This document is the user’s manual for the OS/78 V3 operating system. Its purpose is to acquaint new users with the operating system designed for the DECstation 78/88 minicomputers. This manual presents the background material for getting on the air, and a detailed description of the OS/78 commands that are used to direct computer operations. It also describes the OS/78 Editor, PAL8 assembly language, and the two high-level languages supported by the system, BASIC and FORTRAN IV. Command examples and demonstration programs are contained throughout the manual.


OPERATING SYSTEM AND VERSION: OS/78 V3

To order additional copies of this document, contact the Software Distribution Center, Digital Equipment Corporation, Maynard, Massachusetts 01754
The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

The software described in this document is furnished under a license and may only be used or copied in accordance with the terms of such license.

No responsibility is assumed for the use or reliability of software on equipment that is not supplied by DIGITAL or its affiliated companies.

Copyright © 1979 by Digital Equipment Corporation

The postage-prepaid READER'S COMMENTS form on the last page of this document requests the user's critical evaluation to assist us in preparing future documentation.

The following are trademarks of Digital Equipment Corporation:

DIGITAL DEC DECsystem-10 MASSBUS
DECTape DIBOL OMNIBUS
DECUS EDUSYSTEM OS/8
UNIBUS FLIP CHIP PHA
COMPUTER LABS FOCAL RSTS
COMTEX INDAC RSX
DDT LAB-8 TYPESET-8
DECCOMM DECSYSTEM-20 TYPESET-11
ASSIST-11 RTS-8 TMS-11
VAX VMS ITPS-10
DECnet IAS SBI
DATATRIEVE TRAX PDT
CONTENTS

PREFACE

CHAPTER 1 INTRODUCTION TO THE SYSTEM

1.1 DECSTATION 78/88 MINICOMPUTER SYSTEM HARDWARE
1-1

1.2 OS/78 SOFTWARE
1-2

CHAPTER 2 GETTING ON THE AIR WITH OS/78

2.1 STARTING THE SYSTEM
2-1

2.1.1 Loading the Diskette
2-1

2.1.2 Loading the RL01K Cartridge Disk Pack
2-3

2.1.3 Unloading the RL01K Cartridge Disk Pack
2-5

2.1.4 Starting the System
2-5

2.1.4.1 Setting Terminal Characteristics
2-5

2.1.4.2 Bootstrapping OS/78
2-6

2.1.5 Using the Terminal
2-7

2.2 MAKING A BACKUP COPY OF THE OS/78 SOFTWARE
2-8

2.2.1 DECstations with RX01 and RX02 Disk Systems Only
2-8

2.2.2 DECstations with RL01 Disk Systems Only
2-8

2.2.3 DECstations with RX02 and RL01 Disk Systems
2-9

2.3 DEMONSTRATION PROGRAMS
2-10

2.4 DEVICE AND FILE NAMES
2-10

2.4.1 Devices
2-10

2.4.2 File Names and Extensions
2-11

2.5 ENTERING MONITOR COMMANDS
2-12

2.5.1 OS/78 Command Format
2-13

2.5.2 Incorrect Commands
2-14

2.5.3 Correcting Errors
2-15

2.5.4 Using Input/Output Options
2-15

2.5.4.1 Equal Sign Construction
2-16

2.5.4.2 Slash Construction
2-16

2.5.4.3 Parentheses Construction
2-16

2.5.4.4 Square Bracket Construction
2-16

2.5.4.5 General-Purpose Dash Options
2-16

2.5.5 Remembering Previous Arguments
2-17

2.5.6 Using Wildcards
2-18

2.5.6.1 Wildcard Input File Specifications
2-19

2.5.6.2 Wildcard Output File Specifications
2-19

2.5.7 Indirect Commands (At Symbol (@) Construction)
2-20

2.5.8 Asterisk (*) Prompting Symbol
2-21

2.6 USING DEFAULTS
2-21

2.7 WHERE TO GET MORE INFORMATION
2-22

2.8 FILE-STRUCTURED DEVICES
2-23

2.8.1 Using RX01 and RX02 Diskettes
2-24

2.8.2 Using the RL01 Cartridge Disk Pack
2-24

2.8.3 Using the VXAO Extended-Memory Device
2-25

2.9 FILES
2-25
2.9.1 File Directories 2-25
2.9.2 File Types 2-25
2.9.3 ASCII File Format 2-26
2.10 OS/78 COMMAND SUMMARY 2-26

