Switch Summary	A-10
	A.10	SRCCOM	A-11
	A.10.1	Switch Summary	A-11
	A.11	PATCH	A-11
	A.11.1	Command Summary	A-11
APPENDIX	**B**	**ASSEMBLER, INSTRUCTION, AND CHARACTER CODE SUMMARIES**	**B-1**
	B.1	ASCII CHARACTER SET	B-1
	B.2	RADIX-50 CHARACTER SET	B-4
	B.3	ASSEMBLER SPECIAL CHARACTERS	B-5
	B.4	ADDRESS MODE SYNTAX	B-5
	B.5	INSTRUCTIONS	B-6
	B.5.1	Double Operand Instructions	B-7
	B.5.2	Single Operand Instructions	B-8
	B.5.3	Rotate/Shift	B-8
	B.5.4	Operate Instructions	B-10
	B.5.5	Trap Instructions	B-11
	B.5.6	Branch Instructions	B-12
	B.5.7	Register Destination	B-12

CONTENTS (Cont.)

			Page
	B.5.8	Register-Offset	B-13
	B.5.9	Subroutine Return	B-13
	B.5.10	Source-Register	B-13
	B.5.11	Floating-Point Source Double Register	B-14
	B.5.12	Source-Double Register	B-14
	B.5.13	Double Register-Destination	B-15
	B.5.14	Number	B-16
	B.5.15	Priority	B-16
	B.5.16	Stack Oriented Floating Point (OP R)	B-16
	B.6	MACRO DIRECTIVES	B-16
	B.7	ASSEMBLER DIRECTIVES	B-17
	B.8	ASEMBL/CREF SWITCHES	B-19
	B.8.1	Listing Control Switches	B-19
	B.8.2	Function Control Switches	B-20
	B.8.3	CREF Switches	B-20
	B.9	OCTAL/DECIMAL CONVERSIONS	B-21
APPENDIX	**C**	**SYSTEM MACRO FILE**	**C-1**
APPENDIX	**D**	**PROGRAMMED REQUEST SUMMARY**	**D-1**
	D.1	PARAMETERS	D-1
	D.2	REQUEST SUMMARY	D-1
APPENDIX	**E**	**DUMP**	**E-1**
	E.1	CALLING AND USING DUMP	E-1
	E.1.1	DUMP Switches	E-1
	E.1.2	Examples	E-2
	E.2	DUMP ERROR MESSAGES	E-5
APPENDIX	**F**	**SOURCE COMPARE (SRCCOM)**	**F-1**
	F.1	CALLING AND USING SRCCOM	F-1
	F.1.1	Extensions	F-1
	F.1.2	Switches	F-1
	F.2	OUTPUT FORMAT	F-2
	F.3	SRCCOM ERROR MESSAGES	F-5
APPENDIX	**G**	**PATCH**	**G-1**
	G.1	CALLING AND USING PATCH	G-1
	G.2	PATCH COMMANDS	G-2
	G.2.1	Patch a New File	G-2
	G.2.2	Exit from PATCH	G-2
	G.2.3	Examine, Change Locations in the File	G-2
	G.2.4	Set Bottom Address	G-3
	G.2.5	Set Relocation Registers	G-3
	G.3	EXAMPLES USING PATCH	G-4
	G.4	PATCH ERROR MESSAGES	G-6

CONTENTS (Cont.)

Page

INDEX Index-1

FIGURES

			Page
FIGURE	2-1	HT-11 System Memory Map	2-4
	5-1	Assembly Source Listing Showing Local Symbol Blocks	5-12
	5-2	Example of ASEMBL Line Printer Listing (132-column Line Printer)	5-24
	5-3	Example of Page Heading From ASEMBL 80-column Line Printer (same format as Terminal Listing)	5-25
	5-4	Symbol Table	5-26
	5-5	Assembly Listing Table of Contents	5-28
	5-6	ASEMBL Source Code	5-62
	5-7	CREF Listing Output	5-63
	6-1	Linker Load Map	6-6
	6-2	Overlay Scheme	6-7
	6-3	Memory Diagram Showing BASIC Link with Overlay Regions	6-8
	6-4	Run-Time Overlay Handler	6-9
	6-5	Library Searches	6-12
	6-6	Alphabetized Load Map	6-14
	7-1	General Library File Format	7-10
	7-2	Library Header Format	7-10
	7-3	Format of Entry Point Table	7-11
	7-4	Library End Trailer	7-11

TABLES

			Page
TABLE	2-1	Prompting Characters	2-2
	2-2	Permanent Device Names	2-2
	2-3	File Name Extensions	2-3
	2-4	Special Function Keys	2-7
	2-5	SET Command Options	2-13
	3-1	EDIT Key Commands	3-2
	3-2	Command Arguments	3-4
	4-1	PIP Switches	4-2
	5-1	Legal Separating Characters	5-6
	6-1	Linker Switches	6-3
	7-1	LIBR Switches	7-2
	8-1	Forms of Relocatable Expressions (r)	8-3
	8-2	Internal Registers	8-8
	8-3	Radix 50 Terminators	8-8
	9-1	Summary of Programmed Requests	9-11
	9-2	Requests Requiring the USR	9-14
	E-1	DUMP Switches	E-1
	F-1	SRCCOM Switches	F-2
	G-1	PATCH Commands	G-2

PREFACE

This manual describes the use of the HT-11 Operating System. It assumes the reader is familiar with computer software fundamentals and has had some exposure to assembly-language programs.

The user who is unfamiliar with HT-11 should first read those chapters of interest (see "Chapter Summary" below) to become familiar with system conventions. Having gained familiarity with HT-11, the user can then reread the manual for specific information.

CHAPTER SUMMARY

Chapter 1 describes general system operations.

Chapter 2 introduces the user to system conventions and monitor/memory layout. It describes in detail the keyboard commands for controlling jobs and implementing user programs.

Chapters 3 through 8 describe the system utility programs EDIT, PIP, ASEMBL, EXPAND, LINK, LIBR, and ODT. These programs (a text editor, file transfer program, assembler, macro expander, linker, librarian, and debugging program) aid the user in creating text files and producing assembly-language programs.

Chapter 9, which describes programmed requests, is of particular interest to the experienced programmer. It describes call sequences that allow the user to access system monitor services from within assembly-language programs.

The appendices summarize the contents of the manual and describe additional system utility programs that can be used for extended system operations. These programs include SRCCOM (a source file comparison program); PATCH and PATCHO (patching programs); and DUMP (a file dump program).

DOCUMENTATION CONVENTIONS

Conventions used throughout this manual include the following:

1. Examples reflect actual computer output whenever possible. When necessary, computer output is underlined to differentiate from user responses.
2. A line feed (character or key) is represented in the text as ⟨LF⟩; a carriage return (character or key) is represented as ⟨CR⟩. Unless otherwise indicated, all commands and command strings are terminated by a carriage return.
3. Terminal and teleprinter are general terms used throughout all HT-11 documentation to represent any terminal device.
4. Several characters in system commands are produced by typing a combination of keys concurrently; for example, the CTRL key is held down while typing an O to produce a command which causes suppression of teleprinter output. Key combinations such as this are documented as CTRL O, CTRL C, SHIFT N, and so forth.

CHAPTER 1
HT-11 OVERVIEW

HT-11 is a single-user programming and operating system designed for the PDP-11 series of computers. It includes system programs or "tools" for program development using MACRO assembly language or the high-level languages BASIC and FORTRAN IV (when available). The HT-11 system programs are summarized in Section 1.2 and are discussed in detail in individual chapters and appendices of this manual.

1.1 PROGRAM DEVELOPMENT

Computer systems such as HT-11 are often used extensively for program development. The programmer makes use of the programming "tools" available on his system to develop programs which will perform functions specific to his needs. The number and type of "tools" available on any given system depend on a good many factors – the size of the system, its application and its cost, to name a few. Systems based on the PDP-11, however, provide several basic program development aids: these generally include an editor, assembler, linker, debugger, and often a librarian; a high level language (such as FORTRAN IV or BASIC) is also usually available.

An editor is used to create and modify textual material. Text may be the lines of code which make up a source program written in some programming language, or it may be data; text may be reports, or memos, or in fact may consist of any subject matter the user wishes. In this respect using an editor is analogous to using a typewriter – the user sits at a keyboard and types text. But the advantages of an editor far exceed those of a typewriter because once text has been created, it can be modified, relocated, replaced, merged, or deleted – all by means of simple editing commands. When the user is satisfied with his text, he can save it on a storage device where it is available for later reference.

If the editor is used for the purpose of writing a source program, development does not stop with the creation of this program. Since the computer cannot understand any language but machine language (which is a set of binary command codes), an intermediary program is necessary which will convert source code into the instructions the computer can execute. This is the function of an assembler.

The assembler accepts alphanumeric representations of PDP-11 coding instructions (i.e., mnemonics), interprets the code, and produces as output the appropriate object code. The user can direct the assembler to generate a listing of both the source code and binary output, as well as more specific listings which are helpful during the program debugging process. In addition, the assembler is capable of detecting certain common coding errors and of issuing appropriate warnings.

The output produced by the assembler is called object output because it is composed of object (or binary) code. On PDP-11 systems, the object output is called a module and contains the user's source program in the binary language which is acceptable to a PDP-11 computer.

Source programs may be complete and functional by themselves; however, some programs are written in such a way that they must be used in conjunction with other programs (or modules) in order to form a complete and logical flow of instructions. For this reason the object code produced by the assembler must be relocatable – that is, assignment of memory locations must be deferred until the code is combined with all other necessary object modules. It is the purpose of the linker to perform this relocation.

The linker combines and relocates separately assembled object programs. The output produced by the linker consists of a load module, which is the final linked program ready for execution. The user can, at his option, request a load map which displays all addresses assigned by the linker.

Very rarely is a program created which does not contain at least one unintentional error, either in the logic of the program or in its coding. Errors may be discovered by the programmer while he is editing his program, or the assembler may find errors during the assembly process and inform the programmer by means of error codes. The linker may also catch certain errors and issue appropriate messages. Often, however, it is not until execution that the user discovers his program is not working properly. Programming errors may be extremely difficult to find, and for this reason a debugging tool is usually available to aid the programmer in determining the cause of his error.

A debugging program allows the user to interactively control the execution of his program. With it, he can examine the contents of individual locations, search for specific bit patterns, set designated stopping points during execution, change the contents of locations, continue execution, and test the results, all without the need of re-editing and re-assembling.

When programs are successfully written and executed, they may be useful to other programmers. Often routines which are common to many programs (such as I/O routines) or sections of code which are used over and over again, are more useful if they are placed in a library where they can be retrieved by any interested user. A librarian provides such a service by allowing creation of a library file. Once created, the library can be expanded, updated, or listed.

High level languages simplify the programmer's work by providing an alternate means of writing a source program other than assembly-language mnemonics. Generally, high level languages are easy to learn – a single command may cause the computer to perform many machine language instructions. The user does not need to know about the mechanics of the computer to use a high level language. In addition, some high level languages (like BASIC) offer a special immediate mode which allows the user to solve equations and formulas as though he were using a calculator. Assembling and linking are done automatically so that the user can concentrate on solving the problem rather using the system.

These are a few of the programming tools offered by most computer systems. The next section summarizes specific programming aids available to the user of HT-11.

1.2 SYSTEM SOFTWARE COMPONENTS

The following is a brief summary of the HT-11 system programs:

1. The Text Editor (EDIT, described in Chapter 3) is used to create or modify source files for use as input to language processing programs such as the assembler or FORTRAN. EDIT contains powerful text manipulation commands for quick and easy editing of a text file.
2. EXPAND (Chapter 5) brings the capabilities of macros to the HT-11 system. (Macros are instructions in a source or command language which are equivalent to a specified sequence of machine instructions or commands.) ASEMBL accepts source files written in the assembly language and generates a relocatable object module to be processed by the Linker before loading and execution. Cross reference listings of assembled programs may be produced using CREF in conjunction with the assembler.
3. The Linker (LINK, described in Chapter 6) fixes (i.e., makes absolute) the values of relocatable symbols and converts the relocatable object modules of compiled or assembled programs and subroutines into a load module which can be loaded and executed by HT-11. LINK can automatically search library files for specified modules and entry points; it can produce a load map (which lists the assigned absolute addresses) and can provide automatic overlay capabilities to very large programs.
4. The Librarian (LIBR, see Chapter 7) allows the user to create and maintain his own library of functions and routines. These routines are stored on a random access device as library files, where they can be referenced by the Linker.
5. The Peripheral Interchange Program (PIP, see Chapter 4) is the HT-11 file maintenance and utility program. It is used to transfer files between all devices which are part of the HT-11 system, to rename or delete files, and to obtain directory listings.

6. SRCCOM (Source Compare, described in Appendix F) allows the user to perform a character-by-character comparison of two or more text files. Differences can be listed in an output file or directly on the line printer or terminal, thus providing a fast method of determining, for example, if all edits to a file have been correctly made.
7. The PATCH utility program (Appendix G) is used to make minor modifications to memory image files (output files produced by the Linker); it is used on files which do or do not have overlays.
8. ODT (On-line Debugging Technique, described in Chapter 8) aids in debugging assembled and linked object programs. It can print the contents of specified locations, execute all or part of the object program, single step through the object program, and search the object program for bit patterns.
9. DUMP (Appendix E) is used to print for examination all or any part of a file in octal words, octal bytes, ASCII and/or RAD50 characters (see Chapter 5).

CHAPTER 2
SYSTEM COMMUNICATION

The monitor is the hub of HT-11 system communications; it provides access to system and user programs, performs input and output functions, and enables control of the job.

The user communicates with the monitor through programmed requests and keyboard commands. The keyboard commands (described in Section 2.7) 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.

Programmed requests (described in detail in Chapter 9) are source program instructions which request monitor services. These instructions allow user assembly-language programs to utilize the available monitor features.

2.1 START PROCEDURE

The monitor can be loaded into memory from disk as follows:

1. Power up the system
2. When the terminal prints $, type DX and a carriage return (specifies floppy disk):

   $ DX ⟨CR⟩

3. The monitor then prints the identification message on the terminal:

   HT-11 H01A

After the message has printed, the system device should be WRITE ENABLED. The monitor is ready to accept keyboard commands.

2.2 SYSTEM CONVENTIONS

Special character commands, file naming procedures and other conventions that are standard for the HT-11 system are described in this section. The user should be familiar with these conventions before running the system.

2.2.1 Data Formats

The HT-11 system makes use of four types of data formats: ASCII, object, memory image, and load image.

Files in ASCII format conform to the American National Standard Code for Information Interchange, in which each character is represented by a 7-bit code. Files in ASCII format include program source files created by the Editor, listing and map files created by various system programs, and data files consisting of alphanumeric characters. A chart containing ASCII character codes appears in Appendix B.

Files in object format consist of data and PDP-11 machine-language code. Object files are those output by the assembler or FORTRAN compiler and are used as input to the Linker.

The Linker can output files in memory image format (.SAV) or load image format (.LDA).

A memory image file (.SAV) is a 'picture' of what memory will look like when a program is loaded. The file itself requires the same number of disk blocks as the corresponding number of 256-word memory blocks.

A load image (or .LDA) file may be produced for compatibility with the PDP-11 Paper Tape System and is loaded by the absolute binary loader. LDA files can be loaded and executed in stand-alone environments.

2.2.2 Prompting Characters

The following table summarizes the characters typed by HT-11 to indicate to the user that the system is awaiting user response:

Table 2-1 Prompting Characters

Character	Meaning
.	The Keyboard Monitor is waiting for a command (see Section 2.3.2).
*	The Command String Interpreter is waiting for a command string specification as explained in Sections 2.3.3 and 2.5.
↑	When the terminal is being used as an input file, the up-arrow prompts the user to enter information from the keyboard. Typing a CTRL Z marks the end-of-file.

2.2.3 Physical Device Names

Devices are referenced by means of a standard two-character device name. Table 2-2 lists each name and its related device. If no unit number is specified for devices which have more than one unit, unit 0 is assumed.

Table 2-2 Permanent Device Names

Permanent Name	I/O Device
DK:	The default logical storage device for all files. DK is initially the same as SY: (see below), but the assignment (as a logical device name) can be changed with the ASSIGN Command (Section 2.7.1.4).
DKn:	The specified unit of the same device type as DK.
DXn:	H27 Floppy disk (n is 0 or 1).
LP:	Line printer.
PP:	High-speed paper tape punch.
PR:	High-speed paper tape reader.
SY:	System device; the device and unit from which the system is bootstrapped. The assignment as a logical device name can be changed with the ASSIGN command (Section 2.7.1.4).
SYn:	The specified unit of the same device type as that from which the system was bootstrapped.
TT:	Terminal keyboard and printer.

In addition to the fixed names shown in Table 2-2, devices can be assigned logical names. A logical name takes precedence over a 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. Refer to Section 2.7.1.4 for instructions on assigning logical names to devices.

2.2.4 File Names and Extensions

Files are referenced symbolically by a name of one to six alphanumeric characters followed, optionally, by a period and an extension of up to three alphanumeric characters. (Excess characters in a filename may cause an error message.) The extension to a filename generally indicates the format of a file. It is a good practice to conform to the standard filename extensions for HT-11. If an extension is not specified for an input or output file, most system programs assign appropriate default extensions. Table 2-3 lists the standard extensions used in HT-11.

Table 2-3 File Name Extensions

Extension	Meaning
.BAD	Files with bad (unreadable) blocks; this extension can be assigned by the user whenever bad areas occur on a device. The .BAD extension makes the file permanent in that area, preventing other files from using it and consequently becoming unreadable.
.BAK	Editor backup file.
.BAS	BASIC source file (BASIC input).
.DAT	BASIC or FORTRAN data file.
.DIR	Directory listing file.
.DMP	DUMP output file.
.FOR	FORTRAN IV source file (FORTRAN input).
.LDA	Absolute binary file (optional Linker output).
.LLD	Library listing file.
.LST	Listing file (ASEMBL or FORTRAN output).
.MAC	EXPAND source file (EXPAND or SRCCOM input).
.MAP	Map file (Linker output).
.OBJ	Relocatable binary file (ASEMBL, FORTRAN IV output, Linker input, LIBR input and output).
.PAL	Output file of EXPAND (the MACRO expander program), input file of ASEMBL.
.SAV	Memory 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.

If a filename with a blank extension is to be used in a command line in which a default extension is assumed (by either the monitor or a system program), the user must insert a period after the filename to indicate that there is no extension. For example, to run the file TEST, type:

```
RUN TEST.
```

If the period after the filename is not given, the monitor assumes the .SAV extension and attempts to run a file named TEST.SAV.

2.2.5 Device Structures

HT-11 devices are categorized by the physical structure of the device and the way in which the device allows information to be processed.

All HT-11 devices are either random-access or sequential-access devices. *Random-access* devices allow blocks of data to be processed in a random order – that is, independent of the data's physical location on the device or its location relative to any other information. All disks fall into this category. Random-access devices are sometimes also called *block-replaceable* devices, because individual data blocks can be manipulated (rewritten) without affecting other data blocks on the device. *Sequential-access* devices require that data be processed sequentially; the order of processing data must be the same as the physical order of the data. HT-11 devices that are considered sequential devices are paper tape, line printer, and terminal.

File-structured devices are those devices that allow the storage of data under assigned filenames. HT-11 devices that are file-structured include all disks. *Nonfile-structured* devices, on the other hand, are those used to contain a single logical collection of data. These devices are used generally for reading and listing information, and include line printer, terminal, and paper tape devices.

2.3 MONITOR SOFTWARE COMPONENTS

The main HT-11 monitor software components are:

Resident Monitor (RMON)
Keyboard Monitor (KMON)
User Service Routine (USR) and Command String Interpreter (CSI)
Device Handlers

The reader may find Figure 2-1 helpful while reading the following descriptions.

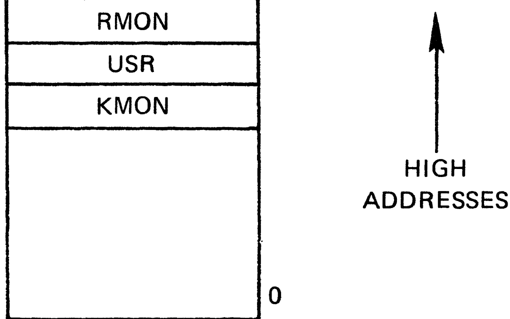


Figure 2-1 HT-11 System Memory Map

2.3.1 Resident Monitor (RMON)

The Resident Monitor is the only permanently memory-resident part of HT-11. The programmed requests for all services of HT-11 are handled by RMON. RMON also contains the terminal service, error processor, system device handler, EMT processor, and system tables.

2.3.2 Keyboard Monitor (KMON)

The Keyboard Monitor provides communication between the user at the keyboard and the HT-11 system. Monitor commands allow the user to assign logical names to devices, run programs, and load device handlers. A dot at the left margin of the terminal page indicates that the Keyboard Monitor is in memory and is waiting for a user command.

2.3.3 User Service Routine (USR)

The User Service Routine provides support for the HT-11 file structure. It loads device handlers, opens files for read or write operations, deletes and renames files, and creates new files. The Command String Interpreter (the use of which is described in Section 2.5) is part of the USR and can be accessed by any program to interpret device and file I/O information.

The USR is only required at the beginning and end of file operations. At other times its memory space may be reclaimed in a process called USR swapping.

2.3.4 Device Handlers

Device handlers for the HT-11 system are programs that perform the actual transfer of data to and from peripheral devices. All device handlers, except the system device handler, normally reside on the system device and are brought into memory only when they are needed.

2.4 GENERAL MEMORY LAYOUT

When the HT-11 System is first bootstrapped from the system device, memory is arranged as shown in Figure 2-1. The job is the HT-11 module KMON.

The LOAD and UNLOAD commands can modify the memory map. LOAD causes device handlers to be made resident between the USR and RMON until an UNLOAD command is performed.

HT-11 maintains a free memory list to manage memory. Thus, when a handler is unloaded, the space the handler occupied is returned to the free memory list.

2.5 ENTERING COMMAND INFORMATION

Once the monitor has been loaded and a system program started, the user must enter the appropriate command information before any operation can be performed.

In most cases, the Command String Interpreter immediately prints an asterisk at the left margin. The user must then type a command string in the general format:

```
OUTPUT=INPUT/SWITCH
```

(A few system programs – EDIT, PATCH, PATCHO – require that this command information be entered in a slightly different format. Complete instructions are provided in the appropriate chapter.)

In all cases, the format for OUTPUT is:

```
dev:filnam.ext[n],...dev:filnam.ext[n]
```

INPUT is:

```
dev:filnam.ext,...dev:filnam.ext
```

and SWITCH is:

```
/s:oval or /s!dval
```

where:

dev:	in each case is an optional two to three-character name from Table 2-2 or a user-assigned name (see Section 2.7.1.4) whose usage conforms to the NOTE below.

filnam.ext — in each case is the name of a file (consisting of one to six alphanumeric characters followed optionally by a dot and a zero to three-character extension). As many as three output and six input files may be allowed.

[n] — is an optional declaration of the number of blocks (n) desired for an output file. n is a decimal number (<65,535) enclosed in square brackets immediately following the output filnam.ext to which it applies.

/s:oval or /s!dval — is one or more optional switches whose functions vary according to the program in use (refer to the switch option table in the appropriate chapter). oval is either an octal number or one to three alphanumeric characters (the first of which must be alphabetic) which will be converted to radix-50. dval is a decimal value preceded by an exclamation point.

Throughout this manual, the /s:oval construction is used; however, the /s!dval format is always valid. Generally, these switches and their associated values, if any, should follow the device and filename to which they apply.

If the same switch is to be repeated several times with different values, as for example, /L:MEB/L:TTM/L:CND, the line may be abbreviated as /L:MEB:TTM:CND; octal, RAD50, and decimal values may be mixed.

= — if required, is a delimiter that separates the output and input fields. The < sign may be used in place of the = sign. The separator can be omitted entirely if there are no output files.

NOTE

As illustrated in the general format of a command line, the command line consists of an output list, a separator (= or <), and an input list. Omission of a device specification in either the input or output list is handled as follows:

DK: is assumed if the first file in a list has no explicit device. DK (or the device associated with the first file) is default until another device is indicated; that device then becomes default until a new one is used, and so on. If the following command is entered, for example, to ASEMBL:

```
*DX0:FIRST.OBJ,LP:=TASK.1,DX1:TASK.2,TASK.3
```

it is interpreted as though all devices had been indicated as follows:

```
*DX0:FIRST.OBJ,LP:=DK:TASK.1,DX1:TASK.2,
DX1:TASK.3
```

2.6 KEYBOARD COMMUNICATION (KMON)

Special function keys and keyboard commands allow the user to communicate with the HT-11 monitor and allocate system resources, manipulate memory images, and start programs.

The special functions of certain terminal keys used for communication with the Keyboard Monitor are explained in Table 2-4.

Table 2-4 Special Function Keys

Key	Function
CTRL C	CTRL C echoes as ^C on the terminal and is used to interrupt program execution and return control to the keyboard monitor. If the program to be interrupted is waiting for terminal input, typing one CTRL C is sufficient to interrupt execution; in all other cases, two CTRL Cs are necessary.
CTRL O	Echoes ^O on the terminal and causes suppression of teleprinter output while continuing program execution. Teleprinter output is re-enabled when one of the following occurs: 1. A second CTRL O is typed, 2. A return to the monitor occurs, or 3. The running program issues a Reset CTRL O (.RCTRLO) directive (see Chapter 9). (HT-11 system programs reset CTRL O each time a new command string is entered.)
CTRL Q	Does not echo. Resumes printing characters on the terminal from the point at which printing was previously stopped (via CTRL S).
CTRL S	Does not echo. Temporarily suspends output to the terminal until a CTRL Q is typed.
CTRL U	Deletes the current input line and echoes as ^U followed by a carriage return at the terminal. (The current line is defined to be all characters back to, but not including, the most recent line feed, CTRL C or CTRL Z.)
CTRL Z	Echoes ^Z on the terminal and terminates input when used with the terminal device handler (TT). The CTRL Z itself does not appear in the input buffer. If TT is not being used, CTRL Z has no special meaning.
RUBOUT	Deletes the last character from the current line and echoes a backslash plus the character deleted. Each succeeding RUBOUT deletes and echoes another character. An enclosing backslash is printed when a key other than RUBOUT is typed. This erasure is done right to left up to the beginning of the current line.

CTRL commands are entered by holding the CTRL key down while typing the appropriate letter.

2.6.1 Type-Ahead

The monitor has a type-ahead feature which allows terminal input to be entered while a program is executing. For example:

```
.R PIP
*DX1:TAPE=PR:/A
DX1:/L
*13-FEB-78
TAPE              78 13-FEB-78
 422 FREE BLOCKS
```

While the first command line is executing, the second line (DX1:/L) is entered by the user. This terminal input is stored in a buffer and used when the first operation has completed.

If a single CTRL C is typed while in this mode, it is put into the buffer. The program currently executing exits when a terminal input request needs to be satisfied. A double CTRL C returns control to the monitor immediately.

If type-ahead input exceeds 80 characters, the terminal bell rings and no characters are accepted until part of the type-ahead buffer is used by a program or characters are deleted. No input is lost. Type-ahead is particularly useful in specifying multiple command lines to system programs, as shown in the preceding example. If a job is terminated by typing two CTRL Cs, any unprocessed type-ahead is discarded.

NOTE

If type-ahead is used in conjunction with EDIT or BASIC, there is no terminal echo of the characters but they are stored in the buffer until a new command is needed. The characters are echoed only when actually used by the program.

2.7 KEYBOARD COMMANDS

Keyboard commands allow the user to communicate with the monitor. Keyboard commands can be abbreviated; optional characters in a command are delimited (in this section only) by braces. Keyboard commands require at least one space between the command and the first argument. All command lines are terminated by a carriage return.

2.7.1 Commands to Allocate System Resources

2.7.1.1 DATE Command – The DATE command enters the indicated date to the system. This date is then assigned to newly created files, new device directory entries (which may be listed with PIP), and listing output until a new DATE command is issued.

The form of the command is:

DAT{E} {dd-mmm-yy}

where dd-mmm-yy is the day, month and year to be entered. dd is a decimal number in the range 1–31; mmm is the first three characters of the name of the month, and yy is a decimal number in the range 73–99. If no argument is given, the current date is printed.

Examples:

.DATE 21-FEB-78	Enter the date 21-FEB-78 as the current system date.
.DAT 21-FEB-78	Print the current date.

If the date is entered in an incorrect format, the ?DAT? error message is printed.

TIME

2.7.1.2 TIME Command – The TIME command allows the user to find out the current time of day kept by HT-11 or to enter a new time of day. If the time is entered in an incorrect format, the ?TIM? message is printed.

The form of the command is:

TIM{E} {hh:mm:ss}

where hh:mm:ss represents the hour, minute, and second. Time is represented as hours, minutes, and seconds past midnight in 24-hour format (e.g., 1:25:00 P.M. is entered as 13:25:00). If any of the arguments are omitted, 0 is assumed. If no argument is given, the current time of day is output.

Examples:

.TIM 8:15:23	Sets the time of day to 8 hours, 15 minutes and 23 seconds.
.TIM 08:25:27	Approximately 10 minutes later, the TIME command outputs this time.
.TIM 18:5	Sets the time of day to 18:05:00.

INITIALIZE

2.7.1.3 INITIALIZE Command – The INITIALIZE command is used to reset several system tables and do a general "clean-up" of the area. In particular, this command makes non-resident those handlers which were not loaded (via LOAD), purges the I/O channels, disables CTRL O, performs a hard reset, clears locations 40–53, and resets the KMON stack pointer.

The form of the command is:

IN{ITIALIZE}

The INITIALIZE command can be used prior to running a user program, or when the accumulated results of previously issued GET commands (see Section 2.7.2.1) are to be discarded.

Example:

.IN .R PROG	Initializes system

ASSIGN

2.7.1.4 ASSIGN Command – The ASSIGN command assigns a user-defined (logical) name as an alternate name for a physical device. This is especially useful when a program refers to a device which is not available on a certain system. Using the ASSIGN command, I/O can be redirected to a device which is available. Only one logical name can be assigned per ASSIGN command, but several ASSIGN commands (14 maximum) can be used to assign different names to the same device. This command is also used to assign FORTRAN logical units to device names.

The form of the command is:

ASS {IGN} {{dev} :udev }

where:

dev	is any standard HT-11 (physical) device name (refer to Table 2-2) with the exception of DK and SY.

udev	is a 1–3 character alphanumeric (logical) name to be used in a program to represent dev (if more than three characters are given, only the first three are actually used). DK and SY may be used as logical device names.
:	is a delimiter character (can be a colon, equal sign, and, if separating physical and logical devices, space).

The placement of the delimiter is very important in the ASSIGN command; it must be placed exactly as shown in the following examples:

ASSIGN DX1 INP	Physical device DX1 is assigned the logical device name INP. Whenever a reference to INP: is encountered, device DX1: is used.
.ASSIGN DX1:DK	Physical device name DX1 is assigned the default device name DK. Whenever DK is referenced or defaulted to, DX1 is used. (Note that the initial assignment of DK is thus changed.)
.ASSIGN LP=9	FORTRAN logical unit 9 becomes the physical device name LP. All references to unit 9 use the line printer for output.

Assignment of logical names to logical names is not allowed.

If only a logical device name is indicated in the command line, that particular assignment (only) is removed. Thus:

.ASSIGN :9	Deassigns the logical name 9 from its physical device (LP, in the case above).
.ASSIGN =DK	Removes assignment of logical name DK from its physical device (DX1, in the case above).

If neither a physical device name nor a logical device name is indicated, all assignments to all devices are removed.

.ASSIGN	All previous logical device assignments are removed.

CLOSE

2.7.1.5 CLOSE Command – The CLOSE command causes all currently open output files to become permanent files. If a tentative open file is not made permanent, it will be deleted. The CLOSE command is most often used after CTRL C has been typed to abort a job and to preserve any new files that job had open prior to the CTRL C.

The form of the command is:

CLO{SE}

The CLOSE command makes temporary directory entries permanent.

Example:

```
.R EDIT
*EWTEXT$$
*IABCD$$
*^C

.CLOSE
```

The Editor has a temporary file open (TEXT), which is preserved by .CLOSE.

LOAD

2.7.1.6 LOAD Command – The LOAD command is used to make a device handler resident in memory. Time to fetch the handler is saved when a handler is resident, although memory area for the handler must be allocated.

The form of the command is:

LOA{D} dev

where:

dev	represents any legal HT-11 device name.

LOAD is valid for use with user-assigned names. For example:

```
.ASSIGN DX1:XY

.LOA XY
```

UNLOAD

2.7.1.7 UNLOAD Command – The UNLOAD command is used to make handlers that were previously LOADed non-resident, freeing the memory they were using.

The form of the command is:

UNL{OAD} dev {,dev,...}

where:

dev	represents any legal HT-11 device name.

Example:

.UNLOAD LP,PP	The lineprinter and paper tape punch handlers are released and the area which they used is freed.

SET

2.7.1.8 SET Command – The SET command is used to change device handler characteristics and certain system configuration parameters.

The form of the command is:

SET dev: {NO} option {=value} { , {NO} option {=value} , . . . }

where:

dev:	represents any legal HT-11 physical device name (or USR).
{NO}option	is the feature or characteristic to be altered.
=value	is a decimal number required in some cases.

A space may be used in place of or in addition to the colon, equal sign, or comma. Note that the device indicated (with the exception of USR) must be a physical device name and is not affected by logical device name assignments which may be active. The name of the characteristic or feature to be altered must be legal for the indicated device (see Table 2-5) and may not be abbreviated.

The SET command locates the file SY:dev.SYS and permanently modifies it. No modification is done if the command entered is not completely valid. If a handler has already been loaded when a SET command is issued for it, the modifications will not take effect until the handler is unloaded and a fresh copy called in from the system device.

Table 2-5 lists the system characteristics and parameters which may be altered (those modes designated as "normal" are the modes as set in the distribution copies of the drivers).

The following variant of the SET command is used to prevent the job from ever placing the USR in a swapping state (note that USR replaces a device specification in the command line):

SET USR {NO} SWAP

This is useful because programs requiring the USR run much faster in a NOSWAP environment, provided they can spare the USR's 2K memory requirement; for some programs, this environment is necessary just so they can run.

When the monitor is bootstrapped, it is in the SWAP condition, i.e., the job may place the USR in a swapping state via a SETTOP.

2.7.2 Commands to Manipulate Memory Images

GET

2.7.2.1 GET Command – The GET command loads the specified memory image file (not ASCII or object) into memory from the indicated device.

The form of the GET command is:

GE{T} dev:filnam.ext

where:

dev:	represents any legal HT-11 device name. If a device is not specified, DK: is assumed.
filnam.ext	represents a valid HT-11 filename and extension. If an extension is not specified, the extension .SAV is assumed.

Table 2-5 SET Command Options

Device	Option	Alteration
LP	CR	Allows carriage returns to be sent to the printer. The CR option should be set for any FORTRAN program using formatted I/O, to allow the overstriking capability for any line printer. This is the normal mode.
LP	NOCR	Inhibits sending carriage returns to the line printer. Some line printer controllers cause a line feed to perform the functions of a carriage return, so using this option can produce a significant increase in printing speed.
LP	CTRL	Causes all characters, including nonprinting control characters, to be passed to the line printer. This is the normal mode.
LP	NOCTRL	Ignores nonprinting control characters.
LP	FORM0	Causes a form feed to be issued before a request to print block zero. This is the normal mode.
LP	NOFORM0	Turns off FORM0 mode.
LP	HANG	Causes the handler to wait for user correction if the line printer is not ready or becomes not ready during printing. This is the normal mode. New users should note that when expecting output from the line printer and it appears as though the system is not responding or is in an idle state, the line printer should be checked to see if it is on and ready to print.
LP	NOHANG	Generates an immediate error if the line printer is not ready.
LP	LC	Allows lower-case characters to be sent to the printer. This option should be used if the printer has a lower-case character set. This is the normal mode.
LP	NOLC	Causes lower-case characters to be translated to upper case before printing.
LP	WIDTH=n	Sets the line printer width to n, where n is a number between 30 and 255. Any characters printer past column n are ignored. The NO modifier is not permitted.
TTY	SCOPE	Causes the monitor to echo RUBOUTs as backspace-space-backspace.
TTY	NOSCOPE	Causes the monitor to echo RUBOUTs as backslash followed by the character deleted. This is the normal mode.

The GET command is typically used to load a program into memory for modification and/or debugging. The GET command can also be used in conjunction with the Base, Examine, Deposit, and START commands to test patches, and can be used with SAVE to make patches permanent. Multiple GETs can be used to combine programs. Thus:

.GET ODT.SAV	Loads ODT into memory
.GET PROG	Loads PROG.SAV into memory with ODT
.START (ODT's starting address)	Starts execution with ODT (see Chapter 8).

The GET command cannot be used to load overlay segments of programs; it may only be used to load the root segment (that part which will not be overlaid; refer to Chapter 6, Linker).

Multiple GETs can be used to build a memory image of several programs. If identical locations are required by any of the programs, the later programs overlay the previous ones. The GET command loads memory in multiples of 256-word blocks.

Examples:

GET DX1:FILE1.SAV	Loads the file FILE1.SAV into memory from DX1.
GET NAME1	Loads the file NAME1.SAV from device DK.

BASE

2.7.2.2 Base Command – The B command sets a relocation base. This relocation base is added to the address specified in subsequent Examine or Deposit commands to obtain the address of the location to be referenced. This command is useful when referencing linked modules with the Examine and Deposit commands. The base address can be set to the address where the module of interest is loaded. The form of the command is:

B {location}

where:

location — represents an octal address used as a base address for subsequent Examine and Deposit commands.

NOTE

A space must follow the B command even if an address is not specified (the B<space> command is equivalent to B 0).

Any non-octal digit terminates an address. If location is odd, it is rounded down by one to an even address.

The base is cleared whenever user-program execution is initiated.

Examples:

.BΔ	Sets base to 0 (Δ represents space).
.B 200	Sets base to 200.
.B 201	Sets base to 200.

EXAMINE

2.7.2.3 Examine Command – The E command prints the contents of the specified location(s) in octal on the terminal. The form of the Examine command is:

E location m{–location n}

where:

location — represents an octal address which is added to the relocation base value (the value set by the B Command) to get the actual address examined. Any non-octal digit terminates an address. An odd address is truncated to become an even address.

If more than one location is specified (location m-location n), the contents of location m through location n inclusive are printed. The second location specified (location n) must not be less than the first location specified, otherwise an error message is printed. If no location is specified, the contents of location 0 are printed. Examination of locations outside the job's area is illegal.

Examples:

```
.E 1000
127401
```

Prints contents of location 1000 (added to the base value if other than 0).

```
.E 1001-1012
127401 007624 127400 000000 000000 000000
```

Prints the contents of locations 1000 (plus the base value if other than 0) through 1013.

DEPOSIT

2.7.2.4 Deposit Command – The Deposit command deposits the specified value(s) starting at the location given.

The form of the command is:

D location=value1{,value2,...valuen}

where:

location — represents 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 — represents the new contents of the location. 0 is assumed if a value is not indicated.

If multiple values are specified (value1,...,valuen), they are deposited beginning at the location specified. An odd address is truncated by one to an even address. All values are stored as word quantities.

Any character that is not an octal digit may be used to separate the locations and values in a DEPOSIT command. However, two (or more) non-octal separators cause 0's to be deposited at the location specified (and those following). For example:

.D 56,,,	Deposits 0's in locations 56, 60, and 62.

The user should be aware of situations like the above, which cause system failure since the terminal vector (location 60) is zeroed.

An error results when the address specified references a location outside the job's area.

Examples:

.D 1000=3705	Deposits 3705 into location 1000
.B 1000	Sets relocation base to 1000
.D 1500=2503	Puts 2503 into location 2500
.B 0	Resets base to 0

SAVE

2.7.2.5 SAVE Command – The SAVE command writes specified user memory areas to a named file and device in save image format. Memory is written from location 0 to the highest memory address specified by the parameter list or to the program high limit (contents of location 50 in the system communication area).

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

The form of the command is:

SAV{E} dev:filnam.ext {parameters}

where:

dev:	represents one of the standard HT-11 block-replaceable device names. If no device is specified, DK is assumed.
file.ext	represents the name to be assigned to the file being saved. If the file name is omitted, an error message is output. If no extension is specified, the extension .SAV is used.
parameters	represent memory locations to be saved. HT-11 transfers memory in 256-word blocks beginning on boundaries that are multiples of 256 (decimal). If the locations specified make a block of less than 256 words, enough additional locations are transferred to make a 256-word block.

Parameters can be specified in the following format:

area1,area2-arean

where:

area1 area2-arean	represent an octal number (or numbers separated by dashes). If more than one number is specified, the second number must be greater than the first.

The SAVE command saves the job parameters stored in the following locations. If the user wishes to alter these parameters, the DEPOSIT command can be used:

Area	Location
Start address	40
Initial stack	42
JSW	44
USR address	46
High address	50

If these values are changed, it is the user's responsibility to reset them to their original values. See Chapter 9 for more information concerning these addresses.

Examples:

```
.SAVE 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.

```
.SAVE DX1:NAM.NEW 10000
```

Saves locations 10000 through 10777 on DX1: with the name NAM.NEW.

```
.D 44:20000
.SAV SY:PRAM 1000-5777
```

Sets the reenter bit in the JSW and saves locations 1000 through 5777.

2.7.3 Commands to Start a Program

RUN

2.7.3.1 RUN Command – The RUN command loads the specified memory image file into memory and starts execution at the start address specified in location 40.

The form of the command is:

RU{N} dev:filnam.ext

where:

dev:	is any standard device name specifying a block-replaceable device. If dev: is not specified, the device is assumed to be DK.
filnam.ext	is the file to be executed. If an extension is not specified, the extension .SAV is assumed.

The RUN command is equivalent to a GET command followed by a START command (with no address specified).

NOTE

If a file containing overlays is to be RUN from a device other than the system device, the handler for that device must be loaded (see Section 2.7.1.6) before the RUN command is issued.

Examples:

.RUN DX1:SRCH.SAV	Loads and executes the file SRCH.SAV from DX1.
.RUN PROG	Loads PROG.SAV from DK and executes the program.
.GET PROG1 .RUN PROG2	Loads PROG1.SAV from device DK without executing it. Then combines PROG1 and PROG2.SAV in memory and begins execution at the starting address for PROG2.

2.7.3.2 R Command – This command is similar to the RUN command except that the file specified must be on the system device (SY:).

The form of the command is:

R filnam.ext

No device may be specified. If an extension is not given, the extension .SAV is assumed.

Examples:

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

2.7.3.3 START Command – The START command begins execution of the program currently in memory (i.e., loaded via the GET command) at the specified address. START does not clear or reset memory areas.

The form of the command is:

ST{ART} {address}

where:

address — is an octal number representing any 16-bit address. If the address is omitted, or if 0 is given, the starting address in location 40 will be used.

If the address given does not exist or is not an even address, a trap to location 4 occurs. In this case a monitor error message appears. If no address is given, the program's start address from location 40 is used.

Examples:

.GET FILE.1 .START 1000	Loads FILE.1 into memory and starts execution at location 1000.
.GET FILEA .GET FILEB .ST	Loads FILEA.SAV, then combines FILEA.SAV with FILEB.SAV and starts execution at FILEB's start address.

2.7.3.4 REENTER Command – The REENTER command starts the program at its reentry address (the start address minus two). REENTER does not clear or reset any memory areas and is generally used to avoid reloading the same program for repetitive execution. It can be used to return to a program whose execution was stopped with a CTRL C.

The form of the command is:

RE{ENTER}

If the reenter bit (bit 13) in the Job Status Word (location 44) is not set, the REENTER command is illegal.

For most system programs, the REENTER command restarts the program at the command level.

If desired, the reentry point in a user program can branch to a routine which initializes the tables and stack, fetches device handlers etc., and then continues normal operation.

Example:

```
.R PIP
*/F
MONITR.SYS
[directory prints]
.
. (^C  typed)
.  ^C
 REENTER
*
```

CTRL C interrupts the PIP directory listing and transfers control to the monitor level. REENTER returns control to PIP;

2.8 MONITOR ERROR MESSAGES

The following error messages indicate fatal conditions that can occur during system boot:

Message	Meaning
?B-I/O ERROR	An I/O error occurred during system boot.
?B-NO BOOT ON VOLUME	No bootstrap has been written on volume.
?B-NO MONITR.SYS	No monitor exists on volume being booted.
?B-NOT ENOUGH MEMORY	There is not enough memory for the system being booted.

The following error messages are output by the Keyboard Monitor.

Message	Meaning
?ADDR?	Address out of range in E or D command.
?DAT?	The DATE command argument was illegal, or no argument was given and the date has not yet been set.
?ER RD OVLY?	An I/O error occurred while reading a KMON overlay to process the current command. This is a serious error, indicating that the system file MONITR.SYS is unreadable.
?FIL NOT FND?	File specified in R, RUN, or GET command not found.
?FILE?	No file named where one is expected.
?ILL CMD?	Illegal Keyboard Monitor command or command line too long.
?ILL DEV?	Illegal or nonexistent device.
?OVR MEM?	Attempt to GET or RUN a file that is too big.
?PARAMS?	Bad parameters were typed to the SAVE command.
?SV FIL I/O ER?	I/O error on .SAV file in SAVE (output) or R, RUN, or GET (input) command. Possible errors include end-of-file, hard error, and channel not open.
?SY I/O ER?	I/O error on system device (usually reading or writing swap area).
?TIM?	The TIME command argument was illegal.

The following messages are output by the HT-11 Resident 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 R or RUN command.

The format for fatal monitor error messages is:

?M-text PC   where PC is the address+2 of the location where the error occurred.

Note that ?M errors can be inhibited in certain cases by the use of the .SERR macro; see Chapter 9 for details.

Message	Meaning
?M-BAD FETCH	Either an error occurred while reading in a device handler from SY, or the address at which the handler was to be loaded was illegal.
?M-DIR IO ERR	An error occurred doing I/O in the directory of a device (e.g., .ENTER on a write-locked device).

Message	Meaning
?M-DIR OVFLO	No more directory segments were available for expansion (occurs during file creation (.ENTER)).
?M-FP TRAP	A floating-point exception trap occurred, and the user program had no .SFPA exception routine active (see Chapter 9).
?M-ILL CHAN	A channel number was specified which was too large.
?M-ILL EMT	An EMT was executed which did not exist; i.e., the function code was out of bounds.
?M-ILL USR	The USR was called from a completion routine. This error does not have a soft return (i.e., .SERR will not inhibit this message; see Chapter 9).
?M-NO DEV	A READ/WRITE operation was tried but no device handler was in memory for it.
?M-OVLY ERR	A user program with overlays failed to successfully read an overlay.
?M-SWAP ERR	A hard I/O error occurred while the system was attempting to write a user program to the system swap blocks. This is usually caused by a write-locked system device. This may cause the system to halt.
?M-SYS ERR	An I/O error occurred while trying to read KMON/USR into memory, indicating that the monitor file is situated on the system device in an area that has developed one or more bad blocks. The monitor prints the message and loops trying to read KMON. The message is a warning that the system device is bad. If, after several seconds, it is apparent that attempts to read KMON are failing, halt the processor. It may be impossible to boot the volume because of the bad area in the monitor file. Use another system device to verify the bad blocks and follow the recovery procedures described in section 4.2.11.1 of Chapter 4.
?M-TRAP TO 4 ?M-TRAP TO 10	The job has referenced illegal memory or device registers, an illegal instruction was used, stack overflow occurred, a word instruction was executed with an odd address, or a hardware problem caused bus time-out traps through location 4.

If CSI errors occur and input was from the terminal, an error message is printed on the terminal.

Message	Meaning
?DEV FUL?	Output file will not fit.
?FIL NOT FND?	Input file was not found.
?ILL CMD?	Syntax error.
?ILL DEV?	Device specified does not exist.

2.8.1 Monitor HALTS

The monitor will halt only if I/O errors occur during swap operations to the system device. If it halts, look for a write-locked system device.

The monitor halt can be detected by its address, which is high in memory, above the resident base address (contents of location 54).

When a monitor halt occurs, the system must be rebooted.

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 a storage device, makes specified changes and writes ASCII files to a storage device or lists them on the line printer or terminal.

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 buffers where the page becomes available for editing. Editing commands are then used to:

Locate text to be changed,
Execute and verify the changes,
Output a page of text to the output file,
List an edited page on the line printer or terminal.

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.

Type CTRL C to halt the Editor at any time and return control to the monitor. To restart the Editor type .R EDIT or the .REENTER command in response to the monitor's dot. The contents of the buffers are lost when the Editor is restarted.

3.2 MODES OF OPERATION

Under normal usage, the Editor operates in one of two different modes: Command Mode or Text Mode. In Command Mode all input typed on the keyboard is interpreted as commands instructing the Editor to perform some operation. In Text Mode all typed input is interpreted as text to replace, be inserted into, or be appended to the contents of the Text Buffer.

Immediately after being loaded into memory and started, the Editor is in Command Mode. An asterisk is printed at the left margin of the console terminal page indicating that the Editor is waiting for the user to type a command. All commands are terminated by pressing the ESCape key twice in succession. Execution of commands proceeds from left to right. Should an error be encountered during execution of a command string, the Editor prints an error message followed by an asterisk at the beginning of a new line indicating that it is still in Command Mode and awaiting a legal command. The command in error (and any succeeding commands) is not executed and must be corrected and retyped.

Some terminals do not have and ESCape key. On these terminals, the ALTMODE key should be used.

Text mode is entered whenever the user types a command which must be followed by a text string. These commands insert, replace, or otherwise manipulate text; after such a command has been typed, all succeeding characters are considered part of the text string until an ESCape is typed. The ESCape terminates the text string and causes the Editor to reenter Command Mode, at which point all characters are considered commands again.

3.3 SPECIAL KEY COMMANDS

The EDIT key commands are listed in Table 3-1. Control commands are typed by holding down the CTRL key while typing the appropriate character.

Table 3-1 EDIT Key Commands

Key	Explanation
ESCape	Echoes $, A single ESCape terminates a text string. A double ESCape executes the command string. For example, *GMOV A, B$ – 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.
CTRL O	Echoes ↑O and a carriage return. Inhibits printing on the terminal until completion of the current command string. Typing a second CTRL O resumes output.
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 character from the current line; echoes a backslash followed by the character deleted. Each succeeding RUBOUT typed by the user deletes and echoes another character. An enclosing backslash is printed when a key other than RUBOUT is typed. This erasure is done right to left up to the last carriage return/line feed combination. RUBOUT may be used in both Command and Text Modes.
TAB	Spaces to the next tab stop. Tab stops are positioned every eight spaces on the terminal; typing the TAB key causes the carriage to advance to the next tab position.
CTRL X	Echoes ↑X and a carriage return. CTRL X causes the Editor to ignore the entire command string currently being entered. The Editor prints a <CR><LF> and an asterisk to indicate that the user may enter another command. For example: *IABCD EFGH^X * A CTRL U would only cause deletion of EFGH; CTRL X erases the entire command.

3.4 COMMAND STRUCTURE

EDIT commands fall into six general categories:

Category	Commands	Section
Input/Output	Edit Backup	3.6.1.3
	Edit Read	3.6.1.1
	Edit Write	3.6.1.2
	End File	3.6.1.9
	Exit	3.6.1.10
	List	3.6.1.7
	Next	3.6.1.6
	Read	3.6.1.4
	Verify	3.6.1.8
	Write	3.6.1.5
Pointer location	Advance	3.6.2.3
	Beginning	3.6.2.1
	Jump	3.6.2.2
Search	Find	3.6.3.2
	Get	3.6.3.1
	Position	3.6.3.3
Text modification	Change	3.6.4.4
	Delete	3.6.4.2
	Exchange	3.6.4.5
	Insert	3.6.4.1
	Kill	3.6.4.3
Utility	Edit Lower	3.6.5.6
	Edit Upper	3.6.5.6
	Edit Version	3.6.5.5
	Execute Macro	3.6.5.4
	Macro	3.6.5.3
	Save	3.6.5.1
	Unsave	3.6.5.2

The general format for the first five categories of EDIT commands is:

nCtext$

or

nC$

where n represents one of the legal arguments listed in Table 3-2, C is a one- or two-letter command, and text is a string of successive ASCII characters.

As a rule, commands are separated from one another by a single ESCape, however, if the command requires no text, the separating is not necessary. Commands are terminated by a single ESCape; typing a second ESCape begins execution.

3.4.1 Arguments

An argument is positioned before a command letter and is used either to specify the particular portion of text to be affected by the command or to indicate the number of times the command should be performed. With some commands, this specification is implicit and no arguments are needed; other editing commands require an argument. Table 3-2 lists the formats of arguments which are used by commands of this second type.

Table 3-2 Command Arguments

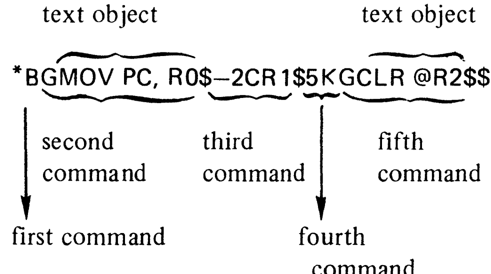
Format	Meaning
n	n stands for any integer in the range –16383 to +16383 and may, except where noted, be preceded by a + or –. If no sign precedes n, it is assumed to be a positive number. Whenever an argument is acceptable in a command, its absence implies an argument of 1 (or –1 if only the – is present).
0	0 refers to the beginning of the current line.
/	/ refers to the end of text in the current Text Buffer.
=	= is used with the J, D and C commands only and represents –n, where n is equal to the length of the last text argument used.

The roles of all arguments are explained more specifically in following sections.

3.4.2 Command Strings

All EDIT command strings are terminated by two successive ESCape characters. Spaces, carriage returns and line feeds within a command string may be used freely to increase command readability but are ignored unless they appear in a text string. Commands used to insert text can contain text strings that are several lines long. Each line is terminated with a <CR> <LF> and the entire command is terminated with a double ESCape.

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



Execution of a command string begins when the double ESCape is typed and proceeds from left to right. Except when they are part of a text string, spaces, carriage return, line feed, and single ESCape are ignored. For example:

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

may be typed as:

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

with equivalent execution.

3.4.3 The Current Location Pointer

Most EDIT commands function with respect to a movable reference pointer which is normally located between the most recent character operated upon and the next character in the buffer. At any given time during the editing procedure, this pointer can be thought of as representing the current position of the Editor in the text. Most commands use this pointer as an implied argument. Commands are available for moving the pointer anywhere in the text, thereby redefining the current location and allowing greater facility in the use of other commands.

3.4.4 Character- and Line-Oriented Command Properties

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.

When using 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 a forward direction (in relation to the pointer), negative arguments the number of characters in a backward direction. Carriage return and line feed characters are treated the same as any other character. For example, assume the pointer is positioned as indicated in the following text (↑ represents the current position of the pointer):

```
MOV     #VECT,R2<CR><LF>↑
CLR     @R2<CR><LF>
```

The EDIT command −2J backs the pointer by two characters.

```
MOV     #VECT,R2<CR><LF>
                ↑
CLR     @R2<CR><LF>
```

The command 10J advances the pointer forward by ten characters and places it between the CR and LF characters at the end of 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 command is used. The J (Jump) command is explained in Section 3.6.2.2.

```
MOV     #VECT,R2<CR><LF>
            ↑
CLR     @R2<CR><LF>
```

When using line-oriented commands, a numeric argument represents the number of lines involved in the operation. The Editor recognizes a line of text as a unit when it detects a <CR> <LF> combination in the text. When the user types a carriage return, the Editor automatically inserts a line feed. Positive arguments represent the number of lines forward (in relation to the pointer); this is accomplished by counting carriage return/line feed combinations beginning at the pointer. So, 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 the 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 moves the pointer over 2 <CR> <LF> 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 a backward direction (in relation to the pointer). Consequently, if the pointer is at the beginning of the line, a line argument of –1 means "the previous line" (moving backward past the first <CR> <LF> and up to but not including the second <CR> <LF>; 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 backs the pointer by 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 backs it by only 1 line.

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

3.4.5 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:

```
C1$C2$n<C3$C4$>C5$$
```

where C1, C2, . . . C5 represent commands and n represents an iteration argument. Commands C1 and C2 are each executed once, then commands C3 and C4 are executed n times. Finally command C5 is executed once and the command line is finished. The iteration argument (n) 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 and preceding it with an iteration argument (n) is equivalent to typing that portion of the string n times. 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 which will cause an error message since the brackets are not properly matched:

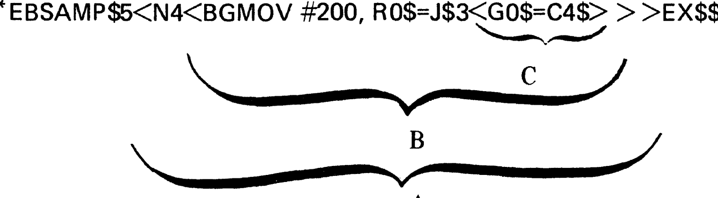
```
><><
<<<>>
```

During command repetition, execution proceeds from left to right until a right bracket is encountered. EDIT then returns to the last left bracket encountered, decrements the iteration 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 innermost brackets and then works its way back again. The most common use for iteration brackets is found in commands such as Unsave, that do not accept repeat counts. For example:

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

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 are to be changed to MOV #244,R4. The following command line is entered:

```
*EBSAMP$5<N4<BGMOV #200, R0$=J$3<G0$=C4$> > >EX$$
```



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 0s to 4s. 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.

3.5 MEMORY USAGE

The memory area used by the Editor is divided into four logical buffers as follows:

Buffer	
MACRO BUFFER	High Memory
SAVE BUFFER	
FREE MEMORY	
COMMAND INPUT BUFFER	
TEXT BUFFER	Low Memory

The Text Buffer contains the current page of text being edited, and the Command Input Buffer holds the command currently being typed at the terminal. If a command currently being entered by the user is within 10 characters of exceeding the space available in the Command Buffer, the message:

```
* CB ALMOST FULL *
```

is printed. If the command can be completed within 10 characters, the user may finish entering the command; otherwise he should type the ALTMODE key twice to execute that portion of the command line already completed. The message is printed each time a character is entered in one of the last 10 spaces.

If the user attempts to enter more than 10 characters the message:

```
?CB FULL?
```

is printed and all commands typed within the last 10 characters are ignored. The user again has 10 characters of available space in which to correct the condition.

The Save Buffer contains text stored with the Save (S) command, and the Macro Buffer contains the command string macro entered with the Macro (M) command. (Both commands are explained in Section 3.6.5.)

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 0M or 0U (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.

3.6 EDITING COMMANDS

3.6.1 Input/Output Commands

Input commands are used to create files and read them into the Text Buffer where they become available for editing or listing. Output commands cause text to be listed on the terminal or lineprinter or written out to a storage device. Some commands are specifically designed for either input or output functions, while a few commands serve both purposes.

Once editing is completed and the page currently in the Text Buffer is written to the output file, that page of text is unavailable for further editing until the file is closed and reopened.

3.6.1.1 Edit Read – The Edit Read command opens an existing file for input and prepares it for editing. Only one file can be open for input at a time.

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 a file is currently open for input, that file is closed; any edits made to the file are preserved.

Edit Read does not input a page of text nor does it affect the contents of the other user buffers (see Section 3.5.)

Edit Read can be used on a file which is already open to close that file for input and reposition EDIT at its beginning. The first Read command following any Edit Read command inputs the first page of the file.

Examples:

*ERDX1:SAMP. MAC$$	Opens SAMP.MAC on device DX1: for input.
*ERSOURCE$$	Opens SOURCE on device DK: for input.

3.6.1.2 Edit Write – The Edit Write command sets up a file for output of newly created or edited text. However, no text is output and the contents of the user buffers are not affected. Only one file can be open for output at a time. Any current output files are closed.

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, one half the largest available space is used; if this is not adequate for the output file size, the EF and EX commands will not close the output file, and all edits will be lost. It is thus recommended that the [n] construction be used whenever there is doubt as to whether enough space is available on the device for the output file.

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

Examples:

*EWDK:TEST.MAC$$	Opens the file TEST.MAC on device DK: for output.
*EWFILE.BAS[11]$$	Opens the file FILE.BAS (allocating 11 blocks) on the device DK: for output.

3.6.1.3 Edit Backup – The Edit Backup command is used to open an existing file for editing and at the same time create a backup version of the file. Any currently open file will be closed. No text is read or written with this command.

The form of the command is:

EBdev:filnam.ext[n]$

The device designation, filename and extension are limited to 19 characters. If dev: is not specified, DK: is assumed. [n] is optional and represents the length of the file to be opened; if not specified, one-half the largest available space is used.

The file indicated in the command line must already exist on the device designated since text will be read from this file as input. At the same time, an output file is opened under the same filename and extension. After an EB command has been successfully executed, the original file (used as input) is renamed with the current filename and a .BAK extension; any previous file with this filename and a .BAK extension is deleted. The new output file is closed and assigned the name as 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.

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 (allocating 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 filename are illegal (the filename is considered to be a text string). All dev:file.ext specifications for EB, ER and EW commands conform to the HT-11 conventions for file naming and are identical to filenames entered in command strings used with other system programs.

3.6.1.4 Read – The Read command (R) causes a page of text to be read from the input file (previously specified in an ER or EB command) and appended to the current contents, if any, of the Text Buffer.

The form of the command is:

R

No arguments are used with the R command and the pointer is not moved. Text is input 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; or
2. The Text Buffer is within 500 characters of being full. (When this condition occurs, Read inputs up to the next <CR> <LF> combination, then returns to Command Mode. An asterisk is printed as though the Read were complete, but text will not have been fully input); or
3. An end-of-file condition is detected, (the *EOF* message is printed when all text in the file has been read into memory and no more input is available).

The maximum number of characters which can be brought into memory with an R command is approximately 6,000 for an 8K system. Each additional 4K of memory allows approximately 8,000 additional characters to be input. An error message is printed if the Read exceeds the memory available or if no input is available.

3.6.1.5 Write – The Write command (W) moves lines of text from the Text Buffer to the output file (as specified in the EW or EB command). The format of the command is:

nW	Write all characters beginning at the pointer and ending at the nth <CR> <LF> to the output file.
–nW	Write all characters beginning on the –nth line and terminating at the pointer 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.

The pointer is not moved and the contents of the buffer are not affected. If the buffer is empty when the Write is executed, no characters are output.

Examples:

*5W$$	Writes the next 5 lines of text starting at the pointer, to the current output file.
*–2W$$	Writes the previous 2 lines of text, ending at the pointer, to the current output file.
*B/W$$	Writes the entire Text Buffer to the current output file.

3.6.1.6 Next – The Next command acts as both an input and output command since it performs both functions. First it writes the current Text Buffer to the output file, then clears the buffer, and finally reads in the next page of the input file. 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. If fewer than n pages are available in the input file, all available pages are input to the buffer, output to the current file, and deleted from the buffer; the pointer is left positioned at the beginning of an empty buffer, and an error message is printed. This command is equivalent to a combination of the Beginning, Write, Delete and Read commands (B/W/DR). Next can be used to space forward, in page increments, through the input file.

Example:

*2N$$	Writes the contents of the current Text Buffer to the output file. Read and write the next page of text. Clear the buffer and then read in another page.

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

nL	Print all characters beginning at the pointer and ending with the nth <CR> <LF>.
–nL	Print all characters beginning with the first character on the –nth line and terminating 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.

The pointer is not moved after the command is executed.

Examples:

*–2L$$	Prints all characters starting at the second preceding line and ending at the pointer.
*4L$$	Prints all characters beginning at the pointer and terminating at the 4th <CR> <LF>.

Assuming the pointer location is:

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

The command:

```
*–1L$$
```

Prints the previous 1 1/2 lines up to the pointer:

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

3.6.1.8 Verify – The Verify command prints the current text line (the line containing the pointer) 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

No arguments are used. 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)+
```

3.6.1.9 End File – The End File command closes the current output file. This command does no input/output operations and does not move the pointer. The buffer contents are not affected. The output file is closed, containing only the text previously output.

The form of the command is:

EF

No arguments are used. Note that an implied EF command is included in EW and EB commands.

3.6.1.10 EXit – The EXit command is used to terminate editing, copy the text buffer and the remainder of the input file to the output file, close input and output files, and return control to the monitor. It performs consecutive Next commands until the end of the input file is reached, then closes both the input and output files.

The command format is:

EX

No arguments are used. 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. (It can however, be made permanent using the monitor CLOSE command.)

An example of the contrasting uses of the EF and EX commands follows. Assume an input file, SAMPLE, contains several pages of text. The user wishes to make the first and second pages of the file into separate files called SAM1 and SAM2, respectively; the remaining pages of text will then make up the file SAMPLE. This can be done using these commands:

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

The user might note that the EF commands are not necessary in this example since the EW command closes a currently open output file before opening another.

3.6.2 Pointer Relocation Commands

Pointer relocation commands allow the current location pointer to be moved within the Text Buffer.

3.6.2.1 Beginning – The Beginning command moves the current location pointer to the beginning of the Text Buffer.

The command format is:

B

There are no arguments.