CHAPTER 3 OS/78 COMMANDS 3-1
3.1 ASSIGN COMMAND 3-2
3.2 BASIC COMMAND 3-3
3.3 BOOT COMMAND 3-4
3.4 CANCEL COMMAND 3-5
3.5 COMPARE COMMAND 3-6
3.5.1 Using COMPARE's Options 3-8
3.6 COMPILE COMMAND 3-10
3.7 COPY COMMAND 3-12
3.7.1 File Transfer 3-12
3.7.2 Examples of COPY Commands 3-12
3.7.3 Predeletion 3-13
3.7.4 Postdeletion 3-13
3.7.5 Considering the Date on a Transfer 3-14
3.7.6 File Protection During a Transfer Operation 3-14
3.7.7 Terminating COPY Operations 3-15
3.8 CREATE COMMAND 3-18
3.9 CREF COMMAND 3-19
3.10 DATE COMMAND 3-21
3.11 DEASSIGN COMMAND 3-22
3.12 DELETE COMMAND 3-23
3.13 DIRECT COMMAND 3-25
3.13.1 Directory Listing Selected by Date 3-26
3.13.2 Directory Format Selection 3-27
3.14 DUPLICATE COMMAND 3-28
3.14.1 Changing Devices Before and After Executing the DUPLICATE Command 3-28
3.14.2 Performing a Read Check 3-29
3.14.3 Transfer Without Checking for Identical Contents 3-29
3.14.4 Check for Identical Contents Without Transferring 3-30
3.14.5 Formatting Diskettes (RX02 Drives Only) 3-30
3.15 EDIT COMMAND 3-31
3.16 EXECUTE COMMAND 3-34
3.17 FORMAT COMMAND 3-35
3.18 GET COMMAND 3-37
3.19 HELP COMMAND 3-38
3.20 LIST COMMAND 3-40
3.21 LOAD COMMAND 3-42
3.21.1 PAL8 Absolute Binary Files 3-42
3.21.1.1 Concatenated Programs (/S option) 3-42
3.21.1.2 Loading Programs in Specified Areas (/ff and =ffnnnn options) 3-42
3.21.1.3 Programs not Using Fields 0 and 1 (/8 and /9 options) 3-42
3.21.1.4 Editing or Patching a SAVE File (/I option) 3-43
3.21.1.5 Execution After Loading (/G option) 3-43
3.21.1.6 Clearing Memory After Loading (/R option) 3-44
3.21.2 FORTRAN Relocatable Binary Files 3-44
3.21.2.1 Specifying Additional Files (/C option) 3-44
3.21.2.2 Including System Symbols in the Symbol Map (/S option) 3-44
3.21.2.3 Execution After Loading (/G option) 3-45
3.22 MAP COMMAND 3-46
3.23 MEMORY COMMAND 3-50
3.24 ODT COMMAND 3-52
3.25 PAL COMMAND 3-53
3.26 QUEUE COMMAND 3-54
3.27 R COMMAND 3-55
3.28 RENAME COMMAND 3-56
3.29 REQUEST COMMAND 3-57
3.30 RUN COMMAND 3-58
3.31 SAVE COMMAND 3-59
3.31.1 Restrictions on Arguments of the SAVE Command 3-60
3.32 SET COMMAND 3-60
3.32.1 SET Command Forms 3-62
3.32.1.1 ARROW 3-62
3.32.1.2 COL n 3-63
3.32.1.3 ECHO 3-63
3.32.1.4 ESC 3-64
3.32.1.5 HEIGHT m 3-64
3.32.1.6 INIT cmd 3-64
3.32.1.7 LC 3-64
3.32.1.8 PAGE 3-65
3.32.1.9 PAUSE 3-65
3.32.1.10 READONLY 3-65
3.32.1.11 SCOPE 3-66
3.32.1.12 WIDTH = n 3-66
3.32.1.13 HANDLER 3-66
3.33 SQUISH COMMAND 3-70
3.34 START COMMAND 3-71
3.35 SUBMIT COMMAND 3-72
3.35.1 Processing and Terminating a Batch Input File 3-72
3.35.2 Spooling 3-73
3.36 TERMINATE COMMAND 3-75
3.37 TYPE COMMAND 3-76
3.37.1 Using TYPE Options 3-77
3.38 UA, UB, UC COMMANDS 3-78
3.39 ZERO COMMAND 3-79