For example, assume the buffer contains:

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

The B command:

```
*B$$
```

moves the pointer to the beginning of the Text Buffer:

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

3.6.2.2 Jump – The Jump command moves the pointer over the specified number of characters in the Text Buffer.

The form of the command is:

(+ or –)	nJ	Move the pointer (backward or forward) 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.

Negative arguments move the pointer toward the beginning of the buffer, positive arguments toward the end. Jump treats carriage return, line feed, 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.
*B$GABC$=J$$	Move the pointer so that it immediately precedes the first occurrence of 'ABC' in the buffer.

3.6.2.3 Advance – The Advance command is similar to the Jump command except that it moves the pointer a specified number of lines (rather than single characters) and leaves it positioned at the beginning of the line.

The form of the command is:

nA	Advance the pointer forward n lines and position it at the beginning of the nth line.
–nA	Move the pointer backward past n <CR> <LF> combinations and position it at the beginning of the –nth line.
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).

Examples:

*3A$$	Moves the pointer ahead three lines.

Assuming the buffer contains:

```
CLR      @R2
           ↑
```

The command:

```
*0A$$
```

Moves the pointer to:

```
↑CLR     @R2
```

3.6.3 Search Commands

Search commands are used to locate specific characters or strings of characters within the Text Buffer.

3.6.3.1 Get – The Get command starts at the pointer and searches the current Text Buffer for the nth occurrence of a 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) must be positive and is assumed to be 1 if not otherwise specified. 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↑

After search commands, the pointer is left immediately following the text object. Using a search command in combination with =J will place the pointer before the text object, as follows:

    *GTEST$=J$$

This command combination places the pointer before 'TEST'.

3.6.3.2 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. When 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) must be positive and is assumed to be 1 if not otherwise specified.

By deliberately specifying a nonexistent search string, the user can close out his file; that is, he can copy all remaining text from the input file to the output file.

Find is a combination of the Get and Next commands.

Example:

*2FMOBV 6 (R1),@R2$$	Searches the entire input file for the second occurrence of the text string MOVB 6 (R1), @R2. Each unsuccessfully searched buffer is written to the output file.

3.6.3.3. 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) must be positive, and is assumed to be l if not otherwise specified. 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; it 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 from the second half of the input file, a Position search will save time.

The difference between the Find and Position commands 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.

3.6.4 Text Modification Commands

The following commands are used to insert, relocate, and delete text in the Text Buffer.

3.6.4.1 Insert — The Insert command causes the Editor to enter Text Mode and allows text to be inserted immediately following the pointer. Text is inserted until an ESCape is typed and the pointer is positioned immediately after the last character of the insert. The command format is:

```
Itext$
```

No arguments are used with the Insert command, and the text string is limited only by the size of the Text Buffer and the space available. All characters except ESCape are legal in the text string. ESCape 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 memory, an error message is printed.

Example:

```
*IMOV   #BUFF, R2
MOV     #LINE, R1
MOVB    -1 (R2), RO$$
*
```

Inserts the specified text at the current location of the pointer and leaves the pointer positioned after R0.

3.6.4.2 Delete – The Delete command removes a specified number of characters from the Text Buffer. Characters are deleted starting at the pointer; upon completion, the pointer is positioned at the first character following the deleted text.

The form of the command is:

(+ or –) nD	Delete n characters (forward or backward from the pointer).
0D	Delete from beginning of current line to the pointer (equivalent to 0K).
/D	Delete from pointer to end of Text Buffer (equivalent to /K).
=D	Delete –n characters, where n equals the length of the last text argument used.

Examples:

`*–2D$$`	Deletes the two characters immediately preceding the pointer.
`*B$FMOV R1$=D$`	Deletes the text string 'MOV R1'. (=D used in combination with a search command will delete the indicated text string).

Assuming a buffer of:

```
ADD     R1,(R2)+
CLR     ↑@R2
```

the command:

```
*0D$$
```

leaves the buffer with:

```
ADD     R1,(R2)+
↑@R2
```

3.6.4.3 Kill – The Kill command removes n lines from the Text Buffer. Lines are deleted starting at the location pointer; upon completion of the command, the pointer is positioned at the beginning of the line following the deleted text. The command format is:

nK	Delete lines beginning at the pointer and ending at the nth <CR> <LF>.
–nK	Delete lines beginning with the first character in the –nth line and ending at the pointer.
0K	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).

Example:

`*2K$$`	Delete lines starting at the current location pointer and ending at the 2nd <CR> <LF>.

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.

3.6.4.4 Change – The Change command replaces n characters, starting at the pointer, with the specified text string and leaves the pointer positioned immediately following the changed text.

The form of the command is:

(+ or –)	nCtext$	Replace n characters (forward or backward from the pointer) 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 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 ESCape which terminates the text string.

If the C command is to be executed more than once (i.e., it is enclosed in angle brackets) and if there is enough space available so that the command can be entered, it will be executed at least once (provided it appears first in the command string). If repetition of the command exceeds the available memory, an error message is printed. The Change command is identical to executing a Delete command followed by an Insert (nDItext$).

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$$

leaves the buffer with:

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

=C can be used in conjunction with a search command to replace a specific text string as follows:

*GFIFTY : $=CFIVE : $	Find the occurrence of the text string FIFTY: and replace it with the text string FIVE:.

3.6.4.5 Exchange – The Exchange command exchanges n lines, beginning at the pointer, with the indicated text string and leaves the pointer positioned after the changed text.

The form of the command is:

nXtext$	Exchange all characters beginning at the pointer and ending at the nth <CR> <LF> with the indicated text.
–nXtext$	Exchange all characters beginning with the first character on the –nth line and ending at the pointer with the indicated text.
0Xtext$	Exchange the current line from the beginning to the pointer with the specified text (equivalent to 0C).
/Xtext$	Exchange 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 ESCape which terminates the text.

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

If the X command is enclosed within angle brackets so that it will be executed more than once, and if there is enough memory space available so that the X command can be entered, it will be executed at least once (provided it is first in the command string). If repetition of the command exceeds the available memory, an error message is printed.

Example:

```
*2XADD   R1,(R2)+
CLR      @R2
$$
*
```

Exchanges the two lines to the right of the pointer location with the text string.

3.6.5 Utility Commands

3.6.5.1 Save – The Save command starts at the pointer and copies the specified number of lines into the Save Buffer (described previously in Section 3.5).

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 an overflow of the Save Buffer, an error message is printed.

Example:

Assume 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; it may then be inserted in a program wherever needed by using the U command.

3.6.5.2 Unsave – The Unsave command inserts the entire contents of the Save Buffer into the Text Buffer at the pointer location and leaves the pointer positioned following the inserted text.

The form of the command is:

U	Insert in the Text Buffer the contents of the Save Buffer.
0U	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 (only by the 0U command) and may be Unsaved as many times as desired.

If there is no text in the Save Buffer and the U command is given, the ?*NO TEXT*? error message is printed. If the Unsave command causes an overflow of the Text Buffer, the ?*NO ROOM*? error message is displayed.

3.6.5.3 Macro – The Macro command inserts a command string into the EDIT Macro Buffer. The Macro command is of the form:

M/command string/	Store the command string in the Macro Buffer.
0M or M//	Clear the Macro Buffer and reclaim the area for text.

/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 itself.

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 stores the command string so that it can be executed later by the Execute Macro (EM) command. Macro does not affect the contents of the Text or Save Buffers.

All characters except the delimiter are legal Macro command string characters, including single ESCape to terminate text commands. All commands, except the M and EM commands, are legal in a command string macro.

In addition to the 0M command, typing the M command immediately followed by two identical characters (assumed to be delimiters) and two ESCape characters also clears the Macro Buffer.

Examples:

`*M//$$`	Clears the Macro Buffer
`*M/GR0$-C1$/$$`	Stores a Macro to change R0 to R1.

NOTE

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

```
*M GMOV R0$=CADD R1$ $$
?*NO FILE*?
```

In this case, 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 used as the second delimiter, terminating the macro. EDIT then returns an error when the R0$= becomes an illegal command structure.

3.6.5.4 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 returns control to the next command in the original command string.

Examples:

```
*M/BGR0$-C1$/$$
*B1000EM$$
?*SRCH FAIL IN MACRO*?
*
```

Executes the MACRO stored in the previous example. An error message is returned when the end of buffer is reached. (This macro effectively changes all occurrences of R0 in the Text Buffer to R1.)

```
*IMOV PC, R1$2EMICLR @R2$$
*
```

In a new program, inserts MOV PC, R1 then executes the command in the Macro Buffer twice before inserting CLR @R2.

3.6.5.5 Edit Version – The Edit Version command displays the version number of the Editor in use on the terminal.

The form of the command is:

EV$

Example:

```
*EV$$
H02—01
*
```

3.6.5.6 Upper- and Lower-Case Commands – Users who have any upper/lower-case terminal as part of their hardware configuration may take advantage of the upper- and lower-case capability of this terminal. Two editing commands, EL and EU, permit this.

When the Editor is first called (R EDIT), upper-case mode is assumed; all characters typed are automatically translated to upper case. To allow processing of both upper- and lower-case characters, the Edit Lower command is entered:

```
*EL$$
*i Text and commands can be entered in UPPER and lower case.$$
*
```

The Editor now accepts and echoes upper- and lower-case characters received from the keyboard, and outputs text on the teleprinter in upper- and lower-case.

To return to upper-case mode, the Edit Upper command is used:

```
*EU$$
```

Control also reverts to upper-case mode upon exit from the Editor (via EF, EX, or CRTL C).

Note that when an EL command has been issued, Edit commands can be entered in either upper- or lower-case. Thus, the following two commands are equivalent:

```
*GTEXT$=Cnew text$V$$

*gTEXT$=cnew text$v$$
```

The Editor automatically translates (internally) all commands to upper-case independent of EL or EU.

3.7 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$$
B   */L$$
    ; TEST PROGRAM

    START:   MOV #1000, %6       ; INITIALIZE STACK
             MOV #MSG, %0        ; POINT R0 TO MESSAGE
             JSR %7, MSGTYP      ; PRINT IT
             HALT                ; STOP
    MSG:     . ASCII/IT WORKS/
             . BYTE 15
             . BYTE 12
             . BYTE 0
C   *B 1J 5D$$
D   *GPROGRAM$$
    *0L$$
E   ; PROGRAM*I TO TEST SUBROUTINE MSGTYP. TYPES
    ; "THE TEST PROGRAM WORKS"
    ; ON THE TEMI\IM\RMINAL$$
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 %7, 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.

3.8 EDIT ERROR MESSAGES

The Editor prints an error message whenever one of the error conditions listed next occurs. 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 MSG?

is printed and no commands are executed. The user must retype the command.

If the command string is syntactically correct, execution is started. Execution errors are still possible, however (buffer overflow, I/O errors, etc.), and if such an error occurs, a message of the form:

?*ERROR MSG*?

is printed. In this case, all commands preceding the one in error are executed, while the command in error and those following are not executed. Most errors will generally be of the syntax type and can be corrected before execution.

When an error occurs during execution of a Macro, the message format is:

?message IN MACRO?
or
?*message IN MACRO*?

depending on when it is detected.

Message	Explanation
CB ALMOST FULL	The command currently being entered is within 10 characters of exceeding the space available in the Command Buffer.
?CB FULL?	Command exceeds the space allowed for a command string in the Command Buffer.
?*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 the CLOSE monitor 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 ARG?	The argument specified is illegal for the command used. A negative argument was specified where a positive one was expected or argument exceeds the range + or – 16,383.
?ILL CMD?	EDIT does not recognize the command specified.
?*ILL DEV*?	Attempted to open a file on an illegal device.
?ILL MAC?	Delimiters were improperly used, or an attempt was made to enter an M command during execution of a Macro or an EM command while an EM was in progress.
?*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.
?*NO TEXT*?	Attempted to call in text from the Save Buffer when there was no text available.
?*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.

CHAPTER 4
PERIPHERAL INTERCHANGE PROGRAM (PIP)

The Peripheral Interchange Program (PIP) is the file transfer and maintenance utility for HT-11. PIP is used to transfer files between any of the HT-11 devices (listed in Table 2-2), merge and delete files from these devices, and list, zero, and compress device 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 prints an asterisk at the left margin of the terminal and waits to receive a line of filenames and command switches. PIP accepts up to six input filenames and three output filenames; command switches are generally placed at the end of the command string but may follow any filename in the string. There is no limit to the number of switches which may be indicated in a command line, as long as only one operation (insertion, deletion, etc.) is represented.

Since PIP performs file transfers for all HT-11 data formats (ASCII, object, and image) there are no assumed extensions for either input or output files; all extensions, where present, must be explicitly specified.

Following completion of a PIP operation, the Command String Interpreter prints an asterisk at the left margin of the teleprinter and waits for another PIP command line. Typing CTRL C at any time returns control to the Keyboard Monitor. To restart PIP, type R PIP or the REENTER command in response to the monitor's dot.

4.1.1 Using the "Wild Card" Construction

PIP follows the standard file specification syntax explained in Section 2.5 (Chapter 2) with one exception: the asterisk character can be used in a command string to represent filenames or extensions. The asterisk (called the "wild card") in a file specification means "all". For instance, "*.MAC" means all files with the extension .MAC, regardless of filename. "FORTN.*" means all files with the filename FORTN regardless of extension. "*.*" means all files, regardless of name or extension.

The wild card character is legal in the following cases only (switches are explained in the next section):

1. Input file specification for the copy and multiple copy operations (i.e., no switch, /I, /B, and /A).
2. File specification for the delete operation (/D).
3. Input and output file specifications for the rename operation (/R).
4. Input and output file specifications for the multiple copy operation (/X).
5. Input file specifications for the directory list operations (/L, /E, /F).

Operations on files implied by the wild card asterisk are performed in the order in which the files appear in the directory. System files with the extension .SYS and files with bad blocks and the extension .BAD 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 (regardless of their filenames) to be deleted from the device DK.
`*.TST=*.BAK/R`	Renames all files with a .BAK extension (regardless of filenames) so that these files now have a .TST extension (maintaining the same filenames).
`*DX1:*.*/X/Y=*.*`	Transfers all files, including system files, (regardless of filename or extension) from device DK to device DX1.
`*.MAC,*.OBJ/L`	Lists all files with .MAC and .OBJ extensions.

4.2 PIP SWITCHES

The various operations which can be performed by PIP are summarized in Table 4-1. If no switch is specified, PIP assumes the operation is a file transfer in image (/I) mode. Detailed explanations of the switches follow the table.

Table 4-1 PIP Switches

Switch	Section	Explanation
/A	4.2.1	Copies file(s) in ASCII mode; ignores nulls and rubouts; converts to 7-bit ASCII; CTRL Z (32 octal) treated as logical end-of-file on input.
/B	4.2.1	Copies files in formatted binary mode.
/C	4.2.1	May be used in conjunction with another switch to cause only files with current date (as designated using the monitor DATE command) to be included in the specified operation.
/D	4.2.3	Deletes file(s) from specified device.
/E	4.2.5	Lists the device directory including unused spaces and their sizes.
/F	4.2.5	Prints a short directory (filenames only) of the specified device.
/G	4.2.1	Ignores any input errors which occur during a file transfer and continues copying.
/I or no switch	4.2.1	Copies file(s) in image mode (byte by byte). This is the default switch.
/K	4.2.11	Scans the specified device and types the absolute block numbers (in octal) of any bad blocks on the device.
/L	4.2.5	Lists the directory of the specified device, including the number of files, their dates, and the number of blocks used by each file.
/N:n	4.2.6	Used with /Z to specify the number of directory segments (n) to allocate to the directory.
/O	4.2.9	Bootstraps the specified device.

(Continued on next page)

Table 4-1 PIP Switches (Cont.)

Switch	Section	Explanation
/Q	4.2.1	When used in conjunction with another PIP operation, causes PIP to type each filename which is eligible for a wild card operation and to ask for a confirmation of its inclusion in the operation. Typing a "Y" causes the named file to be included in the operation; typing anything else excludes the file. The command line is not processed until the user has confirmed each file in the operation.
/R	4.2.4	Renames the specified file.
/S	4.2.7	Compresses the files on the specified directory device so that free blocks are combined into one area.
/T	4.2.3	Extends number of blocks allocated for a file.
/U	4.2.8	Copies the bootstrap from the specified file into absolute blocks 0 and 2 of the specified device.
/V	4.2.10	Types the version number of the PIP program being used.
/W	4.2.5	Includes the absolute starting block and any extra directory words in the directory listing for each file on the device (numbers in octal). Used with /F, /L, or /E.
/X	4.2.2	Copies files individually (without concatenation).
/Y	4.2.1	Causes system files and .BAD files to be operated on by the command specified. Attempted modifications or deletions of .SYS or .BAD files without /Y are not done and cause the message ?NO SYS ACTION? to be printed.
/Z:n	4.2.6	Zeroes (initializes) the directory of the specified device; n is used to allocate extra words per directory entry. When used with /N, the number of directory segments for entries may be specified.

4.2.1 Copy Operations

A command line without a switch causes files to be copied onto the destination device in image mode (byte by byte). This operation is used to transfer memory image (save format) files and any files other than ASCII or formatted binary. For example:

*ABC<XYZ — Makes a copy of the file named XYZ on device DK and assigns the name ABC. (Both files exist on device DK following the operation).

*SY:BACK.BIN=PR:/I — Copies a tape from the papertape reader to the system device in image mode and assigns it the name BACK.BIN.

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

*DX1:F1<F2/A — Copies F2 from device DK onto device DX1 in ASCII mode and assigns the name F1.

Nulls and rubouts are ignored in an ASCII mode file transfer. CTRL Z (32 octal) is treated as logical end-of-file if encountered in the input file.

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 or FORTRAN and for .LDA files produced by the Linker. For example:

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

When performing formatted binary transfers, PIP verifies checksums and prints the message ?CHK SUM? if a checksum error occurs.

If neither /A nor /B is used in a copy operation that involves a paper tape device, the size of the output file in the operation depends upon the memory size of the system. The transfer mode defaults to image mode and PIP attempts to do a single read to fill its input buffer. When a read from the paper tape reader encounters end-of-tape, no count of words transferred can be returned; PIP assumes its input buffer is full and copies it to the output device. The output file size thus depends upon the input buffer size, which is determined by the memory size of the system. The output file will have several blocks of zeroes after the end of the paper tape image. If copying to the punch, large amounts of blank tape will be punched after the input tape image is output. The extra length is harmless, but can be avoided by use of /A or /B. Image mode files (for example, .SAV files) cannot reliably be transferred to or from paper tape.

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

*DK:AA<DX1:BB,CC,DD/I
: Transfers files BB, CC and DD to device DK as one file and assigns this file the name AA.

*DX1:MERGE=DX0:FILE2,FILE3/A
: Merges ASCII files FILE2 and FILE3 on DX0 into one ASCII file named MERGE on device DX1.

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 but ignore all input errors. For example:

*ABC<DX1:TOP/G
: Copies file TOP in image mode from device DX1 to device DK and assigns the name ABC. Any errors during the copy operation are ignored.

*DX1:COMB<DX0:F1,F2/A/G
: Copies files F1 and F2 in ASCII mode from device DX0 to device DX1 as one file with the name COMB. Ignores input errors.

The wild card construction may be used for input file specifications during copy operations. Be sure to use the /Y switch if system files (.SYS) are to be copied. For example:

DX1:PROG1<.MAC
: Copies, in image mode, all files with a .MAC extension from device DK to device DX1 and combines them under the name PROG1.

**.*=DX1:*.*/G/Y/X
: Copies to device DK, in image mode, all files (including .SYS files) from device DX1; ignores any input errors.

If only files with the current date are to be copied (using the wild card construction), the /C switch must also be used in the command line. For example:

DX1:NN3=ITEM1./C,ITEM2/A	Copies, in ASCII mode, all files having the filename ITEM1 and the current date, (the date entered using the monitor DATE command) and copies ITEM2 (regardless of its date) from device DK to device DX1 and combines them under the name NN3.
DX1:.*=*.*/C/X	Copies all files with the current date from DK to DX1. Note that commands of this nature are an efficient way to backup all new files after a session at the computer.

The /Q switch is used in conjunction with another PIP operation and the wild card construction to list all files and allow the user the opportunity to confirm individually which of these files should be processed during the wild card expansion. Typing a "Y" causes the named file to be processed; typing anything else excludes the file. For example:

```
**.OBJ<DX1:*.OBJ/Q/X
FIRST  .OBJ?Y
GETR   .OBJ?
BORD   .OBJ?
CARJ   .OBJ?Y
```

Copies the files FIRST.OBJ and CARJ.OBJ to the DK in image mode from disk and ignores the others.

The file allocation scheme for HT-11 normally allows half the entire largest available space or the second largest space, or a maximum size (a constant which may be patched in the HT-11 monitor), whichever is largest, for a new file. The user can, using the [n] construction explained in Chapter 2, force HT-11 to allow the entire largest possible space by setting n=177777. If n is set equal to any other value (other than 0 which is default and gives the normal allocation described first above), that size will be allocated for the file.

Therefore, assume that the directory for a given device shows a free area of 200 blocks and that PIP returns an ?OUT ER? message when a transfer is attempted to that device with a file which is longer than 100 blocks but less than 200 blocks. 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 (refer to Chapter 2, Section 2.5 for an explanation of the [n] construction).
2. Use the /X switch during the transfer to force PIP to allocate the correct number of blocks for the output file. This procedure will operate correctly if the input device is a disk.

For example, assume that file A is 150 blocks long and that a directory listing shows that there is a 200 block <unused> space on DX1:

```
.R PIP
*DX1:A=A
?OUT ER?
```

File longer than 100 blocks.

```
*DX1:A[150]=A
```

or

```
*DX1:A=A/X
```

Either command causes a correct transfer.

4.2.2 Multiple Copy Operations

The /X switch allows the transfer of several files at a time onto the destination device as individual files. The /A, /G, /C, /Q, /B and /Y switches can be used with /X. If /X is not indicated, all output files but the first will be ignored.

Examples:

```
*FILE1,FILE2,FILE3<DX1:FILEA,FILEB,FILEC/X
```

Copies, in image mode, FILEA, FILEB and FILEC from device DX1 to device DK as separate files called FILE1, FILE2 and FILE3, respectively.

```
*DX1:F1.*=F2.*/X
?NO SYS ACTION?
*
```

Copies, in image mode, all files named F2 (except files with .SYS or .BAD extensions) from device DK to device DX1. Each file is assigned the filename F1 but retains its original extension.

```
*DX0:*.*=DX1:*.*/X
?NO SYS ACTION?
```

Copies, in image mode, all files on device DX1 to device DX0 (except files with .SYS or .BAD extensions); the files are copied separately and retain the same names and extensions.

```
*DX1:FILE1,FILE2<FILEA.*/A/G/X
```

This command line assumes there are two files with the filename FILEA (and any extension excluding .SYS or .BAD extensions) and copies these files in ASCII mode to device DX1. The files are transferred in the order they are found in the directory; the first file found is copied and assigned the name FILE1, and the second is assigned FILE2. If there is a third, it is ignored and a fourth causes an ?OUT FIL? error.

```
*DX1:*.SYS=*.SYS/X/Y
```

Copies all system files from device DK to device DX1.

File transfers performed via normal operations 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, the /X switch should be used whenever possible (i.e., when no concatenation is desired) as an aid to reducing disk fragmentation.

```
*A=B
```

and

```
*A=B/X
```

perform the same operation; however, using the second construction whenever possible increases the system disk-usage efficiency.

For example, assume the directory of DX1 is:

```
*/E
9-MAY-79
MONITR.SYS    32 5-MAY-79      < UNUSED >    2<NO DATE>
PR    .SYS     2 5-MAY-79      < UNUSED > 444<NO DATE>
2 FILES. 34 BLOCKS
446 FREE BLOCKS
*
```

To copy the file PP.SYS (2 blocks long) from DK to DX1, the command:

```
*DX1:PP.SYS=PP.SYS/Y
```

can be entered, and the new directory is:

```
*/E
9-MAY-79
MONITR.SYS    32 5-MAY-79        < UNUSED >    2<NO DATE>
PR     .SYS    2 5-MAY-79        PP     .SYS   2 9-MAY-79
< UNUSED >   442<NO DATE>
3 FILES. 36 BLOCKS
444 FREE BLOCKS
*
```

If the command:

```
*DX1:PP.SYS=PP.SYS/Y/X
```

had been entered, the new directory would appear:

```
*/E
9-MAY-79
MONITR.SYS    32 5-MAY-79        PP     .SYS   2 9-MAY-79
PR     .SYS    2 5-MAY-79        < UNUSED > 444<NO DATE>
3 FILES. 36 BLOCKS
444 FREE BLOCKS
*
```

4.2.3 The Extend and Delete Operations

The /T switch is used to increase the number of blocks allocated for the specified file. The file associated with the /T switch must be followed by a numeric argument of the form [n] where n is a decimal number indicating the number of blocks to be allocated to the file at the completion of the extend operation.

The format of the /T switch is:

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

A file can be extended in this manner only if it is followed by an unused area of sufficient size (on whichever device it is located) to accommodate the additional length of the extended file. It may be necessary to create this space by moving other files on the device using the /X switch.

Specifying the /T switch in conjunction with a file that does not currently exist creates a file of the designated length.

Error messages are printed if the /T command makes 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.
*DX1:XYZ[100]</T	Assigns 100 blocks to the file named XYZ on device DX1.

The /D switch is used to delete one or more files from the specified device. The wild card character (*) can be used in conjunction with this 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 on block-replaceable devices, 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:

filena.ext[n]=/T

where filena.ext is the name desired and n is the length of the deleted file. For example:

```
*DX1:/E
4-JUN-79
A       .MAC   18 3-JUN-79     B      .MAC   17 3-JUN-79
C       .MAC   19 3-JUN-79     < UNUSED > 426<NO DATE>
3 FILES, 54 BLOCKS
426 FREE BLOCKS

*DX1:B.MAC/D

*DX1:/E
 4-JUN-79
A     .MAC   18 3-JUN-79     < UNUSED > 17 <NO DATE>
C     .MAC   19 3-JUN-79     < UNUSED >426 <NO DATE>
2 FILES, 37 BLOCKS
443 FREE BLOCKS
*
```

File B.MAC could now be recovered by:

*DX1: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 preceding command is sufficient. If not, all larger unused spaces preceding the desired file must be given dummy names before the recovery can be made.

October 15, 1979
Part D

For instance, assume the previous example with the exception that A.MAC has a 33 block unused file before it, so that the directory looks like:

```
*DX1:/E
 4-JUN-79
< UNUSED >    33 <NO DATE>       A      .MAC    18 3-JUN-79
< UNUSED >    17 <NO DATE>       C      .MAC    19 3-JUN-79
< UNUSED >   393 <NO DATE>
2 FILES, 37 BLOCKS
443 FREE BLOCKS
*
```

A recover of B.MAC would require:

```
*DX1:DUMMY[33]=/T
*DX1:B.MAC[17]=/T
```

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

Examples:

*FILE1.SAV/D	Deletes FILE1.SAV from device DK.
DX1:.*/D	Deletes all files from device DX1 except those with a .SYS or .BAD extension. If there is a file with a .SYS or .BAD extension, the message ?NO SYS ACTION? is printed to remind the user that these files have not been deleted.
**.MAC/D	Deletes all files with a .MAC extension from device DK.
*DX0:R1,DX1:AA/D	Deletes the files specified from the associated devices.
DX1:.*/D/Y	Deletes all files from device DX1.

4.2.4 The Rename Operation

The /R switch is used (in a manner similar to the multiple copy command described in Section 4.2.2) to rename a file given as input with the associated name given in the output specification. There must be an equal number of input and output files and they must reside on the same device, or an error message will be printed. The /Y switch must be used in conjunction with /R if .SYS files are to be renamed.

The Rename command is particularly useful when a file on a disk contains bad blocks. By renaming the file with a .BAD extension, the file permanently resides in that area of the device so that no other attempts to use the bad area will occur. Once a file is given a .BAD extension it cannot be moved during a compress operation. .BAD files are not renamed in wild card operations unless /Y is used.

Examples:

*DX1:F1,X1<DX1:F0,X0/R

Renames F0 to F1 and X0 to X1 on device DX1.

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

4.2.5 Directory List Operations

The /L switch lists the directory of the specified device. The listing contains the current date, all files with their associated creation dates, total free blocks on the device if disk, the number of files listed, and number of blocks used by the files. File lengths, number of blocks and number of files are indicated as decimal values. If no output device is specified, the directory is output to the terminal (TT:).

Outputs complete directory of device DX1 to the terminal.

```
*DX1:/L
1-AUG-79
MONITR.SYS   32 5-MAY-79        PP     .SYS     2 9-MAY-79
PR     .SYS   2 5-MAY-79        F2     .REL    15 4-JUL-79
MERGE .BAS    2 4-JUL-79        COMB   .OBJ     2 4-JUL-79
6 FILES, 55 BLOCKS
425 FREE BLOCKS
```

Outputs a complete directory of device DX1 to a file, DIRECT, on the device DK.

```
*DIRECT=DX1:/L
```

Outputs a complete directory of all files on device DK using the .MAC extension.

```
**.MAC/L
1-AUG-79
VTMAC .MAC     7 8-JUL-79       FILE2 .MAC   1 9-JUL-79
2 FILES, 8 BLOCKS
472 FREE BLOCKS
```

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

```
*/E
9-SEP-79
BATCH.HLP    2  3-SEP-79        CHESS.SAV   20  2-SEP-79
PAT1 .FOR   10  5-SEP-79        < UNUSED >   3 <NO DATE>
IRAD5.MAC    8  7-SEP-79        < UNUSED >  30 <NO DATE>
TRIG .OBJ    2  6-SEP-79        STP   .OBJ   2  6-SEP-79
< UNUSED >  15 <NO DATE>        BAC   .OBJ   2  6-SEP-79
< UNUSED >   4 <NO DATE>        LIBRI.OBJ  137  6-SEP-79

                       . . .
```

The /F switch lists only filenames, omitting the file lengths and associated dates.

Example:

```
*DX0:/F
TRACE .MAC    CARGO .REL    BMAP  .OBJ    AAA   .TST    NEW   .DAT
```

The /L, /E and /F commands have no effect on the files of the specified device. If the /W switch is used in conjunction with the /L or /E switches, the absolute starting block of the file and extra words (in octal) will be included in the listing. For example:

```
*DX1:/L/W
10-SEP-79
DSQRT .OBJ     1 10-SEP-79  16 0        MAIN  .OBJ   1 10-SEP-79  17 0
BASICR.OBJ    11 10-SEP-79  20 0        OTSV2 .OBJ   3 10-SEP-79  33 0
```

NOTE: When you allocate more than a single word per directory, the display is larger than a conventional console screen. The listing device must be capable of printing records greater than 80 characters in width.

The first three columns indicate the filename and extension, block length, and date. The fourth column shows the absolute starting block (in octal), and the fifth column shows the contents of each extra word per directory entry (in octal). (This is allocated using the /Z:n switch; see Section 4.2.6.)

Using the /L, /E, or /F switch in conjunction with a device and filename causes the filename, and optionally the date and file length to be output rather than a directory of the entire device. For example:

```
*F1.SAV/L
```

causes:

```
 4-JUN-78
F1     .SAV   18  4-JUN-78
124 FREE BLOCKS
*
```

to be output, providing the file exists on device DK.

Directories are made up of segments which are two blocks long. Full directory listings with multiple segments contain blank lines as segment boundaries.

4.2.6 The Directory Initialization Operation

The /Z switch clears and initializes the directory of an HT-11 directory-structured device and must always be the first operation performed on a new (that is, previously unused) device. The form of the switch is:

```
/Z:n
```

where n is an optional octal number to increase the size of each directory entry on a directory-structured device. If n is not specified, each entry is 7 words long (for filename and file length information) and 70 entries can be made in a directory segment. When extra words are allocated, the number of entries per directory segment decreases. The formula for determining the number of entries per directory segment is:

507/((# of extra words)+7)

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

When /Z is used, PIP responds as follows:

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

For example:

```
*DX1:/Z
DX1:/Z          ARE YOU SURE ?
```

Answer Y and a carriage return to perform the initialization. An answer beginning with a character other than Y is considered to be no.

Example:

```
*DX1:/Z
DX1:/Z          ARE YOU SURE ?Y<CR>
*
```

Zeroes the directory on device DX1 and allocates no extra words for the directory.

The /N switch is used with /Z to specify the number of directory segments for entries in the directory. The form of the switch is:

/N:n

where n is an octal number less than or equal to 37. Initially HT-11 allocates four directory segments, each two blocks (512 words) long.

Example:

`*DX1:/Z:2/N:6`	Zeroes the directory on device DX1, allocates two extra words per directory entry and allocates six directory segments.

4.2.7 The Compress Operation

The /S switch is used to compress the directory and files on the specified device, condensing all the free (unused) blocks into one area. Input errors are reported on the console terminal unless the /G switch is used; output errors are always reported. In either case, the compress continues.

/S can also be used to copy DX disks, though the output diskette must first be initialized using /Z to write the appropriate volume identification. (It is important to note that the /S switch destroys any previous directory on the output device. The new directory on the output device has the same number of segments as the directory on the input device.) /S does not copy the bootstrap onto the volume.

To increase the number of directory blocks in a two-volume compress (that is, from one volume to another rather than from one volume to itself), use the /N:n switch in conjunction with the /S switch (any attempts to decrease the directory size are ignored).

/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 are then left in place when a /S is executed.

If a compress operation is performed on the system 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.

NOTE

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. If a device is being compressed in place, input and output errors are reported on the terminal and the operation continues to completion.

Examples:

*SY:/S ?REBOOT?	Compresses the files on the system device SY:
*DX0:A<DX1:/S	Transfers and compresses the files from device DX1 to device DX0. Device DX1 is not changed. The filename A is a dummy specification required by the Command String Interpreter.

4.2.8 The Bootstrap Copy Operation

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

Example:

```
*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 filename.

4.2.9 The Boot Operation

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

Example:

```
*DK:/0                  Reboots the device DK.
```

If a boot switch is specified on an illegal device, the message:

?BAD BOOT?

is printed. Legal devices are SY, DK, and DX0-DX1.

4.2.10 The Version Switch

The Version switch (/V) outputs a version number message (representing the version of PIP in use) to the terminal using the form:

PIP V01-XX

The rest of the command line, if any, is ignored.

4.2.11 Bad Block Scan (/K)

The bad block switch (/K) scans the specified device and types the absolute block numbers of those blocks on the device which return hardware errors. The block numbers typed are octal; the first block on a device is 0(8). Note that if no errors occur, nothing will be output. A complete scan of a disk takes several minutes.

Example:

```
*DX1:/K                 Scan disk drive 1 for bad blocks.
BLOCK      140 IS BAD
*DX:/K                  Scan drive 0. No blocks are bad.
*
```

4.2.11.1 Recovery from Bad Blocks – As a disk ages, the recording surface wears. Eventually unrecoverable I/O errors occur during attempts to read or write a bad disk block. PIP protects against usage of bad disk areas by ignoring files with a .BAD extension (unless the /Y switch is used). Once a bad block is uncovered in an I/O operation, it can be located using the /K switch and a .BAD file can be created which encompasses the bad block.

When a hardware I/O error is detected, the recovery procedure is as follows:

1. Use the PIP /K switch to scan the device and print on the terminal the absolute block numbers (in octal) of the bad blocks. For example:

   ```
   .R PIP
   *DX1:/K
   BLOCK      23 IS BAD
   *
   ```

2. Obtain an extended directory with the /W switch, showing the starting block numbers of all the files on the disk.
3. If a bad block occurs in a file with valuable information, copy the file to another file using the /G switch. In most cases, only 1 bit (character) of the file is affected.

4. If the file is small, it can then be renamed with a .BAD extension to prevent further use of that disk area.
5. If the file is large or the bad block occurs in an empty area, a 1-block .BAD file can be created using the /T switch as follows:
   a. Delete the bad file (if any).
   b. If the bad block is at block n of the free area, create a file of length n-1 with the /T switch. Remember that there must be no spaces larger than n-1 blocks before the desired one (refer to Section 4.2.3). Also note that the block numbers printed in the /K and /W operations are octal, while the argument to the /T operation is decimal.
   c. Create a 1-block .BAD file with the /T switch to cover the bad block.
   d. Delete any temporary files created during the operation.

For example, assume the extended directory is:

```
*
*
NEWSRC.BAT    8 11-SEP-79  55     RTTEMP.BAT   27 11-SEP-79  65
PIP   .MAC  150 12-SEP-79 120     < UNUSED >  154 <NO DATE>
VERIFY.SAV    3 12-SEP-79 600     PIP   .OBJ   15 12-SEP-79 603
MKPIP .CTL    1 12-SEP-79 622     MKV2RK.CTL    4 12-SEP-79 623
VTLIB .OBJ   10 12-SEP-79 627     A             4 12-SEP-79 641
PIP   .LST   50  3-SEP-79 645
*
*
```

and a bad block is detected at block 670 (octal) of the file PIP.LST. To recover, make a copy, ignoring the error, and delete the bad file:

```
*DX1:PIPA.LST=DX1:PIP.LST/G
*DX1:PIP.LST/D
```

The directory now reads:

```
*
*
NEWSRC.BAT    8 11-SEP-79  55     RTTEMP.BAT   27 11-SEP-79  65
PIP   .MAC  150 12-SEP-79 120     PIPA  .LST   50 18-SEP-79 346
< UNUSED >  104 <NO DATE>         VERIFY.SAV    3 12-SEP-79 600
PIP   .OBJ   15 12-SEP-79 603     MKPIP .CTL    1 12-SEP-79 622
MKV2RK.CTL    4 12-SEP-79 623     VTLIB .OBJ   10 12-SEP-79 627
A             4 12-SEP-79 641
*
*
```

An unused area following A contains block 670 (octal), which is bad. Continuing in PIP:

```
*DX1:TEMP.002[104]=/T
*DX1:TEMP.003[19]=/T
```

This fills the unused areas with temporary files. Specifying TEMP.003 with a length of 19 blocks makes the file just long enough to precede the bad block (i.e., 645 (octal) and 19 (decimal) equal 670, which would be the starting block number of the next file created). The directory now contains:

```
*
*
NEWSRC.BAT     8 11-SEP-79  55      RTTEMP.BAT    27 11-SEP-79  65
PIP   .MAC   150 12-SEP-79 120      PIPA  .LST    50 18-SEP-79 346
TEMP  .002   104 18-SEP-79 430      VERIFY.SAV     3 12-SEP-79 600
PIP   .OBJ    15 12-SEP-79 603      MKPIP .CTL     1 12-SEP-79 622
MKV2RK.CTL     4 12-SEP-79 623      VTLIB .OBJ    10 12-SEP-79 627
A              4 12-SEP-79 641      TEMP  .003    19 18-SEP-79 645
*
*
```

Continuing with PIP:

```
*DX1:FILE.BAD[1]=/Y/T                       Create a bad file.
```

The directory now contains:

```
*
*
NEWSRC.BAT     8 11-SEP-79  55      RTTEMP.BAT    27 11-SEP-79  65
PIP   .MAC   150 12-SEP-79 120      PIPA  .LST    50 18-SEP-79 346
TEMP  .002   104 18-SEP-79 430      VERIFY.SAV     3 12-SEP-79 600
PIP   .OBJ    15 12-SEP-79 603      MKPIP .CTL     1 12-SEP-79 622
MKV2RK.CTL     4 12-SEP-79 623      VTLIB .OBJ    10 12-SEP-79 627
A              4 12-SEP-79 641      TEMP  .003    19 18-SEP-79 645
FILE  .BAD     1 18-SEP-79
*
*
```

Next delete all temporary files and rename PIPA.LST to PIP.LST. The final directory now contains:

```
*
*
NEWSRC.BAT    8 11-SEP-79  55     RTTEMP.BAT   27 11-SEP-79  65
PIP   .MAC  150 12-SEP-79 120     PIP   .LST   50 18-SEP-79 346
< UNUSED >  104 <NO DATE>         VERIFY.SAV    3 12-SEP-79 600
PIP   .OBJ   15 12-SEP-79 603     MKPIP .CTL    1 12-SEP-79 622
MKV2RK.CTL    4 12-SEP-79 623     VTLIB .OBJ   10 12-SEP-79 627
A             4 12-SEP-79 641     < UNUSED >   19 <NO DATE>
FILE  .BAD    1 18-SEP-79 760
*
*
```

Disks with many bad blocks can often be reused by reformatting them. First copy all desired files, since reformatting destroys all information contained on a volume.

4.3 PIP ERROR MESSAGES

The following error messages are output on the terminal when PIP is used incorrectly:

Errors	Meaning
?BAD BOOT?	A boot switch was specified on an illegal device.
?BOOT COPY?	An error occurred during an attempt to write bootstrap with /U switch.
?CHK SUM?	A checksum error occurred in a formatted binary transfer.
?COR OVR?	Memory 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. Check volume for off-line and try the operation again.
?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 during a delete, copy, or rename operation, or no input files with the expected name or extension were found during a *.* expansion.
?ILL CMD?	The command specified was not syntactically correct; a device name is missing which should be specified, a switch argument is too large, a filename is specified where one is inappropriate, or a nonfile-structured device is specified for a file-structured operation.

?ILL DEV?	Illegal or nonexistent device.
?ILL DIR?	The device did not contain a properly initialized directory structure. Use /Z.
?ILL REN?	Illegal rename operation. Usually caused by different device names on the input and output sides of the command string.
?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 may be caused by an attempt to compress a larger device to a smaller one or by 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 [n] 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/.BAD ACTION?

The /Y switch was not included with a command specified on a .SYS or .BAD file. The command is executed for all but the .SYS and .BAD files. A *.* transfer is most likely to cause this message.

?REBOOT?

.SYS files have been transferred, renamed, compressed or deleted from the system device. It may be necessary to reboot the system.

NOTE

The message is typed immediately after execution of the relevant command has begun, but the actual reboot operation must not be performed until PIP returns with the prompting asterisk for the next command. If the system is halted and rebooted before the prompting asterisk returns, disk information may be lost.

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 or if the /N:n switch was used to increase the number of directory segments. The need to reboot can be permanently avoided by placing all .SYS files at the beginning of the system device, then avoiding their involvements in PIP operations by not using the /Y switch.

dev:/Z ARE YOU SURE?

Confirmation must be given by the user before a device can be zeroed.

CHAPTER 5
THE ASSEMBLY PROCESS

Three HT-11 system programs perform the tasks collectively known as the assembly process.

EXPAND makes the first pass over a source program containing macros, applying a user's macro definition or one from the system library each time the source program references a macro. EXPAND writes the program source, with macros expanded, to its output file.

The EXPANDed program or one originally without macros (both have .PAL file extensions) undergoes two passes by ASEMBL. This system program outputs a single relocatable binary object file and can also produce an assembly listing with symbol table.

The system program CREF (Cross REFerence) appends an index of symbol usage to the assembly listing when specified as part of the assembly output.

Some notable features of ASEMBL 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 declaration for linking among object modules
7. Conditional assembly directives
8. Program sectioning directives
9. Extensive listing control, including cross reference listing

Operating instructions for the three programs EXPAND, ASEMBL, and CREF appear in Sections 5.7 and 5.8.

5.1 SOURCE PROGRAM FORMAT

A source program is composed of a sequence of source lines; each source line contains a single assembly-language statement followed by a statement terminator. A terminator may be either a line feed character (which increments the line count by 1) or a form feed character (which resets the line count and increments the page count by 1).

NOTE

EDIT automatically appends a line feed to every carriage return encountered in a source program. For listing format, ASEMBL automatically inserts a carriage return before any line feed or form feed not already preceded by one.

An assembly-language line can contain up to 132(decimal) 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 an assembly language statement is:

```
label:     operator   operand(s)   ;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:

```
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.)

Source statements may be formatted with EDIT so that use of the TAB character causes the statement fields to be aligned. For example:

Label Field	Operator Field	Operand Field	Comment Field
CHECK:	BIT	#1,R0	;IS NUMBER ODD?
	BEQ	EVEN	;NO, IT'S EVEN
	MOV	#–1,ODDFLG	;ELSE SET FLAG
EVEN:	RTS	PC	;RETURN

5.1.1.1 Label Field – A label is a user-defined symbol that is unique within the first six characters and 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 (fixed in memory independently of the position of the program) or relocatable (not fixed in memory), depending on whether the location counter value (see Section 5.2.6) is currently absolute or relocatable.

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(octal), the statement:

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

assigns the value 100(octal) to the label ABCD. Subsequent reference to ABCD references location 100(octal). In this example if the location counter was declared relocatable within the section, the final value of ABCD would be 100(octal) plus a value assigned by LINK when it relocates the code, called the relocation constant. (The final value of ABCD would therefore not be known until link-time. This is discussed later in this chapter and in Chapter 6.)

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

```
ABC:        ERREX:        MASK:        MOV        A,B
```

cause each of the three labels — ABC, ERREX, and MASK — to be equated to the value 100(octal).

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 an instruction mnemonic or an assembler directive. The operator may be preceded by zero, 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 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 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). Multiple delimiters separating operands are not legal (with the exception of spaces and tabs — any combination of spaces and/or tabs represents a single delimiter). An operand may be preceded by an operator, a 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 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 (CTRL FORM) after the nth line. (See also Section 5.5.1.2 for a description of assembly listing output.)

5.2 SYMBOLS AND EXPRESSIONS

This section describes the various components of legal expressions: the assembler character set, symbol construction, numbers, operators, terms and expressions.

5.2.1 Character Set

The following characters are legal in 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 (with the exception of local symbols; See Section 5.2.5).
4. The following special characters:

Character	Designation	Function
carriage return		formatting character
line feed		source statement terminator
form feed		source statement terminator
vertical tab		source statement terminator
:	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
,	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
↑	uparrow	universal unary operator, argument indicator

5.2.1.1 Separating Characters – Reference is made in the remainder of the chapter to legal separating characters. These terms are defined in Table 5-1 and following.

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.
,	comma	A comma is a legal separator for both expressions and argument operands.
<...>	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. (The angle bracket construction should be used when the argument contains unary operators.)
↑\...\	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.

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 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 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 (operators applying to only one operand) are as follows:

Unary Operator	Explanation	Example	
+	plus sign	+A	(positive value of A, equivalent to A)
–	minus sign	–A	(negative, 2's complement, value of A)
↑	uparrow, 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 1-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 described above can be used adjacent to each other in a term. For example:

```
↑C↑O12
–↑O5
```

Legal binary operators are as follows:

Binary Operator	Explanation	Example
+	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. Division and multiplication are signed operations. 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 Symbols

ASEMBL maintains the Permanent Symbol Table (PST) and the User Symbol Table (UST). The PST contains all the permanent symbols and is part of the assembler load module. The UST is 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 B) and assembler directives (sections 5.5 and 5.6, Appendix B). 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 Symbols – User-defined symbols are those used as labels 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.

User-defined 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; it is recommended that $ and . not be inserted in user-defined symbols.

The following rules apply to the creation of user-defined 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 either one of the symbol types. To determine the value of the symbol, the assembler searches the symbol tables in the following order:

1. Permanent Symbol Table
2. 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.

These search orders allow redefinition of Permanent Symbol Table entries as user-defined symbols.

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 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 ASEMBL 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. 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.

NOTE

If the program jumps to or references the label of a direct assignment statement, it is actually referencing the following instruction statement. For example:

```
        .=.+1000
C:      D=3
E:      MOV  #D,ABLE
         .
         .
        JMP  C
```

This code causes a jump to the label E.

3. Only one symbol can be defined by any one direct assignment statement.
4. Only one level of forward referencing is allowed. That is, the following arrangement is illegal:

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

X and Y are both undefined throughout pass 1. X is undefined throughout pass 2 and causes an 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
```

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 cannot 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 to be referenced by branch instructions. 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.

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 .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 statement altering the location counter:

.=expression

The expression defining the location counter must not contain forward references or symbols that vary from one pass to another. If an expression is assigned to the current'location counter in a relocatable CSECT, an error flag is generated. (The construction .=.+expression must be used.)

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

```
Line     Octal
Number   Expansion        Source Code                    Comments

 1
 2
 3
 4
 5
 6
 7
 8
 9        000000           R0=%0
10                         .SBTTL   SECTOR INITIALIZATION
11        000000'          .CSECT   IMPURE           ;IMPURE STORAGE AREA
12 00000          IMPURE:
13        000000'                   .CSECT   IMPPAS  ;CLEARED EACH PASS
14 00000          IMPPAS:
15        000000'          .CSECT   IMPLIN           ;CLEARED EACH LINE
16 00000          IMPLIN:
17        000000'          .CSECT   XCTPRG           ;PROGRAM
18                                                   ;INITIALIZATION
19 00000          XCTPRG:
20 00000  012700           MOV      #IMPURE,R0
          000000'
21 00004  005020 1$:       CLR      (R0)+            ;CLEAR IMPURE AREA
22 00006  022700           CMP      #IMPTOP,R0
          000000'
23 00012  101374           BHI      1$
24
25        000000'          .CSECT   XCTPAS           ;PASS INITIALIZATION
26 00000          XCTPAS:
27 00000  012700           MOV      #IMPPAS,R0
          000000'
28 00004  005020 1$:       CLR      (R0)+            ;CLEAR IMPURE PART
29 00006  022700           CMP      #IMPTOP,R0
          000000'
30 00012  101374           BHI      1$
31
32        000000'          .CSECT   XCTLIN           ;LINE INITIALIZATION
33 00000          XCTLIN:
34 00000  012700           MOV      #IMPLIN,R0
          000000'
35 00004  005020 1$:       CLR      (R0)+
36 00006  022700           CMP      #IMPTOP,R0
          000000'
37 00012  101374           BHI      1$
38
39        000000'          .CSECT   MIXED            ;MIXED MODE SECTOR
40 00000  000000 IMPTOP:   .WORD 0
41        000001'          .END
```

Figure 5-1 Assembly Source Listing 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.

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
COUNT:    .WORD 0                ;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
INDEX:    .WORD 0                ;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(octal) bytes of storage space in the program. The next instruction is stored at 1100. (The .BLKW and .BLKB directives can also be used to reserve blocks of storage; see Section 5.5.5.3.)

5.2.7 Numbers

The 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 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 ↑F operator (see Section 5.5.6.2) are stored in the following format:

15	14 — 7	6 — 0
SIGN BIT	8-BIT EXPONENT	7-BIT MANTISSA

Refer to the *H11 Operation Manual* 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 whose 16-bit value is used.
2. A symbol that is interpreted according to the following hierarchy:
   a. a period that 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 that 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. An expression 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 (for example, to differentiate between A*B+C and A*<B+C>) or to apply a unary operator to an entire expression (–<A+B>).

5.2.9 Expressions

Expressions are combinations of terms that are joined together by binary operators and that reduce to a 16-bit value. The operands of a .BYTE directive 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 the 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>>
```

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 has 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 the assembly location counter (in relocatable sections) 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.

An example of the three expression types follows:

```
          .ASECT
.=100
ABSSYM=.                              ;THE VALUE OF ABSSYM IS
                                      ;NOT RELOCATABLE, BECAUSE
                                      ;WE ARE IN AN ASECT

          .CSECT MAIN                 ;START RELOCATABLE
                                      ;PROGRAM SECTION

          .GLOBL EXTVAL               ;EXTVAL IS DEFINED ELSEWHERE,
                                      ;ITS VALUE WILL NOT BE KNOWN
                                      ;UNTIL LINK TIME

BEGSYM:   .BLKW 4                     ;THE VALUES OF BEGSYM
          .ASCII /ABCD/               ;AND ENDSYM ARE
          .EVEN                       ;RELOCATABLE, BECAUSE
ENDSYM=.                              ;THE ADDRESS AT WHICH
                                      ;"MAIN" WILL BE LOADED
                                      ;IS NOT DETERMINED UNTIL
                                      ;LINK TIME

SIZE = ENDSYM-BEGSYM                  ;HOWEVER, THE
                                      ;VALUE OF SIZE IS KNOWN
                                      ;(IT IS 12.) AT ASSEMBLY
                                      ;TIME AND IS ABSOLUTE

RELEXP = ENDSYM-BEGSYM+.              ;RELEXP (=.+12.) IS
                                      ;RELOCATABLE

EXTEXP:   .WORD EXTVAL+4              ;THE EXPRESSION 'EXTVAL+4"
                                      ;IS EXTERNAL (OR GLOBAL)
                                      ;BECAUSE EXTVAL IS DEFINED
                                      ;IN ANOTHER PROGRAM UNIT.

CHARA='A                              ;THE VALUE OF CHARA
                                      ;IS ABSOLUTE
```

5.3 RELOCATION AND LINKING

The output of the 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 words that are to be modified (as described in the previous paragraph) are marked with an apostrophe in the assembly listing. A G in the listing indicates that the value is external, or that a global is added to that value. Thus, the binary text output looks as follows:

```
005065      CLR     EXTERNAL(R5)     ;VALUE OF EXTERNAL SYMBOL
000000G
                                     ;ASSEMBLED ZERO; WILL BE
                                     ;MODIFIED BY THE LINKER.

005065      CLR     EXTERNAL+6(R5)   ;THE ABSOLUTE PORTION OF THE
000006G
                                     ;EXPRESSION (000006) IS ADDED
                                     ;BY THE LINKER TO THE VALUE OF
                                     ;THE EXTERNAL SYMBOL

005065      CLR RELOCATABLE(R5)      ;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; 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 any general address specification which produces a 6-bit mode address field as described in the *H11 Operation Manual.*

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
                ;THE WORD AT THE ADDRESS
CLR     (R1)    ;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     (R2)+   ;CONTAINED IN THE SPECIFIED
                ;REGISTER AND INCREMENTS
                ;THAT REGISTER'S CONTENTS
                ;BY TWO.

CLRB    (R4)+   ;CLEARS THE BYTE AT THE
                ;ADDRESS SPECIFIED BY THE
                ;CONTENTS OF R4 AND
                ;INCREMENTS R4 BY ONE.
```

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

MOV     R0,–(R0)     ;THE QUANTITY 76 IS MOVED
                     ;TO LOCATION 76
```

The use of these forms should be avoided as they are not compatible with other PDP-11 models.

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     @(R3)+  ;CONTENTS OF REGISTER 3 POINT
                             ;TO ADDRESS OF WORD TO BE
                             ;CLEARED, AND REGISTER 3 IS
                             ;THEN 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 auto-increment mode).

Format for A: –(ER)

```
Examples:    CLR     –(R0)   ;DECREMENT CONTENTS OF
             CLR     –(R0+3) ;0, 3, AND 2 BY TWO
             CLR     –(R2)   ;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     @-(R2)   ;DECREMENT CONTENTS OF
                             ;REGISTER 2 BY TWO BEFORE
                             ;USING AS A 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(R3)   ;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(R4)  ;IF REGISTER 4 HOLDS 100 AND
                             ;LOC 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 THE SYMBOL X TO
                             ;REGISTER 0
```

The operation of this mode can be explained by the following example. The statement MOV #100,R3 assembles as two words. These are:

```
012703
000100
```

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 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      ;MOV THE 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:        016703
Location 22:        000054
```

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 memory.

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 assembles into 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.

Form	Mode	Meaning
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

n represents the register number.

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

Form	Mode	Meaning
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

n represents 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 .ENABL 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 1-word instructions. The high byte contains the op code and the low byte contains an 8-bit signed offset which specifies the branch address relative to the PC. Upon execution of a branch instruction, 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 instructions 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 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 program through the .LIST and .NLIST directives. These are of the form:

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

where arg represents one or more optional arguments.

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

Argument	Default	Function
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 (supersedes BEX).
BEX	list	Controls listing of binary extensions; that is, prevents listing those locations and binary contents beyond the first line of an expansion. 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.
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.
TOC	list	Controls 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	Terminal mode	Controls listing output format. 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 Terminal 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 an 80-column line printer is shown in Figure 5-3 (this is the same format as a terminal listing). 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, 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.

HTEXEC HT-11 ASSEMBL H01-1 7-SEP-78 17:26:14 PAGE 21
GET PHYSICAL SOURCE LINE

```
HTEXEC HT-11 ASSEMBL H01-1    7-SEP-78  17:26:14  PAGE 21
GET PHYSICAL SOURCE LINE

 1                                               .SBTTL  GET PHYSICAL SOURCE LINE
 2
 3         104240                                WINST=EMT+240
 4 001764                               GETPLI:
 5 001764  104403                                TRAP    SRC
 6 001766  005000                                CLR     R0
 7 001770  032737  000004   000012'              BIT     #IO.EOF,IOFTBL+SRCCHN    ;END OF FILE?
 8 001776  001424                                BEQ     2$                  ;NO
 9 002000  013700  002362'                       MOV     CHAN+SRCCHN,R0      ;GET CURRENT INPUT CHAN
10 002004  005200                                INC     R0                  ;MOVE TO NEXT CHAN
11 002006  020027  000010                        CMP     R0,#8.              ;LAST CHAN?
12 002012  101017                                BHI     1$                  ;YES, FLAG END OF INPUT
13 002014  005037  000026'                       CLR     RECNUM+SRCCHN       ;RESET RECORD (BLK) NUMBER
14 002020  013737  002310'  '002306'             MOV     BLKTBL+<SRCCHN*4>,PTRTBL+<SRCCHN*4>
15 002026  010037  002362'                       MOV     R0,CHAN+SRCCHN
16 002032  052700  104240                        BIS     #WINST,R0           ;CREATE A WAIT CALL FOR NEXT CHAN
17 002036  010017                                MOV     R0,@PC              ;AND STORE IN NEXT LOCATION
18 002040  104240                                WINST
19 002042  103403                                BCS     1$                  ;BRANCH IF NO MORE INPUT
20 002044  012700  177777                        MOV     #-1,R0              ;FLAG END OF FILE
21 002050                               2$:      RETURN
22 002052  012700  000001               1$:      MOV     #1,R0               ;FLAG END OF INPUT
23 002056                                        RETURN
```

Figure 5-2 Example of ASEMBL Line Printer Listing (132-Column Line Printer)

```
HTEXEC  HT-11 ASSEMBL H01-1   5-SEP-78 22:30:23 PAGE 21
GET PHYSICAL SOURCE LINE

1                                 .SBTTL   GET PHYSICAL SOURCE LINE
2
3            104240               WINST=EMT+240
4 001764               GETPLI:
5 001764     104403               TRAP     SRC
6 001766     005000               CLR      R0
7 001770     032737               BIT      #IO.EOF,IOFTBL+SRCCHN     ;END OF FILE?
             000004
             000012'
8 001776     001424               BEQ      2$                   ;NO
9 002000     013700               MOV      CHAN+SRCCHN,R0       ;GET CURRENT INPUT CHAN
             002362'
10 02004     005200               INC      R0                   ;MOVE TO NEXT CHAN
11 02006     020027               CMP      R0,#8.               ;LAST CHAN?
             000010
12 02012     101017               BHI      1$                   ;YES, FLAG END OF INPUT
13 02014     005037               CLR      RECNUM+SRCCHN        ;RESET RECORD (BLK) NUMBER
             000026'
14 02020     013737               MOV      BLKTBL+<SRCCHN*4>,PTRTBL+<SRCCHN*4>
             002310'
             002306'
15 02026     010037               MOV      R0,CHAN+SRCCHN
             002362'
16 02032     052700               BIS      #WINST,R0            ;CREATE A WAIT CALL FOR NEXT CHA
             104240
17 02036     010017               MOV      R0,@PC               ;AND STORE IN NEXT LOCATION
18 02040     104240               WINST
19 02042     103403               BCS      1$                   ;BRANCH IF NO MORE INPUT
20 02044     012700               MOV      #-1,R0               ;FLAG END OF FILE
             177777
21 02050               2$:        RETURN
22 02052     012700    1$:        MOV      #1,R0                ;FLAG END OF INPUT
             000001
23 02056                          RETURN
```

Figure 5-3 Example of Page Heading from ASEMBL 80-Column Line Printer
(same format as Terminal Listing)

```
HTEXEC HT-11 ASEMBL H01-1  5-SEP-78  22:30:23  PAGE 29+
SYMBOL TABLE

ABSEXP=   ****** G          ARGCNT=   ****** G          ASSEM=    ****** G
BINCHN=   000004            BINDAT    002322R    004    BLKTBL=   002310R    004
BPMB   =  000020            BUFTBL    000374RG   003    CHAN      002362R    004
CHNSPC    000312R    003    CHRPNT=   ****** G          CLK50  =  000040
CMILEN=   000123            CNTTBL    000360RG   003    CONFIG=   000300
CONT      000040RG   010    CORERR    001726R    010    CPL    =  000120
CR     =  000015            CRFBUF    002076RG   004    CRFC   =  000040
CRFCHN=   000012            CRFCNT    000004RG   007    CRFDAT    002352R    004
CRFE   =  000100            CRFFLG    000000R    007    CRFLEN=   000204
CRFM   =  000010            CRFP   =  000020            CRFPNT    000064R    003
CRFR   =  000004            CRFS   =  000002            CRFSPC    000114R    003
CRFTAB    000026R    003    CRFTST    000002RG   007    CSIERR    000214R    003
CTLTBL    000000R    003    DATE      001000R    010    DATTIM    001004RG   004
DEFEXT    000104R    003    DEVFUL    000252R    003    DIV60     001240R    010
DNC    =  ****** G          EDMASK=   ****** G          ED.ABS=   ****** G
EMTERR=   000052            ENDP1  =  ****** G          ENDP2 =   ****** G
ENDSWT    000434R    010    ERR       001662R    010    ERRB      000102R    010
ERRBTS=   ****** G          ERRCNT=   ****** G          EXMFLG=   ****** G
FF     =  000014            FILNF     000264R    003    FIN       001434RG   010
FINCL     001636R    010    FINMSG    001030R    004    FINMS1    001052R    004
FINMS2    001070R    004    FINP1     000776R    010    FINP2     000776R    010
FINSML    002124RG   010    FRECOR    000006R    007    GETPLI    001756RG   010
GETR50=   ****** G          GSARG  =  ****** G          HDRTTL    001102RG   004
HIGHAD=   000050            ILLCMD    000226R    003    ILLDEV    000240R    003
IMPURT    000042R    007    IMPUR$    000000R    007    INIOF     000106R    010
                    .
                    .
                    .
HTEXEC HT-11 ASEMBL H01-1  5-SEP-78  22:30:23  PAGE 29+
SYMBOL TABLE

TIME      000210R    003    TIMTIM    001016R    004    TIMWRD    000204R    003
TMPCNT=   000014            TSTSTK    001704RG   010    TTLBRK=   ****** G
TTLBUF=   ****** G          TTLLEN=   000040            TTYBUF=   000616
USRLOC=   000046            VT     =  000013            WINST  =  104240
WRTERR    002306R    010    XBAW   =  000000            XEDPIC =  000000
XMIT0  =  ****** G          $CLOUT    003006RG   010    $EDABL=   ****** G
$FLUSH    002732RG   010    $NLIST=   ****** G          $READ     002422RG   010
$READW    002422RG   010    $WAIT     002730RG   010    $WRITE    002134RG   010
$WRITW    002134RG   010
. ABS.    000000     000
          000000     001
DPURE     000000     002
DPURE$    000410     003
MIXED$    002376     004
SWTSE$    000000     005
SWTSEC    000000     006
IMPUR$    000042     007
MAIN$     003024     010
ERRORS DETECTED:  0
FREE MEMORY:  13431. WORDS
,LP:/C/L:BEX=RP4:RTPAR, RPARAM, RCIOCH, RTEXEC
```

Figure 5-4   Symbol Table

5.5.1.2 Page Headings – The assembler outputs each page in the format shown in Figure 5-3. On the first line of each listing page the assembler prints (from left to right):

1. title taken from .TITLE directive (most recent one encountered)
2. assembler version identification
3. the date and time of day if entered
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 print a heading in the output listing and to assign a name to the object module. The heading printed on the first line of each page of the listing is taken from the first 31 characters of the argument in the .TITLE directive. The first six characters (symbol name) of this same line are also used as the name of the object module. These six characters must be Radix-50 characters (any characters beyond the first six are ignored). Non-Radix-50 characters are not acceptable.

For example:

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

causes PROG TO PERFORM DAILY ACCOUNTIN to be printed in the heading for each page and causes the object module of the assembled program to be 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, ASEMBL 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 a table of contents is shown in Figure 5-5.

```
.MAIN. HT-11 ASEMBL H01-1  5-SEP-78  22:30:23
TABLE OF CONTENTS

 1- 29      HT-11 MACRO PARAMETER FILE
 1- 37      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 not used or supported by the HT-11 system, but is handled by ASEMBL for compatibility with other systems. .IDENT provides a means of labeling the object module produced as a result of an assembly. In addition to the name assigned to the object module with the .TITLE directive, a character string (up to six characters, treated like a .RAD50 string) can be specified between paired delimiters. For example:

```
.IDENT /V005A/
```

The character string:

V005A

is converted to Radix-50 notation and output to the global symbol directory of the object module.

When more than one .IDENT directive is found in a given program, the last .IDENT found determines the symbol which is passed as part of the object module identification.

5.5.1.6 Page Ejection (.PAGE Directive) – There are several means of obtaining a page eject in an assembly listing:

1. After a line count of 58 lines, ASEMBL 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.
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.