CHAPTER 4 THE SYMBOLIC EDITOR 4-1

4.1 INTRODUCTION 4-1
4.2 CALLING THE EDITOR 4-1
4.2.1 Creating a New File -- The CREATE Command 4-1
4.2.2 Editing an Existing File -- The EDIT Command 4-2
4.3 MODES OF OPERATION 4-2
4.3.1 Text Mode 4-3
4.3.2 Command Mode 4-5
4.3.2.1 Input Commands 4-5
4.3.2.2 Listing Commands 4-6
4.3.2.3 Output Commands 4-7
4.3.2.4 Editing Commands 4-9
4.3.2.5 Search Commands 4-10
4.3.2.6 Special Command Mode Characters 4-11
4.4 SEARCHING A TEXT 4-13
4.4.1 Single-Character Search -- The S Command 4-13
4.4.2 The Character String Search 4-14
4.4.2.1 Intrabuffer String Search 4-14
4.4.2.2 Interbuffer String Search -- J Command 4-17
4.5 A SAMPLE EDITING JOB 4-18
4.6 EDITOR OPTIONS 4-21
4.7 EDITOR ERROR MESSAGES 4-21
4.8 SUMMARY OF EDITOR COMMANDS AND SPECIAL CHARACTERS 4-23

CHAPTER 5 THE PAL8 ASSEMBLER 5-1
CHAPTER 6  BASIC

6.1 OVERVIEW 6-1
6.1.1 Writing a BASIC Program 6-1
6.1.2 The BASIC Character Set 6-1
6.1.3 Using BASIC 6-3
6.2 BASIC COMMANDS 6-3
6.2.1 Entering a New Program -- the NEW Command 6-4
6.2.2 Calling for an Old Program -- the OLD Command 6-4
6.2.3 Running a Program -- the RUN Command 6-5
6.2.4 Displaying a Program -- the LIST Command 6-5
6.2.5 Deleting Text -- the DELETE Command 6-7
6.2.6 Substituting Text -- the EDIT Command 6-7
6.2.7 Automatic Line Numbering -- the SEQUENCE Command 6-8
6.2.8 Merging Text -- the WEAVE Command 6-9
6.2.9 Storing a Program -- the SAVE Command 6-10
6.2.10 Renaming a Program -- the NAME Command 6-10
6.2.11 Erasing the Workspace -- the SCRATCH Command 6-11
6.2.12 Leaving BASIC -- the BYE Command 6-11
6.2.13 Resequencing a Program -- Calling RESEQ 6-11
6.2.14 Key Commands 6-12
6.2.14.1 Correcting Typing and Format Errors -- DELETE, CTRL/U 6-12
6.2.14.2 Eliminating Program Lines -- RETURN 6-12
6.2.14.3 Interrupting Program Execution -- CTRL/C 6-12
6.2.14.4 Controlling Program Listings on the Terminal -- CTRL/S, CTRL/Q, and CTRL/O 6-12
6.3 ELEMENTS OF BASIC 6-13
6.3.1 Constants 6-13
6.3.1.1 Numeric Constants 6-13
6.3.1.2 String Constants 6-14
6.3.2 Variables 6-14
6.3.2.1 Numeric Variables 6-15
6.3.2.2 String Variables 6-15
6.3.2.3 Subscripted Variables 6-16
6.3.3 Expressions 6-17
6.3.3.1 Arithmetic Expressions 6-17
6.3.3.2 Relational Expressions 6-18
6.3.3.3 String Concatenation 6-18
6.4 OS/78 COMMERCIAL ARITHMETIC 6-19
6.4.1 BASIC String Numbers 6-20
6.4.2 Truncation Versus Rounding Off 6-21
6.4.3 Using Relational Operators with String Numbers 6-21
6.5 FORMATTING BASIC STATEMENTS 6-22
6.6 THE ASSIGNMENT STATEMENT -- LET 6-23
6.7 THE DIMENSION STATEMENT 6-24
6.8 THE COMMENT STATEMENT -- REMARK AND EXCLAMATION POINT 6-26
6.9 INPUT AND OUTPUT 6-26
6.9.1 Input 6-26
6.9.2 READ, DATA, and RESTORE 6-28
6.9.3 PRINT 6-29
6.9.3.1 Printing Zones -- Format Control Characters 6-30
6.9.3.2 Printing Numbers and Strings 6-31
6.9.3.3 Printing with the TAB and PNT Functions 6-31
6.9.4 PRINT USING 6-32
6.9.4.1 Representing Numeric Fields 6-33
6.9.4.2 Representing Special Functions 6-34

vii
| 6.9.4.3 | Letters and Numbers | 6-35 |
| 6.9.4.4 | Printing Format String Characters | 6-35 |
| 6.9.4.5 | Numeric Field List Use | 6-36 |
| 6.10 | CONTROL STATEMENTS | 6-36 |
| 6.10.1 | Unconditional Transfer -- GOTO | 6-36 |
| 6.10.2 | Conditional Transfer -- IF GOTO | 6-37 |
| 6.10.3 | Semiconditional Transfer -- ON GOTO | 6-38 |
| 6.10.4 | Looping -- FOR, STEP, and NEXT | 6-39 |
| 6.10.4.1 | Nested Loops | 6-40 |
| 6.10.5 | Stopping -- END and STOP | 6-41 |
| 6.10.6 | Jumping to Subroutines -- GOSUB and RETURN | 6-42 |
| 6.10.7 | Semiconditional Jump to Subroutine -- ON GOSUB | 6-43 |
| 6.11 | FUNCTIONS | 6-43 |
| 6.11.1 | Numeric Functions | 6-44 |
| 6.11.1.1 | Calculating Sine -- SIN | 6-45 |
| 6.11.1.2 | Calculating Cosine -- COS | 6-45 |
| 6.11.1.3 | Calculating the Arctangent -- ATN | 6-45 |
| 6.11.1.4 | Calculating the Tangent -- TAN | 6-46 |
| 6.11.1.5 | Finding the Square Root -- SQR | 6-46 |
| 6.11.1.6 | The Exponential Function -- EXP | 6-46 |
| 6.11.1.7 | Calculating the Natural Logarithm -- LOG | 6-47 |
| 6.11.1.8 | The Integer Function -- INT | 6-47 |
| 6.11.1.9 | The Absolute Value Function -- ABS | 6-48 |
| 6.11.1.10 | The Sign Function -- SGN | 6-48 |
| 6.11.1.11 | Random Numbers -- RND | 6-48 |
| 6.11.2 | String Functions | 6-49 |
| 6.11.2.1 | Finding the Length of a String -- LEN | 6-50 |
| 6.11.2.2 | Finding a Substring -- POS | 6-51 |
| 6.11.2.3 | Returning a String -- SEG$ | 6-52 |
| 6.11.2.4 | Converting a Character to ASCII Code -- ASC | 6-52 |
| 6.11.2.5 | Converting ASCII Code to a Character -- CHR$ | 6-53 |
| 6.11.2.6 | Converting Numbers from String to Numeric Format | 6-53 |
| 6.11.2.7 | Converting a Number to a String -- STR$ | 6-54 |
| 6.11.2.8 | Converting Lower to Upper Case | 6-54 |
| 6.11.2.9 | Converting Octal to Decimal -- OCT | 6-54 |
| 6.11.2.10 | Converting Decimal to Octal -- OCS$ | 6-55 |
| 6.11.2.11 | Returning a String Integer -- FIX$ | 6-55 |
| 6.11.3 | Issuing Commands to the OS/78 Monitor -- CCL | 6-55 |
| 6.11.4 | The PNA Function and the DEF Statement | 6-56 |
| 6.11.5 | The Debugging Function -- TRC | 6-57 |
| 6.11.6 | Calling for the Date -- the DAT$ Function | 6-57 |
| 6.11.7 | Logical Functions | 6-58 |
| 6.11.7.1 | AND | 6-58 |
| 6.11.7.2 | OR | 6-58 |
| 6.11.8 | The KEYS Function | 6-59 |
| 6.11.9 | The CUR$ Function | 6-59 |
| 6.11.10 | The PMS$ Function | 6-60 |
| 6.11.11 | The COL Function | 6-60 |
| 6.11.12 | Printing Functions | 6-60 |
| 6.11.12.1 | The TAB Function | 6-61 |
| 6.11.12.2 | The PNT Function | 6-61 |
| 6.12 | FILE STATEMENTS | 6-62 |
| 6.12.1 | File Control | 6-63 |
| 6.12.1.1 | Opening a File | 6-63 |
| 6.12.2 | File I/O | 6-64 |
| 6.12.2.1 | Reading Data from a File -- INPUT$ | 6-65 |
| 6.12.2.2 | Writing Data on a File -- PRINT$ | 6-65 |
| 6.12.2.3 | Formatting Output to a File -- PRINT$ USING | 6-67 |
| 6.12.2.4 | Resetting a File -- RESTORE$ | 6-67 |
| 6.12.2.5 | Checking for Open Files -- IF OPEN$ | 6-68 |
| 6.12.2.6 | Checking for End-of-File -- the IF END$ Statement | 6-68.1 |
6.13  DIRECT RECORD I/O  6-69
6.13.1  Specifying Record-Size with FILEV#  6-69
6.13.2  Opening a File for Record I/O  6-70
6.13.3  Defining Record Fields  6-70
6.13.4  Reading and Updating Records -- PUT# and GET#  6-71
6.13.5  CLOSE# Statement  6-71
6.13.6  Working with Record I/O  6-72
6.14  SEGMENTING PROGRAMS -- THE CHAIN STATEMENT  6-74
6.15  BASIC ESCAPE SEQUENCES  6-76
6.15.1  Calling for ESCape Sequences with the PNT Function  6-76
6.15.2  Using the KEY$ Function  6-77
6.15.3  Defining a User Function  6-78
6.16  CREATING A MEMORY-IMAGE OF A BASIC PROGRAM WITH COMPILE  6-79
6.17  BASIC OPTIONS  6-79
6.18  INTERACTION BETWEEN BASIC AND BATCH  6-80
6.19  BASIC OVERLAYS -- BASIC.OV  6-80
6.20  SUMMARY OF BASIC COMMANDS  6-81
6.21  SUMMARY OF BASIC STATEMENTS  6-81
6.22  SUMMARY OF BASIC FUNCTIONS  6-83
6.23  ERROR MESSAGES  6-86
6.23.1  Editor Error Messages  6-86
6.23.2  Compiler Error Messages  6-87
6.23.3  Run-Time System Error Messages  6-87