5.5.2 Functions: .ENABL and .DSABL Directives

Several functions are provided by ASEMBL through the .ENABL and .DSABL directives. These directives use 3-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 list describes the symbolic arguments and their associated functions in the MACRO language:

Symbolic Argument	Function
ABS	Enabling of this function produces absolute binary output; (i.e., for input to the Paper Tape Software System absolute binary loader using a .BIN extension instead of .OBJ). The default case is .DSABL ABS.
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.
CDR	The statement .ENABL CDR causes source columns 73 and greater to be treated as comments. This accommodates sequence numbers in columns 72-80.
FPT	Enabling of this function 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.

Symbolic Argument	Function
LSB	Enable or disable a local symbol block. 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 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
.ASCIZ
.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
                        ;EXPRESSIONS IN SUCCESSIVE BYTES.
```

A legal expression must have an absolute value (or contain a reference to an external symbol) and must result in eight 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 eight 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 eight bits, in which case, the Linker prints an error message. For example:

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

Here, X has an absolute value,

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

and can be linked later with another program:

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

If an operand following the .BYTE directive is null, it is interpreted as a zero. For example (assume assembly begins at relocatable 0):

```
.=.+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 EXPRESSON
                        ;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, 0 IN LOCATIONS
                             ;500, 502, AND 504
```

A blank operator field (any operator not recognized as an 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
LABEL:  +MOV,LABEL           ;010000, IS STORED IN 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 an instruction mnemonic or assembler directive. Therefore, if an instruction mnemonic 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: A and U. 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 5.6.3.3.)

```
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 binary word being moved into R0:

```
0100001001000001
```

Note that the first character is placed in the low-order byte and the second character in the high-order byte.

The " character is never followed by a carriage return, null, rubout, line feed, or form feed. For example:

```
;DEVICE NAME TABLE

DEVNAM:  .WORD    "DX         ;RX DISK
DEVNKB:  .WORD    "TT         ;TERMINAL KEYBOARD
         .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/
```

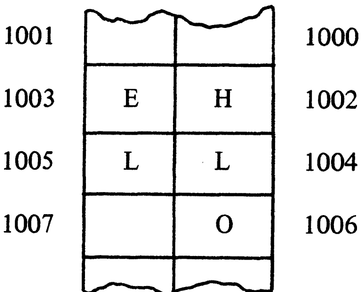
where: character string — is a string of any acceptable printing ASCII characters including spaces. The string may not include null characters, rubout, return, line feed, vertical tab, or form feed. Nonprinting characters can be expressed in digits of the current radix and delimited by angle brackets. (Any legal, defined expression is allowed between angle brackets.)

/ / — 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
```

The order of the characters as they are stored in memory is illustrated below.




```
.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:

	Directive	Result	Explanation
.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 an 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 as follows:

```
CR=15
LF=12
.
.
.
        MOV     #HELLO,R1
```

```
                MOV        #LINBUF, R2
X:              MOVB       (R1) +, (R2) +           ;MOVE A CHARACTER OF THE
                                                    ;MESSAGE STRING INTO THE
                                                    ;OUTPUT BUFFER
                BNE X                               ;BRANCH BACK IF BYTE
                                                    ;NOT EQUAL TO 0
.
.
.
HELLO:          .ASCIZ     <CR><LF>/ASEMBL-11  H01-1/<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 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. (A warning code is printed if it is not a matching delimiter, however.) For example:

```
****** A
20 00040  003223      .RAD50    /ABC
21                                        ;PACK ABC INTO ONE WORD
22 00042  003220      .RAD50    /AB/      ;PACK AB (SPACE) INTO ONE WORD.
23 00044  000000      .RAD50    / /       ;PACK THREE SPACES INTO ONE WORD
24 00046  003223      .RAD50    /ABCD/    ;PACK ABC INTO FIRST WORD AND
   00050  014400
                                          ;D (SPACE) (SPACE) INTO SECOND WORD
```

Each character is translated into its Radix-50 equivalent as indicated:

Character	Radix-50 Equivalent (octal)	ASCII (octal)
(space)	0	40
A-Z	1-32	101-132
$	33	44
.	34	56
undefined	35	undefined
0-9	36-47	60-71

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:

Radix-50 value = ((C1*50) + C2)*50 + C3

For example:

Radix-50 value of ABC is ((1*50) + 2)*50 + 3 or 3223

See Appendix D for a table to quickly determine Radix-50 equivalents.

Use of angle brackets is encouraged in the .ASCII, .ASCIZ, 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 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
```

A given radix is valid throughout a program until changed. Where a possible conflict exists within 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: ↑D, ↑O, and ↑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. For example, the creation of a mask word might best be done in the binary radix.

ASEMBL has three unary operators to provide a single interpretation in a given radix within another radix as follows:

↑Dx	(x is treated as being in decimal radix)
↑Ox	(x is treated as being in octal radix)
↑Bx	(x is treated as being in binary radix)

For example:

```
↑D123
↑O 47
↑B 00001101
↑O<A+3>
```

Notice that while the uparrow 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.

A temporary radix change from octal to decimal may be made 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 that 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
              .BYTE N*2                 ;COUNT=2N BYTES
BUFF:         .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
              MOVB –1(R2), R0           ;GET COUNT STORED IN BUFF –1 IN R0
AGAIN:        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
              .
              .
              CLRB –(R2)                ;OUT OF ROOM IN BUFFER, CLEAR LAST
DONE:                                   ;WORD
              .
              .
LINE:         .ASCIZ /TEXT/
```

In this case, .ODD is used to place the buffer byte count in the byte preceding the buffer, as follows:

COUNT	//////////	BUFF–2
		BUFF

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
 2
 3
 4            000000'              .CSECT IMPURE
 5   000000              PASS:     .BLKW
 6                                                ;NEXT GROUP MUST STAY TOGETHER
 7   000002              SYMBOL:   .BLKW 2        ;SYMBOL ACCUMULATOR
 8   000006              MODE:
 9   000006              FLAGS:    .BLKB 1        ;FLAG BITS
10    00007              SECTOR:   .BLKB 1        ;SYMBOL/EXPRESSION TYPE
11    00010              VALUE:    .BLKW 1        ;EXPRESSION VALUE
12    00012              RELLVL:   .BLKW 1
13                                 .BLKW 2        ;END OF GROUPED DATA
14    00020              CLCNAM:   .BLKW 2        ;CURRENT LOCATION COUNTER
15    00024              CLCFGS:   .BLKB 1
16    00025              CLCSEC:   .BLKB 1
17    00026              CLCLOC:   .BLKW 1
18    00030              CLCMAX:   .BLKW 1
19            000001'              .END
```

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 valid only 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.

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. These two directives are of the form:

```
.FLT2    arg1,arg2,. . .
.FLT4    arg1,arg2,. . .
```

where arg1, arg2, etc. represent 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 shows the use of the .FLT4 directive:

```
1
2
3
4       000000   037314    ATOFTB:    .FLT4 1.E-1      ;10^-1
        000002   146314
        000004   146314
        000006   146315
5       000010   036443               .FLT4 1.E-2      ;10^-2
        000012   153412
        000014   036560
        000016   121727
6       000020   034721               .FLT4 1.E-4      ;10^-4
        000022   133427
        000024   054342
        000026   014545
7       000030   031453               .FLT4 1.E-8      ;10^-8
        000032   146167
        000034   010604
        000036   060717
8       000040   022746               .FLT4 1.E-16     ;10^-16
        000042   112624
        000044   137304
        000046   046741
9       000050   005517               .FLT4 1.E-32     ;10^-32
        000052   130436
        000054   126505
        000056   034625
```

5.5.6.2 Temporary Numeric Control: ↑F and ↑C – Like the temporary radix control operators, operators are available to specify either a 1-word floating-point number or the one's complement of a 1-word number (↑C). For example:

```
FL3.7:     ↑F3.7
```

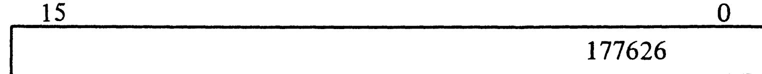
creates a 1-word floating-point number at location FL3.7 containing the value 3.7 as follows:

	15	14 ... 7	6 ... 0
SIGN BIT	S	EXPONENT	MANTISSA

This 1-word floating-point number is the first word of the 2- or 4-word floating-point number format shown in the *H11 Operation Manual,* and the statement:

```
CMP151:     ↑C151
```

stores the one's complement of 151 in the current radix (assume current radix is octal) as follows:



Since these control operators are unary operators, their arguments may be integer constants or symbols, and the operators may be expressed successively. For example:

```
↑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    = 040200
 ↑F-1.0   = 140200
-↑F1.0    = 137600
-↑F-1.0   = 037600
```

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.

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 exp statement should terminate the first object module and .END statements should terminate any other object modules.

5.5.7.2 .EOT – Under the HT-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 load module is the result of the link). 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.

The .ASECT and .CSECT program section directives are provided in ASEMBL to allow the user to specify an absolute or relocatable section. These directives are formatted as follows:

```
.ASECT
.CSECT
.CSECT  symbol
```

The single absolute section can be declared with an:

```
.ASECT
```

directive. No name can be associated with the absolute section specified by means of the .ASECT directive. The single unnamed relocatable program section can be declared with a:

```
.CSECT
```

directive.

All named 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, etc.	are legal symbolic names, separated by commas, tabs, 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.

```
;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.:

```
D=6
CLR      EXT+D
.WORD    EXT-2
CLR      @EXT+D
```

A global symbol defined within the program can be used in the evaluation of a direct assignment statement, but an external symbol cannot. Since ASEMBL determines at the end of pass 1 whether a given global symbol is defined within the program or is expected to be external, a construction such as the following will cause errors at link time:

```
          .GLOBL FREE
          .GLOBL LIMITS
FREE=LIMITS+2                    ;FREE WILL NOT BE
          .                      ;DEFINED UNTIL PASS 2
          .
          .
LIMITS:   .LIMIT
          .
          .
          .
```

FREE will be flagged as an undefined global at link time. To allow correct linking, define FREE after LIMITS:.

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			
Positive	**Complement**	**Arguments**	**Assemble Block If**
EQ	NE	expression	expression=0 (or ≠ 0)
GT	LE	expression	expression>0 (or ≤ 0)
LT	GE	expression	expression<0 (or ≥ 0)

Conditions		Arguments	Assemble Block If
Positive	**Complement**		
DF	NDF	symbolic argument	symbol is defined (or undefined)
B	NB	macro-type argument	argument is blank (or nonblank)
Z	NZ	expression	same as EQ/NE
G		expression	same as GT/LE
L		expression	same as LT/GE

NOTE

A macro-type argument is enclosed in angle brackets or within an up-arrow construction (as described in Section 5.2.1.1). For example:

```
<A,B,C>
↑/124/
```

For example:

```
ALPHA=-1
        .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     ;ASSEMBLE IF BOTH SYM1
 .                          ;AND SYM2 ARE DEFINED
 .
.ENDC
```

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:

Subconditional	Function
.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                      ;ASSEMBLE THE FOLLOWING CODE ONLY IF
                          ;SYM IS UNDEFINED
 .
 .
.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
                         ;IGNORED WITHIN NESTED. UNSATISFIED
  .                      ;CONDITIONAL BLOCK
  .
  .
.IFT                     ;NOT SEEN
  .
  .
.ENDC
.ENDC
```

However,

```
.IF       DF, X          ;TESTS TRUE
.IF       DF, Y          ;TESTS FALSE
.IFF                     ;IS ASSEMBLED
                         ;OUTER CONDITIONAL SATISFIED.
  .
  .
.IFT                     ;NOT ASSEMBLED
  .
  .
.ENDC
.ENDC
```

5.5.11.2 Immediate Conditionals – An immediate conditional directive is a means of writing a 1-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 5.5.11.

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, as in the following example:

```
LABEL:
                .IIF  DF,FPP  BEQ ALPHA
```

or included as part of the conditional statement:

```
.IFF            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 ASEMBL. It is advisable that future programs be developed using the format for ASEMBL conditional assembly directives.

Directive	Arguments	Assemble Block If
.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 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 WITH THE EXPAND UTILITY PROGRAM

EXPAND is a system program which processes the macro references in an assembly language source file. Using the system library file SYSMAC.SML, EXPAND produces an output file (with a .PAL extension) in which all legal macro references are expanded into macro-free source code. ASEMBL can then process the program.

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.

In general it is recommended that macro definitions not contain or rely on radix settings from the .RADIX directive (Section 5.5.4.1). The temporary radix control characters should be used within a macro definition. (↑D, ↑O, and ↑B are described in Section 5.5.4.2).

5.6.1.1 .MACRO – The first statement of a macro definition must be a .MACRO directive. 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 must begin with a dot (.) and 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). Dummy argument names must begin with a dot (.).

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 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.

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 .MESSAGE        ;TYPE A MESSAGE
JSR         R5,.TYPMSG
.WORD       .MESSAGE
.ENDM
```

5.6.1.3 Macro Definition Formatting – A form feed character used as a line terminator in a macro source statement (or as the only character on a line), causes a page eject when the source program is listed. Used within a macro definition, a form feed character also causes a page eject. A page eject is not performed, however, 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. (Refer to Section 5.2.1.1.)
	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.

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      ;MACRO DEFINITION
  .
  .
  .
.REN     ALPHA,BETA,<C1,C2> ;MACRO CALL
```

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 would 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.

5.6.3.1 Special Characters – Arguments may include special characters without enclosing the argument in a bracket construction if that argument does not contain spaces, tabs, semicolons, or commas. For example:

```
.MACRO  .PUSH   .ARG
MOV     .ARG, -(SP)
.ENDM
  .
  .
  .
.PUSH     X+3(%2)
```

generates the following code:

```
MOV     X+3(%2), -(SP)
```

5.6.3.2 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 .IF B and .IF NB can be used within the macro to detect null arguments.

A macro can be defined with 0 to 30 arguments.

5.6.3.3 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, <EXPAND-11>
```

it expands as follows:

```
XY:                 .ASCIZ  /EXPAND-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 example is 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 expansion. The scan then encounters the second ' character which is followed by a dummy argument and is discarded. The scan of the argument .A terminates 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.)

5.6.4 Macro Libraries: .MCALL

All macro definitions must occur prior to their referencing within the user program. EXPAND 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. 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, and arg2, etc. are the names of the macro definitions required in the current program.

When this directive is encountered, EXPAND searches the system library file, SYSMAC.SML, to find the requested definition(s). EXPAND searches for SYSMAC.SML on the system device (SY:).

See Appendix C for a listing of the system macro file (SYSMAC.SML) stored on the system device.

5.7 CALLING AND USING EXPAND

To run EXPAND, type:

```
R EXPAND
```

in response to the dot printed by the Keyboard Monitor. 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 HT-11 command string syntax (dev:filnam.ext). The default value for each file specification is noted below:

I/O File	Dev	Ext
ofile	DK	PAL
ifile1,. . ., ifile6	device used for last source file specified or DK	MAC

Type CTRL C to halt EXPAND and return control to the monitor. To restart EXPAND, type R EXPAND or the REENTER command in response to the monitor's dot.

EXPAND copies sequentially the specified input files 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 library file (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 reference, 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. EXPAND 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 EXPAND output.

INPUT

```
R1=%1
SP=%6
PC=%7
.MACRO .CALL .SUBR
JSR PC, .SUBR
.ENDM
.MCALL .LOOKUP, .READ
```

OUTPUT

```
;HT-11 MACRO EXPAND H01-1

                R1=%1
                SP=%6
                PC=%7
;               .MACRO .CALL .SUBR
;               JSR PC, .SUBR
;               .ENDM
;               .MCALL .LOOKUP, .READ
;.MACRO .LOOKUP .AREA, .CHAN, .DEVBLK, .SPF
;.IF DF . . .V1
;.IF NB .CHAN
;               MOV     .CHAN, %0
;.ENDC
;               EMT     ^O<20+.AREA>
;.IFF
;.IF NB .AREA
;               MOV     .AREA,%0
;               MOVB    #1, 1(0)
;.ENDC
;.IF NB .CHAN
;               MOVB    .CHAN, (0)
;.ENDC
;.IF NB .DEVBLK
;               MOV     .DEVBLK, 2. (0)
;.ENDC
;.IF NB .SPF
;               MOV     .SPF, 4. (0)
;.IFF
;               CLR     4. (0)
;.ENDC
```

```
                                ;          EMT      ^O375
                                ;.ENDC
                                ;.ENDM
                                ;.MACRO .READ .AREA, .CHAN, .BUFF, .WCNT, .BLK
                                ;.IF DF . . .V1
                                ;.IF NB.WCNT
                                ;          MOV      .WCNT,%0
                                ;.ENDC
                                ;          MOV      #1, –(6.)
                                ;          MOV      .BUFF, –(6.)
                                ;          MOV      .CHAN, –(6.)
                                ;          FMT      ^O<200+.AREA>
                                ;.IFF
                                ;.IF NB .AREA
                                ;          MOV      .AREA, %0
                                ;          MOVB     #8., 1(0)
                                ;.ENDC
                                ;.IF NB .CHAN
                                ;          MOVB     .CHAN, (0)
                                ;.ENDC
                                ;.IF NB .BLK
                                ;          MOV      .BLK, 2. (0)
                                ;.ENDC
                                ;.IF NB .BUFF
                                ;          MOV      .BUFF, 4. (0)
                                ;.ENDC
                                ;.IF NB .WCNT
                                ;          MOV      .WCNT, 6. (0)
                                ;.ENDC
                                ;          MOV      #1, 8. (0)
                                ;          EMT      ^O375
                                ;.ENDC
                                ;.ENDM

         .CSECT MAIN                       .CSECT MAIN
         .GLOBL SQRT                       .GLOBL SQRT
STACK:   .BLKW 100              STACK:     .BLKW 100
AREA:    .BLKW 10               AREA:      .BLKW 10
BUFR:    .BLKW 100              BUFR:      .BLKW 100
INBLK:   .BLKW 5                INBLK:     .BLKW 5
START:   MOV #STACK,SP          START:     MOV #STACK, SP
A:       MOV R1, –(SP)          A:         MOV R1, –(SP)
B:       .CALL SQRT             B:;        .CALL SQRT
                                           JSR PC, SQRT
         .LOOKUP #INBLK, 0      ;          .LOOKUP #INBLK, 0
                                .IF DF . . .V1
                                .IF NB 0
                                           MOV      0, %0
                                .ENDC
                                           EMT      ^O<20+#INBLK>
                                .IFF
                                .IF NB #INBLK
```

```
                                                        MOV         #INBLK, %0
                                                        MOVB        #1, 1(0)
                                    .ENDC
                                    .IF NB 0
                                                        MOVB        0, (0)
                                    .ENDC
                                    .IF NB
                                                        MOV         ,2. (0)
                                    .ENDC
                                    .IF NB
                                                        MOV         ,4. (0)
                                    .IFF
                                                        CLR         4. (0)
                                    .ENDC
                                                        EMT         ^O375
                                    .ENDC
CLR R1          ;BLOCK NUMBER                   CLR R1          ;BLOCK NUMBER
.READ #AREA, #0, #BUFR, #256., R1   ;           .READ #AREA, #0, #BUFR, #256., R1
                                    .IF DF . . . V1
                                    .IF NB #256.
                                                        MOV         #256., %0
                                    .ENDC
                                                        MOV         #1, -(6.)
                                                        MOV         #BUFR, -(6.)
                                                        MOV         #0, -(6.)
                                                        EMT         ^O<200+#AREA>
                                    .IFF
                                    .IF NB #AREA
                                                        MOV         #AREA, %0
                                                        MOVB        #8., 1(0)
                                    .ENDC
                                    .IF NB #0
                                                        MOVB        #0, (0)
                                    .ENDC
                                    .IF NB R1
                                                        MOV         R1, 2. (0)
                                    .ENDC
                                    .IF NB #BUFR
                                                        MOV         #BUFR, 4. (0)
                                    .ENDC
                                    .IF NB #256.
                                                        MOV         #256., 6. (0)
                                    .ENDC
                                                        MOV         #1, 8. (0)
                                                        EMT         ^O375
                                    .ENDC
HALT                                            HALT
.END START                                      .END START
```

5.8 CALLING AND USING ASEMBL

The assembler assembles one or more macro-free ASCII source files into a single relocatable binary object file. Assembler output consists of this binary object file and an optional assembly listing followed by the symbol table listing. CREF (Cross Reference) listings may also be specified as part of the assembly output by means of switch options.

ASEMBL is executed using the HT-11 Monitor R command as follows:

```
.R ASEMBL
```

The assembler responds by typing an asterisk (*) to indicate readiness to accept command string input. 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) in a command line as follows:

```
*dev:obj,dev:list/s:arg=dev:source1,. .,dev:sourcen/s:arg
```

where:

dev:	is any legal HT-11 device for output; must be file-structured for input
obj	is the binary object file
list	is the assembly listing file containing the assembly listing and symbol table
source 1, . .,sourcen	are the ASCII source files containing the ASEMBL source program(s); a maximum of six source files is allowed
/s:arg	represents a switch and argument as explained in Section 5.8.1

A null specification in either of the output file fields signifies that the associated output file is not desired.

One or more switches can be indicated with the appropriate file specification to provide ASEMBL with information about that file.

The default case for each file specification is noted below:

file	device	filename	extension
object	DK:	–	.OBJ
listing	device used for object output	–	.LST
source1	DK:	–	.PAL
source2 . . . sourcen	device used for last source file specified	–	.PAL

Type CRTL C to halt ASEMBL at any time and return control to the monitor. To restart the assembler type R ASEMBL or the REENTER command in response to the monitor's dot.

NOTE

If ↑C was typed while a CREF listing was being produced, the REENTER command may not be accepted. In this case, type R ASEMBL to restart the assembler.

5.8.1 Switches

There are three types of switch options: listing control switches, function switches, and CREF specification switches. The listing control switches (/L,/N) provide capabilities similar to those described in detail in Section 5.5.1.1. The function control switches (/D,/E) provide function control as described in Section 5.5.2; arguments for these switches are summarized in Section 5.8.1.2. CREF control switches allow the user to obtain a detailed cross-referenced listing of his assembled file, and are described in detail in Section 5.8.1.3. Multiple arguments may be specified for a particular switch, if desired, by separating each switch value from the next by a colon. For example:

```
/N:TTM:CND
```

These switches turn off teleprinter mode and suppress printing of unsatisfied conditionals (as described in the next section). Also, the switches are not restricted to appearing near a particular file in the command string; /N:TTM, for example, is legal in all of the following places:

```
*,LP:/N:TTM=source
*,LP:=source/N:TTM
*/N:TTM,LP:=source
```

and they are all equivalent in function.

5.8.1.1 Listing Control Switches – A listing control switch (/L for list or /N for nolist) is indicated in a command line as follows:

```
*dev:obj.ext,dev:list.ext/s:arg=dev:source.ext
```

where s:arg represents /L or /N; the remainder of the command line abbreviations are as described in Section 5.8.

The /N with no argument causes only the symbol table, table of contents and error listings to be produced. The /L switch with no arguments causes .LIST and .NLIST directives that appear in the source program but have no arguments to be ignored. A summary of the arguments which are valid for the listing control switches follows (refer to Section 5.5.1.1 for details):

Argument	Default	Controls Listing of
SEQ	list	Source line sequence numbers
LOC	list	Location counter
BIN	list	Generated binary code
BEX	list	Binary extensions
SRC	list	Source code
COM	list	Comments
CND	list	Unsatisfied conditionals, .IF and .ENDC statements

Argument	**Default**	**Controls Listing of**
LD	nolist	Listing directives with no arguments
TOC	list	Table of Contents
TTM	terminal mode	Listing output format
SYM	list	Symbol table

For example, a command line in the following format suppresses the Table of Contents listing and uses the 132-column listing format:

```
*,LP:/N:TOC:TTM=FILE
```

5.8.1.2 Function Switches – The function control switches (/D for disable and /E for enable) are used to enable or disable certain functions in source input files and are indicated in the command line as follows:

```
*dev:obj.ext,dev:list.ext=dev:source/s:arg
```

/s:arg here represents either /D:arg or /E:arg. A summary of the arguments which are valid for use with the function control switches follows (refer to Section 5.5.2 for details):

Argument	**Default**	**Enables or Disables**
ABS	disable	Absolute binary output
AMA	disable	Assembly of all absolute addresses as relative addresses
CDR	disable	Source columns 73 and greater to be treated as comments
FPT	disable	Floating point truncation
LC	disable	Accepts lower-case ASCII input
LSB	disable	Local symbol block
PNC	enable	Binary output

For example, the following commands assemble a file allowing all 80 columns of each line to be used as input (note that since ASEMBL is a two-pass assembler, the paper tape cannot be read directly from the reader; input from any nonfile-structured device must first be transferred to a file-structured device before assembly):

```
.R PIP
*PTAPE. PAL=PR:/A
*^C

.R ASEMBL
*,LP:=PTAPE. PAL/E:CDR
```

Use of either the function control or listing control switches and arguments at assembly-time will override any corresponding listing or function control directives and arguments in the source program. For example, assume the following appears in the source program:

```
.NLIST BEX
   .
   .              code with binary extensions
   .
.LIST BEX
```

The "BEX" printing will be disabled for the block indicated; however, if /L:BEX is indicated in the assembly command line, both the .NLIST BEX and the .LIST BEX will be ignored and the "BEX" printing will be enabled everywhere in the program.

5.8.1.3 Cross Reference Table Generation (CREF) – A cross reference table of all or a subset of all symbols used in the source program and the statements where they were defined or used can be obtained automatically following an assembly by specifying /C:arg with the assembly listing file specification (and any listing or function control specifications) as follows:

```
*dev:obj.ext,dev:list.ext/s:arg/C:arg=dev:source.ext
```

/s:arg represents any of the other valid switches.

There are five sections to a complete cross reference listing:

1. Cross reference of program symbols (i.e., labels used in the program and symbols used on the left of the "=" operator).
2. Cross reference of register-equate symbols (those symbols which are defined in the program by a "SYMBOL=%N", 0<=N<7, construct). Normally this consists of the symbols R0, R1, R2, R3, R4, R5, SP, and PC.
3. Cross reference of permanent symbols (all operation mnemonics and assembler directives).
4. Cross reference of control sections (those names specified as the operand of a .CSECT directive, plus the blank .CSECT and the absolute section ". ABS." which are always defined by ASEMBL).
5. Cross reference of errors (all errors flagged on the listing are grouped by error type).

Any or all of the above sections may be included in the cross reference listing as desired. The associated switch options and their arguments are listed below:

Switch Argument	Section Type
/C:S	User-defined symbols
/C:R	Register symbols
/C:P	Permanent symbols (instructions, directives)
/C:C	Control sections (.CSECT symbolic names)
/C:E	Error codes
/C<no arg>	Equivalent to /C:S:E

The specification of a /C switch in a command string causes a temporary file, "DK:CREF.TMP", to be generated. If device DK: is write-locked or contains insufficient free space for the temporary file, the user may allocate the temporary file on another device. To do so, a third output file specification is given in the ASEMBL command string; this file is then used instead of DK:CREF.TMP, and is purged after use. For example, a command string of this type:

```
*,LP:,DX1:TEMP.TMP=SOURCE/C
```

causes "DX1:TEMP.TMP" to be used as the temporary file.

Figure 5-6 illustrates assembled source code and Figure 5-7 contains the CREF output. The command line used to produce these listings was:

```
*,EXAMPL/C:S:R:P:C:E/N:BEX=EXAMPL
```

An explanation of the CREF output follows the figures.

```
EXAMPLE OF CROSS REFERENCE LIST HT-11 ASEMBL H01-1     19-SEP-78 PAGE 1

1                      ; HT-11 MACRO EXPAND H01-1
2                              .TITLE  EXAMPLE OF CROSS REFERENCE LISTING
3
4           000000 R0=         %0                              ;DEFINE THE REGISTER SYMBOLS
5           000001 R1=         %1
6           000002 R2=         %2
7           000003 R3=         %3
8           000004 R4=         %4
9           000005 R5=         %5
10          000006 SP=         %6
11          000007 PC=         %7
12
13          000012 LF=         012
14
15                     ;       .MCALL  .TTYIN, .EXIT
16                     ;.MACRO .EXIT
17                     ;               EMT     ^O350
18                     ;.ENDM
19                     ;.MACRO .TTYIN .CHAR
20                     ;               EMT     ^O340
21                     ;               BCS     .-2
22                     ;.ENDM
23
24                     ;       .MACRO  .CALL   .NAME
25                     ;       JSR     PC,.NAME
26                     ;       .ENDM
27
28                             .GLOBL  SUBR1,  SUBR2   ;TWO EXTERNAL SUBROUTINES
29          000000'            .CSECT  PROG            ;DEFINE A CSECT
30 00000    012702 START:      MOV     #BUFFER,R2      ;R2 = ADRS(BUFFER
31 00004           1$:;        .TTYIN                  ;READ A CHAR INTO R0
32 00004    104340                     EMT     ^O340
33 00006    103776                     BCS     .-2
34 00010    110022             MOVB    R0,(R2)+        ;AND STORE IN BUFFER
35 00012    120027             CMPB    R0,#LF          ;WAS IT A LINE FEED?
36 00016    001372             BNE     1$              ;NOPE - KEEP READING
37 00020    105022             CLRB    (R2)+           ;ELSE FLAG END OF LINE
38 00022    012703             MOV     #BUFFER,R3      ;R3 = ADRS FOR SUBR1
39                     ;       .CALL   SUBR1           ;INVOKE "CALL" MACRO
40 00026    004767             JSR     PC,SUBR1
41 00032    103762             BCS     START           ;GET A NEW LINE IF CARRY SET
42                     ;       .CALL   SUBR2           ;ELSE CALL OTHER SUBR
43 00034    004767             JSR     PC,SUBR2
44 00040    010067             MOV     R0,ANSWER       ;STORE RESULT IN ANSWER
45                     ;       .EXIT                   ;RETURN TO HT-11
46 00044    104350                     EMT     ^O350
47
48 00046           ANSWER:     .BLKW                   ;DEFINE ANSWER STORAGE
49 00050           BUFFER:     .BLKB   72.             ;INPUT LINE BUFFER
50
51          000000'            .END    START
```

Figure 5-6 ASEMBL Source Code

```
EXAMPLE OF CROSS REFERENCE LIST HT-11 ASEMBL HØ1-1     19-SEP-78 PAGE 1+
SYMBOL TABLE

ANSWER  ØØØØ46R    ØØ2  BUFFER  ØØØØ5ØR    ØØ2  LF      = ØØØØ12
PC     =%ØØØØØ7         RØ     =%ØØØØØØ         R1     =%ØØØØØ1
R2     =%ØØØØØ2         R3     =%ØØØØØ3         R4     =%ØØØØØ4
R5     =%ØØØØØ5         SP     =%ØØØØØ6         START   ØØØØØØR    ØØ2
SUBR1 = ****** G        SUBR2 = ****** G
. ABS.  ØØØØØØ     ØØØ
        ØØØØØØ     ØØ1
PROG    ØØØ16Ø     ØØ2
ERRORS DETECTED: Ø
FREE MEMORY: 11178. WORDS

,EXAMPL/C:S:R:P:C:E/N:BEX=EXAMPL
```

Figure 5-6 (Cont.) ASEMBL Source Code

```
EXAMPLE OF CROSS REFERENCE LIST HT-11 ASEMBL HØ1-1     19-SEP-78 PAGE S-1
CROSS REFERENCE TABLE (CREF HØ1-1 )


.          1-33
ANSWER     1-44*     1-48#
BUFFER     1-3Ø      1-38      1-49#
LF         1-13#     1-35
START      1-3Ø#     1-41      1-51
SUBR1      1-28#     1-4Ø
SUBR2      1-28#     1-43


EXAMPLE OF CROSS REFERENCE LIST HT-11 ASEMBL HØ1-1     19-SEP-78 PAGE R-1
CROSS REFERENCE TABLE (CREF HØ1-1 )


PC         1-11#     1-4Ø*     1-43*
RØ         1-4#      1-34      1-35      1-44
R1         1-5#
R2         1-6#      1-3Ø*     1-34*     1-37*
R3         1-7#      1-38*
R4         1-8#
R5         1-9#
SP         1-1Ø#


EXAMPLE OF CROSS REFERENCE LIST HT-11 ASEMBL HØ1-1     19-SEP-78 PAGE P-1
CROSS REFERENCE TABLE (CREF HØ1-1 )


.BLKB      1-49
.BLKW      1-48
.CSECT     1-29
.END       1-51
.GLOBL     1-28
.TITLE     1-2
BCS        1-33      1-41
BNE        1-36
CLRB       1-37
CMPB       1-35
EMT        1-32      1-46
JSR        1-4Ø      1-43
MOV        1-3Ø      1-38      1-44
MOVB       1-34
```

Figure 5-7 CREF Listing Output

```
EXAMPLE OF CROSS REFERENCE LIST HT-11 ASEMBL HØ1-1     19-SEP-78 PAGE C-1
CROSS REFERENCE TABLE (CREF HØ1-1 )

         Ø-Ø
. ABS.   Ø-Ø
PROG     1-29
```

Figure 5-7 (Cont.) CREF Listing Output

Cross reference tables, if requested, are generated at the end of an ASEMBL assembly listing. Each table begins on a new page (the tables in Figure 5-7 have been consolidated due to space considerations). Symbols, control sections, and error codes are listed at the left margin of the page; corresponding references are indicated next to them across the page from left to right. A reference is of the form p-l, where p is the page on which the symbol, control section, or error code appears, and l is the line number within the page. A number sign (#) appears next to a reference wherever a symbol has been defined. An asterisk appears next to a reference wherever a destructive reference has been made to the symbol (i.e., the contents of the location defined by that symbol has been altered at that point).

The CREF output requested in the preceding figures included user-defined symbols, control sections, error codes, register symbols, and permanent symbols. Since no errors were generated in this assembly, no CREF output for error codes was produced.

5.9 ERROR MESSAGES

5.9.1 EXPAND Error Messages

The following messages are caused by fatal errors detected by EXPAND. They print on the terminal and cause EXPAND to restart:

Message	Explanation
?BAD SWITCH?	An unrecognized command string switch was specified.
?INPUT ERROR?	Hardware error in reading an input file.
?INSUFFICIENT MEMORY?	Not enough memory to store macro definitions.
?MISSING END IN MACRO?	End of input was encountered while storing a macro definition; probably missing an .ENDM.
?NO INPUT FILE?	There must be at least one input file.
?OUTPUT DEVICE FULL?	No room to continue writing output; try to compress the device with PIP.
?WRONG NUMBER OF OUTPUT FILES?	There must be exactly one output file.

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 in the following form:

?*** ERROR *** message

After each run of EXPAND, the total number of non-fatal errors is printed on the terminal.

Message	Explanation
BAD MACRO ARG	The macro argument is not formatted correctly.
LINE TOO LONG	A line has become longer than 132 characters.
MACRO ALREADY DEFINED	A macro was defined more than once.
MACRO(S) NOT FOUND	Macros listed in an .MCALL statement were not found in SYSMAC.SML (make sure SYSMAC.SML is present on system).
MISSING COMMA IN MACRO ARG	Found spaces or tabs within a macro argument when a comma was expected; try using brackets around the arguments, e.g., <arg with spaces>.
MISSING DOT	A macro name or argument name does not begin with a dot.
NAME DOESN'T MATCH	Optional name given in .ENDM directive does not match name given in corresponding .MACRO directive.
NESTED MACROS	A macro is being defined or invoked within another macro.
NO NAME	A macro definition has no name.
SYNTAX	A macro directive is not constructed correctly.
TOO MANY ARGS	A macro directive has more than 30 arguments.

5.9.2 ASEMBL/CREF Error Messages

ASEMBL 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
```

Error Code	Meaning
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. May also indicate that a local symbol is being defined more than 128 words from the beginning of a local symbol block.
B	Bounding error. Instructions or word data would be assembled at an odd address in memory. The location counter is updated by +1.
D	Multiply-defined symbol referenced. Reference was made to a label (not a local label) that 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 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.
N	Number containing 8 or 9 has decimal point missing.
O	Opcode error. Directive out of context.
P	Phase error. A label's definition or value varies from one pass to another or a local symbol occurred twice within a local symbol block.
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.

Error Message	Explanation
?BAD SWITCH?	The switch specified was not recognized by the program.
?INSUFFICIENT MEMORY?	There are too many symbols in the program being assembled. Try dividing program into separately-assembled subprograms.
?I/O ERROR ON CHANNEL n?	A hardware error occurred while attempting to read from or write to the device on the channel specified in the message. (Channel numbers (0<=n<=10 octal) are assigned to files in the manner described in Section 9.4.4, Chapter 9.) Note that the CREF temporary file is on channel 2 even if it was not specified in the command string (i.e., if the default file DK:CREF.TMP is used).
?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.

All CREF error message begin with C– to distinguish them from ASEMBL error messages. When a CREF error occurs, the error message is printed on the terminal and CREF chains back to ASEMBL; ASEMBL prints an asterisk, at which time another command line may be entered.

Error Message	Explanation
?C–CHAIN–ONLY–CUSP?	An attempt was made either to "R CREF" or to "START" a copy of CREF which was in memory. CREF can only be "chained" to.
?C–CRF FILE ERROR?	An output error occurred while accessing "DK:CREF.TMP", the temporary file passed to CREF.
?C–DEVICE?	An invalid device was specified to CREF.
?C–LST FILE ERROR?	An output error occurred while attempting to write the cross-reference table to the listing file.

CHAPTER 6
LINKER

6.1 INTRODUCTION

The HT-11 Linker converts object modules produced by the HT-11 assembler or FORTRAN IV into a format suitable for loading and execution. This allows the user to separately assemble a main program and each of its 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:

1. Relocate each object module and assign absolute addresses
2. Link the modules by correlating global symbols defined in one module and referenced in another module
3. Create the initial control block for the linked program
4. Create an overlay structure if specified and include the necessary run-time overlay handlers and tables
5. Search user specified libraries to locate unresolved globals
6. Optionally produce a load map showing the layout of the load module

The HT-11 Linker requires two or three 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. If library files are to be linked with input modules, an intermediate pass is needed to force the modules resolved from the library file into the root segment (that part of the program which is never overlaid). During the final pass, the Linker reads the object modules, performs most of the functions listed above, and produces a load module (.LDA for use with the Absolute Loader and save image (.SAV) for the system).

The Linker runs in a minimal HT-11 system of 8K; any additional memory is used to facilitate efficient linking and to extend the symbol table. Input is accepted from any random-access device on the system; there must be at least one random-access device (disk) for save image.

6.2 CALLING AND USING THE LINKER

To call the Linker, type 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 outputs an extra line feed character 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. An extra line feed character precedes the asterisk printed by the CSI.

6.2.1 Command String

The first command string entered in response to the Linker's asterisk has the following format:

*dev:binout,dev:mapout=dev:objl,dev:obj2,.../s1/s2/s3

where:

dev:	is a random-access device for all files except dev:mapout, which can be any legal output device. If dev: is not specified, DK is assumed. If the output is to be LDA format (that is, the /L switch was used), the output file need not be on a random-access device.
binout	is the name to be assigned to the Linker's save image or LDA format output file. This file is optional; if not specified, no binary output is produced. (Save image is the assumed output format unless the /L switch is used.)
mapout	is the optional load map file.
objl,...	are files of one or more object modules to be input to the Linker (these may be library files).
/s1/s2/s3	are switches as explained in Table 6-1 and Section 6.8.

If the /C switch is given, subsequent command lines may be entered as:

```
*objm,objn,.../s1/s2
```

The /C switch is necessary only if the command string will not fit on one line or if the overlay structure is used. If an error occurs in a continued command line (e.g., ?FILE NOT FND?), only the line in error need be retyped.

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 error messages (if any) are printed by the Linker.

The default values for each specification are:

	Device	Filename	Extension
Load Module	DK:	none	SAV or LDA (/L)
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 a fatal error occurs; control is returned to the command string interpreter, an asterisk is printed and a new command string may be entered.

6.2.2 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 must appear on the line indicated if the command is continued on more than one line. They may be positioned anywhere on the line. (A more detailed explanation of each switch is provided in Section 6.8.)

6.3 ABSOLUTE AND RELOCATABLE PROGRAM SECTIONS

A program produced by the HT-11 assembler or FORTRAN IV can consist of an absolute program section, declared by the .ASECT assembler directive, and relocatable program sections declared by the .CSECT assembler directive. A .CSECT directive is assumed at the beginning of the source program. The instructions and data in relocatable sections are normally assigned locations beginning at 1000 (octal). The assignment of addresses can be influenced by command string switches and the size of the absolute section (.ASECT, if present). Each control section is assigned a memory address; the Linker then appropriately modifies all instructions and/or data as necessary to account for the relocation of the control sections.

Table 6-1 Linker Switches

Switch Name	Command Line	Meaning
/A	1st	Alphabetizes the entries in the load map.
/B:n	1st	Bottom address of program is indicated as n.
/C	any	Continues input specification on another command line. Used also with /O.
/F	1st	Instructs the Linker to use the default FORTRAN library, FORLIB.OBJ; note that FORLIB does not have to be specified in the command line.
/I	1st	Includes the global symbols to be searched from the library.
/L	1st	Produces an output file in LDA format.
/M or /M:n	1st	Stack address is to be specified at the terminal keyboard or via n.
/O:n	any but the 1st	Indicates that the program will be an overlay structure; n specifies the overlay region to which the module is assigned.
/S	1st	Allows the maximum amount of space in memory to be available for the Linker's symbol table. (This switch should only be used when a particular link stream causes a symbol table overflow.)
/T or /T:n	1st	Transfer address is to be specified at terminal keyboard or via n.

The HT-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 control section 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; 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 the same as 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.

6.4 GLOBAL SYMBOLS

Global symbols provide the link, or communication, between object modules. Global symbols are created with the .GLOBL assembler directive (see Chapter 5). If the global symbol is defined in an object module (as a label or by direct assignment), 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. Undefined globals are printed on the console terminal after pass 1 (or pass 2 if a library file is also linked).

6.5 INPUT AND OUTPUT

Linker input and output is in the form of modules; one or more input modules (object files produced by either assembler or FORTRAN IV) are used to produce a single output (load) module.

6.5.1 Object Modules

Object files, consisting of one or more object modules, are the input to the Linker (the Linker ignores files which are not object modules). Object modules are created by the HT-11 assembler or FORTRAN IV. The Linker reads each object module at least twice (three times if library files are linked). During the first pass each object module is read to construct a global symbol table and to assign absolute values to the control section names and global symbols. If library files are linked, a second pass is needed to resolve the undefined globals from the library files and force their associated object modules into the root; on the final pass, the Linker reads the object modules, links and relocates the modules and outputs the load module.

6.5.2 Load Module

The primary output of the Linker is a load module which may be loaded and run under HT-11. The load module is output as a save image file (SAV). An absolute load module (LDA) is produced if the module is to be loaded by the Absolute Loader.

The load module for a save image file is arranged as follows:

Root Segment	Overlay Segments (optional)

The first 256-word block of the root segment (main program) contains the memory usage map and the locations used by the Linker to pass program control parameters. The memory usage map outlines the blocks of memory used by the load module and is located in locations 360 to 377.

The control parameters are located in locations 40–50 and contain the following information when the module is loaded:

Address	Information
40:	Start Address of program
42:	Initial setting of R6 (stack pointer)
44:	Job Status Word
46:	USR Swap Address (0 implies normal location)
50:	Highest Memory Address in user's program

Memory locations 0–476 (comprising the interrupt vectors and system communication area) may be assigned initial values by using an .ASECT assembler statement and will appear in block 0 of the load module, but there are restrictions on the use of .ASECTs in this region. The Linker does not permit an .ASECT of location 54 or of locations 360–377 (the memory usage map is passed in those locations).

Any location which is not restricted may be set with an .ASECT, but caution should be used in changing the system communication area. Restricted areas, such as the region 360–377, must be initialized by the program itself. There are no restrictions on .ASECTs if the output format is LDA.

Locations in the region 0–476 which are initialized by an .ASECT in a program may never be loaded when the program is executed. The R, RUN, and GET commands will not load an address protected by the monitor's memory protection map. The addresses normally protected include such important areas as the system device and console device vectors, but protection may be extended dynamically (e.g., by a task issuing a .PROTECT call). The procedure for loading these locations is to do so at run-time using MOV instructions.

6.5.3 Load Map

If requested, a load map is produced following the completion of the initial pass(es) of the Linker. This map, shown in Figure 6-1, diagrams the layout of 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 map begins with the name of the load module and the date of creation. The modules located in the root segment of the load module are listed next, followed by those modules which were assigned to overlays in order by their region number (see Section 6.6). Any undefined global symbols are then listed. The map ends with the transfer address (start address) and high limit of relocatable code.

6.5.4 Library Files

The HT-11 Linker has the capability of automatically searching libraries. Libraries are composed of library files—specially formatted files produced by the Librarian program (Chapter 7) which contain one or more object modules. The object modules provide routines and functions to aid the user in meeting specific programming needs. (For example, FORTRAN has a special library containing all necessary computational functions—TAN, ATAN, etc.) By using the Librarian, libraries can be created and updated so that routines which are used more than once, or routines which are used by more than one program, may be easily accessed. Selected modules from the appropriate library file are linked as needed with the user program to produce one load module. Libraries are further described in Section 6.7 and in Chapter 7.

NOTE

Library files that have been combined under PIP are illegal as input to both the Linker and the Librarian.

6.6 USING OVERLAYS

The HT-11 program overlay facility enables the user to have virtually unlimited memory space for an assembly language or FORTRAN program. A program using the overlay facility can be much larger than would normally fit in the available memory space, since portions of the program (called overlay segments) reside on a backup storage device (disk).

The HT-11 overlay scheme is a strict multi-region arrangement; it is not tree-structured. Figure 6-2 diagrams this scheme. The overlay system which the user constructs from his completed program is composed of a root segment, memory-resident overlay regions, and the overlay segments stored on the backup storage device. The root segment is a required part of every overlay program and contains all transfer addresses; it must therefore never be overlaid. An overlay region corresponds to a run-time area of memory that is shared by two or more subroutines; there is a distinct memory area for each overlay region. Overlay segments are portions of the save image file from which the user's program is run; these are brought into memory as needed.

HT-11 LINK V03-01 LOAD MAP
SQRT .SAV 19-SEP-78

SECTION	ADDR	SIZE	ENTRY	ADDR	ENTRY	ADDR	ENTRY	ADDR
. ABS.	000000	001000	$USRSW	000000	$V005A	000001	$NLCHN	000006
			$LRECL	000210	$TRACE	004737		
	001000	000220						
	001220	001364	$OTI	001246				
	002604	002300	OCI$	002604	ICI$	002612	$GET	002772
			RCI$	003006	OCO$	003712	ICO$	003720
			GCO$	004144	FCO$	004152	ECO$	004156
			DCO$	004164				
	005104	000160	ISN$	005104	$ISNTR	005110	LSN$	005124
			$LSNTR	005130				
	005264	000102	MOI$SS	005264	MOL$SS	005264	MOI$SM	005270
			MOI$SA	005274	MOI$IS	005300	MOL$IS	005300
			REL$	005300	MOI$IM	005304	MOI$IA	005310
			MOI$MS	005314	MOI$MM	005320	MOI$MA	005324
			MOI$0S	005330	MOI$0M	005334	MOI$0A	005340
			MOI$1S	005344	MOI$1M	005352	MOI$1A	005360
	005366	000020	IFR$	005366	IFW$	005400		
	005406	000046	EOL$	005406				
	005454	000062	TVL$	005454	TVF$	005462	TVD$	005470
			TVQ$	005476	TVP$	005504	TVI$	005512
	005536	000036	CAI$	005536	CAL$	005544		
	005574	000174	SQRT	005574				
	005770	000026	MOF$RS	005770	MOF$RM	005776	MOF$RA	006006
			MOF$RP	006012				
	006016	000044	NMI$1M	006016	NMI$1I	006026	BLE$	006034
			BEQ$	006036	BGT$	006044	BGE$	006046
			BRA$	006050	BNE$	006054	BLT$	006056
	006062	000072	FOO$	006062	EXIT	006074	STP$	006074
	006154	000002	$AOTS	006154				
$ERRTB	006156	000100						
$ERRS	006256	002637						
	011116	001534	$FIO	011600				
	012652	000202	$FMTDR	012652	$FMTDW	012702	$INITI	012750
	013054	000416	$CLOSE	013054				
	013472	000106	LCI$	013472	LCO$	013540		
	013600	000302	$GETRE	013600	$TTYIN	013722		
	014102	000262	$PUTRE	014102				
	014364	000106	$FCHNL	014364				
	014472	000674	$OPEN	014472				
	015366	000110	$DUMPL	015366				
	015476	000414	$PUTBL	015476	$GETBL	015676	$EOFIL	016046
	016112	000042	$WAIT	016112				

TRANSFER ADDRESS = 001000
HIGH LIMIT = 016154

Figure 6-1 Linker Load Map

```
A=A/C          = Root
B/O:1/C        = Segment 1  )
                            } = Region 1
C/O:1/C        = Segment 2  )
D/O:2/C        = Segment 3  )
                            } = Region 2
E/O:2          = Segment 4  )
```

Figure 6-2 Overlay Scheme

Overlay regions are specified to the Linker via the /O switch as described in Section 6.8.8. The size of the overlay region is calculated by the Linker to be the size of the largest group of subroutines that can occupy the region at one time. The Linker creates the overlay regions and edits the program to produce the desired overlays at run-time.

Figure 6-3 shows a diagram of memory for a program which has an overlay structure and Figure 6-4 is a listing of the run-time overlay handler.

There is no special code or function call needed to use overlays but the following eight rules must be observed when referencing parts of the users 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 (refer to Chapter 5). For example, if ENTER is a global tag in an overlay segment:

JMP ENTER	is legal, but
JMP ENTER+6	is illegal.

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). A violation of this rule cannot be detected by the assembler or Linker so no error is issued; however, it can cause the program to use incorrect data.
3. When calls are made to overlays, the entire return path must be in memory. This will happen if these 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. However, 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 may be called from overlay segments; in turn, they may call entries from the same overlay segment which called them, or from the root segment, or from another 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). As stated in 1 above, a global symbol must be used to pass control from one segment to the next.
5. Channel 17 (octal) cannot be used by the user program because overlays are read on that channel.
6. Object modules acquired from a library file cannot be placed into overlays.
7. Library files may not be specified on the same command line as an overlay.

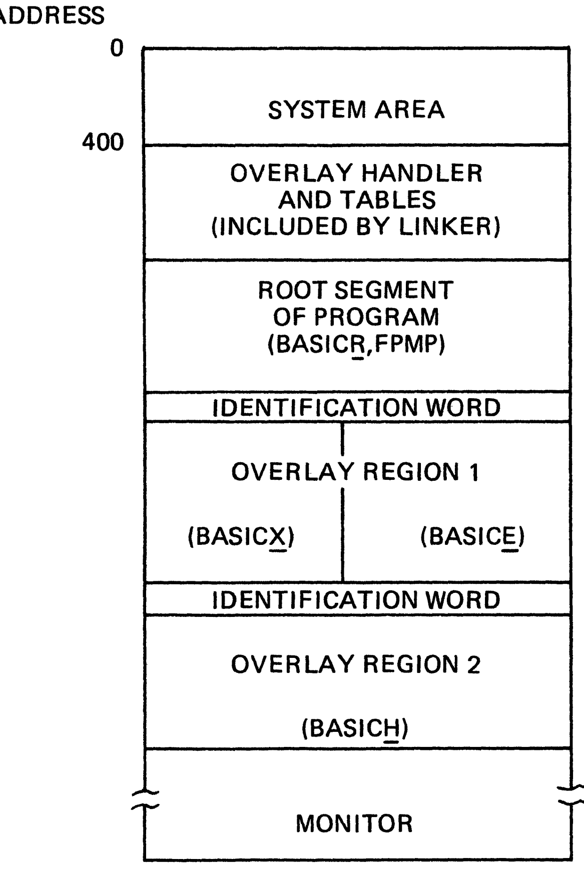


Figure 6-3 Memory Diagram Showing BASIC Link With Overlay Regions

```
        .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,$OVRH         ;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.

          $OVTAB=1000+$OVRHE-$OVRH
$OVRH:    MOV      R0,-(SP)
          MOV      R1,-(SP)
          MOV      R2,-(SP)
$OVRHB:
;         MOV      (R5)+,R0         ;PICK UP OVERLAY NUMBER
          BR       $FIRST           ;FIRST CALL ONLY * * *
          MOV      R0,R1
$OVRHA:   ADD      #$OVTAB-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
          .READW   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       $OVRHB           ;AND RETURN TO CALL IN PROGRESS
$ERR:     EMT      376              ;GENERATE ALWAYS FATAL ERROR
          .BYTE    0,373            ;AND DISREGARD SOFT ERROR
$OVRHE:

;  OVERLAY SEGMENT TABLE FOLLOWS;
;  $OVTAB:            .WORD   <CODE 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 $OVTAB TABLE.
```

Figure 6-4 The Run-Time Overlay Handler

8. Overlay regions must be specified in ascending order and are read-only. Unlike USR swapping, an overlay segment does not save the segment it is overlaying. Any tables, variables, or instructions that are modified within a given overlay segment are re-initialized to their original values in the SAV file if that segment has been overlaid by another segment. Any variables or tables whose values must be maintained across overlays should be placed in the root segment.

The following information should be noted when writing FORTRAN overlays.

1. When dividing a FORTRAN program into a root segment and overlay regions (and subsequently dividing each overlay region into overlay segments), routine placement should be carefully considered. The user should always remember that it is illegal to call a routine located in a different overlay segment in the same overlay region, or an overlay region with a lower numeric value (as specified by the Linker overlay switch, /O:n) from the calling routine. The user should divide each overlay region into overlay segments which never need to be resident simultaneously (i.e., if segments A and B are assigned to region X, they cannot call each other because they occupy the same locations in memory).
2. The FORTRAN main program unit must be placed in the root segment.
3. In an overlay environment, subroutine calls and function subprogram references may refer only to one of the following:

   A FORTRAN library routine (e.g., ASSIGN, DCOS)

   A FORTRAN or assembly language routine contained in the root segment

   A FORTRAN or assembly language routine contained in the same overlay segment as the calling routine

   A FORTRAN or assembly language routine contained in a segment whose region number is greater than that of the calling routine

4. In an overlay environment, COMMON blocks must be placed so that they are resident when referenced. Blank COMMON is always resident since it is always placed in the root segment. All named COMMON must be placed either in the root segment, or into the segment whose region number is lowest of all segments which reference the COMMON block. A named COMMON block cannot be referenced by two segments in the same region unless the COMMON block appears in a segment of a lower region number. The Linker automatically places a COMMON block into the root segment if it is referenced by the FORTRAN main program or by a subprogram that is located in the root segment. Otherwise the Linker places a COMMON block in the first segment encountered in the Linker command string that references that COMMON block.
5. All COMMON blocks which are initialized (by use of DATA statements) must be so initialized in the segment in which they are placed.

Refer to the *HT-11 FORTRAN IV User's Guide* for more details.

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 aforementioned sets of 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 (as described in the *PDP-11 Processor Handbook* and in Chapter 5).

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 6-4) in the same file with the user program to be used at program run-time. This overlay handler plus some tables are inserted into the user's program beginning at the bottom address computed by the Linker. The Linker moves the user's program up in memory by an appropriate amount to make room for the overlay handler and tables, if necessary.

6.7 USING LIBRARIES

Libraries are specified in a command string in the same fashion as normal modules; they may be included anywhere in the command string, with the exception of overlay lines. If a global symbol is undefined at the time the library is encountered in the input stream and a module is included in the library which includes that global definition, that module is pulled from the library and linked into the load image. Only the modules needed to resolve references are pulled from the library; unused modules are not linked.

NOTE

Modules in one library may call modules from another library; however, the libraries must appear in the command string in the order in which they are called. For example, assume module X in library ALIB calls SQRT from the FORTRAN library. To correctly resolve all globals, the order of ALIB and the FORTRAN library should appear in the command line as:

```
   *Z=B,ALIB/F
or *Z=B,ALIB,FORLIB
```

Module B is the root. It calls X from ALIB and brings X into the root. X in turn calls SQRT which is brought from FORLIB into the root.

FORTRAN libraries cannot precede their root segment in a command line as this creates a bad transfer address. For example:

```
*X=ROOT/F
*X=ROOT,FORLIB
```

are legal, but:

```
*X=FORLIB,ROOT
```

is not. Unpredictable results will occur.

6.7.1 User Library Searches

Object modules from the named user libraries built by the Librarian are relocated selectively and linked by the Linker. The HT-11 Linker searches a specified library file during the library pass as follows (refer to Figure 6-5 for a flowchart representation of this process):

1. If there are any undefined globals in the Linker's table when a library is encountered in the command string, proceed to step 2; otherwise skip this library (go to step 5).
2. Read the library directory.
3. If any of the undefined globals can be defined by a module in this library, include the relevant module into the linked output file; otherwise, go to step 5.
4. If any undefined globals remain in the Linker's table and they have not been looked for in the library, return to step 2; otherwise go to step 5.
5. Close the library file.
6. Go to the next element in the command string.

This search method allows modules to appear in any order in the library. Any number of libraries may be specified in a link, and they may be positioned anywhere, with the exception of overlay segments and the restrictions noted in Section 6.7.

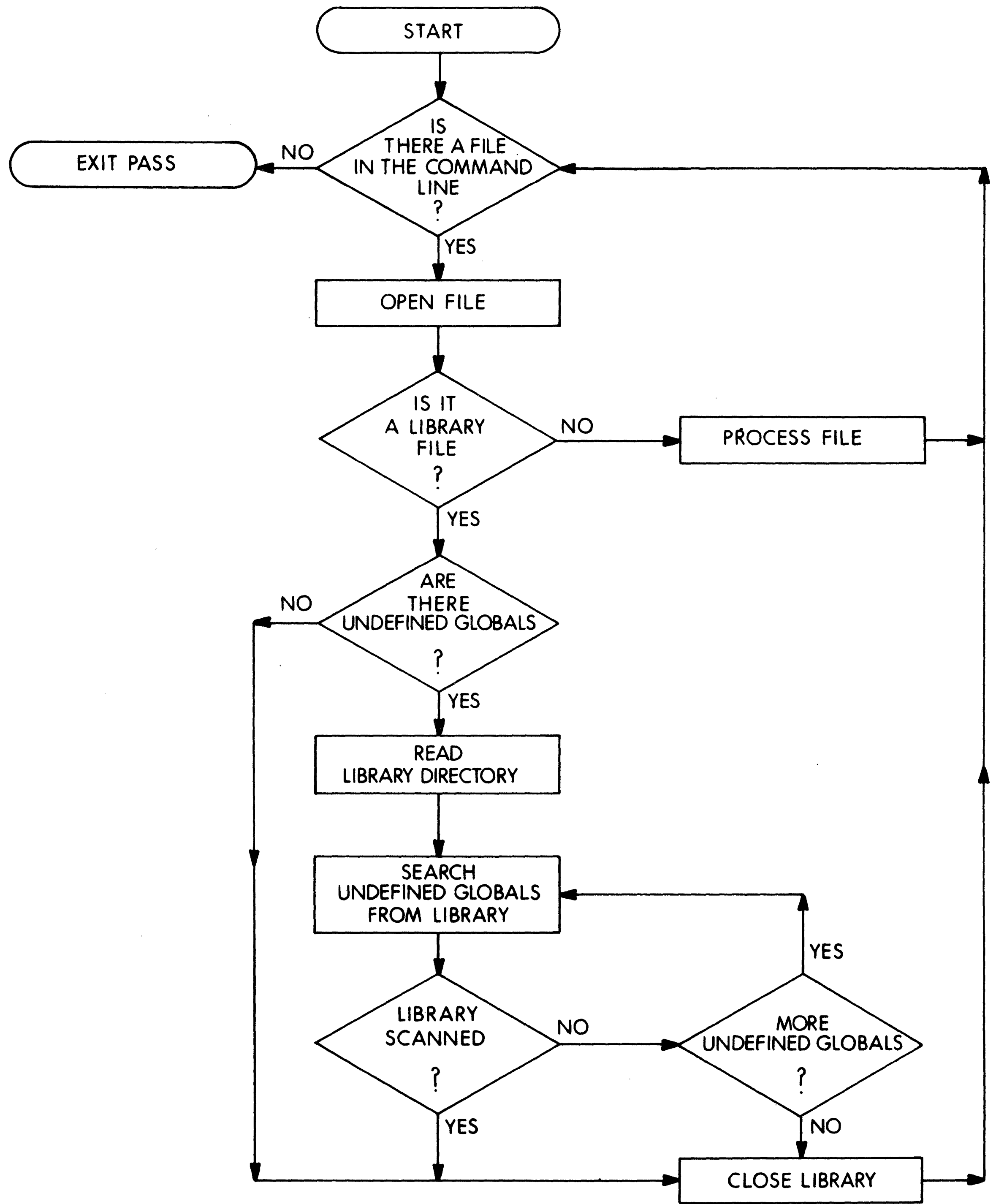


Figure 6-5 Library Searches

NOTE

For faster Linker performance, the user should specify all object files before library files, and all user library files before the system library files. For example:

```
*A=A,B,USELIB/F
```

where A and B are object modules, USELIB is a user-created library file, and /F denotes the default FORTRAN library, FORLIB.

Libraries are input to the Linker as any other input file. Assume the following command string to the Linker:

```
*TASK01.SAV,LP:=MAIN.OBJ,MEASUR.OBJ
```

This causes program MAIN.OBJ to be read from DK: as the first input file. Any undefined symbols generated by program MAIN.OBJ should be satisfied by the library file MEASUR.OBJ specified in the second input file. The load module, TASK01.SAV is put on DK: and a load map goes to the line printer.

6.8 SWITCH DESCRIPTION

The switches summarized in Table 6-1 are described in detail below.

6.8.1 Alphabetize Switch

The /A switch requests the Linker to list linked modules in alphabetical order as follows: .CSECTs, module names, and entry points within modules. The load map is normally arranged in order by module address as shown in Figure 6-1. Figure 6-6 is an example of an alphabetized load map.

6.8.2 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
```

n is a six-digit unsigned octal number which defines the bottom address of the program being linked. An error message results if 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. If the value is odd, an error message is generated.

HT-11 LINK V03-01 LOAD MAP
SQRT .SAV 19-SEP-78

SECTION	ADDR	SIZE	ENTRY	ADDR	ENTRY	ADDR	ENTRY	ADDR
. ABS.	000000	001000	$LRECL	000210	$NLCHN	000006	$USRSW	000000
			$TRACE	004737	$V005A	000001		
	001000	000220						
	001220	001364	$OTI	001246				
	002604	002300	DCO$	004164	ECO$	004156	FCO$	004152
			GCO$	004144	ICI$	002612	ICO$	003720
			OCI$	002604	OCO$	003712	RCI$	003006
			$GET	002772				
	005104	000160	ISN$	005104	LSN$	005124	$ISNTR	005110
			$LSNTR	005130				
	005264	000102	MOI$IA	005310	MOI$IM	005304	MOI$IS	005300
			MOI$MA	005324	MOI$MM	005320	MOI$MS	005314
			MOI$SA	005274	MOI$SM	005270	MOI$SS	005264
			MOI$0A	005340	MOI$0M	005334	MOI$0S	005330
			MOI$1A	005360	MOI$1M	005352	MOI$1S	005344
			MOL$IS	005300	MOL$SS	005264	REL$	005300
	005366	000020	IFR$	005366	IFW$	005400		
	005406	000046	EOL$	005406				
	005454	000062	TVD$	005470	TVF$	005462	TVI$	005512
			TVL$	005454	TVP$	005504	TVQ$	005476
	005536	000036	CAI$	005536	CAL$	005544		
	005574	000174	SQRT	005574				
	005770	000026	MOF$RA	006006	MOF$RM	005776	MOF$RP	006012
			MOF$RS	005770				
	006016	000044	BEQ$	006036	BGE$	006046	BGT$	006044
			BLE$	006034	BLT$	006056	BNE$	006054
			BRA$	006050	NMI$1I	006026	NMI$1M	006016
	006062	000072	EXIT	006074	FOO$	006062	STP$	006074
	006154	000002	$AOTS	006154				
$ERRTB	006156	000100						
$ERRS	006256	002637						
	011116	001534	$FIO	011600				
	012652	000202	$FMTDR	012652	$FMTDW	012702	$INITI	012750
	013054	000416	$CLOSE	013054				
	013472	000106	LCI$	013472	LCO$	013540		
	013600	000302	$GETRE	013600	$TTYIN	013722		
	014102	000262	$PUTRE	014102				
	014364	000106	$FCHNL	014364				
	014472	000674	$OPEN	014472				
	015366	000110	$DUMPL	015366				
	015476	000414	$EOFIL	016046	$GETBL	015676	$PUTBL	015476
	016112	000042	$WAIT	016112				

TRANSFER ADDRESS = 001000
HIGH LIMIT = 016154

Figure 6-6 Alphabetized Load Map

Example:

*OUTPUT,LP:=INPUT/B:500	Causes the input file to be linked starting at location 500 (octal).

6.8.3 Continue Switch

The Continue switch (/C) is used to allow additional lines of command string input. The /C switch is typed at the end of the current line and may be repeated on subsequent command lines as often as necessary to specify all input modules for which memory is available. If memory is exceeded, an error message is output. A /C switch is not entered on the last line of input.

Example:

*OUTPUT,LP:=INPUT/C *	Input is to be continued on the next line; the Linker prints an asterisk.

6.8.4 Default FORTRAN Library Switch

By indicating the /F switch in the command line, the FORTRAN library, FORLIB.OBJ on the default device (SY:), is linked with the other object modules specified; the user does not need to specify FORLIB. For example:

```
*FILE,LP:=AB.OBJ/F
```

The object module AB.OBJ and the FORTRAN library FORLIB are linked together to form a load module called FILE.SAV. (Note that the FORLIB default is SY:FORLIB.OBJ, not DK:FORLIB.OBJ.)

6.8.5 Include Switch

The /I switch allows subsequent entry at the keyboard of global symbols to be taken from any library and included in the linking process. When the /I switch is specified, the Linker prints:

```
LIBRARY SEARCH:
```

Reply with the list of global symbols to be included in the load module; type a carriage return to enter each symbol in the list. A carriage return alone terminates the list of symbols. This provides a method for forcing modules (which are not called by other modules) to be loaded from the library.

Example:

```
*OUTPUT,LP:=INPUT,XLIB/I
```

LIBRARY SEARCH:	Linker prints LIBRARY SEARCH:
A <CR> GETSYM <CR> CHAR <CR> CHFLG <CR> <CR>	User enters A, GETSYM, etc. which are to be included in the linking process. Each symbol is entered by typing a carriage return; the list is terminated by an additional carriage return.

6.8.6 LDA Format Switch

The LDA format switch (/L) causes the output file to be in LDA format instead of save image format. The LDA format file can be output to any device, including devices that are not block-replaceable such as paper tape, and is useful for files which are to be loaded with the Absolute Loader. The default extension .LDA is assigned when the /L switch is used.

The /L switch cannot be used in conjunction with the overlay switch (/O).

Example:

*DK:OUT,LP:=IN,IN2/L	Links disk files IN and IN2; outputs an LDA format file OUT.LDA to the system device and a load map to the line printer.

6.8.7 Modify Stack Address

The stack address, location 42, is the address which contains the user's stack pointer. The /M switch allows terminal keyboard specification of the user's stack address.

The form of the switch is:

```
/M:n
```

n is an even unsigned 6-digit octal number which defines the stack address. If n is not specified, the Linker prints the message:

```
STACK ADDRESS =
```

In this case, specify the global symbol whose value is the stack address. A number cannot be specified, and if a nonexistent symbol is specified, an error message is printed and the stack address is set to the system default (1000 for save files).

Direct assignment (via .ASECT) of the stack address within the program takes precedence over assignment with the /M switch.

Example:

```
*OUTPUT=INPUT/M

STACK ADDRESS = BEG
```

6.8.8 Overlay Switch

The Overlay switch (/O) is used to segment the load module so that the entire program is not memory resident at one time (overlay feature). This allows programs larger than the available memory to be executed. The switch has the form:

```
/O:n
```

where n is an unsigned octal number (up to six digits in length) specifying the overlay region to which the module is assigned. 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 as this is the root segment. Therefore, the /C continuation switch must be used.

Co-resident overlay routines (a group of subroutines which occupy the overlay region and segment at the same time) are specified as follows:

```
*OBJA,OBJB,OBJC/O:n/C
*OBJD,OBJE/O:m/C
.
.
.
```

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 value previously used (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 memory as the first group, but not at the same time. For example, if subroutines in object modules R and S are to be in memory 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 memory as T (but at different times):

```
*MAIN,LP:=ROOT/C
*R,S/O:1/C
*T/O:1
```

The above could also be written as:

```
*MAIN,LP:=ROOT/C
*R/O:1/C
*S/C
*T/O:1
```

Example:

```
*OUTPUT,LP:=INPUT/C          Establishes two overlay regions
*OBJA/O:1/C
*OBJB/O:2
```

Overlays must be specified in order of increasing region number. For example:

```
.R LINK
*A=A/C
*B/O:1/C
*C/O:1/C
*D/O:1/C
*E,F/O:2/C
*G/O:3
```

The following overlay specification is illegal since the overlay regions are given in a random numerical order (an error message is printed in each case):

```
.R LINK
*A=A/C
*D/O:2/C
*B/O:1/C
/O IGNORED
*C/O:1/C
/O IGNORED
*G/O:3/C
*H/O:3/C
*E,F/O:2
/O IGNORED
```

6.8.9 Symbol Table Switch

Use of the symbol table switch in the command line instructs the Linker to allow the largest possible memory area for its symbol table at the expense of making the link process slower. With the /S switch, library directories are not made resident in memory, but are left on disk. For example:

```
*OUTF,LP:=INPUT.OBJ,LIBR1.OBJ,LIBR2.OBJ/S
```

The directories of the library files LIBR1 and LIBR2 are not brought into memory, resulting in more room in the symbol table but longer link time.

If the /S switch is not used and the memory available to the Linker is approximately 10K or larger, the library directory is brought into memory (providing there is room); the directory is kept there until the library has been completely processed, thus reducing the size of the Linker's symbol table. If there is not enough room in memory for the direcctory (as is the case in an 8K system), the Linker determines this and leaves the directory on disk regardless of whether the /S switch was used or not.

The /S switch should be used only if an attempt to link a program failed because of symbol table overflow. Often, use of /S will allow the program to link.

6.8.10 Transfer Address Switch

The transfer address is the address at which a program is to be started when executed via an R or RUN command. The Transfer Address 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. If n is not specified, the message:

TRANSFER ADDRESS =

is printed. In this case, 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 (so that the program cannot be executed).

If the transfer address specified is odd, the program does 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. The transfer address assigned with a /T has precedence over that assigned with a .END assembly directive.

Example:

```
*PROG=PROG1,PROG2,ODT/T
TRANSFER ADDRESS =
O.ODT
```

The files PROG1.OBJ,PROG2.OBJ and ODT.OBJ are linked together and started at ODT's transfer address.

6.9 LINKER 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 only warnings and are included in the load map. If a load map is not requested in the command string, all messages are output to the terminal.

Message	Meaning
ADDITIVE REF OF xxxxxx AT SEGMENT # yyyyyy	Rule 1 of overlay rules explained in Section 6.6 has been violated. xxxxxx represents the entry point; yyyyyy represents the segment number.
?/B NO VALUE?	The /B switch requires an unsigned even 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 OVERLAY AT SEG # yyyyyy	Overlay tries to store text outside its region; check for a .ASECT in overlay. yyyyyy represents the segment number.
?BAD RLD?	There is an invalid relocation directory (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.
?BAD x SWITCH IGNORED?	LINK did not recognize a switch (x) specified in the command line. The switch is ignored and linking continues.
BYTE RELOCATION ERROR AT xxxxxx	Linker attempted to relocate and link byte quantities but failed. xxxxxx represents the address at which the error occurred. 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.
?CORE?	There is not enough memory to accommodate the command or the resultant load module.
?ERROR ERROR?	An error occurred while the Linker was in the process of recovering from a previous system or user error.
?ERROR IN FETCH?	The device is not available.
?FILE NOT FND?	Input file was not found.
?FORLIB NOT FND?	The user indicated via the /F switch that the FORTRAN library, FORLIB, was to be linked with the other object modules in the command line, and the Linker could not find FORLIB.OBJ on the system device.
?HARD I/O ERROR?	A hardware error occurred; try the operation again.

Message	Meaning
?LDA FILE ERROR?	There was a hardware problem with the device specified for LDA output or the device was full.
?/M ODD VAL?	An odd value has been specified for the stack address. Control returns to the Linker and another command line may be indicated.
?MAP FILE ERROR?	There was a hardware problem with the device specified for map output or the device is full.
MULT DEF OF xxxxxx	The symbol, xxxxxx, was defined more than once.
?NO INPUT?	No input files were specified.
/O IGNORED	Overlays have been specified in the wrong order (see overlay restrictions); the overlay switch is ignored.
?OUTPUT FULL?	The output device was full.
?SAV FILE ERR?	The Linker encountered a problem writing the save image file; try the operation again.
?STACK ADDRESS UNDEFINED OR IN OVERLAY?	The stack address specified by the /M switch was either undefined or in an overlay. The stack address is set to the system default.
?SYMBOL TABLE OVERFLOW?	There were too many global symbols used in the program. Retry the link using the /S switch. If the error still occurs, the link cannot take place in the available memory.
?TOO MANY OUTPUT FILES?	The Linker allows specification of only two output files.
TRANSFER ADDRESS UNDEFINED OR IN OVERLAY	The transfer address was not defined or was in an overlay.
UNDEF GLBLS	Undefined globals exist.
UNDEFINED GLOBALS xxxxxx xxxxxx . . .	The globals listed (xxxxxx) were undefined. If a load map is requested, this condition also causes the warning message, UNDEF GLBLS, to be printed on the terminal.

CHAPTER 7
LIBRARIAN

The HT-11 system provides the user with the capability of maintaining libraries which may be composed of functions and routines of his choice. Each library is a file containing a library header, library directory (or entry point table), and one or more object modules. The object modules in a library file may be routines which are repeatedly used in a program, routines which are used by more than one program, or routines which are related and simply gathered together for ease in usage – the contents of the library file are determined by the user's needs. An example of a typical library file is the FORTRAN library, FORLIB.OBJ. This library is provided with the FORTRAN package and contains all the mathematical functions needed for normal usage.

Object modules in a library file are accessed from another program via calls to their entry points; the object modules are linked with the program which uses them by the Linker (Chapter 6) to produce a single load module.

The HT-11 Librarian (LIBR) allows the user to create, update, modify, list, and maintain library files.

7.1 CALLING AND USING LIBR

The HT-11 Librarian is called from the system device by entering the command:

```
R LIBR
```

in response to the dot printed by the Keyboard Monitor. The Command String Interpreter prints an asterisk at the left margin on the console terminal when it is ready to accept a command line.

Type CTRL C to halt the Librarian at any time and return control to the monitor. To restart the Librarian, type R LIBR or the REENTER command in response to the monitor's dot.

7.2 USER SWITCH COMMANDS AND FUNCTIONS

The user maintains library files through the use of switch commands. Functions which can be performed include object module deletion, insertion and replacement, library file creation, and listing of a library file's contents.

7.2.1 Command Syntax

LIBR accepts command strings in the following general format:

```
*dev:lib,dev:list=dev:input/s1/s2/s3
```

where:

dev:	represents a legal HT-11 device specification
lib	represents the library file to be created or updated
list	represents a listing file for the library's contents
input	represents the filenames of the input object modules
/sl, . . .	represents one or more of the switches listed in Table 7-1

Devices and filenames are specified by the user in the standard HT-11 command string syntax, with default extensions assigned as follows:

File	Extension
list file:	.LLD
library file:	.OBJ
input files:	.OBJ

If no device is specified, the default device (DK:) is assumed.

Each input file is made up of one or more object modules, and is stored on a given device under a specific filename and extension. Once an object module has been inserted into a library file, the module is no longer referenced by the name of the file of which it was a part, but by its individual module name. (This module name has been assigned by the assembler either via a .TITLE statement in the assembly source program, or, if no .TITLE statement is present, with the default name .MAIN.; see Chapter 5.) Thus, for example, the input file FORT.OBJ may exist on DX1: and may contain an object module called ABC. Once the module is inserted into a library file, reference is made only to ABC (not FORT.OBJ).

7.2.2 LIBR Switch Commands

Table 7-1 summarizes the switches available for use under HT-11 LIBR. Switches are explained in detail following the table.

Table 7-1 LIBR Switches

Switch	Position In Command String	Meaning
/C	Any line but last	Command continuation; the command is too long for the current line and is continued on the next line
/D	1st line only	Delete; delete modules from a library file
/G	1st line only	Global deletion; delete entry points from the library directory
/R	1st line only	Replace; replace modules in a library file
/U	1st line only	Update; insert and replace modules in a library file

There is no switch to indicate module insertion. The function of inserting a module into a library file is assumed in the absence of other switches.

7.2.2.1 Command Continuation Switch – The Command Continuation switch is necessary whenever there is not enough room to enter a command string on one line and additional lines are needed. The /C switch is typed at the end of the current line and may be repeated at the end of subsequent command lines as often as necessary as long as memory is available; if memory is exceeded, an error message is output. A /C switch is not entered on the last line of input.

Command Format:

```
*dev:lib,dev:list=dev:input1,dev:input2, . . . ,/C
*dev:inputn
```

where:

dev:	represents a device specification
lib	represents the filename of the library to be created or updated
list	represents the filename of a listing file containing the library file's contents
input	represents the filenames of the input modules to be inserted into the library
/C	represents the Continuation switch, indicating that the command is to be continued on the following line

Examples:

```
*ALIB,LIBLST=DX1:MAIN,TEST,FXN/C
*DX1:TRACK
```

In this example, a library file is created on the default device (DK:) under the filename ALIB.OBJ; a listing of the library file's contents is created as LIBLST.LLD also on the default device; the filenames of the input modules are MAIN.OBJ, TEST.OBJ, FXN.OBJ, and TRACK.OBJ, all from DX1.

```
*BLIB=MAIN/C
*DX1:TEST/C
*PR:FXN/C
*DX1:TRACK
```

A library file is created on the default device, (DK:) under the name BLIB. No listing is produced. Input files are MAIN from the default device, TEST from DX1:, FXN from PR: and TRACK from DX1.

Another way of writing this command line is:

```
*BLIB=MAIN,DX1:TEST,PR:FXN/C
*DX1:TRACK
```

7.2.2.2 Creating a Library File – A library file is created whenever a filename is indicated on the output side of a command line which does not represent a list file.

Command Format:

```
*dev:lib=dev:input1, . . . ,dev:inputn
```

where:

dev:	represents a device specification
lib	represents the filename of the library to be created
input	represents the filenames of the input modules to be inserted into the new library

Example:

```
*NEWLIB=FIRST,SECOND
```

A new library called NEWLIB.OBJ is created on the default device (DK:). The modules which will make up this library file are in the files FIRST.OBJ and SECOND.OBJ, both on the default device.

Assume this command line is next entered:

```
*NEWLIB,LIST=THIRD,FOURTH
```

The already existing library file NEWLIB is destroyed when the new library file is created. A listing of the library file's contents is created under the filename LIST, and the object modules in the files THIRD and FOURTH are inserted into the library file NEWLIB.

7.2.2.3 Inserting Modules Into a Library – The Insert function is assumed whenever an input file does not have an associated switch; the modules in the file are inserted into the library file named on the output side of the command string. Any number of input files are allowed. If an attempt is made to insert a file which contains an entry point or .CSECT having the same name as an entry point or .CSECT already existing in the library file, a warning message is printed. However, the library file is updated, ignoring the entry point or .CSECT in error, and control returns to the CSI; the user may enter another command string.

Although the user may insert object modules which exist under the same name (as assigned by the .TITLE statement) this practice is not recommended because of the difficulty involved when replacing or updating these modules (refer to Sections 7.2.2.4 and 7.2.2.7).

NOTE

The library operations of module insertion, replacement deletion, merge, and update are actually performed in conjunction with the library file creation operation. Therefore, the library file to which the operation is directed must be indicated on both the input and output sides of the command line, since effectively a "new" output library file is created each time the operation is performed. The library file must be specified first in the input field.

Command Format:

```
*dev:lib=dev:lib,dev:input1, . . . ,dev:inputn
```

where:

dev:	represents a device specification
lib	represents the filename of an existing library file
input	represents the filenames of the modules to be inserted into the library file

Example:

```
*DXY=DXY,DX1:FA,FB,FC
```

The modules included in the files FA.OBJ, FB.OBJ, and FC.OBJ on DX1: are inserted into a library file named DXY.OBJ on the default device. The library header and Entry Point Table of the library file are updated accordingly (see Section 7.4).

7.2.2.4 Replace Switch – The Replace function is used to replace modules in a library file. All modules contained in the file(s) indicated as input will replace existing modules of the same names in the library file specified as output.

An error message is printed and no modules are replaced if an old module does not exist under the same name as an input module, or if the user specifies the /R switch on a library file. /R must follow each input filename containing modules for replacement.

Command Format:

    *dev:lib=dev:lib,input1/R, . . . ,dev:inputn/R

where:

dev:	represents a device specification
lib	represents the filename of an existing library file
input	represents the names of the files containing modules to be replaced
/R	represents the Replace switch

Examples:

```
*TFIL=TFIL,INA,INB/R,INC
```

This command line indicates that the modules in the file INB.OBJ are to replace existing modules of the same names in the library file TFIL.OBJ. The object modules in the files INA.OBJ and INC.OBJ are to be added. All files are stored on the default device DK:.

```
*XFIL=TFIL,INA,INB/R,INC
```

The same operation occurs here as in the preceding example, except that this updated library file is assigned the new name XFIL.

7.2.2.5 Delete Switch – The Delete switch deletes modules and all their associated entry points from the library.

Command Format:

    *dev:lib=dev:lib/D

where:

dev:	represents the device on which the library file exists
lib	represents the filename of an existing library file
/D	represents the Delete switch; may be positioned anywhere on the input side of the command line

When the /D switch is used, the Librarian prints:

    MOD NAME:

The user should respond with the name of the module to be deleted followed by a carriage return; he may continue until all modules to be deleted have been entered. Typing only a carriage return (either on a line by itself or immediately after the MOD NAME: message) terminates input and initiates execution of the command line.

Examples:

```
*DX1:TRAP=DX1:TRAP/D

MOD NAME:
SGN <CR>
TAN <CR>
<CR>
```

The modules SGN.OBJ and TAN.OBJ are deleted from the library file TRAP.OBJ on DX1:.

```
*LIBFIL=LIBFIL/D,ABC/R,DEF

MOD NAME:
FIRST <CR>
<CR>
```

The module FIRST.OBJ is deleted from the library (LIBFIL); the module ABC.OBJ replaces an old module of the same name in the library, and the modules in the file DEF.OBJ are inserted into the library.

```
*LIBFIL=LIBFIL/D

MOD NAME:
X<CR>
X<CR>
<CR>
```

Two modules of the same name are deleted from the library file LIBFIL (module names are assigned with the .TITLE statement as described in Section 7.2.1).

7.2.2.6 Delete Global Switch – The Delete Global switch gives the user the ability to delete a specific entry point from a library file's Entry Point Table.

Command Format:

```
*dev:lib=dev:lib/G
```

where:

dev:	represents the device on which the library file exists
lib	represents the filename of an existing library file
/G	represents the Delete Global switch; may be positioned anywhere on the input side of the command line

When the /G switch is used, the Librarian prints:

```
ENTRY POINT:
```

The user should respond with the name of the entry point to be deleted followed by a carriage return; he may continue until all entry points to be deleted have been entered. Typing only a carriage return (either on a line by itself or immediately after the ENTRY POINT: message) terminates input and initiates execution of the command line.

Example:

```
*ROLL=ROLL/G

ENTRY POINT:
NAMEA <CR>
NAMEB <CR>
<CR>
```

This command instructs LIBR to delete the entry points NAMEA and NAMEB from the entry point table found in the library file ROLL.OBJ on DK:.

Since entry points are only deleted from the Entry Point Table (and not from the library itself) whenever a library file is updated, all entry points that were previously deleted are restored unless the /G switch is again used to delete them. This feature allows the user to recover from inadvertently deleting the wrong entry point.

7.2.2.7 Update Switch – The Update switch allows the user to update a library file by combining the insert and replace functions. If the object modules included in an input file in the command line already exist in the library file, they are replaced; if not, they are inserted. (No error messages are printed when using the Update function as might occur under the Insert and Replace functions.) /U must follow each input file containing modules to be updated.

Command Format:

```
*dev:lib=dev:lib,dev:input1/U, . . . ,dev:inputn/U
```

where:

dev:	represents a device specification
lib	represents the filename of an existing library file
input	represents the names of files containing object modules to be updated
/U	represents the Update switch

Examples:

```
*BALIB=BALIB,FOLT/U,TAL,BART/U
```

This command line instructs LIBR to update the library file BALIB.OBJ on the default device. First the modules in FOLT.OBJ and BART.OBJ replace old modules of the same names in the library file, or if none already exist under their names, the modules are inserted. Then the modules from the file TAL.OBJ are inserted; an error message is printed if the name of the module in TAL.OBJ already exists.

```
*XLIB=XLIB/D,Z/U/G

MOD NAME:
X <CR>
X <CR>
<CR>

ENTRY POINT:
SEC <CR>
SEC1 <CR>
<CR>
```

There are two object modules of the same name (X) in both Z and XLIB; these are first deleted from XLIB. This ensures that both modules X in file Z are correctly placed into the library. Entry points SEC and SEC1 are also deleted from the Entry Point Table, but automatically return when the library (XLIB) is updated in the future.

7.2.2.8 **Listing the Directory of a Library File** – The user may specify that a listing of the contents of a library file be output by indicating both the library file and a list file in the command line. Since a library file is not being created or updated, it is not necessary to indicate the filename on the output side of the command line; however a comma must be used to designate a null output library file.

Command Formats:

```
*,LP:=dev:lib
```

or

```
*,dev:list=dev:lib
```

where:

dev:	represents a device specification
lib	represents the file name of an existing library file
LP:	indicates the listing is to be sent directly to the line printer
list	represents a list file of the library file's contents

Examples:

```
*,DX1:LIST=LIBFIL
```

This command line outputs to disk 1 as LIST.LLD a listing of the contents of the library file LIBFIL.OBJ on the default device.

```
*,LP:=FLIB
```

This command outputs on the line printer a listing of all modules in the library file FLIB.OBJ stored on the default device. Assuming this library is composed of modules STOP, WAIT, and IMUL, is 2 blocks long, was created on September 6, 1978, and the listing was requested on September 6, 1978, the directory format appears as follows:

```
HT-11 LIBRARIAN       X02-05          6-SEP-78
FLIB                  6-SEP-78        2 BLOCKS

MODULE                ENTRY/CSECT     ENTRY/CSECT     ENTRY/CSECT

STOP                  STP$
WAIT                  SWAIT
IMUL                  MUI$IS          MUI$MS          MUI$PS
                      MUI$SS          $MLI
```

7.2.2.9 **Merging Library Files** – Two or more library files may be merged under one filename by indicating all the library files to be merged in a single command line. The individual library files are not deleted following the merge.

Command Format:

```
*dev:lib=dev:input1, . . . ,dev:inputn
```

where:

dev: represents a device specification

lib represents the name of the library file which will contain all the merged files (if a library file already exists under this name, it must also be indicated in the input side of the command line in order to be included in the merge)

input represents the library files to be merged together

Thus, the command:

```
*MAIN=MAIN,TRIG,STP,BAC
```

combines library files MAIN.OBJ, TRIG.OBJ, STP.OBJ, and BAC.OBJ under the existing library file name MAIN.OBJ; all files are on the default device DK:.

```
*FORT=A,B,C
```

This command creates a library file named FORT.OBJ and merges existing library files A.OBJ, B.OBJ, and C.OBJ under the filename FORT.OBJ.

NOTE

Library files that have been combined under PIP are illegal as input to both the Librarian and the Linker.

7.3 COMBINING LIBRARY SWITCH FUNCTIONS

Two or more library functions may be requested in the same command line. The Librarian performs functions in the following order:

1. /C
2. /D
3. /G
4. /U
5. /R
6. Insertions
7. Listing

Example:

```
*FILE,LP:=FILE/D,MODX,MODY/R

MOD NAME:
XYZ<CR>
A<CR>
<CR>
```

Functions in this example are performed in order, as follows:

1. Delete modules XYZ.OBJ and A.OBJ from the library file FILE.OBJ
2. Replace any duplicate of the module in the file MODY.OBJ
3. Insert the modules in the file MODX.OBJ
4. List the contents of FILE.OBJ on the line printer

7.4 FORMAT OF LIBRARY FILES

A library file is a contiguous file consisting of a header, an Entry Point Table (library directory) and one or more library object modules, as illustrated in Figure 7-1:

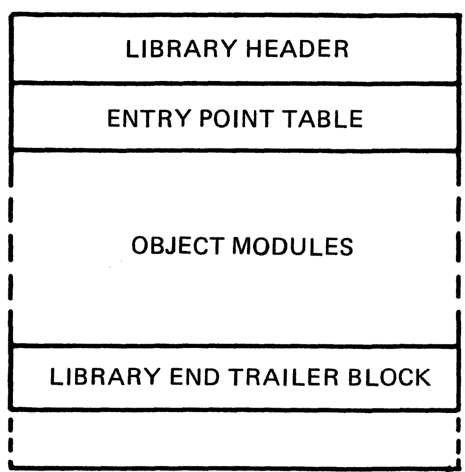


Figure 7-1 General Library File Format

The following paragraphs describe in detail each component of a library file.

7.4.1 Library Header

The header section of a library file contains 17 (decimal) words which describe the current status of the file (refer to Figure 7-2). This includes information relating to the version of the Librarian in use, the date and time of file creation or update, the relative starting address of the Entry Point Table (EPT), the number of EPT entries available and in use, and the placing of the next module to be inserted into the library file. The contents of the library header are updated as the library file is modified, so that LIBR can always quickly and easily access the information it needs to perform its functions. Figure 7-2 illustrates the header format.

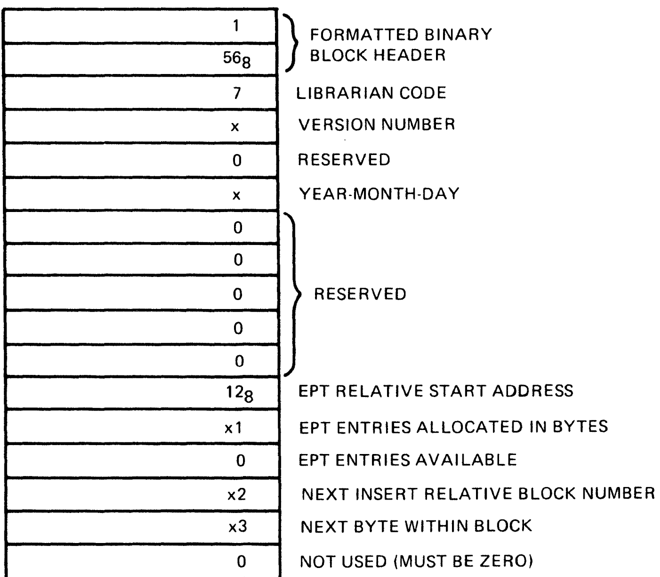


Figure 7-2 Library Header Format

7.4.2 Entry Point Table (Library Directory)

The Entry Point Table is located immediately after the library header. It is composed of four-word entries which include the names, addresses, and entry points of all object modules in the library file. The first two words of an entry in the EPT contain the Radix 50 name by which an entry point, CSECT, or module is referenced. The third word provides a pointer to the object module where an entry point is defined. The fourth word contains the total number of CSECTs in the object module (information needed by the Linker), and the relative byte within the block pointing to the object module's starting point, as shown in Figure 7-3.

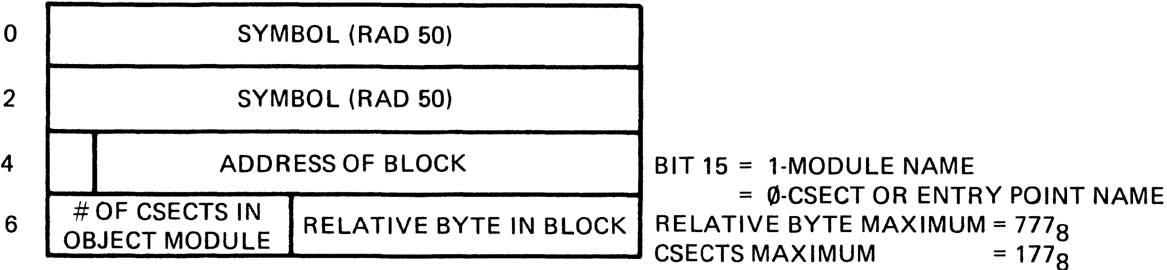


Figure 7-3 Format of Entry Point Table

7.4.3 Object Modules

Object modules follow the Entry Point Table. An object module consists of three main types of data blocks: a global symbol directory, text blocks, and a relocation directory. The information contained in these data blocks is used by the Linker during creation of a load module.

7.4.4 Library End Trailer

Following all object modules in a library file is a specially coded library end trailer which signifies the end of the file. This trailer is illustrated in Figure 7-4.

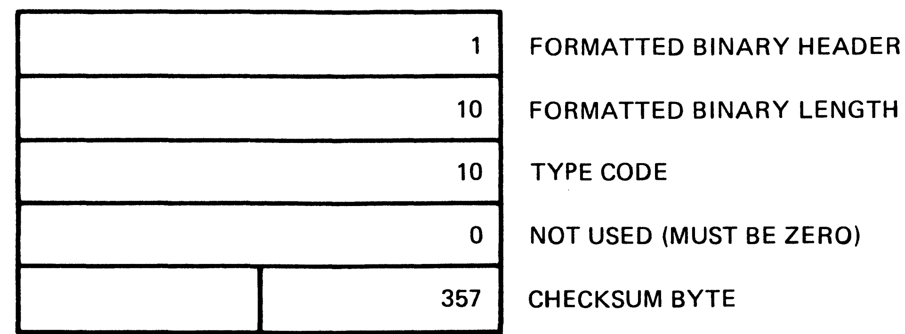


Figure 7-4 Library End Trailer

7.5 LIBR ERROR MESSAGES

The following error messages are printed following incorrect use of LIBR; if any errors result during library processing, the user must reenter the command.

Message	Meaning
?BAD LIBR?	The user has attempted to build a library file containing no directory entries or he has given an illegally constructed library file to the Librarian as input.
?BAD OBJ?	A bad object module was detected during input.
?CSECT ERROR?	The user has extended beyond the allowable .CSECT space for an object module to be placed in the library (i.e., the object module contains greater than 127 (decimal) .CSECTS).
?DEV FULL?	The device is full; LIBR is unable to create or update the indicated library file. The CSI prints an asterisk and waits for the user to enter another command line.
?FIL NOT FND?	One of the input files indicated in the command line was not found. The CSI prints an asterisk; the command may be reentered.
?ILL CMD?	An illegal command was used in the command line. The CSI prints an asterisk; the command may be reentered.
xxxxxx ILL DEL	An attempt was made to delete from the library's directory a module or an entry point that does not exist; xxxxxx represents the module or entry point name. The name is ignored and processing continues.
?ILL DEV?	An illegal device was specified in the command line. The CSI prints an asterisk; the command may be reentered.
xxxxxx ILL INS	An attempt was made to insert a module into a library which contains the same entry point as an existing module. xxxxxx represents the entry point name. The entry point is ignored but the module is still inserted into the library.
xxxxxx ILL REPL	An attempt was made to replace in the library file a module which does not already exist. xxxxxx represents the module name. The module is ignored and the library is built without it.
?IN ERR?	An unrecoverable hardware/software error has occurred while processing an input file. The CSI prints an asterisk and waits for another command to be entered.
?LIBR FIL ILL REPL?	The user has specified that a library file be replaced by another library file. Only object modules can be replaced.
?NO CORE?	Available free memory has been used up. The current command is aborted and the CSI prints an asterisk; a new command may be entered.
?OUT ERR?	An unrecoverable hardware/software error has occurred while processing an output file. This may indicate that there is not enough space left on the device to create the library file, even though there may be enough directory entries. The CSI prints an asterisk and waits for the user to enter another command.

CHAPTER 8
ON-LINE DEBUGGING TECHNIQUE

HT-11 On-Line Debugging Technique (ODT) is a system program that aids in debugging assembled and linked object programs. From the keyboard, the user interacts with ODT and the object program to:

1. Print the contents of any location for examination or alteration.
2. Run all or any portion of an object program using the breakpoint feature.
3. Search the object program for specific bit patterns.
4. Search the object program for words which reference a specific word.
5. Calculate offsets for relative addresses.
6. Fill a single word, block of words, byte or block of bytes with a designated value.

The assembly listing of the program to be debugged should be readily available when ODT is being used. Minor corrections to the program can be made on-line during the debugging session, and the program may then be run under control of ODT to verify any changes made. Major corrections, however (such as a missing subroutine), should be noted on the assembly listing and incorporated in a subsequent updated program assembly.

8.1 CALLING AND USING ODT

ODT is supplied as a relocatable object module. It can be linked with the user program (using the HT-11 Linker) for an absolute area in memory and loaded with the user program. When linked with the user program, ODT should reside in low memory, starting at 1000, to accommodate its stack.

Once loaded in memory with the user program, ODT has three legal start 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 ;G and ;P commands are used to run a program and are explained in Section 8.3.7.

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 memory. A breakpoint inserted in an overlay will be destroyed if it is overlaid during program execution.

Examples:

1. ODT Linked with the User Program:

.GET USER.SAV	User program previously linked to ODT is brought into memory.
.START 1172	Value (1172) of entry point O.ODT (determined from Linker load map) is used to start ODT.
ODT V01-01	
*	

2. Loading ODT with the User Program:

.GET ODT.SAV	ODT is loaded into memory.
.GET USER.SAV	User program is loaded into memory.
.START 40172	Assuming ODT has been linked for a bottom address of 40000, ODT starts.

```
ODT V01-02
*
```

3. 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.

4. Reentering ODT:

.START 1176 BE001212 *	Assuming ODT was linked for a bottom address of 1000, the value of O.ODT 1172+4 is used as the start address.

8.1.1 Return to Monitor, CTRL C

If ODT is awaiting a command, a CTRL C from the keyboard calls the HT-11 Keyboard Monitor. The monitor responds with a ↑C on the terminal and awaits a Keyboard Monitor command. (The monitor REENTER command may be used to reenter ODT only if the user program has set the reenter bit. Otherwise ODT is reentered at address O.ODT+4 as shown above.)

8.1.2 Terminate Search, CTRL U

If typed during a search printout, a CTRL U terminates the search and ODT prints an asterisk.

8.2 RELOCATION

When the assembler produces a binary 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, HT-11 ODT provides an automatic relocation facility.

The basis of the relocation facility lies in eight 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 Chapter 6, the HT-11 Linker.

8.2.1 Relocatable Expressions

A relocatable expression is evaluated by ODT as a 16-bit (6-digit octal) number and may be typed in any one of the three forms presented in Table 8-1. In this table, the symbol n stands for an integer in the range 0 to 7 inclusive, and the symbol k stands for an actal 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:

k (number typed)	Values
1	000001
−1	177777
400	000400
−177730	000050
1234567	034567

Table 8-1 Forms of Relocatable Expressions (r)

	r	Value of r
A)	k	The value of r is simply the value of k.
B)	n,k	The value of r is the value of k plus the contents of relocation register n. If the n part of this expression is greater than 7, ODT uses only the last octal digit of n.
C)	C or C,k or n,C or C,C	Whenever the letter C is typed, ODT replaces C with the contents of a special register called the Constant Register. This value has the same role as the k or n that it replaces (i.e., when used in place of n it designates a relocation register). The Constant Register is designated by the symbol $C and may be set to any value, as indicated below.

In the following examples, assume in each case that relocation register 3 contains 003400 and that the constant register contains 000003.

r	Value of r
5;C	000005
−17;C	177761
3,0;C	003400
3,150;C	003550
3,−1;C	003377
C;C	000003
3,C;C	003403
C,0;C	003400
C,10;C	003410
C,C;C	003403

NOTE

For simplicity most examples in this section use Form A. All three forms of r are equally acceptable, however.

8.3 COMMANDS AND FUNCTIONS

When ODT is started (as explained in Section 8.1) it indicates readiness to accept commands by printing an asterisk on the left margin of the terminal page. Most of the ODT commands can be issued in response to the asterisk. For example, a word can be examined and changed if desired, the object program can be run in its entirety or in segments, or memory can be searched for certain words or references to certain words. The discussion below explains these features. In the following examples, characters output by ODT are underlined to differentiate from user input.

8.3.1 Printout Formats

Normally, when ODT prints addresses (as with the commands ↓, ↑, ←, @, <, and >) it attempts to print them in relative form (Form B in Table 8-1). 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 contain 1000 and 1004 respectively, and all other relocation registers contain numbers much higher. Then the following sequence might occur (the slash command causes the contents of the location to be printed; the line feed command (<LF>) accesses the next sequential location):

```
*774/000000 <LF>
000776 /000000 <LF>
1, 000000 /000000 <LF>        (absolute location 1000)
1, 000002 /000000 <LF>        (absolute location 1002)
2, 000000 /000000             (absolute location 1004)
```

The printout 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 8.3.4, Accessing Internal Registers).

8.3.2 Opening, Changing, and Closing Locations

An open location is one whose contents ODT prints for examination, making those contents available for change. In a closed location, the contents are no longer available for change. Several commands are used for opening and closing locations.

Any command used to open a location when another location is already open causes the currently open location to be closed. 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., <LF>, ↑, RETURN, ←, @, >, <).

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 an 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 <CR>
*
```

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 <CR>
*/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. 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:

```
*1002/000004 \004 =
```

The LINE FEED Key, <LF>

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 <LF>          ( <LF> 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, ↑ or ^

If the up-arrow (or circumflex) is typed when a location is open, ODT closes the open location and opens the previous location. To continue from the example above:

```
*001002/012740 ↑
001000 /002340
```

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 the previous byte.

The Back-Arrow, ← or —
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 addressed location:

```
*1006/000006 ←
001016 /000405
```

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 Chapter 5.

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 <LF>      (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 by PC)
000202 /123456
```

Open the Addressed Location, @
The at symbol @ may be used to optionally modify a location, close it, and then use its contents as the address of the location to open next.

```
*1006/001044 @          (open location 1044 next)
001044 /000500

*1006/001044 2100@      (modify to 2100 and open location 2100)
002100 /000167
```

Relative Branch Offset, >
The right-angle bracket, >, will optionally modify a location, close it, and then use its low-order byte as a relative branch offset to the next word to be opened. For example:

```
*1032/000407 301>       (modify to 301 and interpret as a relative branch)
000636 /000010
```

Note that 301 is a negative offset (–77). The offset is doubled before it is added to the PC; therefore, 1034+(–176)=636.

Return to Previous Sequence, <
The left-angle bracket, <, allows the user to optionally modify a location, close it, and then open the next location of the previous sequence which was 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 <            (return to original sequence)
001034 /001040 @            (@ causes a sequence change)
001040 /000405 \005 = <     (< now operates on byte)
001035 \002 = <             (< acts like <LF>)
001036 \004 =
```

8.3.3 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 <CR>             (R0 was examined and closed)
*

*$4/000474 464<CR>          (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, back-arrow or @ command may be used when a register is open.

8.3.4 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 by typing $S. For example:

```
*$S/000311
```

$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 eight 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.

The $ is used to open certain other internal locations listed in Table 8-2.

8.3.5 Radix 50 Mode, X

The Radix 50 mode of packing certain ASCII characters three to a word is employed by many 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 responses in Table 8-3 can then be typed.

Table 8-2 Internal Registers

Register	Function
$B	location of the first word of the breakpoint table (see Section 8.3.6).
$M	mask location for specifying which bits are to be examined during a bit pattern search (see Section 8.3.9).
$P	location defining the operating priority of ODT (see Section 8.3.15).
$S	location containing the condition codes (bits 0-3) and interrupt priority level (bits 5-7) (explained above).
$C	location of the Constant Register (see Section 8.3.10).
$R	location of Relocation Register 0, the base of the Relocation Register table (see Section 8.3.13).
$F	location of Format Register (see Section 8.3.1).

Table 8-3 Radix 50 Terminators

Response	Effect
RETURN key <CR>	Closes the currently open location.
LINE FEED key <LF>	Closes the currently open location and opens the next one in sequence.
↑ key	Closes the currently open 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 for this last response are:	
. $ Space	0 through 9 A through Z

If any other characters are typed, the resulting binary number is unspecified (that is, no error message is printed and the result is unpredictable). 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 in the opened location by closing the location in any one of the ways listed in Table 8-3.

Example:

```
*1000/042431 X=KBI CBA <CR>
*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/013704
```

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 8.5 for a discussion of command legality and detection of errors.)

8.3.6 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 a 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. A breakpoint is set by typing the address of the desired location of the breakpoint followed by ;B. Thus r;B sets the next available breakpoint at location r. (If all 8 breakpoints have been set, ODT ignores the r;B command.) 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)
*
```

A table of breakpoints is kept by ODT and may be accessed by the user. 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 can be sequentially opened using the LINE FEED key. For example:

```
*$B/001020 <LF>
nnnnnn /001032 <LF>
nnnnnn / nnnnnn        (nnnnnn=address internal to ODT)
```

In this example, breakpoint 2 is not set. The contents printed is an address internal to ODT and can be determined by checking the Linker Load Map (see Chapter 6).

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.

8.3.7 Running the Program, r;G and r;P

Program execution is under control of ODT. There are two commands for running the program: r;G and r;P. The r;G command is used to start execution (Go) and r;P to continue (Proceed) execution after halting at a breakpoint. For example:

```
*1000;G
```

Execution is started at location 1000. The program runs until a breakpoint is encountered or until program completion, unless it gets caught in an infinite loop, in which case it must be either restarted or reentered as explained in Section 8.1.

Upon execution of either the r;G or r;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.

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 setting a proceed count using the k;P command; this command specifies the number of times the breakpoint is to be encountered before program execution is suspended (on the kth encounter). The count, k, refers only to the numbered breakpoint which most recently occurred. A different proceed count may be specified for the breakpoint when it is encountered. Thus:

```
B3;001010       (execution halted at breakpoint 3)
*1026;3B        (reset breakpoint 3 at location 1026)
*4;P            (set proceed count to 4 and continue execution; loop through breakpoint
B3;001026       three times and halt on fourth occurrence of the breakpoint)
*
```

Following the table of breakpoints (as explained in Section 8.3.6) is a table of proceed command repeat counts for each breakpoint. These repeat counts can be inspected by typing $B/ and nine LINE FEEDs. The repeat count for breakpoint 0 is printed (the first seven LINE FEEDs cause the table of breakpoints to be printed; the eighth types the single instruction mode, explained in the next section, and the ninth LINE FEED begins the table of proceed command repeat counts). The repeat counts for breakpoints 1 through 7, and the repeat count for the single-instruction trap follow in sequence. Before a proceed count is assigned a value by the user, it is set to 0; after the count has been executed, it is set to −1. Opening any one of these provides an alternative way of changing the count as the location, once open, can have its contents modified in the usual manner by typing the new contents and then the RETURN key.

For example:

```
        .
        .
nnnnnn /001036 <LF>        (address of breakpoint 7)
nnnnnn /006630 <LF>        (single instruction address)
nnnnnn /000000 15 <LF>     (count for breakpoint 0; change to 15)
nnnnnn /000000 <LF>        (count for breakpoint 1)
        .
        .
        .
nnnnnn /000000 <LF>        (count for breakpoint 7)
nnnnnn /nnnnnn             (repeat count for single instruction mode; the single instruction address is an
                           address internal to the user program if single instruction mode is used)
```

The address indicated as the single instruction address and the repeat count for single instruction mode are explained next.

8.3.8 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.

n;P — Proceeds with program run for next n instructions before reentering ODT (if n is missing, it is assumed to be 1). Trap instructions and associated handlers can affect the Proceed repeat count. See Section 8.4.2.

;S — Disables single instruction mode.

When the repeat count for single instruction mode is exhausted and the program suspends execution, ODT prints:

B8;nnnnnn

where nnnnnn 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 may 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 an assumed 1.

8.3.9 Searches

With ODT all or any specified portion of memory can be searched for any specific bit pattern or for references to a particular 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 FEEDs) 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 <LF>        (test high-order eight bits)
nnnnnn /000000 1000 <LF>      (set low address limit)
nnnnnn /000000 1040 <CR>      (set high address limit)
*400; W                       (initiate word search)
001010 /000770
001034 /000404
*
```

In the above example, nnnnnn is an address internal to ODT; this location varies and is meaningful only for reference purposes. In the first line above, the slash was used to open $M which now contains 177400; the LINE FEEDs opened the next two sequential locations which now contain the upper and lower limits of the search.

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 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 <LF>               (open mask register only to gain access to search limits)
nnnnnn /001000 1010 <LF>
nnnnnn /001040 1060 <CR>
*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 effective address references 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.

8.3.10 The Constant Register, r;C

It is often desirable to convert a relocatable address into its value after relocation 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 6-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 follow:

*–4432; C=173346	(the two's complement of 4432 is placed in the Constant Register)
*6632/062701 C <CR>	(the contents of the Constant Register are stored in location 6632)
*1000; 1R	(relocation Register 1 is set to 1000)
*1, 4272; C=005272	(relative location 4272 is reprinted as an absolute location and stored in the Constant Register)

8.3.11 Memory 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, and ASCII space.

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.

For example, assume relocation register 1 contains 7000, 2 contains 10000, and 3 contains 15000. The following sequence sets word locations 7000-7776 to zero, and byte locations 10000-14777 to ASCII spaces.

*$M/000000 <LF>	(open mask register to gain access to search limits)
nnnnnn /000000 1, 0 LF>	(set lower limit to 7000)
nnnnnn /000000 2, –2 <LF>	(set upper limit to 7776)
*0; C=000000	(Constant Register set to zero)
*; F	(Locations 7000-7776 set to zero)
*$M/000000 <LF>	
nnnnnn /007000 2, 0 <LF>	(set lower limit to 10000)
nnnnnn /007776 3, –1 <CR>	(set upper limit to 14777)
*40; C=000040	(Constant Register set to 40 (SPACE))
*; I	
*	(Byte locations 10000-14777 are set to value in low-order 8 bits of Constant Register)

8.3.12 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 <CR>
*/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(decimal) to 127(decimal) 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 <CR>
*/103777
```

Note that the modified low-order byte 377 must be combined with the unmodified high-order byte.

8.3.13 Relocation Register Commands, r;nR, ;nR, ;R

The use of the relocation registers is defined in Section 8.2. 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, as follows:

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)
*
```

Once a relocation register is defined, it can be used to reference relocatable values. For example:

```
*2000; 1R          (puts 2000 into relocation register 1)
*1, 2176/002466    (examines contents of location 4176)
*1, 3712; 0B       (sets a breakpoint at location 5712)
```

In certain uses, programs may be relocated to an address below that at which they were assembled. This could occur with PIC code (Position Independent Code) which is moved without the use of the Linker. In this case the appropriate relocation bias would be the two'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.

Assume a program was assembled at location 5000 and was moved to location 1000. Then the sequence:

```
*1000; 1R
*1, −5000; 1R
*
```

enters the two'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 a line feed opens the other relocation registers in sequence. When a relocation register is opened in this way, it may be modified like any other memory location.

8.3.14 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 7000; then:

```
*1, 500/000000 1R=1, 171000
```

The value 171000 is the content of 1,500, relative to the base 7000. An example of the use of both relocation calculators follows.

If relocation register 1 contains 1000, and relocation register 2 contains 2000, then to calculate the relocatable addresses of location 3000 and its contents, relative to 1000 and 2000, the following can be performed.

```
*3000/000410 1!=1, 002000 2!=2, 001000 1R=1, 177410 2R=2, 176410
```

8.3.15 ODT 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 operates.

To set ODT to the desired priority level, open $P. ODT prints the present contents, which may then be changed:

```
*$P/000006 377 <CR>
*
```

If $P is not specified, its value is 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, ODT operates at a low priority; if an interrupt occurs from a high priority routine, the breakpoints in the high priority routine will not be recognized since they were removed when the low priority breakpoint occurred. That is, interrupts set at a priority higher than the one at which ODT is running will occur and any breakpoints will not be recognized. ODT disables all breakpoints from the program whenever it gains control.

Breakpoints are enabled when ;P and ;G commands are executed. For example:

```
*$P/000007 5
*1000; B
*2000; B
*1000; G
B0; 001000
*                        (an 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.

8.3.16 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:

<CR> — ODT outputs a carriage return/line feed and an asterisk and waits for another command.

<LF> — ODT opens the byte following the last byte output.

Up to n characters of text — ODT inserts the text into memory, starting at location r. If fewer 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 the next available byte and a carriage return/line feed/asterisk.

ODT does not check the magnitude of n.

8.4 PROGRAMMING CONSIDERATIONS

Information in this section is not necessary for the efficient use of ODT. However, it does provide a better understanding of how ODT performs some of its functions and in certain difficult debugging situations, this understanding is necessary.

8.4.1 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.

8.4.2 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 was set must not be referenced by the program in any way since ODT 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 terminal 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 terminal 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 (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 regain control 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 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 8.1). 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.

Note the following rules concerning the ODT break routine:

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.

8.4.3 Searches
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 to 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 to 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.

8.4.4 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.

1 026 3665 3
(on 2)

To restore the TT:

1. Wait for completion of any I/O from ODT.
2. Restore the TKS.
3. Restore the TPS.

NOTES

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.

8.5 ODT 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/
?M-TRAP TO 4 003362
```

references nonexistent memory, thereby causing a trap through the vector at location 4. If this vector has not been properly initialized, unpredictable results occur.

Typing something other than a legal command causes ODT to ignore the command, print:

```
(echoes illegal command)?
*
```

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:

```
BEnnnnnn
*
```

and waits for another command. BEnnnnnn denotes Bad Entry from location nnnnnn. 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.

CHAPTER 9
PROGRAMMED REQUESTS

A number of services at the machine language level which the monitor regularly provides to system programs are also available to user-written programs. These include services for file manipulation, command interpretation, and facilities for input and output operations. User programs call these monitor services by means of "programmed requests", which are assembler macro calls written into the user program and interpreted by the monitor at program execution time.

The macro definitions for programmed requests are included in the file SYSMAC.SML; Appendix C provides a listing of SYSMAC.SML. Refer to Chapter 5 for general information related to the use of macro calls.

9.1 FORMAT OF A PROGRAMMED REQUEST

The basis of a programmed request is the EMT instruction, used to communicate information to the monitor. When an EMT is executed, control is passed to the monitor, which extracts appropriate information from the EMT and executes the function required. The low-order byte of the EMT instruction contains a code which is interpreted as:

Low-Order Byte of EMT	Meaning
377	Reserved; HT-11 ignores this EMT and returns control to the user program immediately.
376	Used internally by the HT-11 monitor; this EMT code should never be used by user programs.
375	Programmed request with several arguments: R0 must point to a list of arguments which designates the specific function.
374	Programmed request with one argument: R0 contains a function code in the high-order byte and a channel number (see Section 9.2.1) or 0 in the low-order byte.
360–373	Used internally by the HT-11 monitor; these EMT codes should never be used by user programs.
340–357	Programmed request with arguments on the stack and/or in R0.
0–337	Reserved

A programmed request consists of a macro call followed, where necessary, by one or more arguments. Arguments supplied to a macro call must be legal assembler expressions since arguments will be used as source fields in MOV instructions when the macros are expanded. The following two formats are used:

1. PRGREQ ARG1,ARG2,. . . ARGN
2. PRGREQ AREA,ARG1,ARG2,. . . ARGN

Form 1 above contains the arguments ARG1 through ARGN; no argument list pointer is required. Macros of this form generate either an EMT 374 or one of the EMTs 340–357. Certain arguments for this form may be omitted; refer to the listing of SYSMAC.SML in Appendix C.

In form 2 above, AREA is a pointer to the argument list which contains the arguments ARG1 through ARGN. This form always causes an EMT 375 to be generated. Blank fields are permitted; however, if the AREA argument is blank, the macro assumes that R0 points to a valid argument block (see Section 9.2.3). If any of the fields ARG1 to ARGN are blank, the corresponding entries in the argument list are left untouched. Thus,

```
.PRGREQ AREA,A1,A2
```

points R0 to the argument block at AREA and fills in the first and second arguments, while:

```
.PRGREQ AREA
```

points R0 to the block, but does not fill in any arguments.

The call:

```
.PRGREQ ,A1
```

assumes R0 points to the argument block and fills in the A1 argument, but leaves the A2 argument alone. The call:

```
.PRGREQ
```

generates only an EMT 375 and assumes that both R0 and the block to which it points are properly set up.

The arguments to HT-11 programmed request macros all serve as the source field of a MOV instruction which moves a value into the argument block or R0. For example:

```
.PRGREQ CHAR
```

expands into:

```
MOV  CHAR,R0
EMT  357
```

Care should be taken to make certain that the arguments specified are legal source fields and that the address accurately represents the value desired. If the value is a constant, immediate mode [#] should be used; if the value is in a register, the register mnemonic [Rn] should be used; if the value is indirectly addressed, the appropriate register convention is necessary [@Rn], and if the value is in memory, the label of the location whose value is the argument is used.

Following are some examples of both correct and incorrect macro calls. Consider the general request:

```
.PRGREQ .AREA, .ARG1, . . . ARGN
```

A more common way of writing a request of this form is:

```
.PRGREQ #AREA,#ARG1, . . . #ARGN
```

In this format, the address of AREA is put directly into the argument list. AREA is the tag which indicates the beginning of the argument block. For example:

```
        .PRGREQ #AREA,#4
              .
              .
              .
AREA:   .BLKW 3
```

When a direct numerical argument is required, the # causes the correct value to be put into the argument block. For example:

```
        .PRGREQ #AREA,#4
```

is correct, while:

```
        .PRGREQ #AREA,4
```

is not. This form interprets the 4 as meaning "move the contents of location 4 into the argument block", where the number 4 itself should be moved into the block.

If the request is written as:

```
        .PRGREQ AREA,#4
```

it is interpreted as "use the contents of location AREA as the list pointer", when the address of AREA is actually desired. This expansion could be used with the following form:

```
                .PRGREQ LIST1,#4
                      .
                      .
                      .
        LIST1:  AREA
        AREA:   .BLKW3
```

In this case, the content of location LIST1 is the address of the argument list. Similarly, this form is correct:

```
                .PRGREQ LIST1,NUMBER
        LIST1:  AREA
        NUMBER: 4
```

In this case, the contents of the locations LIST1 and NUMBER are the argument list pointer and data value, respectively.

NOTE

All registers except R0 are preserved across a programmed request. (In certain cases, R0 may contain information passed back by the monitor; however, unless the description of a request indicates that a specific value is returned in R0, it may be assumed that the contents of R0 are unpredictable upon return from the request). With the exception of calls to the CSI, the position of the stack pointer is also preserved across a programmed request.

9.2 SYSTEM CONCEPTS

Some basic operational characteristics and concepts of HT-11 are described below.

9.2.1 Channel Number (chan)

A channel number is a logical identifier in the range 0 to 377 (octal) for a file or "set of data" used by the HT-11 monitor. Thus, when a file is opened on a particular device, a channel number is assigned to that file. To refer to an open file, it is only necessary to refer to the appropriate channel number for that file.

9.2.2 Device Block (dblk)
A device block is a four-word block of radix-50 information which specifies a physical device and file name for an HT-11 programmed request. (Refer to Chapter 5 for an explanation of .RAD50 strings.) For example, a device block representing a file FILE.EXT on device DK: could be written as:

```
.RAD50   /DK /
.RAD50   /FIL/
.RAD50   /E  /
.RAD50   /EXT/
```

The first word contains the device name, the second and third words contain the file name, and the fourth contains the extension. Device, name, and extension must each be left-justified in the appropriate field. This string could also be written as:

```
.RAD50 /DK FILE  EXT/
```

Note that spaces must be used to fill out each field. Note also that the colon and period separators do not appear in the actual RAD50 string. They are used only by the monitor keyboard interface to delimit the various fields.

9.2.3 EMT Argument Blocks
Programmed requests which call the monitor via EMT 375 use R0 as a pointer to an argument list. In general, the argument list appears as follows:

contents		address
function code	channel number	x
argument 1		x+2
argument 2		x+4
.		.
.		.
.		.

R0 points to location x. The even (low-order) byte of location x contains the channel number named in the macro call. If no channel number is required, the byte is set to 0. The odd (high-order) byte of x is a code specifying the function to be performed. Locations x+2, x+4, etc. contain arguments to be interpreted. These are described in detail under each request.

Requests which use EMT 374 set up R0 with the channel number in the even byte and the function code in the odd byte. They require no other arguments.

9.2.4 Important Memory Areas

9.2.4.1 Vector Addresses (0–37, 60–477) – Certain areas of memory between 0 and 477 are reserved for use by HT-11. KMON does not load these locations from the save image file when it initiates a program, i.e., R, RUN, and GET will not load these words. However, no hardware memory protection is supplied. Thus, programs should never alter the contents of the indicated areas at run-time.

Locations	Contents
0,2	Monitor restart. Executes .EXIT request and returns control to KMON.
4,6	Time out or bus error trap; HT-11 sets this to point to its internal trap handler.
10,12	Reserved instruction trap; HT-11 sets this to point to its internal trap handler.

Locations	Contents
30,32	EMT trap vector and status.
40–57	HT-11 system communication area (see below).
60,62	TTY input interrupt vector and status.
64,66	TTY output interrupt vector and status.
100,102	Line Time Clock vector and status.

These areas are not replaced by HT-11. If they are destroyed by a program, the system must be re-bootstrapped, or the program must restore them.

9.2.4.2 Resident Monitor – Section 2.4 of Chapter 2 describes the placement of monitor components when the monitor is brought into memory; included is the approximate size of each monitor component and the size of the area available for handlers and user programs.

9.2.4.3 System Communication Area – HT-11 uses bytes 40-57 to hold information about the program currently executing, as well as certain information used only by the monitor. A description of these bytes follows:

Bytes	Meaning and Use
40,41	Start address of job. When a file is linked into an HT-11 memory image, this word is set to the starting address of the job either with the Linker /T switch or as an argument in the .END statement of the program.
42,43	Initial value of the stack pointer. If it is not set by the user program in an .ASECT, it defaults to 1000 or the top of the .ASECT, whichever is larger. The initial stack pointer can also be set with the Linker /M switch option.
44,45	Job Status Word. Used as a flag word for the monitor. Certain bits are maintained by the monitor exclusively while others must be set or cleared by the user job. Those bits in the following list which are marked by an asterisk are bits which must be set by the user job.

Since the currently unassigned bits may be used in future releases of HT-11, user programs should not use these bits for internal flags.

Bit Number	Meaning
15	USR swap bit. The monitor sets this bit when programs do not require the USR to be swapped. See Section 9.2.5 for details on USR swapping.
14	Lower-case bit. When set (for example, by EDIT when the EL command is typed), disables conversion of lower-case to upper-case.
*13	Reenter bit. When set, this bit indicates that the program may be restarted from the terminal with the REENTER command.

Bit Number	Meaning
*12	Special mode TT bit. When set, this bit indicates that the job is in a "special" keyboard mode of input. Refer to the explanation of the .TTYIN/.TTINR requests for details.
11–10	Unused
9	Overlay Bit. Set if the job uses the Linker overlay structure.
8	CHAIN bit. If this bit is set in a job's save image, words 500–776 are loaded from the save file when the job is started even if the job is entered via CHAIN. (These words are normally used to pass parameters across CHAINs.) The bit is set when a job is running if and only if the job was actually entered with CHAIN.
*7	Error halt bit. When set, this bit indicates a halt on an I/O error. If the user desires to halt when any I/O device error occurs, this bit should be set.
6–0	Unused

46,47 USR load address. Normally 0, this word may be set to any valid word address in the user's program. See Section 9.2.5, Swapping Algorithm, for details of use.

50,51 High memory address. The monitor maintains the highest address the user program can use in this word. The Linker sets it initially. It is modified only via the .SETTOP (Set Top of Memory) 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 and the carry bit is set. Each monitor call has its own set of possible errors. It is recommended that the user program reference byte 52 with absolute addressing, rather than relative addressing. For example:

```
ERRWRD = 52
TSTB ERRWRD        ;RELATIVE ADDRESSING
TSTB @#ERRWRD      ;ABSOLUTE ADDRESSING
```

NOTE

Location 52 must always be addressed as a byte, never as a word, since byte 53 will be used in future releases of HT-11.

53 Reserved for future system use.

54,55 Address of the beginning of the Resident Monitor. HT-11 always loads the resident into the highest available memory locations; this word points to its first location. It must never be altered by the user. Doing so will cause HT-11 to malfunction.

56 Fill character (7-bit ASCII). Some high-speed terminals require filler (null) characters after printing certain characters. Byte 56 should contain the ASCII 7-bit representation of the character after which fillers are required.

57 Fill count. This byte specifies the number of fill characters required. If bytes 56 and 57 = 0, no fillers are required.

Bytes	Meaning and Use

Examples of fill characters are:

No. of Fills	Value of Word 56
10 after carriage return	5015
4 after carriage return	2015
2 after carriage return	1015
4 after line feed	2012
2 after line feed	1012
1 after line feed	412

9.2.5 Swapping Algorithm

Programmed requests are divided into two categories according to whether or not they require the USR to be in memory (see Table 9-2). Any request which requires the USR in memory may also require that a portion of the user program be saved temporarily on the system device scratch blocks (i.e., be "swapped out") to provide room for the USR. The USR will be read into the swapped region.

During most normal operations, this swapping is invisible to the user and he need not be concerned about it. However, it is possible to optimize programs so that they require little or no swapping. If the USR is not swapped, the job will not be slowed down by the swapping process.

The following should be considered if a swap operation is necessary:

1. If a .SETTOP request specifies an address beyond the point at which the USR normally resides, a swap will be required when the USR is called. More details concerning the .SETTOP request are in Section 9.4.27.
2. If the user either assembles an address into word 46 or moves a value there while the program is running, HT-11 uses the contents of that word as an alternate place to swap the USR. If location 46 is 0, this indicates that the USR will be at its normal location in high memory.

NOTES

1. If the USR does not require swapping, the value in location 46 is ignored. Swapping is a relatively time-consuming operation and is avoided, if possible.
2. Care should be taken when specifying an alternate address in location 46. The system does not verify the legality of the USR swap address. Thus, if the area to be swapped overlays the Resident Monitor, the system is destroyed.
3. The user should also take care that the USR is never swapped over any of the following areas: the program stack; any parameter block for calls to the USR; any I/O buffers, device handlers, or completion routines being used when the USR is called.

The following is an example of the way a program can avoid unnecessary USR swapping.

```
              .MCALL      .REGDEF,.SETTOP,.EXIT
              .REGDEF
RMPTR=54                         ;POINTER TO RMON IS AT 54.
USRLOC=266                       ;POINTER TO USR LOCATION IS
                                 ;AT 266 BYTES INTO RMON.
START:
              MOV         @#RMPTR,R1         ;R1 -> RESIDENT MONITOR
              MOV         USRLOC(R1),R0      ;R0 -> USR
              TST         -(R0)              ;POINT JUST BELOW
              CMP         R0,@#50            ;DOES USR SWAP OVER US?
              BHI         1$                 ;NO, OK
              MOV         #-2,R0             ;YES, USR MUST SWAP
1$:           .SETTOP                        ;ASK FOR MEMORY UP TO USR
              MOV         R0,HILIM           ;R0 = HIGH LIMIT OF MEMORY
                                             ;ACTUALLY GRANTED BY MONITOR.
              .EXIT
HILIM:        .WORD       0                  ;CONTAINS HI LIMIT OF MEMORY
              .END        START
```

9.2.6 Offset Words

There are several words which always have fixed positions relative to the start of the Resident Monitor. It is often advantageous for user programs to be able to access these words. This is done with the code:

```
RMON = 54
MOV @ #RMON,register
MOV OFFSET(register),register
```

Here, register is any general register and OFFSET is a number from the following list:

OFFSET (Bytes)	Contents
262	System date. (See .DATE request.)
266	Pointer to start of normal USR area. This is where the USR will reside when it is non-swapping. It is useful to be able to perform a .SETTOP in a job such that the USR is always resident. (An example is in Section 9.2.5.)
270	Address of I/O exit routine for all devices. The exit routine is an internal queue management routine through which all device handlers exit once the I/O transfer is complete. Any new devices added to HT-11 must also use this exit location.
275	Unit number of system device (device from which system was last bootstrapped).
276	Monitor version number. The user can always access the version number to determine if the most recent monitor is in use.
277	Update number. Patches to the monitor always increment the update number. This provides a means of checking that all patches have been made. (This number should be accessed by MOVB rather than MOV.)

OFFSET (Bytes)	Contents
300	Configuration word. This is a string of 16 bits used to indicate information about either the hardware configuration of the system, or a software condition. The bits and their meanings are:

Bit #	Meaning
5	0 = 60-cycle clock 1 = 50-cycle clock
9	1 = USR is permanently resident (via a SET USR NOSWAP)
11	1 = Processor is an 11/03
15	1 = KW11L clock is present (always set if 11/03)

The other bits are reserved for future use and should not be accessed by user programs.

OFFSET (Bytes)	Contents
304-313	These locations contain the addresses of the terminal control and status registers. The order is:

304	Keyboard status
306	Keyboard buffer
310	Printer status
312	Printer buffer

These locations can be changed, for example, to reflect a second terminal; thus HT-11 can be made to run on any terminal present on the system which is connected to the machine via the DL11 multiple terminal interface.

OFFSET (Bytes)	Contents
314	The maximum file size allowed in a 0 length .ENTER. This can be adjusted by the user program or by using the PATCH program to be any reasonable value. The default value is 177777 (octal) blocks, allowing an essentially unlimited file size.

9.2.7 File Structure

HT-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 occupies 9 contiguous blocks on the device.

A contiguous area on a device can be in one of the following categories:

1. Permanent file. This is a file which has been .CLOSEd on a device. Any named files which appear in a PIP directory listing are permanent files.
2. Tentative file. Any file which has been created via .ENTER, but not .CLOSEd, is a tentative file entry. When the .CLOSE request is given, the tentative entry becomes a permanent file. If a permanent file already exists under the same name, the old file is deleted. If a .CLOSE is never given, the tentative file is treated like an empty entry.
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 /E directory listing as 〈UNUSED〉 N, where N is the decimal block length of the empty area.