CHAPTER 7  FORTRAN IV

7.1  OVERVIEW  7-1
7.1.1  The COMPILER  7-4
7.1.1.1  Compiler Options  7-5
7.1.1.2  Compiler Error Messages  7-5
7.1.2  The LOADER  7-7
7.1.2.1  Specifying I/O Devices  7-7
7.1.2.2  Running Subprograms  7-8
7.1.2.3  LOADER Options  7-10
7.1.2.4  Loader Error Messages  7-11
7.1.3  The Run-Time System (FRTS)  7-12
7.1.3.1  Run-Time System Options  7-15
7.1.3.2  Run-Time System Error Messages  7-17
7.1.4  The Library  7-19
7.2  THE FORTRAN IV SOURCE LANGUAGE  7-19
7.2.1  The FORTRAN Character Set  7-20
7.2.2  Elements of a FORTRAN Program  7-21
7.2.2.1  Statements  7-21
7.2.2.2  Comments  7-22
7.2.3  FORTRAN Lines  7-22
7.2.3.1  Using a Text Editor  7-22
7.2.3.2  Statement Label Field  7-23
7.2.3.3  Comment Indicator  7-23
7.2.3.4  Continuation Indicator Field  7-24
7.2.3.5  Statement Field  7-24
7.2.3.6  Identification Field  7-25
7.2.4  Blank Lines  7-25
7.2.5  Line Format Summary  7-25
7.3  FORTRAN STATEMENT COMPONENTS  7-26
7.3.1  Symbolic Names  7-26
7.3.2  Data Types  7-27
7.3.3  Constants  7-27
7.3.3.1  Integer Constants  7-28
7.3.3.2  Real Constants  7-28
7.3.3.3  Logical Constants  7-30
7.3.3.4  Octal Constants  7-30
7.3.3.5  Hollerith Constants and Alphanumeric Literals  7-31
7.3.4  Variables  7-32
7.3.4.1  Data Type Specification  7-33
7.3.4.2  Default Data Types  7-33
7.3.5  Arrays  7-33
7.3.5.1  Array Declarations  7-34
7.3.5.2  Array Storage (Order of Subscript Progression)  7-35
7.3.5.3  Subscripts  7-36
7.3.5.4  Data Type of an Array  7-37
7.3.5.5  Array Reference Without Subscripts  7-37
7.3.5.6  Adjustable Arrays  7-37
7.4  EXPRESSIONS  7-38
7.4.1  Arithmetic Expressions  7-38
7.4.1.1  Rules for Writing Arithmetic Expressions  7-39
7.4.1.2  Evaluation Hierarchy  7-40
7.4.1.3  Data Type of an Arithmetic Expression  7-40
7.4.2  Relational Expressions  7-41
7.4.3  Logical Expressions  7-42
7.4.3.1  Logical Operator Hierarchy  7-43
7.4.4  Use of Parentheses  7-44
7.5  ASSIGNMENT STATEMENTS  7-45
7.5.1  Arithmetic Assignment Statement  7-45
7.5.2  Logical Assignment Statements  7-46
7.6  SPECIFICATION STATEMENTS  7-46
7.6.1  Type Declaration Statements  7-47
7.6.2  DIMENSION Statement  7-47
7.6.3  EXTERNAL Statement  7-48
7.6.4  COMMON Statement  7-50
7.6.5  EQUIVALENCE Statement  7-52
7.6.5.1  Making Arrays Equivalent  7-53
7.6.5.2  EQUIVALENCE and COMMON Interaction  7-54