Since a contiguous structure does not automatically reclaim unused disk space, the device may eventually become "fragmented". A device is fragmented when there are many empty entries which are scattered over the device. HT-11 PIP has an option which allows the user to collect all empty areas so that they occur at the end of a device. Refer to Chapter 4 for details.

9.2.8 Completion Routines

Completion routines are user-written routines which are entered following an operation. On entry to a completion routine, R0 contains the channel status word for the operation; R1 contains the octal channel number of the operation. The carry bit is not significant.

The restrictions which must be observed when writing completion routines 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 memory 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 overlaid 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.
4. If a completion routine uses registers other than R0 or R1, it must save them upon entry and restore them before exiting.

9.2.9 Using the System Macro Library

User programs for HT-11 should always be written using the system macro library (SYSMAC.SML), supplied with HT-11. This ensures compatibility among all user programs and allows easy modification by redefining a macro. A listing of SYSMAC.SML appears in Appendix C.

9.3 TYPES OF PROGRAMMED REQUESTS

There are three types of services which the monitor makes available to the user through programmed requests. These are:

1. Requests for File Manipulation
2. Requests for Data Transfer
3. Requests for Miscellaneous Services

Table 9-1 summarizes the programmed requests in each of these categories alphabetically. The EMT and function code for each request (where applicable) are included.

Table 9-2 indicates which requests require the USR to be in memory. The CLOSE request on non-file structured devices (LP, PP, TT, etc.) does not require the USR.

9.3.1 System Macros

The following four macros are included in the system macro library, but are not programmed requests in that they cause no EMT instruction to be generated:

.DATE	.SYNCH
.INTEN	
.REGDEF	

They can be used in the same manner as the other macro calls; their explanations follow.

Table 9-1 Summary of Programmed Requests

Mnemonic	EMT & Code		Section	Purpose
File Manipulation Requests				
.CLOSE	374	6	9.4.3	Closes the specified channel.
.DELETE	375	0	9.4.6	Deletes the file from the specified device.
.ENTER	375	2	9.4.8	Creates a new file for output.
.LOOKUP	375	1	9.4.16	Opens an existing file for input and/or output via the specified channel.
.RENAME	375	4	9.4.24	Changes the name of the indicated file to a new name.
.REOPEN	375	6	9.4.25	Restores the parameters stored via a SAVESTATUS request and reopens the channel for I/O.
.SAVESTATUS	375	5	9.4.26	Saves the status parameters of an open file in user memory and frees the channel for future use.
Data Transfer Requests				
.READ	375	10	9.4.22	Transfers data via the specified channel to a memory 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.
.READC	375	10	9.4.22	Transfers data via the specified channel to a memory 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 asynchronously to the routine specified in the .READC request.
.READW	375	10	9.4.22	Transfers data via the specified channel to a memory buffer and returns control to the user program only after the transfer is complete.
.TTYIN .TTINR	340	–	9.4.31	Transfers one character from the keyboard buffer to R0.
.TTYOUT .TTOUTR	341	–	9.4.32	Transfers one character from R0 to the terminal input buffer.
.WRITE	375	11	9.4.34	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.

Table 9-1 (Cont.) Summary of Programmed Requests

Mnemonic	EMT & Code		Section	Purpose
.WRITC	375	11	9.4.34	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 asynchronously to the routine specified in the .WRITC request.
.WRITW	375	11	9.4.34	Transfers data via the specified channel to a device and returns control to the user program only after the transfer is complete.
Miscellaneous Services				
.CDFN	375	15	9.4.1	Defines additional channels for doing I/O.
.CHAIN	374	10	9.4.2	Chains to another program.
.CSIGEN	344	–	9.4.4	Calls the Command String Interpreter (CSI) in general mode.
.CSISPC	345	–	9.4.5	Calls the CSI in special mode.
.DATE	–	–	9.3.1.1	Moves the current date information into R0.
.DSTATUS	342	–	9.4.7	Returns the status of a particular device.
.EXIT	350	–	9.4.9	Exits the user program and returns control to the Keyboard Monitor.
.FETCH	343	–	9.4.10	Loads device handlers into memory.
.GTIM	375	21	9.4.11	Gets time of day.
.GTJB	375	20	9.4.12	Gets parameters of this job.
.HERR	374	5	9.4.13	Specifies termination of the job on fatal errors.
.HRESET	357	–	9.4.14	Terminates I/O transfers and does a .SRESET operation.
.INTEN	–	–	9.3.1.2	Notifies monitor that an interrupt has occurred and to switch to "system state", and sets the processor priority to the correct value.
.LOCK	346	–	9.4.15	Makes the monitor User Service Routines (USR) permanently resident until .EXIT or .UNLOCK is executed. The user program is swapped out if necessary.

Table 9-1 (Cont.) Summary of Programmed Requests

Mnemonic	EMT & Code		Section	Purpose
.PRINT	351	–	9.4.17	Outputs an ASCII string to the terminal.
.PROTECT	375	31	9.4.18	Reserves a vector in the region 0 – 476 for an interrupt address and priority.
.PURGE	374	3	9.4.19	Clears out a channel.
.QSET	353	–	9.4.20	Expands the size of the monitor I/O queue.
.RCTRLO	355	–	9.4.21	Enables output to the terminal.
.REGDEF	–	–	9.3.1.3	Defines the PDP-11 general registers.
.RELEAS	343	–	9.4.23	Removes device handlers from memory.
.SERR	374	4	9.4.13	Inhibits most fatal errors from causing the job to be aborted.
.SETTOP	354	–	9.4.27	Specifies the highest memory location to be used by the user program.
.SFPA	375	30	9.4.28	Sets user interrupt for floating point processor exceptions.
.SRESET	352	–	9.4.29	Resets all channels and releases the device handlers from memory.
.SYNCH	–	–	9.3.1.4	Enables user program to perform monitor programmed requests from within an interrupt service routine.
.TRPSET	375	3	9.4.30	Sets a user intercept for traps to locations 4 and 10.
.UNLOCK	347	–	9.4.15	Releases USR if a LOCK was done. The user program is swapped in if required.
.WAIT	374	0	9.4.33	Waits for completion of all I/O on a specified channel.

Table 9-2 Requests Requiring the USR

Request	USR Required
.CDFN	Yes
.CHAIN	No
.CLOSE (see Note 1)	Yes
.CSIGEN	Yes
.CSISPC	Yes
.DELETE	Yes
.DSTATUS	Yes
.ENTER	Yes
.EXIT	No
.FETCH	Yes
.GTIM	No
.GTJB	No
.HERR	No
.HRESET	Yes
.LOCK (see Note 2)	Yes
.LOOKUP	Yes
.PRINT	No
.PURGE	No
.QSET	Yes
.RCTRLO	No
.READ/.READC/.READW	No
.RELEAS	Yes
.RENAME	Yes
.REOPEN	No
.SAVESTATUS	No
.SERR	No
.SETTOP	No
.SFPA	No
.SPFUN	No
.SRESET	Yes
.TRPSET	No
.TTINR/.TTYIN	No
.TTOUTR/.TTYOUT	No
.UNLOCK	No
.WAIT	No
.WRITE/.WRITC/.WRITW	No

Note 1. Only if channel was opened via .ENTER.
Note 2. Only if USR is in a swapping state.

.DATE

9.3.1.1 .DATE

This request moves the current date information from the system date word into R0. The date word returned is in the following format:

Bit:	15	14 – 10	9 – 5	4 – 0
		MONTH (1-12.)	DAY (1-31.)	YEAR-72 (DECIMAL)

Macro Call: .DATE

Errors:

No errors are returned. A zero result in R0 indicates that no DATE command was entered.

.INTEN

9.3.1.2 .INTEN

This request is used by user program interrupt service routines to:

1. Notify the monitor that an interrupt has occurred and to switch to "system state",
2. Set the processor priority to the correct value.

In HT-11, all external interrupts cause the processor to go to level 7. .INTEN is used to lower the priority to the value at which the device should be run. On return from .INTEN, the device interrupt can be serviced, at which point the interrupt routine returns via an RTS PC. It is very important to note that an RTI will not return correctly from an interrupt routine which specifies an .INTEN.

Macro Call: .INTEN .priority, pic

where:

.priority — is the processor priority at which the user wishes to run his interrupt routine.

pic — is an operational argument which should be non-blank if the interrupt routine is written as a PIC (position independent code) routine. If the routine does not have to be PIC, it is recommended that the PIC field be left blank; the non-PIC version is slightly faster than the PIC version.

Errors:

None.

Example:

Refer to Section 9.3.1.4, .SYNCH, for an example.

.REGDEF

9.3.1.3 .REGDEF

This macro call defines the PDP-11 general registers as R0 through R5, SP, and PC.

Macro Call: .MCALL .REGDEF,...
.REGDEF

Errors:

None.

Example:

Refer to the example for the .SYNCH request. Appendix C shows the expansion of .REGDEF.

.SYNCH

9.3.1.4 .SYNCH

This macro call enables the user program to perform monitor programmed requests from within an interrupt service routine. Unless a .SYNCH is used, issuing programmed requests from interrupt routines is not supported by the system and should not be performed. .SYNCH, like .INTEN and .DATE, is not a programmed request and generates no EMT instructions.

Macro Call: .SYNCH .area

where: .area is the address of a seven-word area which the user must set aside for use by .SYNCH. The 7-word block appears as:

Word 1	HT-11 maintains this word; its contents should not be altered by the user.
Word 2	The current job's number. This can be obtained by a .GTFB call.
Word 3	Unused.
Word 4	Unused.
Word 5	R0 argument. When a successful return is made from .SYNCH, R0 contains the argument.
Word 6	Must be −1.
Word 7	Must be 0.

NOTE

.SYNCH assumes that the user has not pushed anything on the stack between the .INTEN and .SYNCH calls. This rule must be observed for proper operation.

Errors:

The monitor returns to the location immediately following the .SYNCH if the .SYNCH was rejected. The routine is still unable to issue programmed requests, and R4 and R5 are available for use. Errors returned are due to one of the following:

1. Another .SYNCH which specified the same 7-word block is still pending.
2. An illegal job number was specified in the second word of the block.
3. If the job has been aborted or for some reason is no longer running, the .SYNCH will fail.

Normal return is to the word after the error return with the routine in user state and thus allowed to issue programmed requests. R0 contains the argument which was in word 5 of the block. R0 and R1 are free to be used without having to be saved. (R4 and R5 are not free.) Exit from the routine should be done via an RTS PC.

Example:

```
          .MCALL    .REGDEF
          .REGDEF
          .MCALL    .GTJB,.INTEN,.WRITC,.SYNCH,.EXIT,.PRINT
START:    MOV       #JOB,R5        ;OUTPUT OF .GTJB GOES HERE
          .GTJB     #AREA,R5       ;GET JOB NUMBER
          MOV       (R5),SYNBLK+2  ;STORE THE JOB NUMBER INTO SYNCH BLOCK
                                   ;IN HERE WE SET UP INTERRUPT
                                   ;PROCESSING, AND START UP THE
                                   ;INTERRUPTING DEVICE.
                         .
                         .
                         .
INTRPT:   .INTEN    5              ;GO INTO SYSTEM STATE
                                   ;RUN AT LEVEL FIVE
                                   ;INTERRUPT PROCESSING
                                   ;NOTHING CAN GO ON STACK
          .SYNCH    #SYNBLK        ;TIME TO WRITE A BUFFER
          BR        SYNFAIL        ;SYNCH BLOCK IN USE

;RETURN HERE AT PRIORITY 0. NOTE: .SYNCH DOES RTI

          .WRITC    #AREA,CHAN,BUFF,WCNT,#CRTN1,BLK
                                   ;WRITE A BUFFER
          BCS       WTFAIL         ;FAILED SOMEHOW
                         .
                         .
                         .
                                   ;RE-INITIALIZE FOR MORE
          RTS       PC             ;INTERRUPTS AND EXIT
SYNBLK:   .WORD     0
          .WORD     0              ;JOB NUMBER
          .WORD     0
          .WORD     0
          .WORD     5              ;R0 CONTAINS 5 ON SUCCESSFUL
                                   ;SYNCH
          .WORD     -1,0           ;SET UP FOR MONITOR
SYNFAIL:  .
          .
          .
```

9.4 PROGRAMMED REQUEST USAGE

This section provides a description of each of the programmed requests alphabetically. The following parameters are commonly used as arguments in the various calls:

.addr	an address, the meaning of which depends on the request being used
.area	a pointer to the EMT argument list (for those requests which require a list); see Section 9.2.3
.blk	a block number specifying the relative block in a file where an I/O transfer is to begin
.buff	a buffer address specifying a memory location into or from which an I/O transfer is to be performed
.chan	a channel number in the range 0–377 (octal)
.crtn	the entry point of a completion routine; see Section 9.2.8
.dblk	the address of a four-word RAD50 descriptor of the file to be operated upon; see Section 9.2.2
.num	a number, the value of which depends on the request
.wcnt	a word count specifying the number of words to be transferred to or from the buffer during an I/O operation

Additional information concerning these parameters (and others not defined here) is provided as necessary under each request.

9.4.1 .CDFN

The .CDFN request is used to redefine the number of I/O channels. Each job is initially provided with 16 (decimal) I/O channels, numbered 0–15. .CDFN allows the number to be expanded to as many as 255 (decimal) channels.

Note that .CDFN defines new channels; the previously-defined channels are not used. Thus, a .CDFN for 20 (decimal) channels (while the 16 original channels are defined) causes only 20 I/O channels to be available; the space for the original 16 is unused.

Note that if a program is overlaid, channel 15 is used by the overlay handler and should not be modified. (Other channels can be defined and used as usual.)

Macro Call: .CDFN .area, .addr, .num

where:

.addr	is the address where the I/O channels begin
.num	is the number of I/O channels to be created

Request Format:

R0 ⇒ .area:

15	0
.addr	
.num	

The space used to contain the new channels is taken from within the user program. Each I/O channel requires 5 words of memory. Thus, the user must allocate 5*N words of memory, where N is the number of channels to be defined.

It is recommended that the .CDFN request be used at the beginning of a program, before any I/O operations have been initiated. If more than one .CDFN request is used, the channel areas must either start at the same location or not overlap at all. The two requests .SRESET and .HRESET cause the user's channel areas to revert to the original 16 channels defined at program initiation. Hence, any .CDFNs must be reissued after using those directives.

Errors:

Code	Explanation
0	An attempt was made to define fewer channels than already exist.

Example:

```
          .MCALL   .REGDEF
          .REGDEF
          .MCALL   .CDFN,.PRINT,.EXIT
START:    .CDFN    #R0LIST,#CHANL,#40.
          BCS      BADCDF
          .PRINT   #MSG1
          .EXIT
BADCDF:   .PRINT   #MSG2
          .EXIT
MSG1:     .ASCIZ   /.CDFN O.K./
.EVEN
MSG2:     .ASCIZ   /BAD .CDFN/
          .EVEN
R0LIST:   BLKW     3                 ;EMT ARGUMENT LIST
CHANL:    .BLKW    40.*5             ;ROOM FOR CHANNELS
          .END     START
```

The example defines 40 (decimal) channels to start at location CHANL. An error occurs if 40 or more channels are already defined.

.CHAIN

9.4.2 .CHAIN

This request allows a program to pass control directly to another program without operator intervention. Since this process may be repeated, a large "chain" of programs can be strung together.

The area from locations 500–507 contains the device name and file name (in RAD50) to be chained to, and the area from locations 510–777 is used to pass information between the chained programs.

Macro Call: .CHAIN

NOTES

1. No assumptions should be made concerning which areas of memory will remain intact across a .CHAIN. In general, 500–777 is the only area guaranteed to be preserved across a .CHAIN.
2. I/O channels are left open across a .CHAIN for use by the new program. However, I/O channels opened via a .CDFN request are not available in this way. Since the monitor reverts to the original 16 channels during a .CHAIN, programs which leave files open across a .CHAIN should not use .CDFN. Furthermore, non-resident device handlers are released during a .CHAIN, and must be FETCHed again by the new program.
3. A program can determine whether it was CHAINed to or RUN from the keyboard by examining bit 8 of the JSW. This bit is on during program execution only if the program was entered via CHAIN. If a program normally loads into area 500–777, bit 8 of the JSW should be set during program assembly. This causes the monitor to load the area properly. If the bit is not set, locations 500–777 are preserved from the chaining program causing the new program to malfunction.

Errors:

.CHAIN is implemented by simulating the monitor RUN command (described in Chapter 2), and can produce any errors which RUN can produce. If an error occurs, the .CHAIN is abandoned and the Keyboard Monitor is entered.

When using .CHAIN, care should be taken for initial stack placement, since the program being "chained to" is started. The Linker normally defaults the initial stack to 1000 (octal); if caution is not observed, the stack may destroy chain data before it can be used (see Chapter 2, the RUN command).

Example:

```
        .MCALL  .REGDEF
        .REGDEF
        .MCALL  .CHAIN,.TTYIN
START:
        MOV     #500,R1         ;SET UP TO CHAIN
        MOV     #CHPTR,R2       ;DEVICE, FILE NAME TO 500–511
        MOV     (R2)+,(R1)+
        MOV     (R2)+,(R1)+
        MOV     (R2)+,(R1)+
        MOV     (R2)+,(R1)+
LOOP:   .TTYIN                  ;NOW GET A COMMAND LINE
        MOVB    R0,(R1)+        ;AND PASS IT TO THE JOB
        CMPB    R0,#12          ;IN LOCATIONS 512 AND UP
        BNE     LOOP            ;LOOP UNTIL LINE FEED
        CLRB    (R1)+           ;PUT IN A NULL BYTE
        .CHAIN
CHPTR:  .RAD50  /DK /
        .RAD50  /TECO /
        .RAD50  /SAV/
        .END    START
```

.CLOSE

9.4.3 .CLOSE

The .CLOSE request terminates activity on the specified channel and frees it for use in another operation. The handler for the associated device must be in memory.

Macro Call: .CLOSE .chan

Request Format:

R0 ⇒ | 6 | .chan |

A .CLOSE is required on any channel opened for either 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 causes the device directory to be updated to make that file permanent. A file opened via .LOOKUP does not require any directory operations. If the device associated with the specified channel already contains a file with the same name and extension, the old copy is deleted when the new file is made permanent. When an entered file is .CLOSEd, its permanent length reflects the highest block written since it was entered; for example, if the highest block written is block number 0, the file is given a length of 1; if the file was never written, it is given a length of 0. If this length is less than the size of the area which was allocated at .ENTER time, the unused blocks are reclaimed as an empty area on the device.

Errors:

.CLOSE does not return any errors. If the device handler for the operation is not in memory, a fatal monitor error is generated.

Example:

An example which illustrates the .CLOSE request follows the discussion of the .WRITW request in Section 9.4.34.

.CSIGEN

9.4.4 .CSIGEN

The .CSIGEN request calls the Command String Interpreter (CSI) in general mode to process a standard HT-11 command string (see Chapter 2 for the description of a standard command string). In general mode, all file .LOOKUPs and .ENTERs as well as handler .FETCHs are performed. When called in general mode, the CSI first closes channels 0–8 (decimal).

Macro Call: .CSIGEN .devspc, .defext, .cstring

where:

.devspc — is the address of the memory area where the device handlers (if any) are to be loaded.

.defext — is the address of a 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 ASCIZ input string or a #0 if input is to come from the terminal. If the string is in memory, it must not contain a ⟨CR⟩ ⟨LF⟩, but must terminate with a zero byte. If the .cstring field is left blank, input is automatically taken from the terminal.

.CSIGEN loads all necessary handlers and opens the files as specified. 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 may be destroyed. The system, however, is protected from this.

When the EMT is complete, register 0 points to the first available location above the handlers.

The four-word block pointed to by .defext is arranged as:

Word 1: default extension for all input channels

Word 2, 3, and 4: default extensions for output channels 0, 1, 2 respectively

If there is no default for a particular position, the associated word must contain a zero. All extensions are expressed in Radix 50. For example, the following block can be used to set up default extensions for a macro assembler:

```
DEFEXT: .RAD50  "MAC"
        .RAD50  "OBJ"
        .RAD50  "LST"
        .WORD   0
```

In the command string:

```
*DX0:ALPHA,DX1:BETA=DX1:INPUT
```

the default extension for input is MAC; for output, OBJ and LST. The following cases are legal:

```
*DX0:OUTPUT=
*DX1:INPUT
```

In the last example, the equal sign is not necessary in the event that only input files are specified.

When control returns to the user program after a call to .CSIGEN, all the specified files 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 8. A null specification causes the associated channel to remain inactive. For example, in the following string:

```
*,LP:=F1,F2
```

channel 0 is inactive since the first slot is 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 channels 5 through 8 are inactive. The user program can determine whether a channel is inactive by issuing a .WAIT request on the associated channel, which returns an error if the channel is not open.

Switches and their associated values are returned on the stack; see Section 9.4.5.1 for a description of the way switch information is passed.

Errors:

If CSI errors occur and input was from the terminal, an error message describing the fault is printed on the terminal and the CSI retries the command (these messages appear in Section 9.4.5.1). If the input was from a string, the carry bit is set and byte 52 contains the error code. The errors are:

Code	Explanation
0	Illegal command (bad separators, illegal filename, command too long, 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.

Example:

This example uses the general mode of the CSI in a program to copy an input file to an output file. Command input to the CSI is from the terminal.

```
        .MCALL   .REGDEF
        .REGDEF
        .MCALL   .CSIGEN,.READW,.PRINT,.EXIT,.WRITW,.CLOSE,.SRESET
ERRWD=52

START:  .CSIGEN  #DSPACE,#DEXT     ;GET STRING FROM TERMINAL
        MOV      R0,BUFF           ;R0 HAS FIRST FREE LOCATION
        CLR      INBLK             ;INPUT BLOCK #
        MOV      #LIST,R5          ;EMT ARGUMENT LIST
READ:   .READW   R5,#3,BUFF,#256.,INBLK ;READ CHANNEL 3
        BCC      2$                ;NO ERRORS
        TSTB     @#ERRWD           ;EOF ERROR?
        BEQ      EOF               ;YES
        MOV      #INERR,R0
```

```
1$:       .PRINT                      ;ERROR MESSAGE
          CLR      R0                 ;HARD EXIT
          .EXIT
2$:       .WRITW   R5,#0,BUFF,#256.,INBLK ;WRITE THE BLOCK
          BCC      NOERR              ;NO ERROR WRITING
          MOV      #WTERR,R0
          BR       1$                 ;HARD OUTPUT ERROR
NOERR:    INC      INBLK              ;GET NEXT BLOCK
          BR       READ               ;LOOP UNTIL DONE
EOF:      .CLOSE   #0                 ;CLOSE OUTPUT CHANNEL
          .CLOSE   #3                 ;AND INPUT CHANNEL
          .SRESET                     ;RELEASE HANDLER FROM MEMORY
          BR       START              ;GO FOR NEXT COMMAND LINE
DEXT:     .WORD    0,0,0,0            ;NO DEFAULT EXTENSIONS
BUFF:     .WORD    0                  ;I/O BUFFER START
INBLK:    .WORD    0                  ;RELATIVE BLOCK TO READ/WRITE
LIST:     .BLKW    5                  ;EMT ARGUMENT LIST
INERR:    .ASCIZ   /INPUT ERROR/
          .EVEN
WTERR:    .ASCIZ   /OUTPUT ERROR/
          .EVEN
DSPACE=.                              ;HANDLER SPACE
          .END     START
```

.CSISPC

9.4.5 .CSISPC

The .CSISPC request calls the Command String Interpreter in special mode to parse the command string and return file descriptors and switches to the program. In this mode, the CSI does not perform any handler fetches, .CLOSEs, .ENTERs, or .LOOKUPs.

Macro Call: .CSISPC .outspc, .defext, .cstring

where:

.outspc — is the address of the 39-word block to contain the file descriptors produced by .CSISPC. This area may overlay the space allocated to .cstring if desired.

.defext — is the address of a 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 ASCIZ input string or a #0 if input is to come from the terminal. If the string is in memory, it must not contain a ⟨CR⟩ ⟨LF⟩ but must terminate with a zero byte. If .cstring is blank, input is automatically taken from the terminal.

The 39-word file description consists of nine file descriptor blocks (five words for each of three possible output files; four words for each of six possible input files) which correspond to the nine possible files (three output, six input). If any of the nine possible filenames are not specified, the corresponding descriptor block is filled with zeroes.

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,

```
*DX1:LIST.MAC[15]=PR:
```

Using special mode, the CSI returns in the first five word slot:

16336	.RAD50 for DX1
46173	.RAD50 for LIS
76400	.RAD50 for T
50553	.RAD50 for MAC
00017	Octal value of size request

In the fourth slot (starting at an offset of 36 (octal) bytes into .outspc), the CSI returns:

63320	.RAD50 for PR
0	No file name
0	Specified
0	

Since this is an input file, only four words are returned.

Switches and their associated values are returned on the stack. See Section 9.4.5.1.

Errors:

Errors are the same as in general mode. However, since .LOOKUPs and .ENTERs are not done, the error codes which are valid are:

Code	Explanation
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 which is not in HT-11 format to a file under HT-11.

```
          .MCALL   .REGDEF
          .REGDEF
          .MCALL   .CSISPC,.PRINT,.EXIT,.ENTER,.CLOSE

START:    .CSISPC  #OUTSPC,#DEXT,#CSTRNG        ;GET INPUT FROM A
                                                ;STRING IN MEMORY
          BCC      2$
          MOV      #SYNERR,R0        ;SYNTAX ERROR
1$:       .PRINT                     ;ERROR MESSAGE
          .EXIT
2$:       .ENTER   #LIST,#0,#OUTSPC,#64.        ;ENTER FILE UNDER HT-11
          BCC      3$
          MOV      #ENMSG,R0                    ;ENTER FAILED
          BR       1$
3$:       JSR      R5,INPUT          ;ROUTINE INPUT WILL USE
                                     ;THE INFORMATION AT
                                     ;#OUTSPC+36 TO READ INPUT
                                     ;FROM THE NON-HT-11 DEVICE.
                                     ;INPUT IS PROCESSED AND
                                     ;WRITTEN VIA .WRITW REQUESTS
          .CLOSE   #0                ;MAKE OUTPUT FILE PERMANENT
          .EXIT                      ;AND EXIT PROGRAM
CSTRING:  .ASCIZ   "DX0:HTFIL.MAC=DX1:DOS.MAC"
          .EVEN
DEXT:     .WORD    0,0,0,0           ;NO DEFAULT EXTENSIONS
LIST:     .BLKW    5                 ;LIST FOR EMT CALLS
SYNERR:   .ASCIZ   "CSI ERROR"
ENMSG:    .ASCIZ   "ENTER FAILED"
          .EVEN
INPUT:    RTS      R5
OUTSPC=.                             ;CSI LIST GOES HERE
          .END     START
```

9.4.5.1 Passing Switch Information

In both general and special modes of the CSI, switches and their associated values are returned on the stack. A CSI switch is a slash (/) followed by any character. The CSI does not restrict the switch to printing characters, although it is suggested that printing characters be used wherever possible. The switch can be followed by an optional value, which is indicated by a : or ! separator. The : separator is followed by either an octal number or by one to three alphanumeric characters, the first of which must be alphabetic, which are converted to Radix-50. The ! separator is followed by a decimal value. Switches can be associated with files with the CSI. For example:

```
*DK:FOO/A,DX1:FILE.OBJ/A:100
```

In this case, there are two A switches. The first is associated with the input file DK:FOO. The second is associated with the input file DX1:FILE.OBJ, and has a value of 100(8). The stack output of the CSI is as follows:

Word #	Value	Meaning
1 (top of stack)	N	Number of switches found in command string. If N=0, no switches were found.
2	Switch value and file number	Even byte = 7-bit ASCII switch value. Bits 8–14 = Number (0–8) of the file with which the switch is associated. 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 negative, word 3 = switch value. If word 2 was not negative, this word is the next switch value (if it exists).

For example, if the input to the CSI is:

```
*FILE/B:20,FIL2/E=DX:INPUT/X:SY:20
```

on return, the stack is:

Stack Pointer ⇒	4	Four switches appeared.
	101530	Last switch=X; with file 3, has a value.
	20	Value of switch X=20
	101530	Next switch=X; with file 3, has a value.
	075250	Next value of switch X=RAD50 code for SY.
	505	Next switch=E; associated with file 1, no value.
	100102	Switch=B; associated with file 0 and has a value.
	20	Value is 20.

As an extended example, assume the following string was input for the CSI in general mode:

```
*FILE[8],LP:,SY:FILE2[20]=PR:,DX1:IN1/B,DX0:IN2/M:7
```

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 result of this CSI call would be:

1. A file named FILE.OP1 is entered on channel 0 on device DK; channel 1 is open for output to the device LP; a 20-block file named FILE2.OP3 is entered on the system device on channel 2.
2. Channel 3 is open for input from paper tape; channel 4 is open for input from a file IN1.MAC on device DX1; channel 5 is open for input from IN2.MAC on device DX0.

3. The stack contains switches and values as follows:

Stack	Explanation
2	2 switches found in string.
102515	Second switch is M, associated with Channel 5; has a numeric value.
7	Numeric value is 7.
2102	Switch is B, associated with Channel 4; has no numeric value.

If the CSI were called in special mode (Section 9.4.5), the stack would be the same as for the general mode call, and the descriptor table would contain:

```
.OUTSPEC:   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 (DECIMAL)
            63320    ;.RAD50      'PR'
                0
                0
                0
            16337    ;.RAD50      'DX1'
            35217    ;.RAD50      'IN1'
                0    ;.RAD50      '   '
            50553    ;.RAD50      'MAC'
            16336    ;.RAD50      'DX0'
            35220    ;.RAD50      'IN2'
                0    ;.RAD50      '   '
            50553    ;.RAD50      'MAC'
                0
                .
                .
                .
                0    (twelve more zero words are returned)
```

Keyboard error messages which may occur from incorrect use of the CSI when input is from the keyboard include:

Message	Meaning
?ILL CMD?	Syntax error.
?FIL NOT FND?	Input file was not found.
?DEV FUL?	Output file will not fit.
?ILL DEV?	Device specified does not exist.

NOTE

In many cases, the user program does not need to process switches in CSI calls. However, the user at the terminal may inadvertently enter switches. In this case, it is wise for the program to save the value of the stack pointer before the call to the CSI, and restore it after the call. In this way, no extraneous values will be left on the stack.

.DELETE

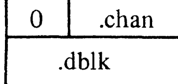
9.4.6 .DELETE

The .DELETE request deletes a named file from an indicated device.

Macro Call: .DELETE .area, .chan, .dblk

Request Format:

R0 ⇒ .area:



NOTE

The channel specified in the .DELETE request must not be in use when the request is made, or an error will occur. 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 nonfile-structured device is ignored. .DELETE requires that the handler to be used be in memory at the time the request is made. When the .DELETE is complete, the specified channel is left inactive.

Errors:

Code	Explanation
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.

```
        .MCALL    .REGDEF
        .REGDEF
        .MCALL    .SRESET,.CSISPC,.DELETE,.PRINT,.EXIT
START:  .SRESET                                  ;MAKE SURE CHANNELS
                                                 ;ARE FREE
        .CSISPC   #OUTSPC,#DEFEXT                ;GET COMMAND LINE
                                                 ;TERMINAL DIALOG WAS
                                                 ;DX:FILE
        .DELETE   #LIST,#0,#INSPC                ;USE CHANNEL 0 TO
                                                 ;DELETE THE FILE
                                                 ;WHICH IS AT THE
                                                 ;FIRST INPUT SLOT.
```

```
            BCC       START                 ;OK, LOOP AGAIN
            .PRINT    #NOFILE               ;NO SUCH FILE
            BR        START
NOFILE:     .ASCIZ    /FILE NOT FOUND/
            .EVEN
DEFEXT:     .RAD50    /MAC/                 ;.MAC INPUT EXTENSION
            .WORD     0,0,0                 ;NO OUTPUT DEFAULTS
LIST:       .BLKW     2                     ;EMT ARG LIST
OUTSPC=.
INSPC=.+36
            .BLKW     39.
            .END      START
```

INSPC is the address of the first input slot in the CSI input table.

.DSTATUS

9.4.7 .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 for the device specified by .devnam and, if found, returns four words of status starting at the address specified by .cblk. The four words returned are:

1. Status Word

   Bits 7-0: contain a number which identifies the device in question. The values (octal) currently defined are:

   0 = Reserved
   1 = Reserved
   2 = Reserved
   3 = Line Printer
   4 = Console Terminal
   5,6 = Reserved
   7 = PC11 High-speed Reader
   10 = PC11 High-speed Punch
   11 = Reserved
   12 = Reserved
   13 = Reserved
   14 = Reserved
   15 = Reserved
   16 = Reserved
   17 = Reserved
   20 = Reserved
   21 = Reserved
   22 = H27 Disk

Bit 15:	1 = Random-access device (disk)
	0 = Sequential-access device (line printer, paper tape, terminal)
Bit 14:	1 = Read-only device (paper tape reader)
Bit 13:	1 = Write-only device (line printer, paper tape punch)
Bit 12:	1 = Non HT-11 directory-structured device
Bit 11:	1 = Enter handler abort entry every time a job is aborted
	0 = Handler abort entry taken only if there is an active queue element belonging to aborted job
Bit 10:	1 = Handler accepts .SPFUN requests (e.g., DX)
	0 = .SPFUN requests are rejected as illegal

2. Handler size

   The size of the device handler, in bytes.

3. Entry point

   Non-zero implies the handler is now in memory; zero implies it must be .FETCHed before it can be used.

4. Device size

   The size of the device (in 256-word blocks) for block-replaceable devices; zero for sequential-access devices.

The device name may be a user-assigned name.

Errors:

Code	**Explanation**
0	Device not found in tables.

Example:

This example shows how to determine if a particular device handler is in memory and, if it is not, how to .FETCH it there.

```
        .MCALL    .REGDEF
        .REGDEF
        .MCALL    .DSTATUS,.PRINT,.EXIT,.FETCH
START:  .DSTATUS  #CORE,#FPTR                    ;GET STATUS OF DEVICE
        BCC       1$
        .PRINT    #ILLDEV                        ;DEVICE NOT IN TABLES
        .EXIT
1$:     TST       CORE+4                         ;IS DEVICE RESIDENT?
        BNE       2$
        .FETCH    #HNDLR,#FPTR                   ;NO, GET IT
        BCC       2$
        .PRINT    #FEFAIL                        ;FETCH FAILED
        .EXIT
2$:     .PRINT    #FECHOK
        .EXIT
CORE:   .BLKW     4          ;DSATUS GOES HERE
FPTR:   .RAD50    /DX0/      ;DEVICE NAME
        .RAD50    /FILE MAC/ ;FILE NAME
```

```
FEFAIL:     .ASCIZ      /FETCH FAILED/
ILLDEV:     .ASCIZ      /ILLEGAL DEVICE/
            .EVEN
FECHOK:     .ASCIZ      /FETCH O.K./
            .EVEN
HNDLR=.                           ;HANDLER WILL GO HERE
            .END        START
```

.ENTER

9.4.8 .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. (Note that if the program is overlaid, channel 15 is used by the overlay handler and should not be modified.)

Macro Call: .ENTER .area, .chan, .dblk, .length

where: .length is the file size specification. The file length allocation is as follows:

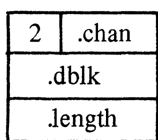
0 – either 1/2 the largest empty entry or the entire second largest empty entry, whichever is largest. (A maximum size for non-specific .ENTERs may be patched in the monitor.)

M – a file of M blocks. M may exceed the maximum mentioned above.

−1 – the largest empty entry on the device.

Request Format:

R0 ⇒ .area:



2	.chan
.dblk	
.length	

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 and the channel may not be used by .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 on the same device 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.

Each job may have up to 256 files open on the system at any time. If required, all 256 may be opened for output with the .ENTER function. .ENTER requires that the device handler be in memory when the request is made. Thus, a .FETCH should normally be executed before a .ENTER can be done. On return, R0 contains the size of the area actually allocated for use.

NOTE

When using the 0 length feature of .ENTER, it must be kept in mind that less than the largest empty space is allocated. This can have an important effect in transferring files between devices which have a relatively small capacity. For example, to transfer a 200-block file to a device on which the largest available empty space is 300 blocks, a 0 length transfer will not work. Since the .ENTER allocates half the largest space, only 150 blocks are really allocated and an output error will occur during the transfer. If a specific length of 200 is requested, however, the transfer will proceed without error.

Errors:

Code	Explanation
0	Channel is in use.
1	In a fixed length request, no space greater than or equal to M was found, or in a non-specific request the device or the directory was found to be full.

Example:

.ENTER may be used to open a file on a specified device, and then write data from memory into that file as follows:

```
        .MCALL  .REGDEF,.ENTER,.WRITW,.CLOSE,.PRINT
        .MCALL  .SRESET,.EXIT,.FETCH
        .REGDEF
START:  .SRESET                         ;MAKE SURE ALL CHANNELS
                                        ;ARE CLOSED.
        .FETCH  #CORSPC,#FPRT           ;FETCH DEVICE HANDLER
        BCS     BADFET                  ;.FETCH ERROR, PROBABLY
                                        ;ILLEGAL DEVICE.
        .ENTER  #AREA,#0,#FPRT          ;OPEN A FILE ON THE DEVICE
                                        ;SPECIFIED. LENGTH 0 WILL
                                        ;GIVE 1/2 OF LARGEST EMPTY
                                        ;SPACE NOW AVAILABLE.
        BCS     BADENT                  ;FAILED. CHANNEL PROBABLY BUSY
        .WRITW  #AREA,#0,#BUFF,#END-BUFF/2,#0
                                        ;WRITE DATA FROM MEMORY, THE
                                        ;SIZE IS # OF WORDS BETWEEN
                                        ;BUFF AND END. START AT BLOCK 0.
        BCS     BADWRT                  ;WRITE FAILURE.
        .CLOSE  #0                      ;CLOSE THE FILE
        .EXIT                           ;AND GO TO KEYBOARD MONITOR.
FPRT:   .RAD50  /DK /                   ;FILE WILL BE ON DK
        .RAD50  /FILE EXT/              ;NAMED FILE, EXT
AREA:   .BLKW   10                      ;EMT ARGUMENT LIST
BADFET: .PRINT  #FMSG
        .EXIT
BADENT: .PRINT  #EMSG
        .EXIT
BADWRT: .PRINT  #WMSG
        .EXIT
FMSG:   .ASCIZ  /BAD FETCH/
EMSG:   .ASCIZ  /BAD ENTER/
WMSG:   .ASCIZ  /WRITE ERROR/
        .EVEN
CORSPC: .BLKW   400                     ;LEAVE 400(8) WORDS
                                        ;FOR DEVICE HANDLER.
BUFF:
        .REPT   400                     ;THIS IS BUFFER TO BE WRITTEN OUT
        .WORD   0,1
        .ENDR
END:
        .END    START
```

.EXIT

9.4.9 .EXIT

The .EXIT request causes the user program to terminate. Any I/O requests and completion routines pending for that job are allowed to complete. If part of the job resides where KMON and USR are to be read, the user job is written onto system device scratch blocks. KMON and USR are then loaded and control goes to KMON. If R0=0 when the .EXIT is done, an implicit INIT command is executed when KMON is entered, disabling the subsequent use of REENTER, START, or CLOSE.

.EXIT also resets any .CDFN and .QSET calls that were done and executes an .UNLOCK if a .LOCK has been done. Thus, the .CLOSE command from the Keyboard Monitor does not operate for programs which perform .CDFN requests.

Macro Call: .EXIT

Errors:

None.

.FETCH

9.4.10 .FETCH

The .FETCH request loads device handlers into memory from the system device.

Macro Call: .FETCH .coradd, .devnam

where:	.coradd	is the address where the device handler is to be loaded.
	.devnam	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 memory, 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. (.RELEAS does not dismiss a handler which was LOADED from the KMON; an UNLOAD must be done.) After a .RELEAS, a .FETCH must be issued in order to use the device again.

Several requests require a device handler to be in memory for successful operation. These include:

.CLOSE	.READC	.READ
.LOOKUP	.WRITC	.WRITE
.ENTER	.READW	.SPFUN
.RENAME	.WRITW	.DELETE

Errors:

Code	**Explanation**
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 memory in preparation for their use by a program. The program sets aside handler space from its free memory area.

```
        .MCALL    .REGDEF,.FETCH,.PRINT,.EXIT
        .REGDEF
START:
        .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 MEMORY
                                        ;POINTER TO POINT TO
                                        ;NEW BOTTOM OF FREE
                                        ;AREA (TOP OF HANDLERS).

        .PRINT    #OK
        .EXIT
OK:     .ASCIZ    /FETCH O.K./
        .EVEN
FERR:   .PRINT    #MSG                  ;PRINT ERROR MESSAGE
        .EXIT                           ;AND EXIT
        HALT
PRNAME: .RAD50    "PR "                 ;DEVICE NAMES
PPNAME: .RAD50    "PP "
MSG:    .ASCIZ    "DEVICE NOT FOUND"    ;ERROR MESSAGE
        .EVEN
FREE:   .+2                             ;POINTER TO FREE MEMORY
        .END      START
```

.GTIM

9.4.11 .GTIM

.GTIM allows user programs to access the current time of day. The time is returned in two words, and is given in terms of clock ticks past midnight.

Macro Call: .GTIM .area, .addr

where: .addr is a pointer to the two words of time to be returned.

Request Format:

R0 ⇒ .area:

21	0
.addr	

The high-order time is returned in the first word, the low-order time in the second word. User programs must make the conversion from clock ticks to hours-minutes-seconds. The basic clock frequency (50 or 60 Hz) may be determined from the configuration word in the monitor (see Section 9.2.6). The system time and date must be manually updated by the user after midnight to maintain the correct time (see Chapter 2).

The clock rate is initially set to 60-cycle.

Errors:

None.

Example:

```
        .MCALL   .REGDEF,.GTIM,.EXIT
        .REGDEF
START:
        .GTIM    #LIST,#TIME

        .EXIT
TIME:   .WORD    0,0                    ;LOW AND HI ORDER TIME
                                        ;RETURNED HERE.

LIST:   .BLKW    2                      ;ARGUMENTS FOR THE EMT

        .END     START
```

9.4.12 .GTJB

The .GTJB request passes certain job parameters back to the user program.

Macro Call: .GTJB .area, .addr

where: .addr is the address of an eight-word block into which the parameters are passed. The values returned are:

Word 1 – Job Number.
2 – High memory limit
3 – Low memory limit
4 – Beginning of I/O channel space
5-8 – Reserved for future use

Request Format:

R0 ⇒ .area:

20	0
.addr	

The job number is always 0 and the low limit 0.

Word 4 describes where the I/O channel words begin. This is normally an address within the Resident Monitor. When a .CDFN is executed, however, the start of the I/O channel area changes to the user-specified area.

Errors:

None.

.HERR/.SERR

9.4.13 .HERR/.SERR

.HERR and .SERR are complementary requests used to govern monitor behavior for serious error conditions. During program execution, certain error conditions may arise which cause the executing program to be aborted (for example, trying to pass I/O to a device which has no handler in memory, or trying to load a device handler over the USR). Normally, these errors cause program termination with one of the ?M– error messages. However, in certain cases it is not feasible to abort the program because of these errors; for example, a multi-user program must be able to retain control and merely abort the user who has generated the error. .SERR accomplishes this by inhibiting the monitor from aborting the job. Instead, it causes an error return to the offending EMT to be taken. On return from that request, the C bit is set and byte 52 contains a negative value indicating the error condition which occurred.

.HERR turns off user error interception and allows the system to abort the job on fatal errors and generate an error message. (.HERR is the default case.)

Macro Calls: .HERR

.SERR

Errors:

Following is a list of the errors which are returned if soft error recovery is in effect:

Code	Explanation
–1	Called USR from completion routine.
–2	No device handler; this operation needs one.
–3	Error doing directory I/O.
–4	FETCH error. Either an I/O error occurred while reading the handler, or tried to load it over USR or RMON.
–5	Error reading an overlay.
–6	No more room for files in the directory.
–7	Tried to perform a monitor operation outside the job partition.
–10	Illegal channel number; number is greater than actual number of channels which exist.
–11	Illegal EMT; an illegal function code has been decoded.

Traps to 4 and 10, and floating point exception traps are not inhibited. These errors have their own recovery mechanism. (See Section 9.4.30.)

Example:

This example causes a normally fatal error to generate errors back to the user program. The error returned is used to print an appropriate message.

```
          .MCALL    .REGDEF,.FETCH,.ENTER,.HERR,.SERR
          .MCALL    .EXIT,.PRINT
          .REGDEF
ST:       .SERR                             ;TURN ON SOFTWARE ERROR
                                            ;RETURNS
          .FETCH    #HDLR,#PTR              ;GET A DEVICE HANDLER
          BCS       FCHERR
          .ENTER    #AREA,#1,#PTR           ;OPEN A FILE ON CHANNEL 1
          BCS       ENERR
          .HERR                             ;NOW PERMIT ?M-ERRORS.
          .EXIT
FCHERR:   MOVB      @#52,R0                 ;WAS IT FATAL
          BMI       FTLERR                  ;YES
          .PRINT    #FMSG                   ;NO . . . NO DEVICE BY THAT NAME
          .EXIT
ENERR:    MOVB      @#52,R0
          BMI       FTLERR
          .PRINT    #EMSG
          .EXIT
FTLERR:   NEG       R0                      ;THIS WILL TURN POSITIVE
          DEC       R0                      ;ADJUST BY ONE
          ASL       R0                      ;MAKE IT AN INDEX
          MOV       TBL(R0),R0              ;PUT MESSAGE ADDRESS INTO R0
          .PRINT                            ;AND PRINT IT.
          .EXIT
TBL:      M1                                ;CAN'T OCCUR IN THIS PROGRAM
          M2                                ;NO DEVICE HANDLER IN MEMORY
          M3                                ;DIRECTORY I/O ERROR
          M4                                ;FETCH ERROR
          M5                                ;IMPOSSIBLE FOR THIS PROGRAM
          M6                                ;NO ROOM IN DIRECTORY
          M7                                ;ILLEGAL ADDRESS
          M10                               ;ILLEGAL CHANNEL
          M11                               ;ILLEGAL EMT
M1:                                         ;CAN'T OCCUR IN THIS PROGRAM
M2:       .ASCIZ    /NO DEVICE HANDLER/
M3:       .ASCIZ    "DIRECTORY I/O ERROR"
M4:       .ASCIZ    /ERROR DOING FETCH/
M5:                                         ;NOT APPLICABLE TO THIS PROGRAM
M6:       .ASCIZ    /NO ROOM IN DIRECTORY/
M7:       .ASCIZ    /ADDRESS CHECK ERROR/
M10:      .ASCIZ    /ILLEGAL CHANNEL/
M11:      .ASCIZ    /ILLEGAL EMT/
FMSG:     .ASCIZ    /FETCH FAILED/
EMSG:     .ASCIZ    /ENTER FAILED/
          .EVEN
HDLR:     .BLKW     300                     ;LEAVE 300 (OCTAL) FOR HANDLER
PTR:      .RAD50    /DX1/                   ;DEVICE AND FILE NAME.
          .RAD50    /EXAMPL/
          .RAD50    /MAC/
AREA:     .BLKW     4                       ;EMT AREA
          .END      ST
```

.HRESET

9.4.14 .HRESET

This request performs the same function as .SRESET, after stopping all I/O transfers in progress for that job. (.HRESET is not used to clear a hard-error condition.) Note that a hardware RESET instruction is used to terminate I/O.

Macro Call: .HRESET

Errors:

None.

Example:

See the example for .SRESET (Section 9.4.29) for format.

.LOCK/.UNLOCK

9.4.15 .LOCK/.UNLOCK

.LOCK

The .LOCK request is used to "lock" the USR in memory for a series of operations. If all the conditions which cause swapping are satisfied, the user program is written into scratch blocks and the USR is loaded. Otherwise, the USR which is in memory is used, and no swapping occurs. The USR is not released until an .UNLOCK request is given. A program which has many USR requests to make can .LOCK the USR in memory, make all the requests, and then .UNLOCK the USR; no time is spent doing unnecessary swapping.

Macro Call: .LOCK

Note that the .LOCK request reduces time spent in file handling by eliminating the swapping of the USR in and out of memory. If the USR is currently resident, .LOCK is ignored. After a .LOCK has been executed, an .UNLOCK request must be executed to release the USR from memory. The .LOCK/.UNLOCK requests are complimentary and must be matched. That is, if three .LOCK requests are issued, at least three .UNLOCKs must be done, otherwise the USR will not be released. More .UNLOCKs than .LOCKs may occur without error.

NOTES

1. It is vital that the .LOCK call not come from within the area into which the USR will be swapped. If this should occur, the return from the USR request would not be to the user program, but to the USR itself, since the LOCK function inhibits the user program from being re-read.
2. Once a .LOCK has been performed, it is not advisable for the program to destroy the area the USR is in, even though no further use of the USR is required. This causes unpredictable results when an .UNLOCK is done.

Errors:

None.

Example:

See the example following .UNLOCK.

.UNLOCK

The .UNLOCK request releases the User Service Routine from memory if it was placed there with a .LOCK request. If the .LOCK required a swap, the .UNLOCK loads the user program back into memory. If the USR does not require swapping, the .UNLOCK acts as a no-op.

Macro Call: .UNLOCK

NOTE

It is important that at least as many .UNLOCKS are given as .LOCKs. If more .LOCK requests were done, the USR remains locked in memory. It is not harmful to give more UNLOCKs than are required; those that are extra are ignored.

Errors:

None.

Example:

This example shows the usage of .LOCK, .UNLOCK, and their interaction with the system.

```
          .MCALL    .REGDEF,.LOCK,.UNLOCK,.LOOKUP
          .MCALL    .SETTOP,.PRINT,.EXIT
          .REGDEF
START:
SYSPTR=54
          .SETTOP   @#SYSPTR              ;TRY FOR ALL OF MEMORY
          MOV       R0,TOP                ;R0 HAS THE TOP
          .LOCK                           ;BRING USR INTO MEMORY
          .LOOKUP   #LIST,#0,#FILE1       ;LOOKUP A FILE ON CHANNEL 0
          BCC       1$                    ;ON ERROR, PRINT A
2$:       .PRINT    #LMSG                 ;MESSAGE AND EXIT
          .EXIT
1$:       MOV       #LIST,R0
          INC       (R0)                  ;DO LOOKUP ON CHANNEL 1
          MOV       #FILE2,2(R0)          ;NEW POINTER
          .LOOKUP                    ;ALL ARGS ARE FILLED IN
          BCS       2$
          .UNLOCK                         ;NOW RELEASE USR
          .EXIT
LIST:     .BLKW     3                     ;SPACE FOR ARGUMENTS
FILE1:    .RAD50    /DK /
          .RAD50    /FILE1 MAC/
FILE2:    .RAD50    /DK /
          .RAD50    /FILE2 MAC/
TOP:      .WORD     0
LMSG:     .ASCIZ    /LOOKUP ERROR/
          .EVEN
          .END      START
```

In the above example, .SETTOP tries to obtain as much memory as it can. Most likely this will make the USR non-resident (i.e., unless a SET USR NOSWAP command is done at the keyboard). Thus, if the USR were non-resident, swapping must take place for each .LOOKUP given. Using the .LOCK, the USR is brought into memory and remains there until the .UNLOCK is given.

The second .LOOKUP makes use of the fact that the arguments have already been set up at LIST. Thus, it is possible to increment the channel number, put in a new file pointer and then give a simple .LOOKUP, which does not cause any arguments to be moved into LIST.

9.4.16 .LOOKUP

The .LOOKUP request associates a specified channel with a device and/or file, for the purpose of performing I/O operations. The channel used is then "busy" until one of the following requests is executed:

.CLOSE
.SAVESTATUS
.SRESET
.HRESET
.PURGE
.CSIGEN (if channel is in range 0–8)

Note that if the program is overlaid, channel 15 is used by the overlay handler and should not be modified.

Macro Call: .LOOKUP .area, .chan, .dblk

Request Format:

R0 ⇒ .area:

1	.chan
.dblk	

If the first word of the file name in .dblk is zero and the device is a file-structured device, absolute block 0 of the device is designated as the beginning of the "file". This technique allows I/O to any physical block on the device. If a file name is specified for a device which is not file-structured (i.e., PR:FILE.EXT), the name is ignored.

The handler for the selected device must be in memory for a .LOOKUP. On return from the .LOOKUP, R0 contains the length (number of blocks) of the file just looked up. If the length returned is 0, a nonfile-structured .LOOKUP was done to the device.

Errors:

Code	**Explanation**
0	Channel already open.
1	File indicated was not found on the device.

Example:

In the following example, the file "DATA.001" on device DX1 is opened for input on channel 7.

```
         .MCALL      .REGDEF,.FETCH,.LOOKUP,.PRINT,.EXIT
         .REGDEF
START:
ERRWD=52
         .FETCH      #HSPACE,#DX1N         ;GET DEVICE HANDLER
         BCS         FERR                  ;DX1 IS NOT AVAILABLE
         .LOOKUP     #LIST,#7,#DX1N        ;LOOKUP THE FILE
                                           ;ON CHANNEL 7
         BCC         LDONE                 ;FILE WAS FOUND
         TSTB        @#ERRWD               ;ERROR, WHAT'S WRONG?
         BNE         NFD                   ;FILE NOT FOUND
         .PRINT      #CAMSG                ;PRINT 'CHANNEL ACTIVE'
         .EXIT
NFD:     .PRINT      #NFMSG                ;FILE NOT FOUND
         .EXIT
CAMSG:   .ASCIZ      /CHANNEL ACTIVE/
NFMSG:   .ASCIZ      /FILE NOT FOUND/      ;ERROR MESSAGES
DXMSG:   .ASCIZ      /DX1 NOT AVAILABLE/
         .EVEN
FERR:    .PRINT      #DXMSG
         .EXIT
LDONE:                                     ;PROGRAM CAN NOW
                                           ;ISSUE READS AND
                                           ;WRITES TO FILE
                                           ;DATA.001 VIA
                                           ;CHANNEL 7
         .EXIT
LIST:    .BLKW       5
DX1N:    .RAD50      "DX1"                 ;DEVICE
         .RAD50      "DAT"                 ;FILENAME
         .RAD50      "A  "                 ;FILENAME
         .RAD50      "001"                 ;EXTENSION
HSPACE:                              ;RESERVED SPACE FOR DX
         .=.+400                           ;HANDLER
         .END        START
```

.PRINT

9.4.17 .PRINT

The .PRINT request causes output to be printed at the terminal. The string to be printed may be terminated with either a null (0) byte or a 200 byte. If the null (ASCIZ) format is used, the output is automatically followed by a ⟨CR⟩ ⟨LF⟩. If a 200 byte terminates the string, no ⟨CR⟩ ⟨LF⟩ is generated.

Macro Call: .PRINT .addr

where: .addr is the address of the string to be printed.

Control returns to the user program after all characters have been placed in the output buffer.

Errors:

None.

Example:

```
                .MCALL      .REGDEF,.PRINT,.EXIT
                .REGDEF
START:
                .PRINT      #S2
                .PRINT      #S1

                .EXIT

S1:             .ASCIZ      /THIS WILL HAVE CR-LF FOLLOWING/
S2:             .ASCII      /THIS WILL NOT HAVE CR-LF/
                .BYTE       200
                .EVEN

                .END        START
```

.PROTECT

9.4.18 .PROTECT

The .PROTECT request is used by a job to obtain exclusive control of a vector (two words) in the region 0–476. If it is successful, it indicates that the locations are not currently in use by another job or by the monitor, in which case the job may place an interrupt address and priority into the protected locations and begin using the associated device.

Macro Call: .PROTECT .area, .addr

where: .addr is the address of the word pair to be protected. .addr must be a multiple of four, and must be less than 476 (octal). The two words at .addr and .addr+2 will be protected.

Request Format:

R0 ⇒ .area:

31	0
.addr	

Errors:

Code	Explanation
0	Protect failure; locations already in use.
1	Address greater than 476 or not a multiple of 4.

Example:

This example shows the use of .PROTECT to gain control of the vectors at location 234.

```
        .MCALL      .REGDEF,.PROTECT,.PRINT,.EXIT
        .REGDEF
ST:     MOV         #AREA,-(SP)
        MOV         #234,R5             ;VECTOR ADDRESS
        .PROTECT    (SP),R5             ;PROTECT 234,236
        BCS         ERR                 ;YOU CAN'T
        MOV         #UDCINT,(R5)+       ;INITIALIZE THE VECTORS.
        MOV         #340,(R5)           ;AT LEVEL 7

        .EXIT
ERR:    .PRINT      #NOVEC
        .EXIT
AREA:   .BLKW       5
NOVEC:  .ASCIZ      /VECTORS ALREADY IN USE/
        .EVEN

UDCINT:
        .
        .
        .
```

.PURGE

9.4.19 .PURGE

The .PURGE request is used to de-activate a channel without performing a .HRESET, .SRESET, .SAVESTATUS, or .CLOSE request. It merely frees a channel without taking any other action. If a tentative file has been .ENTERed on the channel, it will be discarded. Purging an inactive channel acts as a no-op.

Macro Call: .PURGE .chan

Errors:

None.

Example:

The following code is used to make certain that channels 0–7 are free:

```
        .MCALL      .REGDEF,.PURGE,.EXIT
        .REGDEF

START:
        CLR         R1                  ;START WITH CHANNEL 0
1$:     .PURGE      R1                  ;PURGE A CHANNEL
        INC         R1                  ;BUMP TO NEXT CHANNEL
        CMP         R1,#8.              ;IS IT AT CHANNEL 8 YET?
        BLO         1$                  ;NO, KEEP GOING

        .EXIT
        .END        START
```

.QSET

9.4.20 .QSET

All HT-11 I/O transfers are done through a centralized queue management system. If I/O traffic is very heavy and not enough queue elements are available, the program issuing the I/O requests may be suspended until a queue element becomes available.

The .QSET request is used to make the HT-11 I/O queue larger (i.e., add available entries to the queue). A general rule to follow is that each program should contain one more queue element than the total number of I/O requests which will be active simultaneously. Note that if synchronous I/O is done (i.e., .READW/.WRITW, etc.) and no timing requests are done, no additional queue elements need be allocated.

Macro Call: .QSET .addr, .qleng

where:

.addr — is the address at which the new elements are to start.

.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 .qleng * 7 words.

Each time .QSET is called, a continuous area of memory is divided into seven-word segments and is added to the queue for that job. .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. The space used for the new elements is allocated from the user's program space. Thus, care must be taken so that the program in no way alters the elements once they are set up. The .SRESET and .HRESET requests discard all user-defined queue elements; therefore any .QSETs must be reissued.

Care should also be taken to allocate enough memory for the queue. The elements in the queue are altered by the monitor; if enough space is not allocated, destructive references will occur in an unexpccted area of memory.

Errors:

None.

Example:

```
        .MCALL      .REGDEF,.QSET,.EXIT
        .REGDEF

START:
        .QSET       #Q1,#5          ;ADD 5 ELEMENTS TO THE QUEUE
                                    ;STARTING AT Q1
        .QSET       #Q3,#3          ;AND 3 MORE AT Q3.

        .EXIT

Q1:     .BLKW       7*5.            ;FIRST QUEUE AREA (35 DECIMAL WORDS)
Q3:     .BLKW       7*3.            ;SECOND QUEUE AREA (21 DECIMAL WORDS)

        .END        START
```

Note that Q1 and Q3 need not have been contiguous.

.RCTRLO

9.4.21 .RCTRLO

The .RCTRLO request restores terminal output after its inhibition by a CTRL O keyboard command. Typing a second CRTL O or returning to the monitor are methods that also restore terminal output.

Macro Call: .RCTRLO

Errors:

None.

Example:

In this example, the user program first 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.

```
        .MCALL   .REGDEF,.RCTRLO,.CSIGEN,.EXIT
        .REGDEF

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                                ;NO DEFAULT EXTENSIONS
        0
        0
        0
DSPACE: .=.+400                          ;HANDLER SPACE

        .END     START
```

.READ/.READC/.READW

9.4.22 .READ/.READC/.READW

HT-11 provides three modes of I/O: .READ/.WRITE, .READC/.WRITC, and .READW/.WRITW. Section 9.4.34 explains the output operations. The input operations are described next.

Note that in the case of .READ and .READC, additional queue elements should be allocated for buffered I/O operations (see .QSET).

.READ

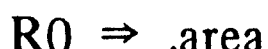
The .READ request transfers a specified number of words from the specified channel to memory. Control returns to the user program immediately after the .READ is initiated. No special action is taken when the transfer is completed.

Macro Call: .READ .area, .chan, .buff, .wcnt, .blk

where:

.buff	is the address of the buffer to receive the data read.
.wcnt	is the number of words to be read.
.blk	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 .blk before it is used again. If .blk=0, TT: gives ^ prompt and LP: gives form feed. (This is true for all .READ and .WRITE requests.)

Request Format:



R0 ⇒ .area:

10	.chan
.blk	
.buff	
.wcnt	
1	

When the user program needs to access the data read on the specified channel, a .WAIT request should be issued. This ensures that the data has been read completely. If an error occurred during the transfer, the .WAIT request indicates the error.

Errors:

Code	Explanation
0	Attempt to read past end-of-file
1	Hard error occurred on channel
2	Channel is not open

Example:

Refer to the .WRITE/.WRITC/.WRITW examples.

.READC

The .READC request transfers a specified number of words from the indicated channel to memory. Control returns to the user program immediately after the .READC is initiated. Execution of the user program continues until the .READC 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 .area, .chan, .buff, .wcnt, .crtn, .blk

where:

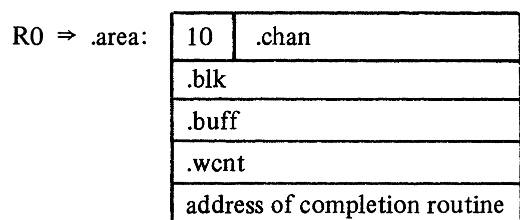
.buff	is the address of the buffer to receive the data read.
.wcnt	is the number of words to be read.

.crtn — is the address of the user's completion routine (refer to Section 9.2.8).

.blk — 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 .blk before it is used again.

Request Format:

R0 ⇒ .area:



10	.chan
.blk	
.buff	
.wcnt	
address of completion routine	

When entering a .READC completion function the following are true:

1. R0 contains the channel status word for the operation. If bit 0 of R0 is set, a hardware error occurred during the transfer. The data may not be reliable.
2. R1 contains the channel number of the operation. This is useful when the same completion function is to be used for several different transfers.

Errors:

Code	**Explanation**
0	Attempt to read past end-of-file
1	Hard error occurred on channel
2	Channel is not open

Example:

Refer to the .WRITE/.WRITC/.WRITW examples.

.READW

The .READW request transfers a specified number of words from the indicated channel to memory. Control returns to the user program when the .READW is complete or if an error is detected.

Macro Call: .READW .area, .chan, .buff, .wcnt, .blk

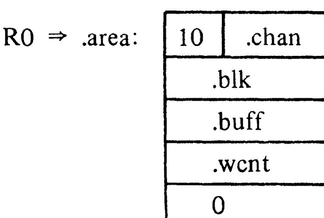
where:

.buff — is the address of the buffer to receive the data read.

.wcnt — is the number of words to be read. The number must be positive.

.blk — 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 .blk before it is used again.

Request Format:



On return from this call, the C bit set indicates an error has occurred. If no error occurred, the data is in memory at the specified address.

NOTE

Upon return from any READ programmed request, R0 will contain no information if the read is from a sequential-access device. If the read is from a random-access device, R0 will contain the actual number of words that will be read (.READ or .READC) or have been read (.READW). This will be less than the requested word count if an attempt is made to read past the end-of-file, but a partial transfer is possible. Therefore, a program should always use the returned word count as the number of words available. For example, suppose a file is 5 blocks long (i.e., it has block numbers 0 to 4) and a request is issued to read 512 words, starting at block 4. The request is shortened to 256 words; no error is indicated. Also note that since the request will be shortened to an exact number of blocks, a request for 256 words will either succeed or fail, but cannot be shortened.

Errors:

Code	Explanation
0	Attempt to read past end-of-file
1	Hard error occurred on channel
2	Channel is not open

Example:

Refer to the .WRITE/.WRITC/.WRITW examples.

.RELEAS

9.4.23 .RELEAS

The .RELEAS request removes the handler for the specified device from memory. The .RELEAS is ignored if the handler is:

1. Part of RMON (i.e., the system device),
2. Not currently resident, or
3. Resident because of a LOAD command to the Keyboard Monitor.

Macro Call: .RELEAS .devname

where: .devname is the pointer to the .RAD50 device name.

Errors:

Code	Explanation
0	Handler name was illegal.

Example:

In the following example, the lineprinter handler (LP) is loaded into memory, used, then released.

```
        .MCALL   .REGDEF,.FETCH,.RELEAS,.EXIT
        .REGDEF

START:  .FETCH   #HSPACE,#LPNAME     ;LOAD LP HANDLER
        BCS      FERR                ;NOT AVAILABLE

; USE HANDLER

        .RELEAS  #LPNAME             ;MARK LP NO LONGER IN
                                     ;MEMORY.
        BR       START
FERR:   HALT                         ;LP NOT AVAILABLE
LPNAME: .RAD50   /LP /               ;NAME FOR LP HANDLER
HSPACE:                              ;BEGINNING OF HANDLER
                                     ;AREA

        .END     START
```

.RENAME

9.4.24 .RENAME

The .RENAME request causes an immediate change of name of the file specified. An error occurs if the channel specified is already open.

Macro Call: .RENAME .area, .chan, .dblk

Request Format:

R0 ⇒ .area:

4	.chan
.dblk	

The .dblk argument consists of two consecutive .RAD50 device and file specifications. For example:

```
          .RENAME    #AREA,#7,#DBLK       ;USE CHANNEL 7
          BCS        RNMERR               ;NOT FOUND
          .
          .
          .
DBLK:     .RAD50     /DX1/
          .RAD50     /OLDFIL/
          .RAD50     /MAC/
          .RAD50     /DX1/
          .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. If a file with the same name as the new file name specified already exists on the indicated device, it is deleted. The second occurrence of the device name DX1 is necessary for proper operation, and should not be omitted. The specified channel is left inactive when the .RENAME is complete. .RENAME requires that the handler to be used be resident at the time the .RENAME request is made. If it is not, a monitor error occurs. Note that .RENAME is legal only on files which are on disk. (.RENAMEs to other devices are ignored.)

Errors:

Code	**Explanation**
0	Channel open
1	File not found

Example:

In the following example, the file DATA.TMP on DX0 is renamed to DATA.001:

```
          .MCALL     .REGDEF,.FETCH,.PRINT
          .MCALL     .EXIT,.RENAME
          .REGDEF

START:    .FETCH     #HSPACE,#NAMBLK      ;GET HANDLER
          BCS        FERR                 ;SOME ERROR
          .RENAME    #AREA,#0,#NAMBLK     ;DO THE RENAME
          BCS        RNMERR               ;ERROR
          .EXIT
FERR:     .PRINT     #FMSG
          .EXIT
RNMERR:   .PRINT     #RNMSG
          .EXIT
AREA:     .BLKW      5                    ;ROOM FOR ARGS.
NAMBLK:   .RAD50     /DX0DATA  TMP/       ;OLD NAME
          .RAD50     /DX0DATA  001/       ;NEW NAME
FMSG:     .ASCIZ     /FETCH?/             ;ERROR MESSAGES
RNMSG:    .ASCIZ     /RENAME?/
          .EVEN
HSPACE=.

          .END       START
```

.REOPEN

9.4.25 .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. The .REOPEN need not be done on the same channel as the original .LOOKUP and .SAVESTATUS.

Macro Call: .REOPEN .area, .chan, .cblk

where: .cblk is the address of the five-word block where the channel status information was stored.

Request Format:

R0 ⇒ .area:

6	.chan
.cblk	

Errors:

Code	Explanation
0	The specified channel is in use. The .REOPEN has not been done.

Example:

Refer to the example following the description of .SAVESTATUS.

.SAVESTATUS

9.4.26 .SAVESTATUS

The .SAVESTATUS request stores five words of channel status information into a user-specified area of memory. These words contain all the information HT-11 requires to completely define a file. When a .SAVESTATUS is done, the data words are placed in memory, 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 .area, .chan, .cblk

where: .cblk is the address of the user memory block (5 words) where the channel status information is to be stored.

Request Format:

R0 ⇒ .area:

5	.chan
.cblk	

The five words stored are the five words normally contained in the channel area, as follows:

Word No.	Contents
1	Channel status word. The contents of the bits of this word are:

Bit No.	Contents
0	1 – a hardware error occurred on this channel.
1-5	Index into monitor tables. This describes the physical device with which the channel is associated.
6	1 – a .RENAME operation is in progress on the channel.
7	1 – a .CLOSE operation must rewrite the directory (i.e., set when a .ENTER is done).
8-12	Contains the directory segment number (1–37(8)) in which the current open file can be found.
13	1 – an end-of-file was found on the channel.
14	Unused.
15	1 – this channel is currently in use (i.e., a file is open on this channel).

Word No.	Contents
2	Starting block number of the file. Zero for sequential-access devices.
3	Length of file (in 256-word blocks).
4	Data length of file; currently unused.
5	Even Byte: I/O count. Count of how many I/O requests have been made on this channel. Odd Byte: Unit number of the device associated with the channel (between 0 – 7).

While the .SAVESTATUS/.REOPEN combination is very useful, care must be observed when using it. In particular, the following cases should be avoided:

1. If a .SAVESTATUS is performed and the same file is then deleted before it is reopened, it becomes available as an empty space which could be used by the .ENTER command. If this sequence occurs, the contents of the file supposedly saved will change.
2. Although the device handler for the required peripheral need not be in memory for execution of a .REOPEN, if the handler is not in memory when a .READ or .WRITE is executed, a fatal error is generated.

Errors:

Code	Explanation
1	The file was opened via .ENTER and a .SAVESTATUS is illegal.

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 their status saved, then they 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 file data, and then .UNLOCKing the USR.

In the program segment below, three input files are specified in the command string; these are then processed one at a time.

```
          .MCALL      .REGDEF,.CSIGEN,.SAVESTATUS,.REOPEN
          .MCALL      .READ,.EXIT
          .REGDEF

START:    MOV         #AREA,R5
          .CSIGEN     #DSPACE,#DEXT            ;GET INPUT FILES

          MOV         R0,BUFF                  ;SAVE POINTER TO FREE MEMORY

          .SAVESTATUS R5,#3,#BLOCK1            ;SAVE FIRST INPUT FILE
          .SAVESTATUS R5,#4,#BLOCK2            ;SAVE SECOND FILE
          .SAVESTATUS R5,#5,#BLOCK3            ;SAVE THIRD FILE

          MOV         #BLOCK1,R4
PROCESS:  .REOPEN     R5,#0,R4                 ;REOPEN FILE ON
                                               ;CHANNEL 0

          .READ       R5,#0,BUFF,COUNT,BLOCK   ;PROCESS FILE ON CHANNEL 0

DONE:     ADD         #12,R4                   ;POINT TO NEXT SAVESTATUS BLOCK
          CMP         R4,#BLOCK3               ;LAST FILE PROCESSED?
          BLOS        PROCESS                  ;NO – DO NEXT
          .EXIT

BLOCK1:   .WORD       0,0,0,0,0                ;MEMORY BLOCKS FOR
BLOCK2:   .WORD       0,0,0,0,0                ;SAVESTATUS INFORMATION
BLOCK3:   .WORD       0,0,0,0,0
AREA:     .BLKW       10

BUFF:     .WORD       0
BLOCK:    .WORD       0
COUNT:    .WORD       256.

DEXT:     .WORD       0,0,0,0
DSPACE=.
          .END        START
```

.SETTOP

9.4.27 .SETTOP

The .SETTOP request allows the user program to request that a new address be specified as a program's upper limit. The monitor determines whether this address is legal and whether or not a memory 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. If .SETTOP specifies a high limit greater than the address of the USR and a SET USR NOSWAP command has not been given, a memory swap is required. Section 9.2.5 gives details on determining where the USR is in memory and how to optimize the .SETTOP.

On return from .SETTOP, both R0 and the word at location 50 (octal) contain the highest memory address allocated for use. If the job requested an address higher than the highest address which is legal for the requesting job, it is adjusted down to that address.

Macro Call: .SETTOP .addr

where: .addr is the address of the word immediately following the free area desired.

NOTES

1. A program should never do a .SETTOP and assume that its new upper limit is the address it requested. It must always examine the returned contents of R0 or location 50 to determine its actual high address.
2. It is imperative that the value returned in R0 or location 50 be used as the absolute upper limit. If this value is ever exceeded, vital parts of the monitor may be destroyed, and the system integrity will be violated.

Errors:

None.

Example:

Following is an example in two parts. The first indicates how a small job (i.e., one with free space between itself and the USR) can be assured of reserving space up to but not including the USR. This in effect gives the job all the space it can without causing the USR to become non-resident.

The second part indicates how to always reserve the maximum amount of space by making the USR non-resident.

```
I)              .MCALL      .REGDEF,.SETTOP,.EXIT
                .REGDEF

    START:
    RMON=54                                     ;POINTER TO START OF RESIDENT
    USR=266                                     ;OFFSET FROM RESIDENT TO POINTER
                                                ;WHERE USR WILL START.
                MOV         @#RMON,R1           ;START OF RMON TO R1
                MOV         USR(R1),R0          ;POINT TO LOWEST USR WORD
                TST         -(R0)               ;POINT TO HIGHEST WORD NOT IN USR
                .SETTOP                         ;AND ASK FOR IT
                MOV         R0,HICORE           ;R0 CONTAINS THE HIGH ADDRESS
                                                ;THAT WAS RETURNED.

II)             .SETTOP     #-2                 ;IF WE ASK FOR A VALUE GREATER
                                                ;THAN START OF RESIDENT, WE
                                                ;WILL GET BACK THE ABSOLUTELY
                                                ;HIGHEST USABLE ADDRESS.
                MOV         R0,HICORE           ;THAT IS OUR LIMIT NOW

                .EXIT
    HICORE:     .WORD       0
                .END        START
```

If a SET USR NOSWAP command is executed, the USR cannot be made non-resident. In this case, in both I and II above, R0 would return a value just below the USR.

Caution should be used concerning technique I, above. If the program is so large that the USR is normally positioned over part of it, the high limit value returned by the .SETTOP may actually be lower than the original limit. The USR is then resident, with a portion of the user program destroyed. The example in Section 9.2.5 shows how to include checks that will avoid this situation.

9.4.28 .SFPA

.SFPA allows users with floating point hardware (FPP on 11/45 and FIS on 11/40 and LSI-11) to set trap addresses to be entered when a floating point exception occurs. If no user trap address is specified and a floating point (FP) exception occurs, a ?M-FP TRAP occurs, and the job is aborted.

Macro Call: .SFPA .area, .addr

where: .addr is the address of the routine to be entered when an exception occurs.

Request Format:

R0 ⇒ .area:

30	0
.addr	

NOTES

1. If the address argument is 0, user floating point routines are disabled and the fatal ?M-FP TRAP error is produced.
2. When the user routine is activated, it is necessary to re-execute an .SFPA request, as the monitor inhibits user traps when any one is serviced. It does this to inhibit any possible infinite loop being set up by repeated FP exceptions.
3. If the 11/45 FPP is being used, the instruction STST – (SP) is executed by the monitor before entering the user's trap routine. Thus, the trap routine must pop the two status words off the stack before doing an RTI. The program can tell if FPP hardware is available by examining the configuration word in the monitor (see Section 9.2.6).

Errors:

None.

Example:

This example sets up a user FP trap address.

```
        .MCALL      .REGDEF,.SFPA,.EXIT
        .REGDEF

START:
        .SFPA       #AREA,#FPTRAP

        .EXIT
```

```
FPTRAP:
                .
                .
                MOV         R0,-(SP)                ;R0 USED BY .SFPA
                .SFPA       #AREA,#FPTRAP
                MOV         (SP)+,R0                ;RESTORE R0
                RTI
                .
AREA:           .BLKW       10
                .END        START
```

.SRESET

9.4.29 .SRESET

The .SRESET (software reset) request performs the following functions:

1. Dismisses any device handlers which were brought into memory via a .FETCH call. Handlers which were loaded via the Keyboard Monitor LOAD command remain resident, as does the system device handler.
2. Purges any currently open files. Files opened for output with .ENTER will never be made permanent.
3. Reverts to using only 16 (decimal) I/O channels. Any channels defined with .CDFN are discarded. A .CDFN must be reissued to open more than 16 (decimal) channels after a .SRESET is performed.
4. Resets the I/O queue to one element. A .QSET must be reissued to allocate extra queue elements.
5. Clears completion queue of any completion routines.

Macro Call: .SRESET

Errors:

None.

Example:

In the example below, .SRESET is used prior to calling the CSI to ensure that all handlers are removed from memory and the CSI is started with a free handler area.

```
                .MCALL      .REGDEF,.CSIGEN,.SRESET,.EXIT
                .REGDEF

START:          .CSIGEN     #DSPACE,#DEXT,#0        ;GET COMMAND STRING
                MOV         R0,BUFFER               ;R0 POINTS TO FREE MEMORY

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.

                .END        START
```

If the .SRESET had not been performed prior to the second call of .CSIGEN, it is possible that the second command string would load a handler over one that the monitor thought was resident from the first command line.

.TRPSET

9.4.30 .TRPSET

.TRPSET allows the user job to intercept traps to 4 and 10 instead of having the job aborted with a ?M-TRAP TO 4 or ?M-TRAP TO 10 message. If .TRPSET is in effect when a trap occurs, the user-specified routine is entered. The sense of the C bit on entry to the routine determines which trap occurred: C bit clear indicates a trap to 4; set indicates a trap to 10. The user routine should exit via an RTI instruction.

Macro Call: .TRPSET .area, .addr

where: .addr is the address of the user's trap routine. If an address of 0 is specified, the user's trap interception is disabled.

Request Format:

R0 ⇒ .area:

3	0
.addr	

NOTES

It is necessary to reissue a .TRPSET request whenever a trap occurs and the user routine is entered. The monitor inhibits servicing user traps prior to entering the first user trap routine. Thus, if a trap should occur from within the user's trap routine, a ?M-TRAP message is generated. The last operation the user routine should perform before an RTI is to reissue the .TRPSET request.

Errors:

None.

Example:

The following example sets up a user trap routine and, when the trap occurs, prints an appropriate error message.

```
        .MCALL    .REGDEF,.TRPSET,.EXIT,.PRINT
        .REGDEF

START:
        .TRPSET   #AREA,#TRPLOC
        MOV       #101,R0          ;SET TO PRODUCE A TRAP
        TST       (R0)+            ;THIS WILL TRAP TO 4.
        .WORD     67               ;THIS WILL TRAP TO 10.
        .EXIT

TRPLOC: MOV       R0,-(SP)         ;R0 USED BY EMTS
        BCS       1$               ;C SET = TRAP TO 10
        .PRINT    #TRP4            ;TRAP TO 4
        BR        2$
```

```
1$:      .PRINT    #TRP10           ;TRAP TO 10
2$:      .TRPSET   #AREA,#TRPLOC    ;RESET TRAP ADDRESS
         MOV       (SP)+,R0         ;RESTORE R0
         RTI

AREA:    .BLKW     10
TRP4:    .ASCIZ    /TRAP TO 4/
TRP10:   .ASCIZ    /TRAP TO 10/
         .EVEN

         .END      START
```

.TTYIN/.TTINR

9.4.31 .TTYIN/.TTINR

These requests are used to transfer characters from the terminal to the user program. The character thus obtained appears right-justified (even byte) in R0.

The expansion of .TTYIN is:

```
EMT 340
BCS .-2
```

while that for .TTINR is:

```
EMT 340
```

If no characters or lines are available when an EMT 340 is executed, return is made with the C bit set. The implication of these calls is that .TTYIN causes a tight loop waiting for a character/line to appear, while the user can either wait or continue processing using .TTINR.

Macro Calls: .TTYIN .char

.TTINR

where: .char is the location where the character in R0 is 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 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:

1. The monitor echoes all characters typed; lower case characters are converted to upper case.
2. CTRL U (↑U) and RUBOUT perform line deletion and character deletion, respectively.
3. A carriage return, line feed, CTRL Z, or CTRL C must be struck before characters on the current line are available to the program. When carriage return is typed, characters on the line typed are passed one-by-one to the user program; both carriage return and line feed are passed to the program.
4. ALTMODEs (octal codes 175 and 176) are converted to ESCAPEs (octal 33).

If bit 12 = 1, the terminal is in special mode. The effects are:

1. The monitor does not echo characters typed except for CTRL C and CTRL O.
2. CTRL U and RUBOUT do not perform special functions.
3. Characters are immediately available to the program.
4. No ALTMODE conversion is done.

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 HT-11.

CTRL S and CTRL Q are intercepted by the monitor.

Errors:

Code	Explanation
0	No characters available in ring buffer.

Example:

Refer to the example following the description of .TTYOUT/.TTOUTR.

.TTYOUT/.TTOUTR

9.4.32 .TTYOUT/.TTOUTR

These requests cause a character to be transmitted from R0 to the terminal. The difference, as in 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, while the .TTOUTR does not wait for room and the character in R0 is not output.

Macro Calls: .TTYOUT .char

.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. Upon return from the request, R0 still contains the character.

If the carry bit is set when execution of the .TTOUTR request is completed, it indicates that there is no room in the buffer and that 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 terminal 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.

Errors:

Code	Explanation
0	Output ring buffer full.

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 until both requests are satisfied, hence there is no time available for any other processing while waiting.

```
          .MCALL        .REGDEF,.TTYIN,.TTYOUT
          .REGDEF

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          #12,R0                  ;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

BUFFER=.

          .END          START
```

Rather than wait for the user to type something at INLOOP or wait for the output buffer to have available space at OUTLOOP, the routine can be recoded using .TTINR and .TTOUTR as follows:

```
          .MCALL        .REGDEF,.TTYIN,.TTYOUT
          .REGDEF
          .MCALL        .TTINR,.TTOUTR,.EXIT

START:    MOV           #BUFFER,R1              ;POINT R1 TO BUFFER
          CLR           R2                      ;CLEAR CHARACTER COUNT
INLOOP:   .TTINR                                ;GET CHAR FROM TERMINAL
          BCS           NOCHAR                  ;NONE AVAILABLE
CHRIN:    MOVB          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:  MOVB          (R1),R0                 ;PUT CHAR IN R0
          .TTOUTR                               ;TYPE IT
          BCS           NOROOM                  ;NO ROOM IN OUTPUT BUFFER
CHROUT:   DEC           R2                      ;DECREASE COUNT
          BEQ           START                   ;DONE IF COUNT=0
          INC           R1                      ;BUMP BUFFER POINTER
          BR            OUTLOOP                 ;AND TYPE NEXT
NO CHAR:
```

```
         .TTINR                        ;PERIODIC CHECK FOR
                                       ;CHARACTER AVAILABILITY
         BCC       CHRIN               ;GOT ONE
          .
          .
          (code to be executed
          while waiting)
          .
          .
          .
         BR        NOCHAR
NOROOM:
         MOVB      (R1),R0             ;PERIODIC ATTEMPT TO TYPE
                                       ;CHARACTER
         .TTOUTR
         BCC       CHROUT              ;SUCCESSFUL
          .
          .
          (code to be executed
          while waiting)

TYPEIT:  .TTYOUT   (R1)                ;PUT CHAR
         BR        CHROUT

BUFFER:  .BLKW     100.
         .END      START
```

.WAIT

9.4.33 .WAIT

The .WAIT request suspends program execution until all input/output requests on the specified channel are completed. The .WAIT request combined with the .READ/.WRITE requests make double-buffering a simple process.

.WAIT also conveys information back through its error returns. An error is returned if either the channel is not currently open or if the last I/O operation resulted in a hardware error.

Macro Call: .WAIT .chan

Request Format:

R0 ⇒ | 0 | .chan |

Errors:

Code	Explanation
0	Channel specified is not open.
1	Hardware error occurred on the previous I/O operation on this channel.

These error codes make the .WAIT request useful in checking channel status.

Example:

For an example of .WAIT used for I/O synchronization, see the examples in the next section.

An 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 ASEMBL might use the following code to determine if a listing file is desired.

```
        .MCALL      .REGDEF,.WAIT,.CSIGEN,.EXIT
        .REGDEF

START:

        .CSIGEN     #DSPACE,#DEXT,#0      ;PROCESS COMMAND STRING
        .WAIT       #0                    ;CHECK FOR FILE IN FIRST FIELD
        BCS         NOBINARY              ;NO BINARY DESIRED

NOBINARY:

        .WAIT       #1                    ;CHECK FOR LISTING SPECIFICATION
        BCS         NOLISTING             ;NO LISTING DESIRED

NOLISTING:

        .WAIT       #3                    ;CHECK FOR INPUT FILE OPEN
        BCS         ERROR                 ;NO INPUT FILE

ERROR:  .EXIT

DEXT:   .RAD50      /PAL/
        .RAD50      /OBJ/
        .RAD50      /LST/
        .WORD       0
DSPACE=.
        .END        START
```

.WRITE/.WRITC/.WRITW

9.4.34 .WRITE/.WRITC/.WRITW

Note that in the case of .WRITE and .WRITC, additional queue elements should be allocated for buffered I/O operations (see .QSET).

.WRITE

The .WRITE request transfers a specified number of words from memory to the specified channel. Control returns to the user program immediately after the request is queued.

Macro Call: .WRITE .area, .chan, .buff, .wcnt, .blk

where:

	.buff	is the address of the memory buffer to be used for output.
	.wcnt	is the number of words to be written.
	.blk	is the number of the block to be written.

Request Format:

R0 ⇒ .area:

11	.chan
.blk	
.buff	
.wcnt	
1	

Notes:

See the note following .WRITW.

Errors:

Code	Explanation
0	Attempted to write past end-of-file.
1	Hardware error.
2	Channel was not opened.

Example:

Refer to the examples following .WRITW.

.WRITC

The .WRITC request transfers a specified number of words from memory to a specified channel. Control returns to the user program immediately after the request is queued. Execution of the user program continues until the .WRITC 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 .area, .chan, .buff, .wcnt, .crtn, .blk

where:

.buff	is the address of the memory buffer to be used for output.
.wcnt	is the number of words to be written.
.crtn	is the address of the completion routine to be entered (see Section 9.2.8).
.blk	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 .blk before it is used again.

Request Format:

R0 ⇒ .area:

11	.chan
.blk	
.buff	
.wcnt	
.crtn	

When entering a .WRITC completion function the following are true:

1. R0 contains the channel status word for the operation. If bit 0 of R0 is set, a hardware error occurred during the transfer. The data may not be reliable.
2. R1 contains the channel number of the operation. This is useful when the same completion function is to be used for several different transfers.

Notes:

See the note following .WRITW.

Errors:

Code	Explanation
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 the examples following .WRITW.

.WRITW

The .WRITW request transfers a specified number of words from memory to the specified channel. Control returns to the user program when the .WRITW is complete.

Macro Call: .WRITW .area, .chan, .buff, .wcnt, .blk

where:

.buff	is the address of the buffer to be used for output.
.wcnt	is the number of words to be written. The number must be positive.
.blk	is the number of the block to be written.

Request Format:

R0 ⇒ .area:

11	.chan
.blk	
.buff	
.wcnt	
0	

NOTE

Upon return from any WRITE programmed request, R0 will contain no information if the write is to a sequential-access device. If the write is to a random-access device, R0 contains the number of words that will be written (.WRITE or .WRITC) or have been written (.WRITW). If a request is made to write past the end-of-file on a random-access device, the word count is shortened and an error is returned. Note that the write will be done and a completion routine, if specified, will be entered, unless the request cannot be partially filled (shortened word count = 0).

Errors:

Code	Explanation
0	Attempted to write past EOF.
1	Hardware error.
2	Channel was not opened.

Examples:

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).

```
        .MCALL    .REGDEF,.FETCH,.READW,.WRITW
        .MCALL    .ENTER,.LOOKUP,.PRINT,.EXIT,.CLOSE,.WAIT
        .REGDEF

ERRWD=52
```

```
START:    .FETCH    #HSPACE,#PRNAME          ;GET PR HANDLER
          BCS       FERR                     ;PR NOT AVAILABLE
          MOV       R0,R2                    ;R0 HAS NEXT FREE LOCATION
          .FETCH    R2,#PPNAME               ;GET PP HANDLER
          BCS       FERR                     ;NOT AVAILABLE
          MOV       #AREA,R5                 ;EMT ARGUMENT AREA
          CLR       R4                       ;R4 IS OUTPUT CHANNEL; 0
          MOV       #1,R3                    ;R3 IS INPUT CHANNEL ;1
          .ENTER    R5,R4,#PPNAME            ;ENTER THE FILE
          BCS       ENERR                    ;SOME ERROR IN ENTER
          .LOOKUP   R5,R3,#PRNAME            ;LOOKUP FILE ON CHANNEL 1
          BCS       LKERR                    ;ERROR IN LOOKUP
          CLR       R1                       ;USE R1 AS BLOCK NUMBER
LOOP:     .READW    R5,R3,#BUFF,#256.,R1     ;READ ONE BLOCK
          BCS       RDERR
          .WRITW    R5,R4,#BUFF,#256.,R1     ;WRITE THAT BLOCK
          BCS       WTERR
          INC       R1                       ;BUMP BLOCK. NOTE: THIS IS
                                             ;NOT NECESSARY FOR NON-FILE
                                             ;DEVICES IN GENERAL. IT IS
                                             ;USED HERE AS AN EXAMPLE OF
                                             ;A GENERAL TECHNIQUE.
          BR        LOOP                     ;KEEP GOING
RDERR:    TSTB      ERRWD                    ;ERROR. IS IT EOF?
          BEQ       1$                       ;YES
          .PRINT    #RDMSG                   ;NO, HARD READ ERROR
          .EXIT
1$:       .CLOSE    R3                       ;CLOSE INPUT AND OUTPUT
          .CLOSE    R4
          .EXIT                              ;AND EXIT.
WTERR:    .PRINT    #WTMSG
          .EXIT
PRNAME:   RAD50     /PR /                    ;NOTE THAT PR NEEDS NO FILE NAME
          .WORD     0                        ;FILE NAME NEED ONLY BE 0.
PPNAME:   .RAD50    /PP /
          .WORD     0
FERR:     .PRINT    #FMSG                    ;ERROR ACTIONS GO HERE. IT IS
          .EXIT                              ;GENERALLY UNDESIRABLE TO
ENERR:    .PRINT    #EMSG                    ;EXECUTE A HALT OR RESET
          .EXIT                              ;INSTRUCTION ON ERROR.
LKERR:    .PRINT    #LMSG
          .EXIT
FMSG:     .ASCIZ    /NO DEVICE?/
EMSG:     .ASCIZ    /ENTRY ERROR?/
LMSG:     .ASCIZ    /LOOKUP ERROR?/
RDMSG:    .ASCIZ    /READ ERROR?/
WTMSG:    .ASCIZ    /WRITE ERROR?/
          .EVEN
AREA:     .BLKW     10
BUFF:     .BLKW     256.
HSPACE=.
          .END      START
```

The same routine can 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.

```
            .MCALL      .REGDEF,.FETCH,.READ,.WRITE
            .MCALL      .ENTER,.LOOKUP,.PRINT,.EXIT,.CLOSE,.WAIT
            .REGDEF

ERRWD=52

START:      .FETCH      #HSPACE,#PRNAME         ;GET PR HANDLER
            BCS         FERR                    ;PR NOT AVAILABLE
            MOV         R0,R2                   ;R0 HAS NEXT FREE LOCATION
            .FETCH      R2,#PPNAME              ;GET PP HANDLER
            BCS         FERR                    ;NOT AVAILABLE
            MOV         #AREA,R5                ;EMT ARGUMENT AREA
            CLR         R4                      ;R4 IS OUTPUT CHANNEL; 0
            MOV         #1,R3                   ;R3 IS INPUT CHANNEL ;1
            .ENTER      R5,R4,#PPNAME           ;ENTER THE FILE
            BCS         ENERR                   ;SOME ERROR IN ENTER
            .LOOKUP     R5,R3,#PRNAME           ;LOOKUP FILE ON CHANNEL 1
            BCS         LKERR                   ;ERROR IN LOOKUP
            CLR         R1                      ;USE R1 AS BLOCK NUMBER
LOOP:       .READ       R5,R3,#BUFF,#256.,R1    ;READ A BUFFER
            BCS         RDERR
            .WAIT       R3                      ;WAIT FOR BUFFER
            BCS         IOERR                   ;ERROR HERE IS HARD ERROR
            .WRITE      R5,R4,#BUFF,#256.,R1    ;WRITE THE BUFFER
            BCS         IOERR                   ;I/O ERROR
            INC         R1
            BR          LOOP                    ;KEEP GOING
RDERR:      TSTB        ERRWD                   ;ERROR, IS IT EOF?
            BNE         IOERR                   ;NO,HARD ERROR
            .CLOSE      R3                      ;CLOSE INPUT AND OUTPUT
            .CLOSE      R4
            .EXIT                               ;AND EXIT.
IOERR:      .PRINT      #IOMSG                  ;NO, HARD READ ERROR
            .EXIT
PRNAME:     .RAD50      /PR /                   ;NOTE THAT PR NEEDS NO FILE NAME
            .WORD       0                       ;FILE NAME NEED ONLY BE 0.
PPNAME:     .RAD50      /PP /
            .WORD       0
FERR:       .PRINT      #FMSG                   ;ERROR ACTIONS GO HERE. IT IS
            .EXIT                               ;GENERALLY UNDESIRABLE TO
ENERR:      .PRINT      #EMSG                   ;EXECUTE A HALT OR RESET
            .EXIT                               ;INSTRUCTION ON ERROR.
LKERR:      .PRINT      #LMSG
            .EXIT
```

```
FMSG:       .ASCIZ      /NO DEVICE?/
EMSG:       .ASCIZ      /ENTRY ERROR?/
LMSG:       .ASCIZ      /LOOKUP ERROR?/
IOMSG:      .ASCIZ      "I/O ERROR?"
WTMSG:      .ASCIZ      /WRITE ERROR?/
            .EVEN
AREA:       .BLKW       10
BUFF:       .BLKW       256.
HSPACE=.
            .END        START
```

.READ and .WRITE are also often used for double-buffered I/O. The basic double-buffering algorithm for input is:

```
                                            Explanation

        READ    BUFFER 1                    Fill BUFFER 1
LOOP:   WAIT    BUFFER 1                    Wait for BUFFER 1 to fill
        READ    BUFFER 2                    Start filling BUFFER 2
        USE     BUFFER 1                    Process BUFFER 1 while BUFFER 2 fills
        WAIT    BUFFER 2                    Wait for BUFFER 2 to fill
        READ    BUFFER 1                    Start filling BUFFER 1
        USE     BUFFER 2                    Process BUFFER 2 while BUFFER 1 fills
        BR      LOOP
```

Correspondingly, the basic double-buffering algorithm for output is:

```
                                            Explanation

        FILL BUFFER 1                       Prepare BUFFER 1 for output
LOOP:   WRITE BUFFER 1                      Start emptying BUFFER 1
        FILL BUFFER 2                       Fill BUFFER 2 while BUFFER 1 empties
        WAIT BUFFER 1                       Wait for BUFFER 1 to empty
        WRITE BUFFER 2                      Start emptying BUFFER 2
        FILL BUFFER 1                       FILL BUFFER 1 while BUFFER 2 empties
        WAIT BUFFER 2                       Wait for BUFFER 2 to empty
        BR LOOP
```

The previous 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 routines.

```
            .MCALL      .REGDEF,.FETCH,.READC,.WRITC
            .MCALL      .ENTER,.LOOKUP,.PRINT,.EXIT,.CLOSE,.WAIT
            .REGDEF

ERRBYT=52
```

```
START:   .FETCH   #HSPACE,#PRNAME           ;GET PR HANDLER
         BCS      FLNK                      ;PR NOT AVAILABLE
         MOV      R0,R2                     ;R0 HAS NEXT FREE LOCATION
         .FETCH   R2,#PPNAME                ;GET PP HANDLER
FLNK:    BCS      FERR                      ;NOT AVAILABLE
         MOV      #AREA,R5                  ;EMT ARGUMENT AREA
         CLR      R4                        ;R4 IS OUTPUT CHANNEL; 0
         MOV      #1,R3                     ;R3 IS INPUT CHANNEL ;1
         .ENTER   R5,R4,#PPNAME             ;ENTER THE FILE
         BCS      ENERR                     ;SOME ERROR IN ENTER
         .LOOKUP  R5,R3,#PRNAME             ;LOOKUP FILE ON CHANNEL 1
         BCS      LKERR                     ;ERROR IN LOOKUP
         CLR      R1                        ;USE R1 AS BLOCK NUMBER
LOOP:    CLR      DFLG                      ;CLEAR DONE/ERROR FLAG
         .READC   R5,R3,#BUFF,#256.,#RDCOMP,R1  ;READ ONE BLOCK
         BCS      EOF                       ;NO ERROR WILL HAPPEN HERE
1$:      TST      DFLG                      ;DONE FLAG SET?
         BEQ      1$                        ;NO, WAIT FOR IT TO BE SET.
         BMI      TOERR                     ;YES, BUT HARD ERROR OCCURRED
EOF:     .CLOSE   R3                        ;CLOSE INPUT AND OUTPUT CHANNELS
         .CLOSE   R4
         .EXIT                              ;ALL DONE
.ENABL   LSB
RDCOMP:  ROR      R0                        ;IF BIT 0 SET
         BCS      RWERR                     ;AN ERROR OCCURRED.
         .WRITC   #AREA,#0,#BUFF,#256.,#WRCOMP,BLKN  ;WRITE THAT BLOCK
         BCC      2$                        ;ERROR HERE IS HARDWARE
RWERR:   MOV      #-1,DFLG                  ;FLAG THE ERROR
2$:      RTS      PC
WRCOMP:  ROR      R0
         BCS      RWERR                     ;HARDWARE ERROR
         INC      BLKN                      ;BUMP BLOCK NUMBER.
         .READC   #AREA,#1,#BUFF,#256.,#RDCOMP,BLKN
         BCC      3$                        ;NO ERROR
         TSTB     ERRBYT                    ;EOF?
         BNE      RWERR                     ;NO, HARD ERROR
         INC      DFLG                      ;SAY WE'RE DONE
3$:      RTS      PC
.DSABL   LSB
FERR:    MOV      #FMSG,R0                  ;ERROR ACTIONS GO HERE. IT IS
         BR       TYPIT                     ;GENERALLY UNDESIRABLE TO
ENERR:   MOV      #EMSG,R0                  ;EXECUTE A HALT OR RESET
         BR       TYPIT                     ;INSTRUCTION ON ERROR.
IOERR:   MOV      #IOMSG,R0
         BR       TYPIT
LKERR:   MOV      #LMSG,R0
TYPIT:   .PRINT
         .EXIT
.NLIST BEX
```

```
FMSG:       .ASCIZ      /NO DEVICE?/
EMSG:       .ASCIZ      /ENTRY ERROR?/
LMSG:       .ASCIZ      /LOOKUP ERROR?/
IOMSG:      .ASCIZ      "I/O ERROR?"
.LIST BEX
.EVEN
DFLG:       .WORD       0
PRNAME:     .RAD50      /PR /                  ;NOTE THAT PR NEEDS NO FILE NAME
            .WORD       0                      ;FILE NAME NEED ONLY BE 0.
PPNAME:     .RAD50      /PP /
            .WORD       0
BLKN:       .WORD       0                      ;BLOCK NUMBER
AREA:       .BLKW       10
BUFF:       .BLKW       256.
HSPACE=.
            .END        START
```

The following example incorporates the .LOOKUP, .READW, and .CLOSE requests. The program opens the file HT11.MAC which is on the system device, SY:, for input on channel 0. The first block is read and the file is then closed.

```
            .MCALL      .REGDEF,.CLOSE,.LOOKUP
            .MCALL      .PRINT,.EXIT,.READW,.FETCH
            .REGDEF

START:      MOV         #LIST,R5               ;EMT ARGUMENT LIST POINTER
            CLR         R4                     ;BLOCK NUMBER
            CLR         R3                     ;CHANNEL #
            .FETCH      #CORADD,#FPTR          ;FETCH DEVICE HANDLER
            BCC         2$
            MOV         #FETMSG,R0             ;FETCH ERROR
1$:         .PRINT                             ;PRINT ERROR MESSAGE
            .EXIT
2$:         .LOOKUP     R5,R3,#FPTR            ;LOOKUP FILE ON CHANNEL 0
            BCC         3$
            MOV         #LKMSG,R0              ;PRINT FAILURE MESSAGE
            BR          1$
3$:         .READW      R5,R3,#BUFF,#256.,R4   ;READ ONE BLOCK
            BCC         4$
            MOV         #RDMSG,R0              ;READ ERROR
            BR          1$
4$:         .CLOSE      R3                     ;CLOSE THE CHANNEL
            .EXIT
```

```
LIST:       .BLKW    5                   ;LIST FOR EMT CALLS
FPTR:       .RAD50   /SY HT11   MAC/     ;RAD50 OF FILE NAME, DEVICE
FETMSG:     .ASCIZ   /FETCH FAILED/      ;ASCII MESSAGES
LKMSG:      .ASCIZ   /LOOKUP FAILED/
RDMSG:      .ASCIZ   /READ FAILED/
            .EVEN
CORADD:     .BLKW    2000                ;SPACE FOR LARGEST HANDLERS
BUFF=.
            .END     START
```

APPENDIX A
COMMAND AND SWITCH SUMMARIES

Command and switch summaries of the various system and utility programs are grouped here for the user's convenience. Refer to the appropriate chapter for details.

A.1 KEYBOARD MONITOR (Chapter 2)

A.1.1 Command Summary

Only those command characters underlined need be entered; all command lines are terminated by typing a carriage return.

Command Format	Explanation
ASSIGN dev:udev	Assigns a user-defined name (udev) as an alternate name for a device (dev). Deassigns synonyms when used without any arguments.
B location	Sets a relocation base (location), which is an octal address to be used as a base address for subsequent Examine and Deposit commands.
CLOSE	Causes all currently open files to become permanent.
DATE dd-mmm-yy	Enters the indicated day-month-year (dd-mmm-yy); this date is then assigned to newly created files, new device directory entries, and listing output. When used without an argument, the current date (as entered) is printed.
D location = value1,value2,...,valuen	Deposits the specified values starting at the given location (location represents an octal address which is added to the base address to obtain the actual address at which values will be deposited).
E location m-location n	Prints the contents of the specified locations in octal on the terminal (location represents an octal address which is added to the base address to obtain the actual address examined).
GET dev:filnam.ext	Loads the specified memory image file (filnam.ext) into memory from the indicated device (dev:).
INITIALIZE	Resets the system tables; makes nonresident all handlers not loaded and purges the I/O channels.
LOAD dev,...	Makes a device handler resident for use.
R filnam.ext	Loads the specific memory image file (filnam.ext) into memory from the system device and starts execution.
REENTER	Starts a program at its reentry address (i.e., its start address –2).

Command Format	Explanation
RUN dev:filnam.ext	Loads the specified memory image file (filnam.ext) into memory from the indicated device (dev:) and starts execution.
SAVE dev:filnam.ext area1,area2-arean	Writes the area(s) of user memory specified into the named file (filnam.ext) in save image format. Memory is transferred in 256-word blocks.
SET dev: {NO} option=value	Used to change device (dev:) handler characteristics and certain system configuration parameters. Consult Chapter 2 for a list of options.
START address	Begins execution of the program currently in memory at the specified address. If an address is not indicated, the starting address in location 40 is used.
TIME hh:mm:ss	Enters time of day in hours, minutes, seconds past midnight (hh:mm:ss). If all three arguments are omitted, the current time of day is output.
UNLOAD dev,dev,...	Makes previously loaded handlers (dev) nonresident and frees the memory space they were using.

A.1.2 Special Function Keys

Key	Function
CTRL C	Echoes ↑C on the terminal, interrupts current program execution, and returns control to the Keyboard Monitor. If a program is waiting for terminal input or is using the device handler TT: for input, typing a single CTRL C interrupts execution and returns control to the monitor command level. Otherwise, two CTRL C's must be typed in order to interrupt execution.
CTRL D	Echoes ↑D and ends a file on PR: .
CTRL O	Echoes ↑O on the terminal and causes suppression of teleprinter output while continuing program execution. Teleprinter output is reenabled when one of the following occurs: 1. A second CTRL O is typed 2. A return to the monitor is indicated via CTRL C 3. The running program issues a .RCTRLO directive (see Chapter 9)
CTRL Q	Does not echo. Resumes printing characters on the terminal from the point at which printing was previously stopped (via CTRL S).
CTRL S	Does not echo. Temporarily suspends output to the terminal until a CTRL Q is typed.
CTRL U	Echoes ↑U followed by a carriage return on the terminal and deletes the current input line.
CTRL Z	Echoes ↑Z on the terminal and terminates input when used with the terminal device handler (TT:).
RUBOUT	Deletes the last character from the current line. Echoes a backslash plus the character deleted; each succeeding RUBOUT deletes and echoes another character; an enclosing backslash is printed when a key other than RUBOUT is typed.

A.2 EDITOR (Chapter 3)

A.2.1 Command Arguments

Format	Meaning
n	A decimal integer (in the range –16383 to +16383) which may, except where noted, be preceded by a + or –. Whenever an argument is acceptable in a command, its absence implies an argument of 1.
0	Refers to the beginning of the current line.
/	Refers to the end of the text in the current Text Buffer.
=	Is used with the J, D and C commands only and represents –n, where n is equal to the length of the last text argument used.

A.2.2 Input and Output Commands

Command	Form	Meaning
EDIT BACKUP	EB dev:filnam.ext[n] $	Opens a file for editing, creating a backup copy (.BAK).
EDIT READ	ER dev:filnam.ext$	Opens a file for input.
EDIT WRITE	EW dev:filnam.ext[n] $	Creates a new file for output.
END FILE	EF	Closes the current output file without performing any further input/output operations.
EXIT	EX	Outputs the remainder of the input file to the output file and returns control to the monitor.
LIST	(–)nL 0L /L	Prints a specified number of lines on the terminal.
NEXT	nN	Outputs the contents of the Text Buffer to the output file, clears the buffer, and reads in the next page of the input file.
READ	R	Reads a page of text from the input file and appends it to the contents of the buffer.
VERIFY	V	Prints the current text line (the line containing the pointer) on the terminal.
WRITE	(–)nW 0W /W	Outputs a specified number of lines of text from the Text Buffer to the output file.

A.2.3 Pointer Relocation Commands

Command	Form	Meaning
ADVANCE	(–)nA 0A /A	Moves the pointer over a specified number of lines in the Text Buffer. The pointer is positioned at the beginning of the line.
BEGINNING	B	Moves the current location pointer to the beginning of the Text Buffer.
JUMP	(–)nJ 0J /J =J	Moves the pointer over a specified number of characters in the Text Buffer.

A.2.4 Search Commands

Command	Form	Meaning
FIND	nFtext$	Beginning at the current location pointer, searches the entire text file for the nth occurrence of the specified character string. Pages of text are read into the Text Buffer, searched, and then written to the output file until the text string is found.
GET	nGtext$	Searches the contents of the Text Buffer, beginning at the current location pointer, for the next occurrence of the text string.
POSITION	nPtext$	Searches the input file for the nth occurrence of the text string; if the text string is not found, the buffer is cleared and a new page is read from the input file.

A.2.5 Text Modification Commands

Command	Form	Meaning
CHANGE	(–)nCtext$ 0Ctext$ /Ctext$	Replaces n characters, beginning at the pointer, with the indicated text string.
DELETE	(–)nD 0D /D =D	Removes a specified number of characters from the Text Buffer, beginning at the current location pointer.
EXCHANGE	(–)nXtext$ 0Xtext$ /Xtext$	Replaces n lines, beginning at the pointer, with the indicated text string.
INSERT	Itext$	Inserts text immediately following the current location pointer; an ESCape terminates the text.
KILL	(–)nK 0K /K	Removes n lines from the Text Buffer beginning at the current location pointer.

A.2.6 Utility Commands

Command	Form	Meaning
EXECUTE MACRO	nEM	Executes the command string specified in the last macro command.
MACRO	M/command string/	Inserts a command string into the Macro Buffer.
	0M M//	Clears the Macro Buffer and reclaims the area for text.
SAVE	nS	Copies the specified number of lines, beginning at the pointer, into the Save Buffer.
UNSAVE	U	Inserts the entire contents of the Save Buffer into the Text Buffer at the position of the current location pointer.
EDIT VERSION	EV	Displays the version number of the Editor on the terminal.
EDIT LOWER	EL	Enables editing in upper- and lower-case.
EDIT UPPER	EU	Returns editing to upper-case only (after EL).

A.2.7 Key Commands

Command	Meaning
ESCape	Echoes $. A single ESCape terminates a text string. A double ESCape executes the command string.
CTRL C	Echoes at the terminal at ↑C and a carriage return. Terminates execution of EDIT commands and returns to monitor command mode.
CTRL O	Echoes ↑O and a carriage return. Inhibits printing on the terminal until completion of the current command string. Typing a second CTRL O resumes output.
CTRL U	Echoes ↑U and a carriage return. Deletes all the characters on the current terminal input line.
RUBOUT	Deletes character from the current line.
TAB	Spaces to the next tab stop. Tab stops are positioned every eight spaces on the terminal.
CTRL X	Echoes ↑X and a carriage return. CTRL X causes the Editor to ignore the entire command string currently being entered. The Editor prints a <CR><LF> and an asterisk to indicate that the user may enter another command.

A.3 PIP (Chapter 4)

A.3.1 Switch Summary

Switch	Explanation
/A	Copies file(s) in ASCII mode; ignores nulls and rubouts; converts to 7-bit ASCII.
/B	Copies files in formatted binary mode.
/C	Used in conjunction with another switch; causes only files with current date to be included in the specified operation.
/D	Deletes file(s) from specified device.
/E	Lists the device directory including unused spaces and their sizes.
/F	Prints a short directory (filenames 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).
/K	Scans the specified device and types the absolute block numbers (in octal) of any bad blocks on the device.
/L	Lists the directory of the specified device.
/N:n	Used with /Z to specify the number of directory blocks (n) to allocate to the directory.
/O	Bootstraps the specified device (DXn).
/Q	Causes PIP to type each filename which is eligible for a wild card operation and to ask for a confirmation of its inclusion in the operation.
/R	Renames the specified file.
/S	Compresses the files on the specified directory device so that free blocks are combined into one area.
/T	Extends number of blocks allocated for a file.
/U	Copies the bootstrap from the specified file into absolute blocks 0 and 2 of the specified device.
/V	Types the version number of the PIP program being used.
/W	Includes the absolute starting block and any extra directory words in the directory listing for each file on the device (numbers in octal). Used with /F, /L, or /E.

Switch	Explanation
/X	Copies files individually (without concatenation).
/Y	Causes system files and .BAD files to be operated on by the command specified.
/Z:n	Zeroes (initializes) the directory of the specified device; n is used to allocate extra words per directory entry. When used with /N, the number of directory segments for entries may be specified.

A.4 ASEMBL/CREF (Chapter 5)

Refer to Appendix B for a complete summary of ASEMBL features. CREF switches are also included in that appendix.

A.5 LINKER (Chapter 6)

A.5.1 Switch Summary

The Linker switches (and the command line on which each must appear) are:

Switch Name	Command Line	Meaning
/A	1st	Alphabetizes the entries in the load map.
/B:n	1st	Bottom address of program is indicated as n.
/C	any	Continues input files on another command line (must be used with /O).
/F	1st	Indicates that the Linker will use the default FORTRAN library, FORLIB.OBJ.
/I	1st	Includes the global symbols to be searched from the library.
/L	1st	Produces an output file in LDA format.
/M:n	1st	Allows terminal keyboard specification of the user's stack address. n represents an optional 6-digit unsigned octal number.
/O:n	any but the 1st	Indicates that the program will be an overlay structure; n specifies the overlay region to which the module is assigned.
/S	1st	Allows the maximum amount of space in memory to be available for the Linker's symbol table. (This switch should only be used when a particular link stream causes a symbol table overflow.)
/T or /T:n	1st	Transfer address is to be specified at terminal keyboard via n.

A.6 LIBRARIAN (Chapter 7)

A.6.1 Switch Summary

The Librarian (LIBR) switches (and the command line on which each must appear) are:

Switch	Command Line	Meaning
/C	Any	The command is too long for the current line and is continued on the next line.
/D	1st	Deletes modules from a library file.
/G	1st	Global deletion; deletes entry points from the library directory.
/R	1st	Replaces modules in a library file.
/U	1st	Update; inserts and replaces modules in a library file.

A.7 ODT (Chapter 8)

A.7.1 Command Summary

In the command format shown below, r represents a relocatable expression and n represents an octal number.

Command	Format	Explanation
RETURN		Closes open location and accepts the next command.
LINE FEED		Closes current location and opens next sequential location.
↑ or ^	↑ or ^	Opens previous location.
← or _	← or _	Indexes the contents of the opened location by the contents of the PC and opens the resulting location.
>	>	Uses the contents of the opened location as a relative branch instruction and opens the referenced location.
<	<	Returns to sequence prior to last @, >, or ← command and opens the succeeding location.
@	@	Uses the contents of the opened location as an absolute address and opens that location.
/	/	Reopens the last opened location.
	r/	Opens the word at location r.
\	\	Reopens the last opened byte.
	r\	Opens the byte at location r.

Command	Format	Explanation
!	! n!	After a word or byte has been opened, prints the address of the opened location relative to relocation register n. If n is omitted, ODT selects the relocation register whose contents are closest to but less than or equal to the address of the opened location.
$	$n/	Opens general register n (0-7).
	$B/	Opens the first word of the breakpoint table.
	$C/	Opens Constant Register.
	$F/	Opens Format Register.
	$P/	Opens Priority Register.
	$R/	Opens first Relocation Register (register 0).
	$S/	Opens Status Register.
A	r;nA	Starting at location r, prints n bytes in their ASCII format; then inputs n bytes from the terminal starting at location r.
B	;B	Removes all Breakpoints.
	r;B	Sets Breakpoint at location r.
	r;nB	Sets Breakpoint n at location r.
	;nB	Removes the nth Breakpoint.
C	r;C	Prints the value of r and stores it in the Constant Register.
E	r;E	Searches for instructions that reference effective address r.
F	;F	Fills memory words with contents of the Constant Register.
G	r;G	Goes to location r and starts program.
I	;I	Fills memory bytes with the low-order 8 bits of the Constant Register.
O	r;O	Calculates offset from currently open location to r.
P	;P	Proceeds with program execution from breakpoint. In single instruction mode only, executes next instruction.

Command	Format	Explanation
	k;P	Proceeds with program execution from breakpoint; stops after encountering the breakpoint k times. In single instruction mode only, executes next k instructions.
R	;R	Sets all Relocation Registers to −1 (highest address value).
	;nR	Sets Relocation Register n to −1.
	r;nR	Sets Relocation Register n to the value of r. If n is omitted, it is assumed to be 0.
	R	Selects the Relocation Register whose contents are closest to but less than or equal to contents of the opened location. Subtracts the contents of the register from the contents of the opened word and prints the result.
	nR	Subtracts the contents of the Relocation Register n from the contents of the opened word and prints the result.
S	;S	Disables single instruction mode; reenables breakpoints.
	;nS	Enables single instruction mode (n can have any value and is not significant); disables breakpoints.
W	r;W	Searches for words with bit patterns which match r.
X	X	Performs a Radix 50 unpack of the binary contents of the current opened word; then permits the storage of a new Radix 50 binary number in the same location.

A.8 PROGRAMMED REQUESTS (Chapter 9)

Appendix D summaries the programmed requests available under HT-11.

A.9 DUMP (Appendix E)

A.9.1 Switch Summary

Switch	Meaning
/B	Outputs octal bytes.
/E:n	Ends output at block n.
/G	Ignores input errors.
/N	Suppresses ASCII output.

Switch	Meaning
/O:n	Outputs only block number n.
/S:n	Starts output with block n.
/W	Outputs octal words.
/X	Outputs RAD50 characters.

A.10 SRCCOM (Appendix F)

A.10.1 Switch Summary

Switch	Meaning
/B	Compares blank lines. Without this switch, blank lines are ignored.
/C	Ignores comments (all text on a line preceded by a semi-colon) and spacing (spaces and tabs).
/F	Includes form feeds in the output file (form feeds are still compared if /F is not used, but they are not included in the output of differences).
/H	Types list of switches available (help text).
/L:n	Specifies the number of lines that determine a match (here n is an octal number <=310). The default value for n is 3.
/S	Ignores spaces and tabs.

A.11 PATCH (Appendix G)

A.11.1 Command Summary

Command	Meaning
/O	Indicates overlay-structured file.
/M	Indicates monitor file.
Vr;nR	Sets relocation register n to value Vr.
b;B	Sets bottom address of overlay file to b.
[s:] r,o/	Opens word location Vr + o in overlay segment s.
[s:] r,o\	Opens byte location Vr + o in overlay segment s.

Command	Meaning
⟨CR⟩	Closes currently open word/byte.
⟨LF⟩	Closes currently open word/byte and opens the next one.
↑ or ^	Closes the currently open word/byte and opens the the previous one.
@	Closes the currently open word and opens the word addressed by it.
F	Begins patching a new file.
E	Exits to HT-11 monitor.

APPENDIX B
ASSEMBLER, INSTRUCTION, AND CHARACTER CODE SUMMARIES

B.1 ASCII CHARACTER SET

Even Parity Bit	7-Bit Octal Code	Character	Remarks
0	000	NUL	Null, Tape Feed, CTRL SHIFT P.
1	001	SOH	Start of Heading; also SOM (Start of Message), CTRL A.
1	002	STX	Start of Text; also EOA (End of Address), CTRL B.
0	003	ETX	End of Text; also EOM (End of Message), CTRL C.
1	004	EOT	End of Transmission (END); Shuts off TWX machines, CTRL D.
0	005	ENQ	Enquiry (ENQRY); also WRU, CTRL E.
0	006	ACK	Acknowledge; also RU, CTRL F.
1	007	BEL	Rings the Bell. CTRL G.
1	010	BS	Backspace; also FEO, Format Effector. Backspaces some machines, CTRL H.
0	011	HT	Horizontal TAB. CTRL I.
0	012	LF	Line Feed or Line Space (New Line); Advances paper to next line; duplicated by CTRL J.
1	013	VT	Vertical TAB (VTAB). CTRL K.
0	014	FF	FORM FEED to top of next page (PAGE). CTRL L.
1	015	CR	Carriage Return to beginning of line. Duplicated by CTRL M.
1	016	SO	Shift Out; Changes ribbon color to red. CTRL N.
0	017	SI	Shift In; Changes ribbon color to black. CTRL O.
1	020	DLE	Data Link Escape. CTRL P (DC0).
0	021	DC1	Device Control 1, turns transmitter (reader) on, CTRL Q (X ON).
0	022	DC2	Device Control 2, turns punch or auxiliary on, CTRL R (TAPE, AUX ON).
1	023	DC3	Device Control 3, turns transmitter (reader) off, CTRL S (X OFF).
0	024	DC4	Device Control 4, turns punch or auxiliary off, CTRL T (AUX OFF).
1	025	NAK	Negative Acknowledge; also ERR, Error, CTRL U.
1	026	SYN	Synchronous Idle (SYNC), CTRL V.
0	027	ETB	End of Transmission Block; also LEM, Logical End of Medium, CTRL W.
0	030	CAN	Cancel (CANCL), CTRL X.
1	031	EM	End of Medium, CTRL Y.
1	032	SUB	Substitute, CTRL Z.
0	033	ESC	Escape, CTRL SHIFT K.
1	034	FS	File Separator, CTRL SHIFT L.

Even Parity Bit	7-Bit Octal Code	Character	Remarks
0	035	GS	Group Separator, CTRL SHIFT M.
0	036	RS	Record Separator, CTRL SHIFT N.
1	037	US	Unit Separator, CTRL SHIFT O.
1	040	SP	Space.
0	041	!	
0	042	”	
1	043	#	
0	044	$	
1	045	%	
1	046	&	
0	047	’	Apostrophe or Acute Accent.
0	050	(	
1	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	

Even Parity Bit	7-Bit Octal Code	Character	Remarks
0	116	N	
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 ALTMODE.
0	176	~	This Code Generated by PREFIX key (if Present)
1	177	DEL	DELETE, RUBOUT.

B.2 RADIX-50 CHARACTER SET

Character	ASCII Octal Equivalent	Radix-50 Equivalent
space	40	0
A–Z	101–32	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
```

Single Character or First Character		Second Character		Third Character	
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

Single Character or First Character		Second Character		Third Character	
$	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

B.3 ASSEMBLER SPECIAL CHARACTERS

Character	Function
form feed	Source line terminator, forces a new listing page
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
, (comma)	Operand field separator
;	Comment field indicator
+	Arithmetic addition operator or autoincrement indicator
—	Arithmetic subtraction operator or autodecrement 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

B.4 ADDRESS MODE SYNTAX

In the following syntax table, n represents an integer between 0 and 7; R is a register expression; E represents any expression; ER represents either a register expression or an absolute expression in the range 0 to 7.

0n	Register	R	Register R contains the operand. R is a register expression.
1n	Deferred Register	@R or (R)	Register R contains the operand address.
2n	Autoincrement	(ER)+	The contents of the register specified by ER are incremented after being used as the address of the operand.
3n	Deferred Autoincrement	@(ER)+	ER contains a pointer to the address of the operand. ER is incremented after use.
4n	Autodecrement	–(ER)	The contents of register ER are decremented before being used as the address of the operand.
5n	Deferred Autodecrement	@–(ER)	The contents of register ER are decremented before being used as a pointer to the address of the operand.
6n	Index by the Register Specified	E(ER)	The value obtained when E is added to the contents of the register specified (ER) is the address of the operand.
7n	Deferred Index by the Register Specified	@E(ER)	E added to ER produces a pointer to the address of the operand.
27	Immediate Operand	#E	E is the operand.
37	Absolute address	@#E	E is the operand address.
67	Relative address	E	E is the address of the operand.
77	Deferred Relative address	@E	E is a pointer to the address of the operand.

B.5 INSTRUCTIONS

The tables of instructions which follow are grouped according to the operands they take and according to the bit patterns of their op-codes.

The following symbols are used to indicate the instruction type format:

OP	Instruction mnemonic
R	Register expression
E	Expression
ER	Register expression or expression 0<=ER<=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	Represents 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 Addresses
| |	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	Negative bit	Set if the result is negative
Z	Zero bit	Set if the result is zero
V	Overflow bit	Set if the operation caused an overflow
C	Carry 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.

Logical operators are represented by the following symbols:

!	Inclusive OR
(!)	Exclusive OR
&	AND
—	Used over a symbol to represent the 1's complement of the symbol

B.5.1 Double Operand Instructions (OP A,A)

Op-Code	Mnemonic	Stands for	Operation	Status Word Condition Codes N	Z	V	C
01SSDD	MOV	MOVe	(SE) → (DE)	*	*	0	–
11SSDD	MOVB	MOVe Byte					
02SSDD	CMP	CoMPare	(SE) – (DE)	*	*	*	*
12SSDD	CMPB	CoMPare Byte					
03SSDD	BIT	BIt Test	(SE) & (DE)	*	*	0	–
13SSDD	BITB	BIt Test Byte					
04SSDD	BIC	BIt Clear	$\overline{(SE)}$ & (DE) → (DE)	*	*	0	–
14SSDD	BICB	BIt Clear Byte					
05SSDD	BIS	BIt Set	(SE) ! (DE) → (DE)	*	*	0	–
15SSDD	BISB	BIt Set Byte					
06SSDD	ADD	ADD	(SE) + (DE) → (DE)	*	*	*	*
16SSDD	SUB	SUBtract	(DE) – (SE) → (DE)	*	*	*	*

B.5.2 Single Operand Instructions (OP A)

Op-Code	Mnemonic	Stands for	Operation	Status Word Condition Codes 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 INCB	INCrement INCrement Byte	(DE) + 1 → (DE)	*	*	*	–
0053DD 1053DD	DEC DECB	DECrement DECrement Byte	(DE) – 1 → (DE)	*	*	*	–
0054DD 1054DD	NEG NEGB	NEGate NEGate Byte	0 – (DE) → (DE)	*	*	*	*
0055DD 1055DD	ADC ADCB	ADd Carry ADd Carry Byte	(DE) + (C) → (DE)	*	*	*	*
0056DD 1056DD	SBC SBCB	SuBtract Carry SuBtract Carry Byte	(DE) – (C) → (DE)	*	*	*	*
0057DD 1057DD	TST TSTB	TeST TeST Byte	(DE)	*	*	0	0

B.5.3 Rotate/Shift

Op-Code	Mnemonic	Stands for	Operation	Status Word Condition Codes N	Z	V	C
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

PDP-11/03 (LSI/11) only:

Op-Code	Mnemonic	Stands for	Operation	Status Word Condition Codes N	Z	V	C
1067DD	MFPS	Move byte From Processor Status word	(DE) ← PSW	*	*	0	–
1064SS	MTPS	Move byte To Processor Status word	(SE) → PSW	*	*	*	*

Op-Code	Mnemonic	Stands for	Operation	Status Word Condition Codes N	Z	V	C
Machines with KT11 Memory Management only:							
0065SS	MFPI	Move From Previous Instruction space	(SE) → (TEMP) (SP) –2 → (SP) (TEMP) → ((SP))	*	*	0	–
1065SS	MFPD	Move From Previous Data space	(SE) → (TEMP) (SP) –2 → (SP) (TEMP) → ((SP))	*	*	0	–
0066DD	MTPI	Move To Previous Instruction space	((SP)) → (TEMP) (SP+2) → (SP) (TEMP) → (DE)	*	*	0	–
1066DD	MTPD	Move To Previous Data space	((SP)) → (TEMP) (SP+2) → (SP) (TEMP) → (DE)	*	*	0	–
Not on 11/04, 11/05, 11/20:							
0067DD	SXT	Sign eXTend	0 → DE if N bit is clear –1 → DE if N bit is set	–	*	0	–
Machines with FP11-B Floating Point only:							
0707DD	NEGD	NEGate Double	–(FDE) → FDE	*	*	0	0
0707DD	NEGF	NEGate Floating	–(FDE) → FDE	*	*	0	0
1701DD	LDFPS	Load FPP program status	DE → FPS	–	–	–	–
1702DD	STFPS	STore Floating Point processor program Status		–	–	–	–

Op-Code	Mnemonic	Stands for	Operation	Status Word Condition Codes N	Z	V	C
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	0
1705DD	TSTF	TeST Floating	(FDE)	*	*	0	0
1706DD	ABSD	make ABSolute Double	|(FDE)| → (FDE)	0	*	0	0
1706DD	ABSF	make ABSolute Floating	|(FDE)| → (FDE)	0	*	0	0

B.5.4 Operate Instructions (OP)

Op-Code	Mnemonic	Stands for	Operation	Status Word Condition Codes N	Z	V	C
000000	HALT	HALT	The computer stops all functions.	–	–	–	–
000001	WAIT	WAIT	The computer stops and waits for an interrupt.	–	–	–	–
000002	RTI	ReTurn from Interrupt (Return from Trap)	The PC and PS are popped off the SP stack: ((SP)) → (PC) (SP) +2 → (SP) ((SP)) → (PS) (SP) +2 → (SP)	*	*	*	*
000005	RESET	RESET	Returns all I/O devices to power-on state.	–	–	–	–
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	–	–	–
000257	CCC	Clear all Condition Codes	0 → N 0 → Z 0 → V 0 → C	0	0	0	0

Op-Code	Mnemonic	Stands for	Operation	Status Word Condition Codes N Z V C
000277	SCC	Set all Condition Codes	1 → N 1 → Z 1 → V 1 → C	1 1 1 1
000240	NOP	No OPeration		– – – –
Machines with FP11-B only:				
170000	CFCC	Copy Floating Condition Codes	Copy FPP condition codes into CPU condition codes.	* * * *
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 integer mode	FPP set for long integer data (32 bits)	– – – –
Not on 11/04, 11/05, 11/20:				
000006	RTT	ReTurn from inTerrupt	Same as RTI instruction but inhibits trace trap	* * * *

B.5.5 Trap Instructions (OP or OP E where 0<=E<=377(8))

Op-Code	Mnemonic	Stands for	Operation	Status Word Condition Codes N Z V C
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 location 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.	* * * *

B.5.6 Branch Instructions (OP E)

(where $-128_{10} < (E-.-2)/2 < 127_{10}$)

Op-Code	Mnemonic	Stands for	Condition To Be Met if Branch is To Occur
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)	$N \textcircled{!} V=0$
0024XX	BLT	Branch if Less Than (zero)	$N \textcircled{!} V=1$
0030XX	BGT	Branch if Greater Than (zero)	$Z ! (N \textcircled{!} V)=0$
0034XX	BLE	Branch if Less than or Equal (to zero)	$Z ! (N \textcircled{!} V)=1$
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 HIgh or Same)	C=0
1034XX	BCS (or BLO)	Branch if Carry Set (or Branch if LOw)	C=1

B.5.7 Register Destination (OP ER,A)

Op-Code	Mnemonic	Stands for	Operation	Status Word Condition Codes 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	– – – –

Op-Code	Mnemonic	Stands for	Operation	Status Word Condition Codes N	Z	V	C
Not on 11/04, 11/05, 11/20:							
074RDD	XOR	eXclusive OR	(R) (!) (DE) → (DE)	*	*	0	–

B.5.8 Register-Offset (OP R,E)

Not on 11/04, 11/05, 11/20:

Op-Code	Mnemonic	Stands for	Operation	N	Z	V	C
077RDD	SOB	Subtract One and Branch	(R) –1 → (R) (PC) – (2*DE) → (PC)	–	–	–	–

B.5.9 Subroutine Return (OP ER)

Op-Code	Mnemonic	Stands for	Operation	N	Z	V	C
00020R	RTS	ReTurn from Subroutine	Put register in PC and pop old contents from SP stack into register.	–	–	–	–

B.5.10 Source-Register (OP A,R)

Only on machines with EIS option:

Op-Code	Mnemonic	Stands for	Operation	Status Word Condition Codes N	Z	V	C
071RSS	DIV	DIVide	(R),(R!1)/(SRC) → (R),(R!1)	*	*	*	*
070RSS	MUL	MULtiply	(R)*(SRC) → (R),(R!1)	*	*	*	*
072RSS	ASH	Arithmetic SHift	R is shifted according to low-order 6-bits of source or				
073RSS	ASHC	Arithmetic SHift Combined	R,R!1 are shifted according to low-order 6-bits of source or	*	*	*	*

B.5.11 Floating-Point Source Double Register (OP A,AC)

Machines with FP11-B only:

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 Convert from Double to Floating	(FSE) → (AC)	*	*	*	0
177(AC+4)SS	LDCFD	LoaD and Convert 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)	*	*	*	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
173(AC)SS	SUBF	SUBtract Floating	(FSE)–(AC) → (AC)	*	*	*	0

B.5.12 Source-Double Register (OP A,AC)

Machines with FP11-B only:

Op-Code	Mnemonic	Stands for	Operation	Status Word Condition Codes FN	FZ	FV	FC
177(AC)SS	LDCID	LoaD and Convert Integer to Double	(SE) → (AC)	*	*	*	0
177(AC)SS	LDCIF	LoaD and Convert Integer to Floating	(SE) → (AC)	*	*	*	0

Op-Code	Mnemonic	Stands for	Operation	Status Word Condition Codes FN	FZ	FV	FC
177(AC)SS	LDCLD	LoaD and Convert Long integer to Double	(SE) → (AC)	*	*	*	0
177(AC)SS	LDCLF	LoaD and Convert Long integer to Floating	(SE) → (AC)	*	*	*	0
176(AC+4)SS	LDEXP	LoaD EXPonent	(SE)+200 → (AC EXP)	*	*	0	0

B.5.13 Double Register-Destination (OP AC,A)

Machines with FP11-B only:

Op-Code	Mnemonic	Stands for	Operation	Status Word Condition Codes FN	FZ	FV	FC
176(AC)DD	STCFD	STore, Convert from Floating to Double	(AC) → (FDE)	*	*	*	0
176(AC)DD	STCDF	STore, Convert from Double to Floating	(AC) → (FDE)	*	*	*	0
175(AC+4)DD	STCDI[1]	STore, Convert from Double to Integer	(AC) → (FDE)	*	*	0	*
175(AC+4)DD	STCDL[1]	STore, Convert from Double to Long integer	(AC) → (FDE)	*	*	0	*
175(AC+4)DD	STCFI[1]	STore, Convert from Floating to Integer	(AC) → (FDE)	*	*	0	*
174(AC+4)DD	STCFL[1]	STore, Convert 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[1]	STore EXPonent	(AC EXP)–200 → (DE)	*	*	0	0

[1] These instruction set both the floating-point and processor condition codes as indicated.