7.7  DATA STATEMENTS AND BLOCK DATA SUBPROGRAMS  7-54
7.8  CONTROL STATEMENTS  7-56
7.8.1  GOTO Statements  7-57
7.8.1.1  Unconditional GOTO Statement  7-57
7.8.1.2  Computed GOTO Statement  7-58
7.8.1.3  ASSIGN and ASSIGNED GOTO Statement  7-58
7.8.2  IF Statements  7-60
7.8.2.1  Arithmetic IF Statement  7-60
7.8.2.2  Logical IF Statement  7-62
7.8.3  DO Statement  7-62
7.8.3.1  DO Iteration Control  7-63
7.8.3.2  Nested DO Loops  7-64
7.8.3.3  Control Transfers in DO Loops  7-65
7.8.3.4  Extended Range  7-65
7.8.4  CONTINUE Statement  7-66
7.8.5  PAUSE Statement  7-67
7.8.6  STOP Statement  7-68
7.8.7  END Statement  7-68
7.9  SUBPROGRAMS  7-69
7.9.1  Subprogram Arguments  7-69
7.9.2  User-Written Subprograms  7-70
7.9.2.1  Arithmetic Statement Functions (ASE)  7-70
7.9.2.2  FUNCTION Subprogram  7-72
7.9.2.3  SUBROUTINE Subprograms  7-73
7.9.3  CALL Statement  7-74
7.9.4  RETURN Statement  7-75
7.9.5  FORTRAN Library Functions  7-76
7.10  INPUT/OUTPUT STATEMENTS  7-76
7.10.1  Defining I/O Operations  7-76
7.10.1.1  Input/Output Devices and Logical Unit Numbers  7-77
7.10.1.2 Format Specifiers 7-77
7.10.1.3 Input/Output Record Transmission 7-77
7.10.2 Input/Output Lists 7-77
7.10.2.1 Simple Lists 7-78
7.10.2.2 Implied DO Lists 7-78
7.10.3 Input/Output Forms 7-80
7.10.3.1 Unformatted Sequential Input/Output 7-80
7.10.3.2 Formatted Sequential Input/Output 7-80
7.10.3.3 Unformatted Direct Access Input/Output 7-80
7.10.4 READ Statements 7-80
7.10.4.1 Unformatted Sequential READ Statement 7-80
7.10.4.2 Formatted Sequential READ Statement 7-81
7.10.4.3 Unformatted Direct Access READ Statement 7-82
7.10.5 WRITE Statements 7-83
7.10.5.1 Unformatted Sequential WRITE Statement 7-83
7.10.5.2 Formatted Sequential WRITE Statement 7-84
7.10.5.3 Unformatted Direct Access WRITE Statement 7-85
7.10.6 Auxiliary Input/Output Statements 7-85
7.10.6.1 BACKSPACE Statement 7-86
7.10.6.2 DEFINE FILE Statement 7-86
7.10.6.3 END FILE Statement 7-87
7.10.6.4 REWIND Statement 7-87
7.11 FORMAT STATEMENTS 7-88
7.11.1 Field Descriptors 7-89
7.11.1.1 I Field Descriptor 7-90
7.11.1.2 F Field Descriptor 7-91
7.11.1.3 E Field Descriptor 7-92
7.11.1.4 G Field Descriptor 7-93
7.11.1.5 L Field Descriptor 7-94
7.11.1.6 A Field Descriptor 7-95
7.11.1.7 H Field Descriptor 7-96
7.11.1.8 Alphanumeric Literals 7-97
7.11.1.9 X Field Descriptor 7-97