B.5.14 Number

Not on 11/04, 11/05, 11/20:

Op-Code	Mnemonic	Stands for	Operation	Status Word Condition Codes N	Z	V	C
0064NN	MARK	MARK	Stack cleanup on return from subroutine.	–	–	–	–

B.5.15 Priority

The following instruction is available on the PDP-11/45 only:

Op-Code	Mnemonic	Stands for	Operation	Status Word Condition Codes N	Z	V	C
00023N	SPL	Set Priority Level	(X) → (PS) (bits 7–5)	–	–	–	–

B.5.16 Stack Oriented Floating Point (OP R)

Only on machines with FIS option:

Op-Code	Mnemonic	Stands for	Operation	Status Word Condition Codes N	Z	V	C
07500R	FADD	Floating ADD	(4(R))+((R)) → ((R))	*	*	0	0
07501R	FSUB	Floating SUBtract	(4(R))–((R)) → ((R))	*	*	0	0
07502R	FMUL	Floating MULtiply	(4(R))*((R)) → ((R))	*	*	0	0
07503R	FDIV	Floating DIVide	(4(R))÷((R)) → ((R))	*	*	0	0

B.6 MACRO DIRECTIVES

Form	Described in Manual Section	Operation
.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.
.MACRO sym,argl, arg2,...	5.6.1.1	Indicates the start of a macro with the specified name containing the dummy arguments specified.
.MCALL	5.6.4	Used to specify the names of all system macro definitions not defined in the current program but required by the program.

B.7 ASSEMBLER DIRECTIVES

Form	Described in Manual Section	Operation
'	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.
↑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.
.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 expl, exp2,...	5.5.3.1	Generates successive bytes of data containing the octal equivalent of the expression(s) specified.
.CSECT symbol	5.5.9	Begins or resumes 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.

Form	Described in Manual Section	Operation
.ENDC	5.5.11	Indicates the end of a condition block.
.EOT	5.5.7.2	Ignored. Indicates End-of-Tape which is detected automatically by the hardware.
.EVEN	5.5.5.1	Ensures that the assembly location counter contains an even address by adding 1 if it is odd.
.FLT2 argl, arg2,...	5.5.6.1	Generates successive two-word floating point equivalents for the floating-point numbers specified as arguments.
.FLT4 argl, arg2,...	5.5.6.1	Generates successive four-word floating point equivalents for the floating-point numbers specified as arguments.
.GLOBL syml, sym2,...	5.5.10	Defines the symbol(s) specified as global symbol(s).
.IDENT symbol	5.5.1.5	Provides a means of labeling the object module produced as a result of assembly.
.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.
.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.
.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.

Form	Described in Manual Section	Operation
.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.
.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).
.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.
.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 expl, exp2,...	5.5.3.2	Generates successive words of data containing the octal equivalent of the expression(s) specified.

B.8 ASEMBL/CREF SWITCHES

B.8.1 Listing Control Switches

Switch	Meaning
/L:arg /N:arg	These switches are used to control listing output.

Arg	Controls Listing of:
SEQ	Source line sequence numbers
LOC	Location counter
BIN	Generated binary code
BEX	Binary extensions
SRC	Source code
COM	Comments
CND	Unsatisfied conditons and all .IF and .ENDC statements.
LD	Listing directives having no arguments
TOC	Table of contents
TTM	Listing output format
SYM	Symbol table
<no arg>	/N with no argument causes only table of contents, symbol table, and error listings to be produced. /L with no argument causes .LIST and .NLIST directives without arguments which appear in the source program to be ignored.

B.8.2 Function Control Switches

Switch	Meaning
/D:arg /E:arg	These switches are used to enable or disable certain functions in source input files.

Arg	Enables or Disables:
ABS	Absolute binary output
AMA	Assembly of all absolute addresses as relative addresses
CDR	Source columns 73 and greater to be treated as comments
FPT	Floating point truncation
LC	Accepts lower case ASCII input
LSB	Local symbol block

B.8.3 CREF Switches

Switch	Produces Cross-Reference of:
/C:S	User-defined symbols
/C:R	Register symbols
/C:P	Permanent symbols
/C:C	Control sections
/C:E	Error codes
/C ⟨no arg⟩	Equivalent to /C:S:E

B.9 OCTAL-DECIMAL CONVERSIONS

	0	1	2	3	4	5	6	7
0000	0000	0001	0002	0003	0004	0005	0006	0007
0010	0008	0009	0010	0011	0012	0013	0014	0015
0020	0016	0017	0018	0019	0020	0021	0022	0023
0030	0024	0025	0026	0027	0028	0029	0030	0031
0040	0032	0033	0034	0035	0036	0037	0038	0039
0050	0040	0041	0042	0043	0044	0045	0046	0047
0060	0048	0049	0050	0051	0052	0053	0054	0055
0070	0056	0057	0058	0059	0060	0061	0062	0063
0100	0064	0065	0066	0067	0068	0069	0070	0071
0110	0072	0073	0074	0075	0076	0077	0078	0079
0120	0080	0081	0082	0083	0084	0085	0086	0087
0130	0088	0089	0090	0091	0092	0093	0094	0095
0140	0096	0097	0098	0099	0100	0101	0102	0103
0150	0104	0105	0106	0107	0108	0109	0110	0111
0160	0112	0113	0114	0115	0116	0117	0118	0119
0170	0120	0121	0122	0123	0124	0125	0126	0127
0200	0128	0129	0130	0131	0132	0133	0134	0135
0210	0136	0137	0138	0139	0140	0141	0142	0143
0220	0144	0145	0146	0147	0148	0149	0150	0151
0230	0152	0153	0154	0155	0156	0157	0158	0159
0240	0160	0161	0162	0163	0164	0165	0166	0167
0250	0168	0169	0170	0171	0172	0173	0174	0175
0260	0176	0177	0178	0179	0180	0181	0182	0183
0270	0184	0185	0186	0187	0188	0189	0190	0191
0300	0192	0193	0194	0195	0196	0197	0198	0199
0310	0200	0201	0202	0203	0204	0205	0206	0207
0320	0208	0209	0210	0211	0212	0213	0214	0215
0330	0216	0217	0218	0219	0220	0221	0222	0223
0340	0224	0225	0226	0227	0228	0229	0230	0231
0350	0232	0233	0234	0235	0236	0237	0238	0239
0360	0240	0241	0242	0243	0244	0245	0246	0247
0370	0248	0249	0250	0251	0252	0253	0254	0255

	0	1	2	3	4	5	6	7
0400	0256	0257	0258	0259	0260	0261	0262	0263
0410	0264	0265	0266	0267	0268	0269	0270	0271
0420	0272	0273	0274	0275	0276	0277	0278	0279
0430	0280	0281	0282	0283	0284	0285	0286	0287
0440	0288	0289	0290	0291	0292	0293	0294	0295
0450	0296	0297	0298	0299	0300	0301	0302	0303
0460	0304	0305	0306	0307	0308	0309	0310	0311
0470	0312	0313	0314	0315	0316	0317	0318	0319
0500	0320	0321	0322	0323	0324	0325	0326	0327
0510	0328	0329	0330	0331	0332	0333	0334	0335
0520	0336	0337	0338	0339	0340	0341	0342	0343
0530	0344	0345	0346	0347	0348	0349	0350	0351
0540	0352	0353	0354	0355	0356	0357	0358	0359
0550	0360	0361	0362	0363	0364	0365	0366	0367
0560	0368	0369	0370	0371	0372	0373	0374	0375
0570	0376	0377	0378	0379	0380	0381	0382	0383
0600	0384	0385	0386	0387	0388	0389	0390	0391
0610	0392	0393	0394	0395	0396	0397	0398	0399
0620	0400	0401	0402	0403	0404	0405	0406	0407
0630	0408	0409	0410	0411	0412	0413	0414	0415
0640	0416	0417	0418	0419	0420	0421	0422	0423
0650	0424	0425	0426	0427	0428	0429	0430	0431
0660	0432	0433	0434	0435	0436	0437	0438	0439
0670	0440	0441	0442	0443	0444	0445	0446	0447
0700	0448	0449	0450	0451	0452	0453	0454	0455
0710	0456	0457	0458	0459	0460	0461	0462	0463
0720	0464	0465	0466	0467	0468	0469	0470	0471
0730	0472	0473	0474	0475	0476	0477	0478	0479
0740	0480	0481	0482	0483	0484	0485	0486	0487
0750	0488	0489	0490	0491	0492	0493	0494	0495
0760	0496	0497	0498	0499	0500	0501	0502	0503
0770	0504	0505	0506	0507	0508	0509	0510	0511

0000 to 0777 (Octal)	0000 to 0511 (Decimal)

Octal	Decimal
10000	4096
20000	8192
30000	12288
40000	16384
50000	20480
60000	24576
70000	28672

	0	1	2	3	4	5	6	7
1000	0512	0513	0514	0515	0516	0517	0518	0519
1010	0520	0521	0522	0523	0524	0525	0526	0527
1020	0528	0529	0530	0531	0532	0533	0534	0535
1030	0536	0537	0538	0539	0540	0541	0542	0543
1040	0544	0545	0546	0547	0548	0549	0550	0551
1050	0552	0553	0554	0555	0556	0557	0558	0559
1060	0560	0561	0562	0563	0564	0565	0566	0567
1070	0568	0569	0570	0571	0572	0573	0574	0575
1100	0576	0577	0578	0579	0580	0581	0582	0583
1110	0584	0585	0586	0587	0588	0589	0590	0591
1120	0592	0593	0594	0595	0596	0597	0598	0599
1130	0600	0601	0602	0603	0604	0605	0606	0607
1140	0608	0609	0610	0611	0612	0613	0614	0615
1150	0616	0617	0618	0619	0620	0621	0622	0623
1160	0624	0625	0626	0627	0628	0629	0630	0631
1170	0632	0633	0634	0635	0636	0637	0638	0639
1200	0640	0641	0642	0643	0644	0645	0646	0647
1210	0648	0649	0650	0651	0652	0653	0654	0655
1220	0656	0657	0658	0659	0660	0661	0662	0663
1230	0664	0665	0666	0667	0668	0669	0670	0671
1240	0672	0673	0674	0675	0676	0677	0678	0679
1250	0680	0681	0682	0683	0684	0685	0686	0687
1260	0688	0689	0690	0691	0692	0693	0694	0695
1270	0696	0697	0698	0699	0700	0701	0702	0703
1300	0704	0705	0706	0707	0708	0709	0710	0711
1310	0712	0713	0714	0715	0716	0717	0718	0719
1320	0720	0721	0722	0723	0724	0725	0726	0727
1330	0728	0729	0730	0731	0732	0733	0734	0735
1340	0736	0737	0738	0739	0740	0741	0742	0743
1350	0744	0745	0746	0747	0748	0749	0750	0751
1360	0752	0753	0754	0755	0756	0757	0758	0759
1370	0760	0761	0762	0763	0764	0765	0766	0767

	0	1	2	3	4	5	6	7
1400	0768	0769	0770	0771	0772	0773	0774	0775
1410	0776	0777	0778	0779	0780	0781	0782	0783
1420	0784	0785	0786	0787	0788	0789	0790	0791
1430	0792	0793	0794	0795	0796	0797	0798	0799
1440	0800	0801	0802	0803	0804	0805	0806	0807
1450	0808	0809	0810	0811	0812	0813	0814	0815
1460	0816	0817	0818	0819	0820	0821	0822	0823
1470	0824	0825	0826	0827	0828	0829	0830	0831
1500	0832	0833	0834	0835	0836	0837	0838	0839
1510	0840	0841	0842	0843	0844	0845	0846	0847
1520	0848	0849	0850	0851	0852	0853	0854	0855
1530	0856	0857	0858	0859	0860	0861	0862	0863
1540	0864	0865	0866	0867	0868	0869	0870	0871
1550	0872	0873	0874	0875	0876	0877	0878	0879
1560	0880	0881	0882	0883	0884	0885	0886	0887
1570	0888	0889	0890	0891	0892	0893	0894	0895
1600	0896	0897	0898	0899	0900	0901	0902	0903
1610	0904	0905	0906	0907	0908	0909	0910	0911
1620	0912	0913	0914	0915	0916	0917	0918	0919
1630	0920	0921	0922	0923	0924	0925	0926	0927
1640	0928	0929	0930	0931	0932	0933	0934	0935
1650	0936	0937	0938	0939	0940	0941	0942	0943
1660	0944	0945	0946	0947	0948	0949	0950	0951
1670	0952	0953	0954	0955	0956	0957	0958	0959
1700	0960	0961	0962	0963	0964	0965	0966	0967
1710	0968	0969	0970	0971	0972	0973	0974	0975
1720	0976	0977	0978	0979	0980	0981	0982	0983
1730	0984	0985	0986	0987	0988	0989	0990	0991
1740	0992	0993	0994	0995	0996	0997	0998	0999
1750	1000	1001	1002	1003	1004	1005	1006	1007
1760	1008	1009	1010	1011	1012	1013	1014	1015
1770	1016	1017	1018	1019	1020	1021	1022	1023

1000 to 1777 (Octal)	0512 to 1023 (Decimal)

2000 to 2777 (Octal)	1024 to 1535 (Decimal)

Octal	Decimal
10000	4096
20000	8192
30000	12288
40000	16384
50000	20480
60000	24576
70000	28672

	0	1	2	3	4	5	6	7
2000	1024	1025	1026	1027	1028	1029	1030	1031
2010	1032	1033	1034	1035	1036	1037	1038	1039
2020	1040	1041	1042	1043	1044	1045	1046	1047
2030	1048	1049	1050	1051	1052	1053	1054	1055
2040	1056	1057	1058	1059	1060	1061	1062	1063
2050	1064	1065	1066	1067	1068	1069	1070	1071
2060	1072	1073	1074	1075	1076	1077	1078	1079
2070	1080	1081	1082	1083	1084	1085	1086	1087
2100	1088	1089	1090	1091	1092	1093	1094	1095
2110	1096	1097	1098	1099	1100	1101	1102	1103
2120	1104	1105	1106	1107	1108	1109	1110	1111
2130	1112	1113	1114	1115	1116	1117	1118	1119
2140	1120	1121	1122	1123	1124	1125	1126	1127
2150	1128	1129	1130	1131	1132	1133	1134	1135
2160	1136	1137	1138	1139	1140	1141	1142	1143
2170	1144	1145	1146	1147	1148	1149	1150	1151
2200	1152	1153	1154	1155	1156	1157	1158	1159
2210	1160	1161	1162	1163	1164	1165	1166	1167
2220	1168	1169	1170	1171	1172	1173	1174	1175
2230	1176	1177	1178	1179	1180	1181	1182	1183
2240	1184	1185	1186	1187	1188	1189	1190	1191
2250	1192	1193	1194	1195	1196	1197	1198	1199
2260	1200	1201	1202	1203	1204	1205	1206	1207
2270	1208	1209	1210	1211	1212	1213	1214	1215
2300	1216	1217	1218	1219	1220	1221	1222	1223
2310	1224	1225	1226	1227	1228	1229	1230	1231
2320	1232	1233	1234	1235	1236	1237	1238	1239
2330	1240	1241	1242	1243	1244	1245	1246	1247
2340	1248	1249	1250	1251	1252	1253	1254	1255
2350	1256	1257	1258	1259	1260	1261	1262	1263
2360	1264	1265	1266	1267	1268	1269	1270	1271
2370	1272	1273	1274	1275	1276	1277	1278	1279

	0	1	2	3	4	5	6	7
2400	1280	1281	1282	1283	1284	1285	1286	1287
2410	1288	1289	1290	1291	1292	1293	1294	1295
2420	1296	1297	1298	1299	1300	1301	1302	1303
2430	1304	1305	1306	1307	1308	1309	1310	1311
2440	1312	1313	1314	1315	1316	1317	1318	1319
2450	1320	1321	1322	1323	1324	1325	1326	1327
2460	1328	1329	1330	1331	1332	1333	1334	1335
2470	1336	1337	1338	1339	1340	1341	1342	1343
2500	1344	1345	1346	1347	1348	1349	1350	1351
2510	1352	1353	1354	1355	1356	1357	1358	1359
2520	1360	1361	1362	1363	1364	1365	1366	1367
2530	1368	1369	1370	1371	1372	1373	1374	1375
2540	1376	1377	1378	1379	1380	1381	1382	1383
2550	1384	1385	1386	1387	1388	1389	1390	1391
2560	1392	1393	1394	1395	1396	1397	1398	1399
2570	1400	1401	1402	1403	1404	1405	1406	1407
2600	1408	1409	1410	1411	1412	1413	1414	1415
2610	1416	1417	1418	1419	1420	1421	1422	1423
2620	1424	1425	1426	1427	1428	1429	1430	1431
2630	1432	1433	1434	1435	1436	1437	1438	1439
2640	1440	1441	1442	1443	1444	1445	1446	1447
2650	1448	1449	1450	1451	1452	1453	1454	1455
2660	1456	1457	1458	1459	1460	1461	1462	1463
2670	1464	1465	1466	1467	1468	1469	1470	1471
2700	1472	1473	1474	1475	1476	1477	1478	1479
2710	1480	1481	1482	1483	1484	1485	1486	1487
2720	1488	1489	1490	1491	1492	1493	1494	1495
2730	1496	1497	1498	1499	1500	1501	1502	1503
2740	1504	1505	1506	1507	1508	1509	1510	1511
2750	1512	1513	1514	1515	1516	1517	1518	1519
2760	1520	1521	1522	1523	1524	1525	1526	1527
2770	1528	1529	1530	1531	1532	1533	1534	1535

3000 to 3777 (Octal)	1536 to 2047 (Decimal)

	0	1	2	3	4	5	6	7
3000	1536	1537	1538	1539	1540	1541	5142	1543
3010	1544	1545	1546	1547	1548	1549	1550	1551
3020	1552	1553	1554	1555	1556	1557	1558	1559
3030	1560	1561	1562	1563	1564	1565	1566	1567
3040	1568	1569	1570	1571	1572	1573	1574	1575
3050	1576	1577	1578	1579	1580	1581	1582	1583
3060	1584	1585	1586	1587	1588	1589	1590	1591
3070	1592	1593	1594	1595	1596	1597	1598	1599
3100	1600	1601	1602	1603	1604	1605	1606	1607
3110	1608	1609	1610	1611	1612	1613	1614	1615
3120	1616	1617	1618	1619	1620	1621	1622	1623
3130	1624	1625	1626	1627	1628	1629	1630	1631
3140	1632	1633	1634	1635	1636	1637	1638	1639
3150	1640	1641	1642	1643	1644	1645	1646	1647
3160	1648	1649	1650	1651	1652	1653	1654	1655
3170	1656	1657	1658	1659	1660	1661	1662	1663
3200	1664	1665	1666	1667	1668	1669	1670	1671
3210	1672	1673	1674	1675	1676	1677	1678	1679
3220	1680	1681	1682	1683	1684	1685	1686	1687
3230	1688	1689	1690	1691	1692	1693	1694	1695
3240	1696	1697	1698	1699	1700	1701	1702	1703
3250	1704	1705	1706	1707	1708	1709	1710	1711
3260	1712	1713	1714	1715	1716	1717	1718	1719
3270	1720	1721	1722	1723	1724	1725	1726	1727
3300	1728	1729	1730	1731	1732	1733	1734	1735
3310	1736	1737	1738	1739	1740	1741	1742	1743
3320	1744	1745	1746	1747	1748	1749	1750	1751
3330	1752	1753	1754	1755	1756	1757	1758	1759
3340	1760	1761	1762	1763	1764	1765	1766	1767
3350	1768	1769	1770	1771	1772	1773	1774	1775
3360	1776	1777	1778	1779	1780	1781	1782	1783
3370	1784	1785	1786	1787	1788	1789	1790	1791

	0	1	2	3	4	5	6	7
3400	1792	1793	1794	1795	1796	1797	1798	1799
3410	1800	1801	1802	1803	1804	1805	1806	1807
3420	1808	1809	1810	1811	1812	1813	1814	1815
3430	1816	1817	1818	1819	1820	1821	1822	1823
3440	1824	1825	1826	1827	1828	1829	1830	1831
3450	1832	1833	1834	1835	1836	1837	1838	1839
3460	1840	1841	1842	1843	1844	1845	1846	1847
3470	1848	1849	1850	1851	1852	1853	1854	1855
3500	1856	1857	1858	1859	1860	1861	1862	1863
3510	1864	1865	1866	1867	1868	1869	1870	1871
3520	1872	1873	1874	1875	1876	1877	1878	1879
3530	1880	1881	1882	1883	1884	1885	1886	1887
3540	1888	1889	1890	1891	1892	1893	1894	1895
3550	1896	1897	1898	1899	1900	1901	1902	1903
3560	1904	1905	1906	1907	1908	1909	1910	1911
3570	1912	1913	1914	1915	1916	1917	1918	1919
3600	1920	1921	1922	1923	1924	1925	1926	1927
3610	1928	1929	1930	1931	1932	1933	1934	1935
3620	1936	1937	1938	1939	1940	1941	1942	1943
3630	1944	1945	1946	1947	1948	1949	1950	1951
3640	1952	1953	1954	1955	1956	1957	1958	1959
3650	1960	1961	1962	1963	1964	1965	1966	1967
3660	1968	1969	1970	1971	1972	1973	1974	1975
3670	1976	1977	1978	1979	1980	1981	1982	1983
3700	1984	1985	1986	1987	1988	1989	1990	1991
3710	1992	1993	1994	1995	1996	1997	1998	1999
3720	2000	2001	2002	2003	2004	2005	2006	2007
3730	2008	2009	2010	2011	2012	2013	2014	2015
3740	2016	2017	2018	2019	2020	2021	2022	2023
3750	2024	2025	2026	2027	2028	2029	2030	2031
3760	2032	2033	2034	2035	2036	2037	2038	2039
3770	2040	2041	2042	2043	2044	2045	2046	2047

	0	1	2	3	4	5	6	7
4000	2048	2049	2050	2051	2052	2053	2054	2055
4010	2056	2057	2058	2059	2060	2061	2062	2063
4020	2064	2065	2066	2067	2068	2069	2070	2071
4030	2072	2073	2074	2075	2076	2077	2078	2079
4040	2080	2081	2082	2083	2084	2085	2086	2087
4050	2088	2089	2090	2091	2092	2093	2094	2095
4060	2096	2097	2098	2099	2100	2101	2102	2103
4070	2104	2105	2106	2107	2108	2109	2110	2111
4100	2112	2113	2114	2115	2116	2117	2118	2119
4110	2120	2121	2122	2123	2124	2125	2126	2127
4120	2128	2129	2130	2131	2132	2133	2134	2135
4130	2136	2137	2138	2139	2140	2141	2142	2143
4140	2144	2145	2146	2147	2148	2149	2150	2151
4150	2152	2153	2154	2155	2156	2157	2158	2159
4160	2160	2161	2162	2163	2164	2165	2166	2167
4170	2168	2169	2170	2171	2172	2173	2174	2175
4200	2176	2177	2178	2179	2180	2181	2182	2183
4210	2184	2185	2186	2187	2188	2189	2190	2191
4220	2192	2193	2194	2195	2196	2197	2198	2199
4230	2200	2201	2202	2203	2204	2205	2206	2207
4240	2208	2209	2210	2211	2212	2213	2214	2215
4250	2216	2217	2218	2219	2220	2221	2222	2223
4260	2224	2225	2226	2227	2228	2229	2230	2231
4270	2232	2233	2234	2235	2236	2237	2238	2239
4300	2240	2241	2242	2243	2244	2245	2246	2247
4310	2248	2249	2250	2251	2252	2253	2254	2255
4320	2256	2257	2258	2259	2260	2261	2262	2263
4330	2264	2265	2266	2267	2268	2269	2270	2271
4340	2272	2273	2274	2275	2276	2277	2278	2279
4350	2280	2281	2282	2283	2284	2285	2286	2287
4360	2288	2289	2290	2291	2292	2293	2294	2295
4370	2296	2297	2298	2299	2300	2301	2302	2303

	0	1	2	3	4	5	6	7
4400	2304	2305	2306	2307	2308	2309	2310	2311
4410	2312	2313	2314	2315	2316	2317	2318	2319
4420	2320	2321	2322	2323	2324	2325	2326	2327
4430	2328	2329	2330	2331	2332	2333	2334	2335
4440	2336	2337	2338	2339	2340	2341	2342	2343
4450	2344	2345	2346	2347	2348	2349	2350	2351
4460	2352	2353	2354	2355	2356	2357	2358	2359
4470	2360	2361	2362	2363	2364	2365	2366	2367
4500	2368	2369	2370	2371	2372	2373	2374	2375
4510	2376	2377	2378	2379	2380	2381	2382	2383
4520	2384	2385	2386	2387	2388	2389	2390	2391
4530	2392	2393	2394	2395	2396	2397	2398	2399
4540	2400	2401	2402	2403	2404	2405	2406	2407
4550	2408	2409	2410	2411	2412	2413	2414	2415
4560	2416	2417	2418	2419	2420	2421	2422	2423
4570	2424	2425	2426	2427	2428	2429	2430	2431
4600	2432	2433	2434	2435	2436	2437	2438	2439
4610	2440	2441	2442	2443	2444	2445	2446	2447
4620	2448	2449	2450	2451	2452	2453	2454	2455
4630	2456	2457	2458	2459	2460	2461	2462	2463
4640	2464	2465	2466	2467	2468	2469	2470	2471
4650	2472	2473	2474	2475	2476	2477	2478	2479
4660	2480	2481	2482	2483	2484	2485	2486	2487
4670	2488	2489	2490	2491	2492	2493	2494	2495
4700	2496	2497	2498	2499	2500	2501	2502	2503
4710	2504	2505	2506	2507	2508	2509	2510	2511
4720	2512	2513	2514	2515	2516	2517	2518	2519
4730	2520	2521	2522	2523	2524	2525	2526	2527
4740	2528	2529	2530	2531	2532	2533	2534	2535
4750	2536	2537	2538	2539	2540	2541	2542	2543
4760	2544	2545	2546	2547	2548	2549	2550	2551
4770	2552	2553	2554	2555	2556	2557	2558	2559

4000 to 4777 (Octal) | 2048 to 2559 (Decimal)

Octal	Decimal
10000	4096
20000	8192
30000	12288
40000	16384
50000	20480
60000	24576
70000	28672

	0	1	2	3	4	5	6	7
5000	2560	2561	2562	2563	2564	2565	2566	2567
5010	2568	2569	2570	2571	2572	2573	2574	2575
5020	2576	2577	2578	2579	2580	2581	2582	2583
5030	2584	2585	2586	2587	2588	2589	2590	2591
5040	2592	2593	2594	2595	2596	2597	2598	2599
5050	2600	2601	2602	2603	2604	2605	2606	2607
5060	2608	2609	2610	2611	2612	2613	2614	2615
5070	2616	2617	2618	2619	2620	2621	2622	2623
5100	2624	2625	2626	2627	2628	2629	2630	2631
5110	2632	2633	2634	2635	2636	2637	2638	2639
5120	2640	2641	2642	2643	2644	2645	2646	2647
5130	2648	2649	2650	2651	2652	2653	2654	2655
5140	2656	2657	2658	2659	2660	2661	2662	2663
5150	2664	2665	2666	2667	2668	2669	2670	2671
5160	2672	2673	2674	2675	2676	2677	2678	2679
5170	2680	2681	2682	2683	2684	2685	2686	2687
5200	2688	2689	2690	2691	2692	2693	2694	2695
5210	2696	2697	2698	2699	2700	2701	2702	2703
5220	2704	2705	2706	2707	2708	2709	2710	2711
5230	2712	2713	2714	2715	2716	2717	2718	2719
5240	2720	2721	2722	2723	2724	2725	2726	2727
5250	2728	2729	2730	2731	2732	2733	2734	2735
5260	2736	2737	2738	2739	2740	2741	2742	2743
5270	2744	2745	2746	2747	2748	2749	2750	2751
5300	2752	2753	2754	2755	2756	2757	2758	2759
5310	2760	2761	2762	2763	2764	2765	2766	2767
5320	2768	2769	2770	2771	2772	2773	2774	2775
5330	2776	2777	2778	2779	2780	2781	2782	2783
5340	2784	2785	2786	2787	2788	2789	2790	2791
5350	2792	2793	2794	2795	2796	2797	2798	2799
5360	2800	2801	2802	2803	2804	2805	2806	2807
5370	2808	2809	2810	2811	2812	2813	2814	2815

	0	1	2	3	4	5	6	7
5400	2816	2817	2818	2819	2820	2821	2822	2823
5410	2824	2825	2826	2827	2828	2829	2830	2831
5420	2832	2833	2834	2835	2836	2837	2838	2839
5430	2840	2841	2842	2843	2844	2845	2846	2847
5440	2848	2849	2850	2851	2852	2853	2854	2855
5450	2856	2857	2858	2859	2860	2861	2862	2863
5460	2864	2865	2866	2867	2868	2869	2870	2871
5470	2872	2873	2874	2875	2876	2877	2878	2879
5500	2880	2881	2882	2883	2884	2885	2886	2887
5510	2888	2889	2890	2891	2892	2893	2894	2895
5520	2896	2897	2898	2899	2900	2901	2902	2903
5530	2904	2905	2906	2907	2908	2909	2910	2911
5540	2912	2913	2914	2915	2916	2917	2918	2919
5550	2920	2921	2922	2923	2924	2925	2926	2927
5560	2928	2929	2930	2931	2932	2933	2934	2935
5570	2936	2937	2938	2939	2940	2941	2942	2943
5600	2944	2945	2946	2947	2948	2949	2950	2951
5610	2952	2953	2954	2955	2956	2957	2958	2959
5620	2960	2961	2962	2963	2964	2965	2966	2967
5630	2968	2969	2970	2971	2972	2973	2974	2975
5640	2976	2977	2978	2979	2980	2981	2982	2983
5650	2984	2985	2986	2987	2988	2989	2990	2991
5660	2992	2993	2994	2995	2996	2997	2998	2999
5670	3000	3001	3002	3003	3004	3005	3006	3007
5700	3008	3009	3010	3011	3012	3013	3014	3015
5710	3016	3017	3018	3019	3020	3021	3022	3023
5720	3024	3025	3026	3027	3028	3029	3030	3031
5730	3032	3033	3034	3035	3036	3037	3038	3039
5740	3040	3041	3042	3043	3044	3045	3046	3047
5750	3048	3049	3050	3051	3052	3053	3054	3055
5760	3056	3057	3058	3059	3060	3061	3062	3063
5770	3064	3065	3066	3067	3068	3069	3070	3071

5000 to 5777 (Octal) | 2560 to 3071 (Decimal)

6000 to 6777 (Octal) | 3072 to 3583 (Decimal)

Octal	Decimal
10000	- 4096
20000	- 8192
30000	- 12288
40000	- 16384
50000	- 20480
60000	- 24576
70000	- 28672

	0	1	2	3	4	5	6	7
6000	3072	3073	3074	3075	3076	3077	3078	3079
6010	3080	3081	3082	3083	3084	3085	3086	3087
6020	3088	3089	3090	3091	3092	3093	3094	3095
6030	3096	3097	3098	3099	3100	3101	3102	3103
6040	3104	3105	3106	3107	3108	3109	3110	3111
6050	3112	3113	3114	3115	3116	3117	3118	3119
6060	3120	3121	3122	3123	3124	3125	3126	3127
6070	3128	3129	3130	3131	3132	3133	3134	3135
6100	3136	3137	3138	3139	3140	3141	3142	3143
6110	3144	3145	3146	3147	3148	3149	3150	3151
6120	3152	3153	3154	3155	3156	3157	3158	3159
6130	3160	3161	3162	3163	3164	3165	3166	3167
6140	3168	3169	3170	3171	3172	3173	3174	3175
6150	3176	3177	3178	3179	3180	3181	3182	3183
6160	3184	3185	3186	3187	3188	3189	3190	3191
6170	3192	3193	3194	3195	3196	3197	3198	3199
6200	3200	3201	3202	3203	3204	3205	3206	3207
6210	3208	3209	3210	3211	3212	3213	3214	3215
6220	3216	3217	3218	3219	3220	3221	3222	3223
6230	3224	3225	3226	3227	3228	3229	3230	3231
6240	3232	3233	3234	3235	3236	3237	3238	3239
6250	3240	3241	3242	3243	3244	3245	3246	3247
6260	3248	3249	3250	3251	3252	3253	3254	3255
6270	3256	3257	3258	3259	3260	3261	3262	3263
6300	3264	3265	3266	3267	3268	3269	3270	3271
6310	3272	3273	3274	3275	3276	3277	3278	3279
6320	3280	3281	3282	3283	3284	3285	3286	3287
6330	3288	3289	3290	3291	3292	3293	3294	3295
6340	3296	3297	3298	3299	3300	3301	3302	3303
6350	3304	3305	3306	3307	3308	3309	3310	3311
6360	3312	3313	3314	3315	3316	3317	3318	3319
6370	3320	3321	3322	3323	3324	3325	3326	3327

	0	1	2	3	4	5	6	7
6400	3328	3329	3330	3331	3332	3333	3334	3335
6410	3336	3337	3338	3339	3340	3341	3342	3343
6420	3344	3345	3346	3347	3348	3349	3350	3351
6430	3352	3353	3354	3355	3356	3357	3358	3359
6440	3360	3361	3362	3363	3364	3365	3366	3367
6450	3368	3369	3370	3371	3372	3373	3374	3375
6460	3376	3377	3378	3379	3380	3381	3382	3383
6470	3384	3385	3386	3387	3388	3389	3390	3391
6500	3392	3393	3394	3395	3396	3397	3398	3399
6510	3400	3401	3402	3403	3404	3405	3406	3407
6520	3408	3409	3410	3411	3412	3413	3414	3415
6530	3416	3417	3418	3419	3420	3421	3422	3423
6540	3424	3425	3426	3427	3428	3429	3430	3431
6550	3432	3433	3434	3435	3436	3437	3438	3439
6560	3440	3441	3442	3443	3444	3445	3446	3447
6570	3448	3449	3450	3451	3452	3453	3454	3455
6600	3456	3457	3458	3459	3460	3461	3462	3463
6610	3464	3465	3466	3467	3468	3469	3470	3471
6620	3472	3473	3474	3475	3476	3477	3478	3479
6630	3480	3481	3482	3483	3484	3485	3486	3487
6640	3488	3489	3490	3491	3492	3493	3494	3495
6650	3496	3497	3498	3499	3500	3501	3502	3503
6660	3504	3505	3506	3507	3508	3509	3510	3511
6670	3512	3513	3514	3515	3516	3517	3518	3519
6700	3520	3521	3522	3523	3524	3525	3526	3527
6710	3528	3529	3530	3531	3532	3533	3534	3535
6720	3536	3537	3538	3539	3540	3541	3542	3543
6730	3544	3545	3546	3547	3548	3549	3550	3551
6740	3552	3553	3554	3555	3556	3557	3558	3559
6750	3560	3561	3562	3563	3564	3565	3566	3567
6760	3568	3569	3570	3571	3572	3573	3574	3575
6770	3576	3577	3578	3579	3580	3581	3582	3583

7000 to 7777 (Octal) | 3584 to 4095 (Decimal)

	0	1	2	3	4	5	6	7
7000	3584	3585	3586	3587	3588	3589	3590	3591
7010	3592	3593	3594	3595	3596	3597	3598	3599
7020	3600	3601	3602	3603	3604	3605	3606	3607
7030	3608	3609	3610	3611	3612	3613	3614	3615
7040	3616	3617	3618	3619	3620	3621	3622	3623
7050	3624	3625	3626	3627	3628	3629	3630	3631
7060	3632	3633	3634	3635	3636	3637	3638	3639
7070	3640	3641	3642	3643	3644	3645	3646	3647
7100	3648	3649	3650	3651	3652	3653	3654	3655
7110	3656	3657	3658	3659	3660	3661	3662	3663
7120	3664	3665	3666	3667	3668	3669	3670	3671
7130	3672	3673	3674	3675	3676	3677	3678	3679
7140	3680	3681	3682	3683	3684	3685	3686	3687
7150	3688	3689	3690	3691	3692	3693	3694	3695
7160	3696	3697	3698	3699	3700	3701	3702	3703
7170	3704	3705	3706	3707	3708	3709	3710	3711
7200	3712	3713	3714	3715	3716	3717	3718	3719
7210	3720	3721	3722	3723	3724	3725	3726	3727
7220	3728	3729	3730	3731	3732	3733	3734	3735
7230	3736	3737	3738	3739	3740	3741	3742	3743
7240	3744	3745	3746	3747	3748	3749	3750	3751
7250	3752	3753	3754	3755	3756	3757	3758	3759
7260	3760	3761	3762	3763	3764	3765	3766	3767
7270	3768	3769	3770	3771	3772	3773	3774	3775
7300	3776	3777	3778	3779	3780	3781	3782	3783
7310	3784	3785	3786	3787	3788	3789	3790	3791
7320	3792	3793	3794	3795	3796	3797	3798	3799
7330	3800	3801	3802	3803	3804	3805	3806	3807
7340	3808	3809	3810	3811	3812	3813	3814	3815
7350	3816	3817	3818	3819	3820	3821	3822	3823
7360	3824	3825	3826	3827	3828	3829	3830	3831
7370	3832	3833	3834	3835	3836	3837	3838	3839

	0	1	2	3	4	5	6	7
7400	3840	3841	3842	3843	3844	3845	3846	3847
7410	3848	3849	3850	3851	3852	3853	3854	3855
7420	3856	3857	3858	3859	3860	3861	3862	3863
7430	3864	3865	3866	3867	3868	3869	3870	3871
7440	3872	3873	3874	3875	3876	3877	3878	3879
7450	3880	3881	3882	3883	3884	3885	3886	3887
7460	3888	3889	3890	3891	3892	3893	3894	3895
7470	3896	3897	3898	3899	3900	3901	3902	3903
7500	3904	3905	3906	3907	3908	3909	3910	3911
7510	3912	3913	3914	3915	3916	3917	3918	3919
7520	3920	3921	3922	3923	3924	3925	3926	3927
7530	3928	3929	3930	3931	3932	3933	3934	3935
7540	3936	3937	3938	3939	3940	3941	3942	3943
7550	3944	3945	3946	3947	3948	3949	3950	3951
7560	3952	3953	3954	3955	3956	3957	3958	3959
7570	3960	3961	3962	3963	3964	3965	3966	3967
7600	3968	3969	3970	3971	3972	3973	3974	3975
7610	3976	3977	3978	3979	3980	3981	3982	3983
7620	3984	3985	3986	3987	3988	3989	3990	3991
7630	3992	3993	3994	3995	3996	3997	3998	3999
7640	4000	4001	4002	4003	4004	4005	4006	4007
7650	4008	4009	4010	4011	4012	4013	4014	4015
7660	4016	4017	4018	4019	4020	4021	4022	4023
7670	4024	4025	4026	4027	4028	4029	4030	4031
7700	4032	4033	4034	4035	4036	4037	4038	4039
7710	4040	4041	4042	4043	4044	4045	4046	4047
7720	4048	4049	4050	4051	4052	4053	4054	4055
7730	4056	4057	4058	4059	4060	4061	4062	4063
7740	4064	4065	4066	4067	4068	4069	4070	4071
7750	4072	4073	4074	4075	4076	4077	4078	4079
7760	4080	4081	4082	4083	4084	4085	4086	4087
7770	4088	4089	4090	4091	4092	4093	4094	4095

APPENDIX C
SYSTEM MACRO FILE

The following is a listing of the system macro library, SYSMAC.SML. This file is stored on the system device, and used by EXPAND when it expands the programmed requests discussed in Chapter 9.

Several macros and arguments are present in SYSMAC.SML for compatibility purposes. These items can be ignored by the HT-11 user.

```
; SYSMAC.SML
; HT-11 V1A SYSTEM MACRO LIBRARY FOR USE WITH EXPAND
;
; EF,JD,EP
;
; COPYRIGHT (C) 1974,1975,1978
;
; DIGITAL EQUIPMENT CORPORATION
; MAYNARD, MASSACHUSETTS Ø1754
;
; THIS SOFTWARE IS FURNISHED UNDER A LICENSE FOR USE ONLY
; ON A SINGLE COMPUTER SYSTEM AND MAY BE COPIED ONLY WITH
; THE INCLUSION OF THE ABOVE COPYRIGHT NOTICE. THIS SOFTWARE,
; OR ANY OTHER COPIES THEREOF, MAY NOT BE PROVIDED OR OTHERWISE MADE
; AVAILABLE TO ANY OTHER PERSON EXCEPT FOR USE ON SUCH SYSTEM AND TO
; ONE WHO AGREES TO THESE LICENSE TERMS. TITLE TO AND OWNERSHIP OF THE
; SOFTWARE SHALL AT ALL TIMES REMAIN IN DIGITAL.
;
; THE INFORMATION IN THIS SOFTWARE IS SUBJECT TO
; CHANGE WITHOUT NOTICE AND SHOULD NOT BE CONSTRUED
; AS A COMMITMENT BY DIGITAL EQUIPMENT CORPORATION.
;
; DIGITAL ASSUMES NO RESPONSIBILITY FOR THE USE
; OR RELIABILITY OF ITS SOFTWARE ON EQUIPMENT
; WHICH IS NOT SUPPLIED BY DIGITAL.
;

.MACRO ..V1..
...V1=1
.ENDM

.MACRO ..V2..
...V2=1
.ENDM

.MACRO ...CM1
.ENDM
.MACRO ...CM2
.ENDM
.MACRO ...CM3
.ENDM
.MACRO ...CM4
.ENDM
```

```
.MACRO .CDFN .AREA,.ADD,.NUM
.IF NB .AREA
                  MOV     .AREA,%Ø
                  MOV     #64ØØ,(Ø)
.ENDC
.IIF NB .ADD,     MOV     .ADD,2.(Ø)
.IIF NB .NUM,     MOV     .NUM,4.(Ø)
                  EMT     ^O375
.ENDM

.MACRO .CHAIN
                  MOV     #4ØØØ,%Ø
                  EMT     ^O374
.ENDM

.MACRO .CHCOPY .AREA,.CHAN,.OCHAN
.ERROR  ;BAD MAC
.ENDM

.MACRO .CNTXSW  .AREA,.ADD
.ERROR  ;BAD MAC
.ENDM

.MACRO .CLOSE   .CHAN
.IF DF ...V1
                  EMT     ^O<16Ø+.CHAN>
.IFF
                  MOV     #3ØØØ,%Ø
.IIF NB .CHAN,    BISB    .CHAN,%Ø
                  EMT     ^O374
.ENDC
.ENDM

.MACRO .CMKT .AREA,.ID,.TIME
.ERROR  ;BAD MAC
.ENDM

.MACRO .CSIGEN .DEVSPC,.DEFEXT,.CSTRING
                  MOV     .DEVSPC,-(6.)
                  MOV     .DEFEXT,-(6.)
.IF B .CSTRING
                  CLR     -(6.)
.IFF
                  MOV     .CSTRING,-(6.)
.ENDC
                  EMT     ^O344
.ENDM

.MACRO .CSISPC .OUTSPC,.DEFEXT,.CSTRING
                  MOV     .OUTSPC,-(6.)
                  MOV     .DEFEXT,-(6.)
.IF B .CSTRING
                  CLR     -(6.)
.IFF
                  MOV     .CSTRING,-(6.)
.ENDC
                  EMT     ^O345
.ENDM
```

```
.MACRO .CSTAT   .AREA,.CHAN,.ADD
.IF NB .AREA
                MOV     .AREA,%Ø
                MOVB    #23.,1(Ø)
.ENDC
.IIF NB .CHAN,  MOVB    .CHAN,(Ø)
.IIF NB .ADD,   MOV     .ADD,2.(Ø)
                EMT     ^O375
.ENDM

.MACRO .DATE
                MOV     @#^O54,%Ø
                MOV     ^O262(Ø),%Ø
.ENDM

.MACRO .DELETE  .AREA,.CHAN,.DEVBLK
.IF DF ...V1
.IIF NB .CHAN,  MOV     .CHAN,%Ø
                EMT     ^O<.AREA>
.IFF
.IF NB .AREA
                MOV     .AREA,%Ø
                CLRB    1(Ø)
.ENDC
.IIF NB .CHAN,  MOVB    .CHAN,(Ø)
.IIF NB .DEVBLK,MOV     .DEVBLK,2.(Ø)
                CLR     4.(Ø)
                EMT     ^O375
.ENDC
.ENDM

.MACRO .DEVICE  .AREA,.ADD
.ERROR  ;BAD MAC
.ENDM

.MACRO .DSTATUS .RETSPC,.DNAME
.IIF NB .DNAME, MOV     .DNAME,%Ø
                MOV     .RETSPC,-(6.)
                EMT     ^O342
.ENDM
```

```
.MACRO .ENTER    .AREA,.CHAN,.DEVBLK,.LEN
.IF DF ...V1
.IIF NB .CHAN,   MOV      .CHAN,%Ø
.IF B .DEVBLK
                 CLR      -(6.)
.IFF
                 MOV      .DEVBLK,-(6.)
.ENDC
                 EMT      ^O<4Ø+.AREA>
.IFF
.IF NB .AREA
                 MOV      .AREA,%Ø
                 MOVB     #2.,1(Ø)
.ENDC
.IIF NB .CHAN,   MOVB     .CHAN,(Ø)
.IIF NB .DEVBLK,MOV       .DEVBLK,2.(Ø)
.IF NB .LEN
                 MOV      .LEN,4.(Ø)
.IFF
                 CLR      4.(Ø)
.ENDC
                 CLR      6.(Ø)
                 EMT      ^O375
.ENDC
.ENDM

.MACRO .EXIT
                 EMT      ^O35Ø
.ENDM

.MACRO .FETCH .ADD,.DNAME
.IIF NB .DNAME, MOV       .DNAME,%Ø
                 MOV      .ADD,-(6.)
                 EMT      ^O343
.ENDM

.MACRO .GTIM     .AREA,.ADD
.IF NB .AREA
                 MOV      .AREA,%Ø
                 MOV      #1Ø4ØØ,(Ø)
.ENDC
.IIF NB .ADD,    MOV      .ADD,2.(Ø)
                 EMT      ^O375
.ENDM

.MACRO .GTJB     .AREA,.ADD
.IF NB .AREA
                 MOV      .AREA,%Ø
                 MOV      #1ØØØØ,(Ø)
.ENDC
.IIF NB .ADD,    MOV      .ADD,2.(Ø)
                 EMT      ^O375
.ENDM

.MACRO .HERR
                 MOV      #24ØØ,%Ø
                 EMT      ^O374
.ENDM

.MACRO .HRESET
                 EMT      ^O357
.ENDM
```

```
.MACRO .LOCK
                EMT     ^O346
.ENDM

.MACRO .INTEN .PRIO,.PIC
.IF NB .PIC
                MOV     @#^O54,-(6.)
                JSR     5.,@(6.)+
.IFF
                JSR     5.,@^O54
.ENDC
                .WORD   ^C<.PRIO*32.>&224.
.ENDM

.MACRO .LOOKUP  .AREA,.CHAN,.DEVBLK
.IF DF ...V1
.IIF NB .CHAN,  MOV     .CHAN,%Ø
                EMT     ^O<2Ø+.AREA>
.IFF
.IF NB .AREA
                MOV     .AREA,%Ø
                MOVB    #1,1(Ø)
.ENDC
.IIF NB .CHAN,  MOVB    .CHAN,(Ø)
.IIF NB .DEVBLK,MOV     .DEVBLK,2.(Ø)
                CLR     4.(Ø)
                EMT     ^O375
.ENDC
.ENDM

.MACRO  .MFPS   .ADD
        MFPS    .ADD
.ENDM


.MACRO .MRKT    .AREA,.TIME,.CRTN,.ID
.ERROR  ;BAD MAC
.ENDM

.MACRO  .MTPS   .ADD
        MTPS    .ADD
.ENDM


.MACRO .MWAIT
.ERROR  ;BAD MAC
.ENDM

.MACRO .PRINT .ADD
.IIF NB .ADD,   MOV     .ADD,%Ø
                EMT     ^O351
.ENDM

.MACRO .PROTECT .AREA,.ADD
.IF NB .AREA
                MOV     .AREA,%Ø
                MOV     #144ØØ,(Ø)
.ENDC
.IIF NB .ADD,   MOV     .ADD,2.(Ø)
                EMT     ^O375
.ENDM
```

```
.MACRO .PURGE .CHAN
                MOV     #1400,%0
.IIF NB .CHAN,  BISB    .CHAN,%0
                EMT     ^O374
.ENDM

.MACRO .QSET .QADD,.QLEN
.IIF NB .QLEN,  MOV     .QLEN,%0
                MOV     .QADD,-(6.)
                EMT     ^O353
.ENDM

.MACRO .RCTRLO
                EMT     ^O355
.ENDM

.MACRO .RCVD .AREA,.BUFF,.WCNT
.ERROR  ;BAD MAC
.ENDM

.MACRO .RCVDC .AREA,.BUFF,.WCNT,.CRTN
.ERROR  ;BAD MAC
.ENDM

.MACRO .RCVDW .AREA,.BUFF,.WCNT
.ERROR  ;BAD MAC
.ENDM

.MACRO .READ    .AREA,.CHAN,.BUFF,.WCNT,.BLK
.IF DF ...V1
.IIF NB .WCNT,  MOV     .WCNT,%0
                MOV     #1,-(6.)
                MOV     .BUFF,-(6.)
                MOV     .CHAN,-(6.)
                EMT     ^O<200+.AREA>
.IFF
.IF NB .AREA
                MOV     .AREA,%0
                MOVB    #8.,1(0)
.ENDC
.IIF NB .CHAN,  MOVB    .CHAN,(0)
.IIF NB .BLK,   MOV     .BLK,2.(0)
.IIF NB .BUFF,  MOV     .BUFF,4.(0)
.IIF NB .WCNT,  MOV     .WCNT,6.(0)
                MOV     #1,8.(0)
                EMT     ^O375
.ENDC
.ENDM
```

```
.MACRO .READC .AREA,.CHAN,.BUFF,.WCNT,.CRTN,.BLK
.IF DF ...V1
.IIF NB .CRTN,   MOV     .CRTN,%Ø
                 MOV     .WCNT,-(6.)
                 MOV     .BUFF,-(6.)
                 MOV     .CHAN,-(6.)
                 EMT     ^O<2ØØ+.AREA>
.IFF
.IF NB .AREA
                 MOV     .AREA,%Ø
                 MOVB    #8.,1(Ø)
.ENDC
.IIF NB .CHAN,   MOVB    .CHAN,(Ø)
.IIF NB .BLK,    MOV     .BLK,2.(Ø)
.IIF NB .BUFF,   MOV     .BUFF,4.(Ø)
.IIF NB .WCNT,   MOV     .WCNT,6.(Ø)
.IIF NB .CRTN,   MOV     .CRTN,8.(Ø)
                 EMT     ^O375
.ENDC
.ENDM

.MACRO .READW .AREA,.CHAN,.BUFF,.WCNT,.BLK
.IF DF ...V1
.IIF NB .WCNT,   MOV     .WCNT,%Ø
                 CLR     -(6.)
                 MOV     .BUFF,-(6.)
                 MOV     .CHAN,-(6.)
                 EMT     ^O<2ØØ+.AREA>
.IFF
.IF NB .AREA
                 MOV     .AREA,%Ø
                 MOVB    #8.,1(Ø)
.ENDC
.IIF NB .CHAN,   MOVB    .CHAN,(Ø)
.IIF NB .BLK,    MOV     .BLK,2.(Ø)
.IIF NB .BUFF,   MOV     .BUFF,4.(Ø)
.IIF NB .WCNT,   MOV     .WCNT,6.(Ø)
                 CLR     8.(Ø)
                 EMT     ^O375
.ENDC
.ENDM

.MACRO .REGDEF
RØ=%Ø
R1=%1
R2=%2
R3=%3
R4=%4
R5=%5
SP=%6
PC=%7
.ENDM

.MACRO .RELEASE .DEVBLK
.IIF NB .DEVBLK,MOV      .DEVBLK,%Ø
                 CLR     -(6.)
                 EMT     ^O343
.ENDM
```

```
.MACRO .RENAME   .AREA,.CHAN,.DEVBLK
.IF DF ...V1
.IIF NB .CHAN,   MOV      .CHAN,%Ø
                 EMT      ^O<1ØØ+.AREA>
.IFF
.IF NB .AREA
                 MOV      .AREA,%Ø
                 MOVB     #4.,1(Ø)
.ENDC
.IIF NB .CHAN,   MOVB     .CHAN,(Ø)
.IIF NB .DEVBLK,MOV       .DEVBLK,2.(Ø)
                 EMT      ^O375
.ENDC
.ENDM

.MACRO .REOPEN   .AREA,.CHAN,.CBLK
.IF DF ...V1
.IIF NB .CHAN,   MOV      .CHAN,%Ø
                 EMT      ^O<14Ø+.AREA>
.IFF
.IF NB .AREA
                 MOV      .AREA,%Ø
                 MOVB     #6.,1(Ø)
.ENDC
.IIF NB .CHAN,   MOVB     .CHAN,(Ø)
.IIF NB .CBLK,   MOV      .CBLK,2.(Ø)
                 EMT      ^O375
.ENDC
.ENDM

.MACRO .SAVESTAT .AREA,.CHAN,.CBLK
.IF DF ...V1
.IIF NB .CHAN,   MOV      .CHAN,%Ø
                 EMT      ^O<12Ø+.AREA>
.IFF
.IF NB .AREA
                 MOV      .AREA,%Ø
                 MOVB     #5.,1(Ø)
.ENDC
.IIF NB .CHAN,   MOVB     .CHAN,(Ø)
.IIF NB .CBLK,   MOV      .CBLK,2.(Ø)
                 EMT      ^O375
.ENDC
.ENDM

.MACRO .RSUM
.ERROR  ;BAD MAC
.ENDM

.MACRO .SDAT     .AREA,.BUFF,.WCNT
.ERROR  ;BAD MAC
.ENDM

.MACRO .SDATC .AREA,.BUFF,.WCNT,.CRTN
.ERROR  ;BAD MAC
.ENDM

.MACRO .SDATW .AREA,.BUFF,.WCNT
.ERROR  ;BAD MAC
.ENDM
```

```
.MACRO .SERR
                MOV     #2ØØØ,%Ø
                EMT     ^O374
.ENDM

.MACRO .SETTOP .ADD
.IIF NB .ADD,   MOV     .ADD,%Ø
                EMT     ^O354
.ENDM

.MACRO .SFPA    .AREA,.ADD
.IF NB .AREA
                MOV     .AREA,%Ø
                MOV     #14ØØØ,(Ø)
.ENDC
.IIF NB .ADD,   MOV     .ADD,2.(Ø)
                EMT     ^O375
.ENDM

.MACRO .SPFUN   .AREA,.CHAN,.CODE,.BUFF,.WCNT,.BLK,.CRTN
.IF NB .AREA
                MOV     .AREA,%Ø
                MOVB    #26.,1(Ø)
.ENDC
.IIF NB .CHAN,  MOVB    .CHAN,(Ø)
.IIF NB .BLK,   MOV     .BLK,2.(Ø)
.IIF NB .BUFF,  MOV     .BUFF,4.(Ø)
.IIF NB .WCNT,  MOV     .WCNT,6.(Ø)
.IF NB .CODE
                MOVB    #^O377,8.(Ø)
                MOVB    .CODE,9.(Ø)
.ENDC
.IF NB .CRTN
                MOV     .CRTN,8.(Ø)
.IFF
                CLR     8.(Ø)
.ENDC
                EMT     ^O375
.ENDM

.MACRO .SRESET
                EMT     ^O352
.ENDM

.MACRO .SPND
.ERROR  ;BAD MAC
.ENDM

.MACRO .SYNCH .AREA
.IIF NB .AREA,  MOV     .AREA,%4
                MOV     @#^O54,%5
                JSR     5.,@^O324(5.)
.ENDM

.MACRO .TLOCK
                MOV     #34ØØ,%Ø
                EMT     ^O374
.ENDM
```

```
.MACRO .TRPSET .AREA,.ADD
.IF NB .AREA
                MOV     .AREA,%Ø
                MOV     #14ØØ,(Ø)
.ENDC
.IIF NB .ADD,   MOV     .ADD,2.(Ø)
                EMT     ^O375
.ENDM

.MACRO .TTINR
                EMT     ^O34Ø
.ENDM

.MACRO .TTYIN .CHAR
                EMT     ^O34Ø
                BCS     .-2
.IIF NB .CHAR,  MOVB    %Ø,.CHAR
.ENDM

.MACRO .TTOUTR
                EMT     ^O341
.ENDM

.MACRO .TTYOUT .CHAR
.IIF NB .CHAR,  MOVB    .CHAR,%Ø
                EMT     ^O341
                BCS     .-2
.ENDM

.MACRO .TWAIT   .AREA,.TIME
.ERROR  ;BAD MAC
.ENDM

.MACRO .UNLOCK
                EMT     ^O347
.ENDM

.MACRO .WAIT    .CHAN
.IF DF ...V1
                EMT     ^O<24Ø+.CHAN>
.IFF
                CLR     %Ø
.IIF NB .CHAN,  BISB    .CHAN,%Ø
                EMT     ^O374
.ENDC
.ENDM
```

```
.MACRO .WRITE   .AREA,.CHAN,.BUFF,.WCNT,.BLK
.IF DF ...V1
.IIF NB .WCNT,  MOV     .WCNT,%Ø
                MOV     #1,-(6.)
                MOV     .BUFF,-(6.)
                MOV     .CHAN,-(6.)
                EMT     ^O<22Ø+.AREA>
.IFF
.IF NB .AREA
                MOV     .AREA,%Ø
                MOVB    #9.,1(Ø)
.ENDC
.IIF NB .CHAN,  MOVB    .CHAN,(Ø)
.IIF NB .BLK,   MOV     .BLK,2.(Ø)
.IIF NB .BUFF,  MOV     .BUFF,4.(Ø)
.IIF NB .WCNT,  MOV     .WCNT,6.(Ø)
                MOV     #1,8.(Ø)
                EMT     ^O375
.ENDC
.ENDM

.MACRO .WRITC .AREA,.CHAN,.BUFF,.WCNT,.CRTN,.BLK
.IF DF ...V1
.IIF NB .CRTN,  MOV     .CRTN,%Ø
                MOV     .WCNT,-(6.)
                MOV     .BUFF,-(6.)
                MOV     .CHAN,-(6.)
                EMT     ^O<22Ø+.AREA>
.IFF
.IF NB .AREA
                MOV     .AREA,%Ø
                MOVB    #9.,1(Ø)
.ENDC
.IIF NB .CHAN,  MOVB    .CHAN,(Ø)
.IIF NB .BLK,   MOV     .BLK,2.(Ø)
.IIF NB .BUFF,  MOV     .BUFF,4.(Ø)
.IIF NB .WCNT,  MOV     .WCNT,6.(Ø)
.IIF NB .CRTN,  MOV     .CRTN,8.(Ø)
                EMT     ^O375
.ENDC
.ENDM

.MACRO .WRITC .AREA,.CHAN,.BUFF,.WCNT,.CRTN,.BLK
.IF DF ...V1
.IIF NB .CRTN,  MOV     .CRTN,%Ø
                MOV     .WCNT,-(6.)
                MOV     .BUFF,-(6.)
                MOV     .CHAN,-(6.)
                EMT     ^O<22Ø+.AREA>
.IFF
.IF NB .AREA
                MOV     .AREA,%Ø
                MOVB    #9.,1(Ø)
.ENDC
.IIF NB .CHAN,  MOVB    .CHAN,(Ø)
.IIF NB .BLK,   MOV     .BLK,2.(Ø)
.IIF NB .BUFF,  MOV     .BUFF,4.(Ø)
.IIF NB .WCNT,  MOV     .WCNT,6.(Ø)
.IIF NB .CRTN,  MOV     .CRTN,8.(Ø)
                EMT     ^O375
.ENDC
.ENDM
```

```
.MACRO .WRITW .AREA,.CHAN,.BUFF,.WCNT,.BLK
.IF DF ...V1
.IIF NB .WCNT,  MOV     .WCNT,%Ø
                CLR     -(6.)
                MOV     .BUFF,-(6.)
                MOV     .CHAN,-(6.)
                EMT     ^O<22Ø+.AREA>
.IFF
.IF NB .AREA
                MOV     .AREA,%Ø
                MOVB    #9.,1(Ø)
.ENDC
.IIF NB .CHAN,  MOVB    .CHAN,(Ø)
.IIF NB .BLK,   MOV     .BLK,2.(Ø)
.IIF NB .BUFF,  MOV     .BUFF,4.(Ø)
.IIF NB .WCNT,  MOV     .WCNT,6.(Ø)
                CLR     8.(Ø)
                EMT     ^O375
.ENDC
.ENDM
```

APPENDIX D
PROGRAMMED REQUEST SUMMARY

D.1 PARAMETERS

The following parameters are used as arguments in various calls. (Any parameters used which are not mentioned here are particular to a request and the appropriate section in Chapter 9 should be consulted.)

Parameter	Description
.addr	an address, the meaning of which depends on the request being used
.area	a pointer to the EMT argument list
.blk	a block number specifying the relative block in a file where an I/O operation is to begin
.buff	a buffer address specifying a memory location into which or from which an I/O transfer is to be performed
.chan	a channel number in the range 0-377 (octal)
.crtn	the entry point of a completion routine
.dblk	the address of a four-word RAD50 descriptor of the file to be opened
.num	a number, the value of which depends on the request
.wcnt	a word count specifying the number of words to be transferred to or from the buffer during an I/O operation

D.2 REQUEST SUMMARY

Refer to Appendix C (SYSMAC.SML) to see how each macro call is expanded in assembly language code.