7.11.1.10 T Field Descriptor 7-97
7.11.2 Scale Factor 7-99
7.11.3 Grouping and Group Repeat Specifications 7-100
7.11.4 Carriage Control 7-100
7.11.5 Format Specification Separators 7-101
7.11.6 Short Field Termination 7-102
7.11.7 Format Control Interaction with Input/Output Lists 7-103
7.11.8 Summary of Rules for Format Statements 7-103
7.12 LIBRARY FUNCTIONS AND SUBROUTINES 7-105
7.12.1 ABS (Single-Precision Absolute Value) 7-105
7.12.2 ACOS (Single-Precision Arc-Cosine Function) 7-105
7.12.3 AINT (Single-Precision Floating-Point to Integer) 7-106
7.12.4 ALOG (Single-Precision Natural Logarithm) 7-106
7.12.5 ALOG10 (Single-Precision Common Logarithm) 7-106
7.12.6 AMAX (Single-Precision Maximum Value) 7-106
7.12.7 AMAX1 (Single-Precision Maximum Value) 7-106
7.12.8 AMIN (Single-Precision Minimum Value) 7-106
7.12.9 AMIN1 (Single-Precision Minimum Value) 7-106
7.12.10 AMOD (Single-Precision A Modulo B) 7-106
7.12.11 ASIN (Single-Precision Arc-Sine) 7-107
7.12.12 ATAN (Single-Precision Arc-Tangent) 7-107
7.12.13 ATAN2 (Single-Precision Arc-Tangent of Two Arguments) 7-107
7.12.14 CGET (Character Get Subroutine) 7-107
7.12.15 CHKEOF (Check for End-of-File Subroutine) 7-108
7.12.16 CLOCK (Initialize Clock Subroutine) 7-108
7.12.17 COS (Single-Precision Cosine Function) 7-108
COSH (Single-Precision Hyperbolic Cosine Function) 7-108
CPU (Character Put Subroutine) 7-109
DATE (OS/78 Date Subroutine) 7-109
DIM (Single-Precision Positive Real Difference) 7-109
EXP (Single-Precision Exponential Function) 7-110
FLOAT (Integer-to-Floating-Point Conversion) 7-110
IABS (Integer Absolute Value Function) 7-110
IDIM (Integer Positive Difference Function) 7-110
IFIX (Single-Precision Floating-Point-to-Integer Function) 7-110
INT (Single-Precision Floating-Point-to-Integer) 7-110
ISIGN (Integer Transfer of Sign Function) 7-110
MAX0 (Single-Precision Maximum Value) 7-111
MAX1 (Single-Precision Maximum Value) 7-111
MIN0 (Single-Precision Minimum Value Function) 7-111
MIN1 (Single-Precision Minimum Value Function) 7-111
MOD (Integer A Modulo B Function) 7-111
SIGN (Single-Precision Transfer of Sign) 7-111
SIN (Single-Precision Sine Function) 7-111
SINH (Single-Precision Hyperbolic Sign) 7-112
SQRT (Single-Precision Square Root Function) 7-112
SINH (Single-Precision Hyperbolic Tangent) 7-112
TIME (Read Time of Day) 7-112
FORTRAN LANGUAGE SUMMARY 7-112