Mnemonic	Function	Macro Call	Error Codes (Byte 52=)
.CDFN	Increases number of I/O channels to as many as 255 (decimal)	.CDFN .area,.addr,.num	0 – attempt to define fewer channels than already exist
.CHAIN	Allows one program to transfer control to another without operator intervention	.CHAIN	Can produce any errors which the monitor RUN command can produce

Mnemonic	Function	Macro Call	Error Codes (Byte 52=)
.CLOSE	Terminates activity on specified channel and frees it for use in another operation; makes tentative files permanent	.CLOSE .chan	Fatal monitor error if device handler is not in memory
.CSIGEN	Calls the CSI in general mode Note: if input is taken from TT:, all errors are printed out	.CSIGEN .devspc,.defext,.cstring	0 – illegal command 1 – device not found 2 – unused 3 – full directory 4 – input file not found
.CSISPC	Calls the CSI in special mode Note: if input is taken from TT:, all errors are printed out	.CSISPC .outspc,.defext,.cstring	0 – illegal command line 1 – illegal device
.DATE	Moves current date information into R0	.DATE	None
.DELETE	Deletes named file from indicated device	.DELETE .area,.chan,.dblk	0 – active channel 1 – file not found
.DSTATUS	Provides information about a device	.DSTATUS .cblk,.devnam	0 – device not found
.ENTER	Allocates space on specified device and creates tentative entry for named file	.ENTER .area,.chan,.dblk,.length	0 – channel in use 1 – no space greater than or equal to the specified length was found
.EXIT	Terminates user program and returns control to monitor	.EXIT	None
.FETCH	Loads device handlers into memory from system device	.FETCH .coradd,.devnam	0 – nonexistent device name or no handler for that device
.GTIM	Allows access to the current time of day	.GTIM .area,.addr	None

Mnemonic	Function	Macro Call	Error Codes (Byte 52=)
.GTJB	Passes certain job parameters back to user program	.GTJB .area,.addr	None
.HERR	Disables error interception and allows system to detect and act on normally fatal errors	.HERR	Monitor Error occurs if: 1. called USR from completion routine 2. no device handler 3. error doing directory I/O 4. FETCH error 5. Error reading overlay 6. no room in directory 7. illegal address 8. illegal channel number 9. illegal EMT
.HRESET	Resets channels, releases device handlers and stops all I/O transfers in progress	.HRESET	None
.INTEN	Notifies monitor that interrupt has occurred, and sets processor priority to correct state	.INTEN .priority, .pic	None
.LOCK	Locks the USR in memory	.LOCK	None
.LOOKUP	Associates a specified channel with a device and/or file on that device	.LOOKUP .area,.chan,.dblk	0 – channel already open 1 – file not found
.PRINT	Outputs a string to the terminal	.PRINT .addr	None
.PURGE	Deactivates a channel without taking any other action	.PURGE .chan	None
.QSET	Enlarges I/O queue for monitor	.QSET .addr,.qleng	None
.RCTRLO	Enables terminal printing	.RCTRLO	None

Mnemonic	Function	Macro Call	Error Codes (Byte 52=)
.READ	Initiates transfer from specified channel to memory; returns to program immediately	.READ .area,.chan,.buff,.wcnt,.blk	0 – attempt to read past end-of-file 1 – hard error on channel 2 – channel not open
.READC	Transfers words from specified channel to memory; returns control to specified routine when complete	.READC .area,.chan,.buff,.wcnt,.crtn,.blk	0 – attempt to read past end-of-file 1 – hard error on channel 2 – channel not open
.READW	Transfers words from specified channel to memory; returns control to user program when transfer complete	.READW .area,.chan,.buff,.wcnt,.blk	0 – attempt to read past end-of-file 1 – hard error on channel 2 – channel not open
.REGDEF	Defines general registers RO-R5, SP,PC	.REGDEF	None
.RELEAS	Removes device handler from memory	.RELEAS .devnam	0 – handler name is illegal
.RENAME	Changes file name	.RENAME .area,.chan,.dblk	0 – channel open 1 – file not found
.REOPEN	Reassociates channel with file on which a SAVESTATUS was performed	.REOPEN .area,.chan,.cblk	0 – channel is in use
.SAVESTATUS	Stores five words (containing data concerning file definition) into memory	.SAVESTATUS .area,.chan,.cblk	1 – SAVESTATUS is illegal
.SERR	Inhibits fatal errors from aborting job	.SERR	-1 – called USR from completion routine -2 – no device handler -3 – error doing directory I/O -4 – FETCH error -5 – error reading overlay -6 – no more room for files in directory -7 – illegal address -10 – illegal channel number -11 – illegal EMT

Mnemonic	Function	Macro Call	Error Codes (Byte 52=)
.SETTOP	Requests additional memory for program	.SETTOP .addr	None
.SFPA	Sets user interrupt address for floating point processor exceptions	.SFPA .area,.addr	None
.SRESET	Resets certain areas of memory, dismisses device handlers brought in by FETCH, purges currently open files, resets to 16 I/O channels, queue to one element	.SRESET	None
.SYNCH	Enables monitor programmed request from within an interrupt service routine; normal return is to 2nd location following .SYNCH	SYNCH .area	Monitor returns to location following .SYNCH if: 1. another .SYNCH specifying same 7-word block is pending 2. illegal job number was specified 3. job is not running
.TRPSET	Allows user job to intercept traps to 4 and 10	.TRPSET .area,.addr	None
.TTYIN	Inputs character from terminal and waits until done	.TTYIN .char	None
.TTINR	Inputs character from terminal	.TTINR	0 – No characters available in ring buffer
.TTOUTR	Outputs character to terminal	.TTOUTR	0 – Output ring buffer full

Mnemonic	Function	Macro Call	Error Codes (Byte 52=)
.TTYOUT	Outputs character to terminal and waits until done	.TTYOUT .char	None
.UNLOCK	Releases USR from memory	.UNLOCK	None
.WAIT	Suspends program execution until all channel I/O is complete	.WAIT .chan	0 – channel not open 1 – hardware error
.WRITC	Initiates transfer from memory to specified channel and returns to user program; when complete, passes control to specified routine	.WRITC .area,.chan,.buff,.wcnt,.crtn,.blk	0 – end-of-file reached 1 – hardware error 2 – channel not open
.WRITE	Initiates transfer from memory to channel; returns control to user program immediately	.WRITE .area,.chan,.buff,.wcnt,.blk	0 – end-of-file reached 1 – hardware error 2 – channel not open
.WRITW	Transfers words from memory to channel; when complete, returns control to user program	.WRITW .area,.chan,.buff,.wcnt,.blk	0 – end-of-file reached 1 – hardware error 2 – channel not open

APPENDIX E
DUMP

HT-11 DUMP is a program which outputs to the terminal or lineprinter all or any part of a file in octal words, octal bytes, ASCII characters and/or RAD50 characters. DUMP is particularly useful for examining data such as directories or files.

E.1 CALLING AND USING DUMP

DUMP is called using the monitor command:

```
R DUMP
```

in response to the dot printed by the Keyboard Monitor. The name of the file which is to be output is entered as follows in response to the asterisk printed by the Command String Interpreter:

dev:output=dev:input/s

where:

dev:	represents any valid device specification (terminal is default for output if no output file is designated).
output	represents the filename and extension assigned to the output file. The default extension for file-structured output is .DMP.
input	represents the input source filename and extension.
/s	represents one or more of the switches listed in Table E-1.

Type CTRL C to halt DUMP at any time and return control to the monitor. To restart DUMP, type R DUMP or the REENTER command in response to the monitor's dot.

E.1.1 DUMP Switches

The following switches can appear in the command string for DUMP:

Table E-1 DUMP Switches

Switch	Meaning
/B	Output octal bytes
/E:n	End output at block number n
/G	Ignore input errors
/N	Suppress ASCII output
/O:n	Output only block number n (same as /E:n, /S:n)
/S:n	Start output with block number n
/W	Output octal words
/X	Output RAD50 characters

If neither /W nor /B is given, /W is assumed. ASCII characters are always dumped unless /N is given. The number n is an octal block number.

If an input filename is given, block numbers are relative to the beginning of the file to which the block belongs. If not, block numbers are absolute block numbers on the device (i.e., the physical block numbers on the corresponding device).

E.1.2 Examples

The following are two examples of DUMP. /B is used in the first example to output octal bytes of the file SQRT.FTN into a file called DIF.DMP on device DX1.

```
R DUMP
*DX1:DIF = SQRT.FTN/B
```

If DIF.DMP is then listed on the line printer (using PIP), it appears as follows:

```
BLOCK NUMBER 0000
000/  001 000 056 000 001 000 011 261 214 072 150 000 000 000 001 257
       .   .   .   .   .   .   .   1   .   :   H   .   .   .   .   /
020/  224 017 110 001 000 000 262 252 304 037 110 004 210 000 374 252
       .   .   H   .   .   .   2   *   D   .   H   .   .   .   ϕ   *
040/  016 024 110 004 006 000 033 254 217 163 110 004 000 000 012 001
       .   .   H   .   .   .   .   ,   .   $   H   .   .   .   .   .
060/  000 036 000 001 000 054 253 100 070 100 004 000 000 123 263 024
       .   .   .   .   .   ,   +   @   8   @   .   .   .   S   3   .
100/  255 150 001 000 000 000 000 000 000 000 003 000 000 032 001 000
       —   H   .   .   .   .   .   .   .   .   .   .   .   .   .   .
120/  016 000 004 000 007 000 000 000 000 000 000 000 346 001 000 016
       .   .   .   .   .   .   .   .   .   .   .   .   F   .   .   .
140/  000 003 000 000 000 067 011 076 371 000 000 167 001 000 014 000
       .   .   .   .   .   7   .   >   Y   .   .   W   .   .   .   .
160/  004 000 004 006 054 253 100 070 226 000 000 000 000 000 000 000
       .   .   .   .   ,   +   @   8   .   .   .   .   .   .   .   .
200/  000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000
       .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .
220/  000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000
       .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .
240/  000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000
       .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .
260/  000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000
       .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .
300/  000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000
       .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .
320   000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000
       .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .
340/  000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000
       .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .
360/  000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000
       .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .
400/  000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000
       .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .
420/  000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000
       .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .
```

```
440/   000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000
        .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .
460/   000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000
        .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .
500/   000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000
        .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .
520/   000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000
        .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .
540/   000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000
        .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .
560/   000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000
        .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .
600/   000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000
        .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .
620/   000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000
        .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .
640/   000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000
        .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .
660/   000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000
        .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .
700/   000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000
        .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .
720/   000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000
        .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .
740/   000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000
        .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .
760/   000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000
        .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .
```

The second example illustrates the use of /X to output RAD 50 and octal values for locations in the file. The numbers in the left column represent the byte displacement. ASCII characters are suppressed.

```
.R DUMP
*LP: = DX:/O:6/X/N

BLOCK NUMBER 00006
000/    000004   000000   000001   000000   000016   002000   051646   035562
          D                  A                 N        YX     MON       ITR
020/    075273   000052   000016   020446   002000   017751   076400   073376
        SYS        AB        N     ELF        YX     EDI      T        SAV
040/    000016   000016   020446   002000   057164   000000   057032   000011
          N        N      ELF        YX     ODT               OBJ        I
060/    000016   020446   002000   075273   050553   074324   000023   000016
          N      ELF        YX     SYS      MAC      SML        S        N
100/    020446   002000   062570   000000   073376   000015   000016   020446
        ELF        YX     PIP               SAV        M        N      ELF
120/    002000   004505   007145   073376   000032   000016   020446   002000
          YX     ASM      BLE      SAV        Z        N      ELF        YX
140/    021420   004164   073376   000014   000016   020446   002000   046166
        EXP      AND      SAV        L        N      ELF        YX     LIN
160/    042300   073376   000031   000015   020446   002000   110215   042300
          K      SAV        Y        M      ELF        YX     WEE      K
```

200/	073376	000020	000016	020446	002000	016125	062000	073376
	SAV	P	N	ELF	YX	DUM	P	SAV
220/	000005	000016	020546	002000	074623	012445	073376	000013
	E	N	EM0	YX	SRC	COM	SAV	K
240/	000015	020546	002000	012625	022600	073376	000005	000014
	M	EM0	YX	CRE	F	SAV	E	L
260/	020546	001000	057164	000000	073376	000433	000000	020446
	EM0	L2	ODT		SAV	GC		ELF
300/	004000	057164	000000	073376	000433	000000	020446	004000
	AKH	ODT		SAV	GC		ELF	AKH
320/	057164	000000	073376	000433	000000	020446	004000	162745
	ODT		SAV	GC		ELF	AKH	61M
340/	162745	162745	162745	162745	162745	162745	162745	162745
	61M	61M	61M	61M	61M	61M	61M	61M
360/	162745	162745	162745	162745	162745	162745	162745	162745
	61M	61M	61M	61M	61M	61M	61M	61M
400/	162745	162745	162745	162745	162745	162745	162745	162745
	61M	61M	61M	61M	61M	61M	61M	61M
420/	162745	162745	162745	162745	162745	162745	162745	162745
	61M	61M	61M	61M	61M	61M	61M	61M
440/	162745	162745	162745	162745	162745	162745	162745	162745
	61M	61M	61M	61M	61M	61M	61M	61M
460/	162745	162745	162745	162745	162745	162745	162745	162745
	61M	61M	61M	61M	61M	61M	61M	61M
500/	162745	162745	162745	162745	162745	162745	162745	162745
	61M	61M	61M	61M	61M	61M	61M	61M
520/	162745	162745	162745	162745	162745	162745	162745	162745
	61M	61M	61M	61M	61M	61M	61M	61M
540/	162745	162745	162745	162745	162745	162745	162745	162745
	61M	61M	61M	61M	61M	61M	61M	61M
560/	162745	162745	162745	162745	162745	162745	162745	162745
	61M	61M	61M	61M	61M	61M	61M	61M
600/	162745	162745	162745	162745	162745	162745	162745	162745
	61M	61M	61M	61M	61M	61M	61M	61M
620/	162745	162745	162745	162745	162745	162745	162745	162745
	61M	61M	61M	61M	61M	61M	61M	61M
640/	162745	162745	162745	162745	162745	162745	162745	162745
	61M	61M	61M	61M	61M	61M	61M	61M
660/	162745	162745	162745	162745	162745	162745	162745	162745
	61M	61M	61M	61M	61M	61M	61M	61M
700/	162745	162745	162745	162745	162745	162745	162745	162745
	61M	61M	61M	61M	61M	61M	61M	61M
720/	162745	162745	162745	162745	162745	162745	162745	162745
	61M	61M	61M	61M	61M	61M	61M	61M
740/	162745	162745	162745	162745	162745	162745	162745	162745
	61M	61M	61M	61M	61M	61M	61M	61M
760/	162745	162745	162745	162745	162745	162745	162745	162745
	61M	61M	61M	61M	61M	61M	61M	61M

E.2 DUMP ERROR MESSAGES

The following errors may occur when using DUMP:

Message	Meaning
?IN ERR?	A hardware error occurred while reading the input file and /G was not specified in the command line.
?OUT ERR?	A hardware error occurred while writing an output file, or the output device was full.
?LP NOT FND?	A line printer handler is not available on the system.

APPENDIX F
SOURCE COMPARE (SRCCOM)

The HT-11 Source Compare program (SRCCOM) is used to compare two ASCII files and to output any differences to a specified output device. It is particularly useful when the two files are different versions of a single program, in which case SRCCOM prints all the editing changes which transpired between the two versions.

F.1 CALLING AND USING SRCCOM

To run SRCCOM type the command:

```
R SRCCOM
```

followed by a carriage return in response to the dot printed by the Keyboard Monitor; the CSI prints an asterisk. Then enter the names of the files which are to be compared using a command string in the following format:

```
dev:output=dev:input1,dev:input2/s
```

where:

dev:	is any valid device specification.
output	is the filename and extension assigned to the output file. If no output file is indicated, output is directed to the terminal.
input1 ...	are the input source filenames and extensions to be compared.
/s	is one of the switches listed in Table F-1.

Source files are examined line by line for groups of lines which match. When a mismatch occurs, all differences are output until n successive lines in the first file are identical to n lines in the second file. The number (n) is a variable which the user can set with the /L switch.

F.1.1 Extensions

No default extension is assigned by SRCCOM to the output file. The default extension for an input file is .MAC, representing a source file in MACRO language.

F.1.2 Switches

Command switches are generally placed at the end of the command string but may follow any filename in the string. The following switches can appear in the command string.

Table F-1

SRCCOM Switches

Switch	Meaning
/B	Compare blank lines. Without this switch, blank lines are ignored.
/C	Ignore comments (all text on a line preceded by a semicolon) and spacing (spaces and tabs). This switch does not cause a line consisting entirely of a comment to become a blank line, and therefore ignored in the line count.
/F	Include form feeds in the output file. (Form feeds are still compared if /F is not used, but they are not included in the output of differences.)
/H	Type list of switches available (help text). No I/O device is necessary since /H always prints the help text on the terminal.
/L:n	Specify the number of lines that determines a match (n is an octal number <=310). All differences occurring before and after a match are output. In addition, the first line of the current match is output after the differences to aid in locating the place within each file at which the differences occurred. The default value for n is 3.
/S	Ignore spaces and tabs.

F.2 OUTPUT FORMAT

The first line of each file is always output as identification and is also compared. A blank line is then printed, followed by the differences between the files, in the following format:

```
1) 1        FILEA
1)          A
****
2) 1        FILEB
2)          A

**********

% FILES ARE DIFFERENT
```

The different lines are listed followed by a reference line which is the same for both files. Note the example below.

The following example uses SRCCOM to compare an edited file and its backup version. The default value for a match is 3 lines. Blank lines are ignored but all other characters are compared.

Following the example is a coded explanation of the comparison.

```
     .R SRCCOM
     *DX1:LINKO. FB,LINKO. BAK
A {  1) 1                  .TITLE     HTLINK ROOT CODE H03-16
  {  2) 1                  .TITLE     HTLINK ROOT CODE H03-15

  {  1) 1                  SEVENK=    31452              ; MINIMUM CORE TO START LINKER
B {  1)
  {  1)
C {  1)                    .MCALL     .CSISPC,. CSIGEN,.SETTOP,.LOCK,.UNLOCK
     ****
  {  2) 1                  SEVENK=    31500              ; JUST BELOW 8K RESIDENT
B {  2)
  {  2)
C {  2)                    .MCALL     .CSISPC,.CSIGEN,.SETTOP,.LOCK,.UNLOCK
     **********
B {  1) 2                  .GLOBL     RSWIT,RELPTR,FBTXT,OVLNUM,RELOVL,RLSTRT
C {  1)                    .GLOBL     RELADR,PNRELO,RELID1,RSIZ1,OVSIZ1,OVLCDE
     ****
B {  2) 2                  .GLOBL     RSWIT,RELPTR,FBTXT,OVLNUM, RELOVL
C {  2)                    .GLOBL     RELADR,PNRELO,RELID1,RSIZ1,OVSIZ1,OVLCDE
     **********
B {  1) 2       RLSTRT:    .BLKW                   ;CURRENT REL BLK OVERLAY NUM
C {  1)         RELPTR:    .BLKW                   ;POINTER TO CURRENT REL BLK LOCATION
     ****
C {  2)2         RELPTR:    .BLKW                   ; POINTER TO CURRENT REL BLK LOCATION
     **********
B {  1) 2       MTITLE:    .ASCII     /HT-11 LINK        X03-16/
C {  1)                    .ASCII     /      LOAD MAP /
     ****
B {  2) 2       MTITLE:    .ASCII     /HT-11 LINK        X03-15/
C {  2)                    .ASCII     /      LOAD MAP /
     ************
  {  1) 12                 .IF DF FB
B {  1)                    MOV        OBLK, RLSTRT       ; IND START OF OVL FOR REL BLK
  {  1)                    .ENDC
C {  1)                    BR         1$
     ****
C {  2) 12                 BR         1$
     **********

D {  %FILES ARE DIFFERENT
```

A Headers, consisting of the first line of each file; for identification purposes.

B n)m. A notation where n is the number of the input file, and m is the page number (less than 256 decimal) of the input file on which the text appears. The right column lists the lines in the files which are different.

C Following a section of differences, a line identical to each file is output for reference purposes.

D Indicates that the files are different (this is printed on the system terminal, not in the output file).

This example uses the /L:n switch and sets the number of lines that determines a match to 2 lines. The first two columns represent the input files:

```
TEST FILE 1
LINE C
LINE E
LINE C
LINE D
LINE F
LINE H
LINE I
LINE J

TEST FILE 2
LINE C
LINE D
LINE C
LINE E
LINE F
LINE G
LINE H
LINE I
LINE J
```

The files are compared and differences listed on the line printer.

```
*LP: =TEST1, TEST2/L:2
1) 1        TEST FILE 1
2) 1        TEST FILE 2

1) 1        LINE E
1)          LINE C
1)          LINE D
1)          LINE F
1)          LINE H
****
2) 1        LINE D
2)          LINE C
2)          LINE E
2)          LINE F
2)          LINE G
2)          LINE H
**********
```

This message prints on the terminal indicating that the files are different.

```
%FILES ARE DIFFERENT
```

F.3 SRCCOM ERROR MESSAGES

The following errors may be reported by SRCCOM:

Messages	Meaning
?COR OVR?	Not enough memory to hold a particular difference section.
?IN ERR?	A hardware error occurred in reading input.
?OUT ERR?	A hardware error occurred in writing output file, or output device full.
?SWITCH ERROR?	An invalid switch was found or a switch other than /L was given a value.
?TOO MUCH DIFFERENCE?	More than 310 (octal) lines of difference between two files were found.

APPENDIX G
PATCH

The PATCH utility program is used to make code modifications to memory image (.SAV) files, including overlay-structured and monitor files. PATCH, like ODT, can be used to interrogate, and then to 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 modified when PATCH is used.

G.1 CALLING AND USING PATCH

To run PATCH, type the command:

```
R PATCH
```

followed by the RETURN key in response to the dot printed by the monitor. PATCH prints a version number message:

```
PATCH H01-02
```

and then prints the message:

```
FILE NAME --
*
```

In response to the asterisk, enter the name of the file to be modified, using the following format:

```
dev:filnam.ext/M/O
```

where:

dev:	represents an optional device specification; if not specified, DK: is assumed.
filnam.ext	represents the name of the file which is to be patched, if an extension is not indicated, .SAV is assumed.
/M	must be used if the file is an HT-11 monitor file.
/O	must be used if the file is an overlay-structured file.

G.2 PATCH COMMANDS

Table G-1 summarizes the PATCH commands.

Table G-1

PATCH Commands

Command	Action
Vr; nR	Set relocation register n to value Vr.
b ; B	Set bottom address of overlay file to b.
[s :] r,o/	Open word location Vr + o in overlay segment s.
[s :] r,o\	Open byte location Vr + o in overlay segment s.
<CR>	Close currently open word/byte.
<LF>	Close currently open word/byte and open the next one.
↑ or ^	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 HT-11 monitor.

Explanations of each command follow. An example of the use of the commands is provided in Section G.3.

G.2.1 Patch a New File
The F command causes PATCH to close the file being patched, and accept a new file name to be patched.

G.2.2 Exit from PATCH
The E command causes PATCH to close the file being patched and return control to the HT-11 monitor.

G.2.3 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 either new location contents (in octal) or another command.

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 the 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 to the new contents (if specified), and await more control input.
<line feed>	Close the current location, and open the next word/byte.
↑ or ^	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).

G.2.4 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 is the case if the program sets location 42 in an .ASECT. To set the bottom address, type:

<bottom address>;B

Note that the B command must be issued before any locations are opened for modification.

G.2.5 Set Relocation Registers
The relocation registers 0-7 are set, as with ODT, by the R command. The R command 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>

G.3 EXAMPLES USING PATCH

The following example shows how to patch a non-overlaid file. Assume the following program (EXAM):

```
.MAIN.     HT-11 ASEMBL HM02-0B       PAGE 1


1
2
3          000015              CR=       15
4          000012              LF=       12
5          000000 '            .CSECT    MAIN
6                              .MCALL    .PRINT,.EXIT
7                              .NLIST    BEX
8  000000       124  MSG:      .ASCII    /THIS IS A SUCCESSFUL PATCH/<CR><LF>
9                              .LIST     BEX
10  00034  000403 START:       BR        EXIT
11  00036                      .PRINT    #MSG
12  00044          EXIT:       .EXIT
13          000034 '           .END      START
```

This program has been assembled with ASSEMBL and linked with LINK; execution causes no output of text:

```
. R EXAM
```

To make a line of text print on the terminal, PATCH is used as follows:

```
R  PATCH

PATCH H01-02

FILE NAME --
*EXAM. SAV
*1000;0R
*0, 34/  403    240
*E
```

Now when the program is executed:

```
R  EXAM
THIS IS A SUCCESSFUL PATCH
```

The next example demonstrates a similar situation, only includes an overlay file. These programs have been assembled and linked; the output of both operations is included:

```
.MAIN.    HT-11 ASEMBL HM02-08   3-SEP-78   PAGE 1

1
2
3         000015              CR=      15
4         000012              LF=      12
5         000007              PC=      %7
6         000000 '            .CSECT   MAIN
7                             .GLOBL   ENTRY, MSG1
8                             .MCALL   .PRINT,.EXIT
9                             .NLIST   BEX
10 00000     124  MSG:        .ASCIZ   /THIS IS A SUCCESSFUL PATCH/<CR><LF>
11 00035     124  MSG1:       .ASCIZ   /THIS IS AN OVERLAY PATCH/
12                            .LIST    BEX
13 00066  000403  START:      BR       EXIT
14 00070                      .PRINT   #MSG
15 00076  004767  EXIT:       JSR      PC,ENTRY
          000000G
16 00102                      .EXIT
17        000065'             .END     START


.MAIN.    HT-11 ASEMBL HM02-08      3-SEP-78   PAGE 1

1
2
3         000015              CR=      15
4         000012              LF=      12
5         000007              PC=      %7
6         000000 '            .CSECT   OVL
7                             .MCALL   .PRINT
8                             .GLOBL   MSG1
9                             .GLOBL   ENTRY
10 00000  000403  ENTRY:      BR       RETURN
11 00002                      .PRINT   #MSG1
12 00010  000207  RETURN:     RTS      PC
13        000001 '            .END

HT-11 LINK    H03-18          LOAD MAP
PTCH .SAV                     03-SEP-78

SECTION ADDR      SIZE     ENTRY    ADDR     ENTRY    ADDR     ENTRY    ADDR

. ABS.   000000   001122
MAIN     001122   000104   MSG1     001157

OVERLAY REGION 000001   SEGMENT  000001
OVL      001230     000012 ENTRY    001230

TRANSFER ADDRESS = 001210
HIGH LIMIT  =  001242
```

Running the program (PTCH) produces no terminal output:

```
. R PTCH
```

But by using PATCH to modify the file as follows:

```
. R PATCH

PATCH H01-02

FILE NAME --
*PTCH. SAV/O
*1230 ;0R
*1 : 0, 0/ 403      240
*E
```

the following line results:

```
. R PTCH
THIS IS AN OVERLAY PATCH
```

G.4 PATCH ERROR MESSAGES

Error messages which may occur under PATCH follow.

Message	Meaning
?ADDR NOT IN SEG?	The address is not in the specified overlay 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 memory 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 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.

Message	Meaning
?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.
?PROG HAS NO 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.

INDEX

Abort entry point, 9-31
Absolute, 5-2
  and relocatable program sections, 6-2
  block numbers, 4-14, E-2
  expression, 5-14, B-5
  load address, 6-1
  load module, 6-4
  mode, 5-20
  quantities, 5-14
  section, 6-2
  starting block, 4-11
Absolute Loader, 6-4
Accessing,
  general registers, 8-7
  internal registers, 8-7
Address mode syntax, B-5
Addressed location, 8-6
Addresses, vector, 9-4
Addressing modes, 5-16
Advance command, 3-14
Allocating,
  blocks for files, 4-5
  extra words, 4-12
  memory for a queue, 9-45
  system resources, 2-8
Alphabetize switch, 6-13
Alphabetized load map, 6-14
Alphanumeric representation, 1-1
ALTMODE, 3-1
Argument, 3-4
  block, 9-2
  dummy, 5-53
  iteration, 3-6
  list, 9-4
  list pointer, 9-1
  negative line, 3-6
  numeric, 3-5
  numerical, 9-3
  positive, 3-5
Arguments, 9-1
  EDIT, 3-4
  missing, 5-53
  number of, 5-53
  real, 5-52
  symbolic, 5-29
  to Macro calls and definitions, 5-52
ASCII,
  character set, B-1, B-4
  conversion of one or two characters, 5-32
  files, 3-1
  format, 2-1
  input and output, 8-16
.ASCII directive, 5-33
.ASCIZ directive, 5-34
.ASECT directive, 5-44, 6-2, 6-4
ASEMBL, 1-2, 5-1
  calling and using, 5-58
  character code, B-1
  directives, 5-50
  error messages, 5-65
  features, 5-1
  file specifications, 5-58
  instructions, B-1
  program section capabilities, 5-8
  source code, 5-62, 5-63
  source statements, 5-1
  special characters, B-5
  switches, 5-59, B-19
Assembler, 1-1, 8-2
  ASEMBL, 5-1
  output, 5-16
Assembler directives, 5-3, 5-22, B-17
  .ASCII, 5-33
  .ASCIZ, 5-34
  .ASECT, 5-42
  .BLKB, 5-39
  .BLKW, 5-39
  .BYTE, 5-30
  .CSECT, 5-42
  .DSABL, 5-29
  .ENABL, 5-29
  .END, 5-42
  .ENDM, 5-51
  .EOT, 5-42
  .EVEN, 5-38
  .FLT2, 5-40
  .FLT4, 5-40
  .GLOBL, 5-44
  .IDENT, 5-28
  .IFF, 5-48
  .IFT, 5-48
  .IFTF, 5-48

Assembler directives (cont.)
  .LIMIT, 5-42
  .LIST, 5-22
  .MACRO, 5-50
  .MCALL, 5-54
  .NLIST, 5-22
  .ODD, 5-38
  .PAGE, 5-29
  .RADIX, 5-36
  .RAD50, 5-35
  .SBTTL, 5-27
  .TITLE, 5-27
  .WORD, 5-31
Assembly,
  language statement, 5-1
  listing, 8-1
  listing table of contents, 5-27
  location counter, 5-11
  pass, 5-11
  source listing showing local symbol blocks, 5-12
ASSIGN command, 2-9
Asterisk,
  wild-card, 4-1
Asynchronous completion routines, 9-10
Autodecrement mode, 5-18
Autodecrement deferred mode, 5-19
Autoincrement mode, 5-17
Autoincrement deferred mode, 5-18
Automatic relocation facility, 8-2

Backslash, 8-5
Backup storage device, 6-5
Back-arrow, 8-6
Bad block files, 4-1
Bad block scan, 4-14
Bad entry, 8-1, 8-20
Base address, 5-20, 8-2
Base command, 2-14
Beginning command, 3-13
Binary,
  code, 1-1
  object module, 8-2
  operators, 5-6, 5-14
  output, 1-1
  radix, 5-13, 5-37
Bit patterns, 1-2, 8-19
  search, 8-8
Blank COMMON, 6-10
Blank,
  extension filename, 2-3
  lines, 5-2
.BLKB directive, 5-39
.BLKW directive, 5-39
Blocks, 2-6
  control, 6-1
  device, 9-4
  EMT arguments, 9-4
  256-word, 2-16
Block numbers,
  absolute, E-2
  physical, E-2
Block-replaceable devices, 2-4, 4-8
Boot operation, 4-13
Bootstrap,
  copy operation, 4-13
  file, 4-12
Bootstrapping the system, 4-13
Bottom address switch, 6-13
Branch,
  address, 5-22
  instruction addressing, 5-22
  instructions, 5-22, B-12
Breakpoints, 8-9, 8-17
  table, 8-8, 8-9
Buffer,
  macro, 3-8
  save, 3-8
  text, 3-1, 3-17
Building,
  a memory image, 2-14
Byte, 8-5
  offset, 5-22
BYTE directive, 5-30

Calculating offsets, 8-13
Calling and using,
  ASEMBL, 5-58
  DUMP, E-1
  EDIT, 3-1
  EXPAND, 5-54
  LIBR, 7-1
  LINK, 6-1
  ODT, 8-1
  PATCH, G-1
  PIP, 4-1
  SRCCOM, F-1
Calls or branches to overlay segments, 6-7
Carry bit, 9-10
.CDFN request, 9-18
Centralized queue management system, 9-45
CHAIN bit, 9-6
.CHAIN request, 9-19
Change command, 3-19
Changing,
  device handler characteristics, 2-11
Channel,
  data, 9-52

Channel (cont.)
- number, 9-3
- status word, 9-10, 9-52

Channels, 9-22
Chapter summary, Preface
Character,
- deletion, 9-59
- transfer, 9-59

Character- and line-oriented command
- properties, 3-5

Character-oriented commands, 3-5
Character set, 5-4
- ASCII, B-1
- Radix-50, B-4

Characters,
- illegal, 5-6
- legal, 5-4
- operator, 5-6
- optional, 2-8
- prompting, 2-2
- special, 5-53
- upper-/lower-case, 3-23

Checking channel status, 9-62
Checksum, 4-4,
Clearing breakpoints, restarting ODT, 8-2
Clock,
- frequency, 9-36
- rate, 9-36
- ticks, 9-35

CLOSE command, 2-10
.CLOSE request, 9-21
Closed location, 8-4
Code,
- binary, 1-1
- modifications, G-1
- object, 1-1
- source, 1-1

Combining,
- files, 4-4
- library switch functions, 7-9

Command and Switch Summaries, A-1
Command,
- arguments (EDIT), 3-4, A-3
- continuation switch, 7-2
- decoder, 8-16
- execution routine, 8-16
- interpretation services, 9-1
- mode, 3-1
- repetition, 3-7
- string format, 2-5
- strings, 3-4, 6-1
- structure (EDIT), 3-3
- switches, 4-1, F-1
- syntax (LIBR), 7-1

Command String Interpreter, 2-5, 6-2, 9-21
Command summary,
- EDIT, A-3
- Keyboard monitor, A-1
- ODT, A-8
- PATCH, A-11

Commands,
- and functions, 8-4
- character-oriented, 3-5
- control, 3-2
- edit control, 3-2
- input/output, 3-3, 3-9
- keyboard, 2-8
- line-oriented, 3-5
- PATCH, G-2
- pointer relocation, 3-13
- search, 3-15
- text modification, 3-3, 3-17
- utility, 3-3, 3-20

Commands to allocate system resources, 2-8
- ASSIGN, 2-9
- CLOSE, 2-10
- DATE, 2-8
- INITIALIZE, 2-9
- LOAD, 2-11
- SET, 2-11
- TIME, 2-8
- UNLOAD, 2-11

Commands to manipulate memory
- images, 2-12
- Base, 2-14
- Deposit, 2-15
- Examine, 2-15
- GET, 2-12
- SAVE, 2-16

Comment field, 5-2, 5-3
Common blocks, 6-10
Completion functions, 9-10
Completion routines, 9-10, 9-34, 9-57
Components,
- system software, 1-2

Compress operation, 4-12
Compressing,
- directories, 4-12
- files, 4-12

Concatenation, 5-53
Condition codes, 8-7
Conditional,
- assembly directives, 5-46
- block, 5-46

Configuration word, 9-9, 9-36, 9-56
Confirming file transfers, 4-5
Constant register, 8-3, 8-8, 8-13

Contiguous
  area, 9-9
  file, 7-10, 9-9
Continuation lines, 5-2
Continue switch, 6-15
Control,
  block, 6-1
  commands, 3-2
  parameters, 6-4
  section names, 6-1
Control section,
  named, 6-3
  unnamed, 6-3
Conventions, system, 2-1
Conversion, Octal-Decimal, B-21
Copy operations, 4-3
  errors, 4-4
  multiple, 4-6
Copying,
  files with the current date, 4-5
  system files, 4-4
Co-resident overlay routines, 6-16
Correct and incorrect macro calls, 9-2
Creating,
  a library file, 7-3
CREF, 1-2
  error messages, 5-65
  listing output, 5-63
  specification switches, 5-59
  switches, 5-61, B-20
Cross reference,
  control sections, 5-61
  errors, 5-61
  listings, 1-2
  permanent symbols, 5-61
  register-equate symbols, 5-61
  table generation, 5-61
.CSECT directive, 5-44, 6-2
CSECT, 7-11
CSI,
  error messages, 9-23
  general mode, 9-21
  special mode, 9-24
  switch separators, 9-26
.CSIGEN request, 9-21
.CSISPC request, 9-24
CTRL C, 2-7, 3-2, 8-2
CTRL O, 2-7, 3-2
CTRL Q, 2-7
CTRL S, 2-7
CTRL U, 2-7, 3-2, 8-2
CTRL X, 3-2
CTRL Z, 2-7

Current,
  location counter, 5-2, 5-11
  location pointer, 3-5

Data,
  format, 2-1
  length, 9-53
  storage directives, 5-30
  transfer requests, 9-11
DATE command, 2-8
.DATE request, 9-15
De-activating a channel, 9-44
Debugger, 1-2
Debugging,
  process, 1-2
  tool, 1-2
Decimal,
  number, 5-13
  radix, 5-13, 5-37
Default,
  extensions, 9-22
  FORTRAN library switch, 6-15
  stack, 9-5
Delete,
  command, 3-17
  global switch, 7-6
  operation, 4-7
  switch, 7-5
.DELETE request, 9-29
Deleting files, 4-7
Delimiting characters, 3-21
Deposit command, 2-15
Destination device, 4-3
Device,
  block, 9-4
  block replaceable, 2-4
  designation, 3-10
  file-structured, 2-4
  HT-11 directory-structured, 2-4
  name, logical, 2-2
  name, physical, 2-10, 9-4
  nonfile-structured, 2-4, 6-15
  random access, 2-4, 6-1
  sequential-access, 2-4
  size, 9-31
  specification, 2-6
Device Handlers, 2-5, 9-34
  loading, 9-34
Device names,
  logical, 2-2
  permanent, 2-2
  physical, 2-2

Device structures, 2-4
Direct assignment statement, 5-8, 5-11
  conventions, 5-9
  format, 5-8
Directives,
  assembler, see assembler directives
  conditional assembly, 5-46
  data storage, 5-30
  immediate conditional, 5-49
  listing control, 5-22
  Macro, 5-50
  PAL-11R, 5-50
  PAL-11S, 5-50
  program boundaries, 5-42
  program section, 5-42
  terminating, 5-42
Directory,
  access motion and code consolidation, 9-53
  initialization operation, 4-11
  list operations, 4-10
  segments, 4-11, 4-12
Directories, compressing, 4-12
Documentation conventions, Preface
Double-buffered I/O, 9-69
Double Operand Instructions, B-7
Double Register-Destination, B-15
.DSABL directive, 5-29
.DSTATUS request, 9-30
Dummy,
  argument, 5-53
  argument list, 5-51
  names, 4-8
DUMP, 1-3, E-1
  calling and using, E-1
  error messages, E-5
  switches, E-2, A-10

E command, G-2
Edit Backup command, 3-10
Edit Lower command, 3-23
Edit Read command, 3-9
Edit Upper command, 3-23
Edit Version command, 3-22
Edit Write command, 3-11
Editor (EDIT), 1-2, 3-1
  arguments, 3-4, A-3
  calling and using, 3-1
  command structure, 3-3
  control commands, 3-2
  editing commands, 3-9
  error messages, 3-25
  example, 3-24
  input/output commands, 3-9, A-3
Editor (EDIT) (cont.)
  key commands, 3-2, A-5
  modes of operation, 3-1
  pointer relocation commands, 3-13, A-4
  search commands, 3-15, A-4
  text modification, 3-17
  utility commands, 3-20, A-5
Effective address search, 8-12
Empty entry, 9-9, 9-29
EMT,
  and TRAP addressing, 5-22
  argument blocks, 9-4
  error code, 9-6
  instruction, 9-1
.ENABL directive, 5-29
.END directive, 5-42
End File command, 3-13
.ENDM directive, 5-51
.ENTER request, 9-32
Entering command information, 2-5
Entry point, 5-8, 5-45, 6-5, 6-7, 9-31
  table, 7-4, 7-10, 7-11
Entry symbol, 6-3
.EOT directive, 5-42
Error,
  code, 9-23
  returns, 9-62
Error halt bit, 9-6
Error messages,
  ASEMBL, 5-65
  CREF, 5-67
  DUMP, E-5
  EDIT, 3-25
  EXPAND, 5-64
  keyboard, 9-28
  monitor, 2-19
  PATCH, G-6
  PIP, 4-17
ESCape, 3-2
Evaluation of an expression, 5-14
Even byte, 9-4
.EVEN directive, 5-38
Examine, change locations in the file, G-2
Examine command, 2-15
Examples,
  ASEMBL line printer listing, 5-24
  EDIT, 3-24
  page heading, 5-25
  PATCH, G-4
EX command, 3-13
Exchange command, 3-20
Execute Macro command, 3-22
Exit command, 3-13

Exit from PATCH, G-2
.EXIT request, 9-34
EXPAND, 1-2, 5-1
  calling and using, 5-54
  error messages, 5-64
Expression, B-5
  absolute, 5-14, B-5
  external, 5-14
  register, 5-17, B-5
  relocatable, 5-14, 8-3
Expressions, 5-14
  symbols and, 5-4
Extend and delete operations, 4-7
Extending file lengths, 4-7
Extensions, 3-10, F-1
  and filenames, 2-3
External,
  expression, 5-14
  symbols, 5-8, 6-3

F command, G-2
Facilities,
  for input and output operations, 9-1
Fatal monitor error messages, 2-20
Fatal system boot error messages, 2-19
.FETCH request, 9-34
Field,
  comment, 5-2, 5-3
  label, 5-2
  operand, 5-3
File,
  allocation scheme, 4-5
  ASCII, 3-1
  compressing, 4-12
  descriptor blocks, 9-24
  length, 9-53
  manipulation requests, 9-11
  manipulation services, 9-1
  memory image, 2-1, 4-3, G-1
  names and extensions, 2-3
  non-overlay, G-3
  overlay, G-1
  permanent, 9-9
  specifications (ASEMBL), 5-25
  structure, 9-9
  temporary, 4-16
  tentative, 9-9
  transfer, 4-1
File-structured HT-11 device, 2-4

Filename, 3-10
  blank extensions, 2-4
  extensions, 2-3
  input, 4-1
  output, 4-1
Fill characters, 9-6
Fill count, 9-7
Find command, 3-16
Floating point,
  exception, 9-56
  hardware, 5-39, 9-56
  numbers, 5-14, 5-39
  source double register, B-14
.FLT2 directive, 5-40
.FLT4 directive, 5-40
Format,
  ASCII, 2-1
  control, 5-4
  load image, 2-1, 6-2, 6-15
  memory image, 2-1
  object, 2-1
  of Entry Point Table, 7-11
  of library files, 7-10
  of programmed requests, 9-1
  register, 8-4
  statement, 5-1
Formats, data, 2-1
Formatted binary copy switch, 4-4
Formatting,
  horizontal, 5-4
  vertical, 5-4
Form feed character, 3-1, 3-11
Forms of relocatable expressions, 8-3
Fragmented device, 9-9
Free area, 4-5
Free memory list, 2-5
Function,
  code, 9-4
  control switches, B-20
  switches, 5-59, 5-60

General,
  address specification, 5-17
  library file format, 7-10
  memory layout, 2-5
  registers, 5-9, 8-7
GET command, 2-12
Get command, 3-15
Global symbol directory, 7-11
Global symbol table, 6-1, 6-3

Global symbols, 5-8, 5-45, 6-1, 6-3
Globals, unresolved, 6-1
.GLOBL directive, 5-44
.GTIM request, 9-35
.GTJB request, 9-36

HALT instructions, 2-22
Handler size, 9-31
Handlers,
  device, 2-5
  removing from memory, 9-49
Hardware bootstrap, 4-13
Hardware,
  memory protection, 9-4
.HERR request, 9-37
High,
  address, 2-17
  level languages, 1-2
  memory address, 9-6
  order time of day, 9-36
Horizontal formatting, 5-4
.HRESET request, 9-39
HT-11 directory-structured
  devices, 2-4
HT-11 system, 1-1
  data formats, file transfers, 4-1
  I/O transfers, 9-45
  librarian, 7-1
  memory map, 2-4

.IDENT directive, 5-28
Identification messages, 2-1
.IFF Directive, 5-48
.IFT Directive, 5-48
.IFTF Directive, 5-48
Illegal characters, 5-6, 8-20
Image mode transfer, 4-3
Immediate conditional directive, 5-49
Immediate mode, 1-2, 5-19, 9-1
Important memory areas, 9-4
Include switch, 6-15
Index Mode, 5-19, 5-20
Index Deferred Mode, 5-19
Individual module name, 7-2
INITIALIZE command, 2-9
Input and output, 3-3, 6-4
  commands (Editor), 3-9, A-3
Input,
  filenames, 4-1
  list, 2-6
  source filename, E-1
Insert command, 3-17
Inserting modules into a library, 7-4

Instruction,
  EMT, 9-1
  mnemonic, 5-3
  offset, 8-13
Instructions, B-6
  branch, B-12
  double operand, B-7
  operate, B-10
  rotate/shift, B-8
  single operand, B-8
  trap, B-11
.INTEN request, 9-15
Internal,
  buffers, 3-1
  Macro buffer, 3-22
  registers, 8-7
  symbol directory, 7-11
  symbolic names, 5-44
  symbols, 5-8
Interrupt,
  priority level, 8-8
I/O count, 9-53
  exit routine, 9-8
Iteration argument, 3-6
  loops, 3-7

Job Status Word, 2-19, 9-5, 9-59
Jump command, 3-14
Jumps, 6-7

Key commands, Editor, 3-2, A-5
Key, LINE FEED, 8-5
Keyboard,
  commands, 2-8
  communication (KMON), 2-6
  error messages, 9-28
  monitor (KMON), 2-4
Keyboard Monitor (KMON), 2-4
  command summary, A-1
  special function keys, A-2
Kill command, 3-18

Label field, 5-2
LDA format, 6-2
  switch, 6-15
Legal,
  characters, 5-4
  separating characters, 5-5, 5-53
  wild card, 4-1
Librarian (LIBR), 1-2, 7-1
  calling and using, 7-1
  error messages, 7-11
  switch commands, 7-1, 7-2, A-8

Library
  directory, 7-1
  end trailer, 7-11
  header, 7-1, 7-10
  processing, 7-11
  searches, 6-12
Library files, 6-5
  creation, 7-3
  directory listing, 7-8
  entry point table, 7-6
  format, 7-10
  inserting modules into, 7-4
  merging, 7-8
.LIMIT directive, 5-42
Line- and character-oriented command
  properties, 3-5
Line deletion, 9-59
  formatting, 5-4
  oriented commands, 3-5
  printer overstriking capability, 2-13
LINE FEED key, 8-5
Linker (LINK), 1-1, 1-2, 6-1, 7-11
  calling and using, 6-1
  error handling and messages, 6-18
  input and output, 6-4
  load map, 6-6, 8-1
  switches, 6-2, A-7
Linked,
  list, 9-45
  program, 6-1
Linking, relocation and, 5-16
List command, 3-12
LIST command, H-3
.LIST directive, 5-22
Listing,
  assembly, 8-1
  control directives, 5-22
  control switches, 5-59, B-19
  cross-reference, 1-2
  the directory of a library file, 7-8
Load address,
  absolute, 6-1
LOAD command, 2-11
Load image format (.LDA), 2-1
Load
  map, 1-1, 6-1, 6-5
  module, 1-1, 6-1, 6-4, 6-15
Loading,
  device handlers, 9-34
  memory image files, 2-17
  ODT with user program, 8-1
  root segment, 2-12
Local symbol block, 5-10
Local symbols, 5-10
Location,
  addressed, 8-6
  closed, 8-4
  counter control, 5-37
  open, 8-4
Locations, opening, changing, and
  closing, 8-4
.LOCK request, 9-39
Locking USR in memory, 9-39
Logical device name, 2-2
  identifier, 9-3
  names, 2-2, 2-9
.LOOKUP request, 9-41
Low and high addresses, 5-42
  -order priority, 8-15
  priority, 8-15
Lower-case bit, 9-5

Machine language, 1-1
Macro,
  buffer, 3-8
  call, 9-1
  definition, 5-50
  definition formatting, 5-51
  libraries, 5-54
  recursive, 5-65
Macro command, 3-8, 3-21
.MACRO directive, 5-50
Making patches permanent, 2-14
Manipulating memory images, 2-12
Mask limit, 8-12
Masking, 8-8
Matching areas, F-1
Maximum file size, 9-9
.MCALL directives, 5-54
Memory block initialization, 8-13
Memory,
  diagram, BASIC link with overlay regions, 6-8
  maps, 2-4, 8-2
  usage, 3-8
  usage map, 6-4
Memory image, G-1
  files, 2-1, 4-3, G-1
  format, 2-1
Merging library files, 7-8
Miscellaneous services, 9-12
Missing arguments, 5-53
Mnemonics, 1-1
Mode,
  absolute, 5-20
  autodecrement, 5-18

Mode (cont.)
autodecrement deferred, 5-19
autoincrement, 5-17
autoincrement deferred, 5-18
command, 3-1
general, 9-21
immediate, 1-2, 5-19
index, 5-19, 5-20
index deferred, 5-19
instruction, 8-13
Radix-50, 8-7
register, 5-17
register deferred, 5-17
relative, 5-20
relative branch, 8-6
relative deferred, 5-21
single instruction, 8-11, 8-13
source operand, 5-20
special, 9-24
text, 3-1, 3-17
type-ahead, 2-7
Modes, addressing, 5-16
of operation (EDIT), 3-1
Modify stack address, 6-16
Modules, 6-4
absolute load, 6-4
load, 6-1, 6-4, 6-15, 7-1
object, 6-4, 6-11, 7-1, 7-11
Monitor, 2-1
error messages, 2-19
HALTs, 2-22
keyboard (KMON), 2-4
memory protection map, 6-4
Resident (RMON), 2-4, 9-5
software components, 2-4
start procedure, 2-1
version number, 9-8
MOV instruction, 9-2
Multiple,
command lines, 2-7
copy operations, 4-6
delimiters, 5-3
GETs, 2-12
labels, 5-2
operands, 5-3, 5-30
.QSET requests, 9-45
Multiply-defined symbols, 5-10

[n] construction, 4-5
Named,
COMMON, 6-10
control sections, 6-3
relocatable program sections, 5-44
Negative,
line arguments, 3-6
numbers, 5-14
Next command, 3-11
.NLIST, 5-22
Nonfile-structured,
devices, 2-4, 6-16
Non-overlay file, G-2
Nonexistent symbol, 6-16
Number, B-16
channel, 9-3
decimal, 5-13
monitor version, 9-8
of arguments, 5-53
update, 9-8
Numbers,
assembler, 5-14
floating-point, 5-14, 5-40
negative, 5-14
octal, 5-13
positive, 5-14
Numeric arguments, 3-5, 9-3
Numeric control, 5-39

Object,
code, 1-1
files, 6-4
format, 2-1
modules, 5-16, 6-4, 6-11, 7-1, 7-11
modules, relocatable, 8-1
modules, starting point, 7-10
output, 1-1
.OBJ format, H-1
Octal,
numbers, 5-13
radix, 5-13, 5-37
Octal-Decimal conversions, B-21
.ODD directive, 5-38
Odd (high-order) byte, 9-4
ODT,
(On-line debugging technique), 1-3
break routine, 8-18
calling and using, 8-1
command summary, A-8
error detection, 8-20
functional organization, 8-16
linking ODT with the user program, 8-1
priority level, 8-15
Offset, 5-11, 5-20
relative branch, 8-6
relative branch instruction, 8-13
Offset words, 9-8

On-line debugging technique
  (ODT) – see ODT
Open file, 2-10
Opening a byte address, G-2
Opening, changing, and closing locations, 8-4
Operand field, 5-3
Operands, multiple, 5-30
Operate Instructions, B-10
Operations on files, 4-1
Operator,
  characters, 5-6
  field, 5-3
Operators,
  binary, 5-6, 5-14
  unary, 5-6, 5-37, 5-41
Optional characters, 2-8
Output,
  filenames, 4-1
  format, F-2
  list, 2-6

Page, 3-1
  eject, 5-29, 5-51
  headings, 5-27
.PAGE directive, 5-29
PAL-11R directive, 5-50
PAL-11S conditional assembly directive, 5-50
Parameter list, 2-16
Parameters used as arguments, D-1
Passing switch information, 9-22, 9-26
PATCH utility program, 1-3, G-1
  commands, G-2
  command summary, A-11
  error messages, G-6
  examples, G-4
PATCH,
  calling and using, G-1
Patching,
  new files, G-2
Percent (%) character, 5-10
Peripheral Interchange Program (PIP), see PIP
Permanent,
  device names, 2-2
  files, 2-10, 9-9
Permanent Symbol Table, 5-7
Physical block numbers, E-2
Physical device, 2-9, 9-4
  names, 2-2
PIC (position independent code), 8-14, 9-15
PIP (Peripheral Interchange Program), 1-2, 4-1
  copy operation, 4-3
  error messages, 4-17
PIP (Peripheral Interchange Program) (cont.)
  switches, 4-2
  switch summary, A-6
  warning messages, 4-18
PIP,
  calling and using, 4-1
POINT command, H-2
Pointer location, 3-3
Pointer relocation commands, 3-13
  for Editor, A-4
Position command, 3-16
Position independent code (PIC), 8-14, 9-15
Positive,
  arguments, 3-5
  numbers, 5-14
.PRINT request, 9-42
Printout formats, 8-4
Priority, B-16
Proceed command, 8-10
Proceed count, 8-10
Processor Status, 8-1
Program Boundaries directive, 5-42
Program,
  counter, 5-16, 8-6
  development aids, 1-1
  execution, 8-10
  runaway, 8-18
  section directives, 5-42
  section names, 5-44
  sections, absolute and relocatable, 6-2
Program starting commands, 2-17
  R, 2-18
  REENTER, 2-19
  RUN, 2-17
  START, 2-18
Programmed requests, 2-4, 9-1
  format, 9-1
  summary, D-1
  usage, 9-18
Programming,
  considerations, 8-16
  errors, 1-2
Prompting characters, 2-2
.PROTECT requests, 9-43
.PURGE request, 9-44
Purging an inactive channel, 9-44

.QSET request, 9-45
Quantities, absolute, 5-14

.RADIX directive, 5-36

Radix,
- binary, 5-14, 5-37
- control, 5-36
- decimal, 5-14, 5-37
- octal, 5-14, 5-37
- specification characters, 5-37

Radix-50,
- character set, B-4
- equivalents, B-4
- mode, 8-7
- notation, 5-28
- terminators, 8-8

.RAD50 directive, 5-35
Random-access,
- devices, 2-4, 6-1, 9-31

R Command, 2-18
.RCTRLO request, 9-46
READ command, 3-10
.READ request, 9-46
.READC request, 9-47
.READW request, 9-48
Real arguments, 5-52
Re-assembling, 1-2
Rebooting the system, 4-12
Reclaiming memory, 2-11
Recovering files, 4-8
Recovery from Bad Blocks, 4-14
Recurring coding sequence, 5-50
Reducing disk fragmentation, 4-6
Re-editing, 1-2
Reenter bit, 2-19, 9-5
REENTER command, 2-19, 3-2
Reference line, F-2
.REGDEF, macro call, 9-16
Region number, 6-7
Region, overlay, 6-7, 6-10
Register Deferred Mode, 5-17
Register,
- destination, B-12
- expression, 5-17, B-5
- mnemonic, 9-2
- mode, 5-17
- symbols, 5-9

Register-Offset, B-13
Registers,
- constant, 8-13
- relocation, G-1
- terminal control and status, 9-9

Reinitializing monitor tables, 4-13
Relative,
- branch, 8-12
- branch offset, 8-6

Relative (cont.)
- deferred mode, 5-21
- mode, 5-20

.RELEAS request, 9-49
Releasing USR from memory, 9-40
Relocatable, 1-1, 5-2
- expressions, 5-15, 8-3
- image file, 2-1
- object module, 8-1
- values, 8-14

Relocation, 8-2
- base, 2-14
- bias, 8-2, 8-14
- calculators, 8-15
- constant, 5-2
- directory, 7-11
- register commands, 8-14
- registers, 8-4, G-1

Relocation and Linking, 5-16
Removing,
- handlers from memory, 9-49
- logical assignments, 2-10

Rename operation, 4-9
.RENAME request, 9-50
.REOPEN request, 9-52
Repeat counts, 8-18
Replace switch, 7-5
Requests,
- for data transfer, 9-10
- for file manipulation, 9-10
- for miscellaneous services, 9-10
- requiring the USR, 9-14

Requests, programmed, 9-1
Requests,
- .CDFN, 9-18
- .CHAIN, 9-19
- .CLOSE, 9-21
- .CSIGEN, 9-21
- .CSISPC, 9-24
- .DELETE, 9-29
- .DSTATUS, 9-30
- .ENTER, 9-32
- .EXIT, 9-34
- .FETCH, 9-34
- .GTIM, 9-35
- .GTJB, 9-36
- .HERR, 9-37
- .HRESET, 9-39
- .LOCK, 9-39
- .LOOKUP, 9-41
- .PRINT, 9-42
- .PROTECT, 9-43

Requests (cont.)
.PURGE, 9-44
.QSET, 9-45
.RCTRLO, 9-46
.READ, 9-46
.READC, 9-46
.READW, 9-46
.RELEAS, 9-49
.RENAME, 9-50
.REOPEN, 9-52
.SAVESTATUS, 9-52
.SERR, 9-37
.SETTOP, 9-54
.SFPA, 9-56
.SRESET, 9-57
.TRPSET, 9-58
.TTINR, 9-59
.TTYIN, 9-59
.TTYOUT, 9-60
.TTOUTR, 9-60
.UNLOCK, 9-39
.WAIT, 9-46, 9-62
.WRITC, 9-64
.WRITE, 9-64
.WRITW, 9-64
Reserving storage area, 5-13
Resident monitor (RMON), 2-4, 9-5
Resident overlay handler, 6-11
Restarting ODT, clearing breakpoints, 8-2
Restarting PIP, 4-1
Return,
to previous sequence, 8-6
to monitor, CTRL C, 8-2
Return path, 6-7
Reusing bad blocks, 4-13
Root segment, 2-14, 6-5, 6-10, G-2
Rotate/Shift Instructions, B-8
Rounding numbers, 5-40
Routines, 1-2
RUBOUT, 2-7, 3-2
Rules for user-defined symbols, 5-7
RUN command, 2-17
Running,
the program (ODT), 8-10
Run-time area of memory, 6-5
Run-time overlay handler, 6-9
Run-time overlay handlers and tables, 6-1
Save buffer, 3-8
SAVE command,
(Monitor), 2-16
(Editor), 3-8, 3-20
Save image, 6-1, 6-2
file (SAV), 6-4
format, 2-16, 6-15
.SAVESTATUS request, 9-52
.SBTTL directive, 5-27
Search,
algorithm (ODT), 8-19
commands (EDIT), 3-3, 3-15, A-3
effective address, 8-12
limit (ODT), 8-12
word, 8-12
Searches, 8-11, 8-19
Segment boundaries, 4-11
Separating characters, 5-5
Separator, 2-6
Sequential-access devices, 2-4, 9-31
.SERR request, 9-37
Set Bottom Address, G-3
SET command, 2-11
options, 2-12
Set Relocation Registers, G-3
.SETTOP request, 9-54
.SET USR NOSWAP command, 9-54
.SFPA request, 9-56
Single absolute section, 5-44
Single instruction,
address, 8-11
mode, 8-11
Single load module, 7-1
Single operand instructions, B-8
Size specification, 9-25
Slash, 8-4
Soft error recovery, 9-37
Software,
components, monitor, 2-4
Source compare (SRCCOM), 1-3, F-1
error messages, F-5
switches, F-1, A-11
Source-Double Register, B-14
Source,
code, 1-1
code, macro-free, 5-1
field, 9-2
lines, 5-1
operand mode, 5-21
program, 1-1, 5-1
program format, 5-2
register, B-13
Special characters, 5-53
Special key commands, EDIT, 3-2

Special function keys, 2-7, A-2
  CTRL C, 2-7
  CTRL O, 2-7
  CTRL Q, 2-7
  CTRL S, 2-7
  CTRL U, 2-7
  CTRL Z, 2-7
  RUBOUT, 2-7
Specifying,
  directory segments, 4-12
  extra words per directory entry, 4-12
SRCCOM, see Source Compare
.SRESET request, 9-57
Stack, 9-34
  address, 6-16
  pointer, 6-16, 9-5
Start address, 6-5, 9-5
START Command, 2-18
Start procedure, 2-1
Starting block number, 9-53
Statement,
  format, 5-2
  terminator, 5-2
Statements,
  assembly language, 5-1
  direct assignment, 5-8, 5-11
Status word, 9-30
Stopping points, 1-2
Storage device, 3-1
Subconditionals, 5-48
Subroutine,
  return, B-13
Summary,
  ASEMBL, B-1
  command, B-1
  programmed requests, 9-11
  switch, A-1
Suspending program execution, 9-62
Swapped region, 9-7
Swapping, 9-39
  algorithm, 9-7
Switch description, 6-13
Switch summary, A-1
  ASEMBL/CREF, B-19
  CREF, B-20
  DUMP, A-10
  Librarian, A-8
  Linker, A-7
  PIP, A-6
  SRCCOM, A-11
Switches, 2-5
  CREF, 5-59, 5-61
  function control, 5-59, 5-60, B-20
  listing control, 5-59, B-19
  Macro, 5-59
  PIP, 4-2
Symbol control, 5-44
Symbol table, 5-26
  global, 6-1, 6-3
  overflow, 6-18
  permanent, 5-7
  switch, 6-18
  user, 5-7
  user-defined, 5-2
Symbol, value, 5-8
Symbolic arguments, 5-29
Symbols, 5-7
  entry, 6-3
  external, 5-8, 6-3
  global, 5-8, 5-44, 6-3
  internal, 5-8
  local, 5-10
  multiply-defined, 5-10
  permanent, 5-7
  register, 5-9
  user-defined, 5-2, 5-7
Symbols and expressions, 5-4
.SYNCH, macro call, 9-16
SYSMAC.SML, D-1, 5-50, 9-1
System,
  build operation, 2-1
  communication, 2-1
  communication area, 6-4, 9-5
  concepts, 9-3
  conventions, 2-1
  date, 9-8
  device, 2-18
  device scratch blocks, 9-34
  disk-usage efficiency, 4-6
  files, 4-5
  macros, 9-10
  macro library, C-1
  programs, 1-1
  software components, 1-2
  state, 9-15
System software components, 1-2
  ASEMBL, 1-2, 5-1
  CREF, 1-2, 5-61
  DUMP, 1-3, E-1
  EDIT, 1-2, 3-1

System software components (cont.)
EXPAND, 1-2, 5-50
Librarian, 1-2, 7-1
Linker, 1-2, 6-1
ODT, 1-3, 8-1
PATCH, 1-3, G-1
PIP, 1-2, 4-1
SRCCOM, 1-3, F-1

TAB character, 3-2, 5-2
Table of,
breakpoints, 8-9, 8-10
mode forms and codes, 5-21
proceed command repeat counts, 8-10
Temporary files, 4-16
Temporary numeric control, 5-41
Temporary radix control, 5-37
Tentative entry, 9-32
Tentative file, 9-9, 9-44
Terminal,
input request, 2-14
interrupt, 8-19
normal mode, 9-59
special mode, 9-59
terminal control and status registers, 9-9
Terminate search, 8-2
Terminating directives, 5-42
Terms, 5-14
Testing patches, 2-14
Text, 1-1
blocks, 7-11
buffer, 3-1, 3-17
Editor, 3-1
mode, 3-1, 3-17
modification, 3-3
modification commands, 3-17
TIME command, 2-8
Time of day, access to, 9-35
.TITLE directive, 5-27
Trace trap instruction, 8-17
Trailing delimiter, 5-35
Transfer address, 6-5, 6-18
Transfer,
image mode, 4-3
Transferring characters, 9-59
Transferring files, 4-6
Transferring memory, 2-16
TRAP addressing, EMT and, 5-22
Trap instructions, 8-11, B-11
Trap interception, 9-58
.TRPSET request, 9-58
TT
printer interrupt, 8-20
.TTYIN request, 9-59
.TTINR request, 9-59
.TTYOUT request, 9-60
.TTOUTR request, 9-60
Turning off user error interception, 9-37
Two-volume compress, 4-12
Type ahead, 2-7
Types of programmed requests, 9-10

Unary operators, 5-6, 5-37, 5-41
Unit number (system device), 9-8
UNLOAD command, 2-11
.UNLOCK request, 9-40
Unnamed control section, 6-3
Unrecoverable hardware/software error, 7-12
Unresolved globals, 6-1
Unsave command, 3-21
Unused areas, 4-10
Up-arrow construction, 5-52
Up-arrow, 8-5
Update,
number, 9-8
switch, 7-7
Upper-/lower-case,
commands, 3-23
mode, 3-23
terminal, 3-23
User command string, 3-1
User-defined symbol, 5-2, 5-7
table, 5-2
User library searches, 6-11
User program, 5-2
protection, 9-22
User Service Routine (USR), 2-5
address, 2-17
swapping, 2-12
User switch commands and functions (LIBR), 7-1
User symbol table, 5-7
Using,
libraries, 6-11
overlays, 6-5
the system macro library, 9-10
the wild-card construction, 4-1
USR area, 9-8
load address, 9-6
USR
swap bit, 9-5
swapping, 9-7, 9-53
Utility commands, Editor, 3-3, 3-20, A-5

Utility program, PATCH, G-1
Utility routines, 8-16

Value specifier, H-2, H-3
Vector addresses, 9-4
Verify command, 3-12
Version,
  number message, 4-14, G-1
  switch, 4-14
Vertical formatting, 5-4

.WAIT request, 9-47, 9-62
Warning messages, PIP, 4-18
Wild card,
  construction, 4-4
  expansion, 4-4
Word,
  address, G-2
  search, 8-12, 8-19
.WORD directive, 5-31
.WRITC request, 9-64
Write command, 3-11
.WRITE request, 9-64
.WRITW request, 9-64

1 026 3665 3