CHAPTER 8
BATCH 8-1

8.1 BATCH PROCESSING UNDER OS/78 8-1
8.2 BATCH COMMANDS 8-2
8.3 THE BATCH INPUT FILE 8-4
8.4 BATCH ERROR MESSAGES 8-6
8.5 RESTRICTIONS UNDER OS/78 BATCH 8-8

CHAPTER 9
OCTAL DEBUGGING TECHNIQUE (ODT) 9-1

9.1 ODT FEATURES 9-1
9.2 CALLING AND USING ODT 9-1
9.3 ODT COMMANDS 9-3
9.3.1 Special Characters 9-3
9.3.1.1 Slash (/) 9-3
9.3.1.2 RETURN -- Close Location 9-3
9.3.1.3 LINE FEED -- Close Location 9-3
9.3.1.4 Semicolon (;) -- Change Location, Close It and Open Next 9-4
9.3.1.5 nnnn+ and nnnn- -- Open Location Relative to Another 9-4
9.3.1.6 Circumflex (^) -- Close Location, Take Contents as Memory Reference Instruction and Open Effective Address 9-4
9.3.1.7 Underline (_) -- Close Location, Open Indirectly 9-4
9.3.2 Illegal Characters 9-5
9.3.3 Control Commands 9-5
9.3.3.1 ffnnnm -- Transfer Control to Program at Location nnnn of Field ff 9-5
9.3.3.2 ffnnnmB -- Set Breakpoint at Location nnnn of Field ff 9-5
F.2.1     Line Printer                  F-3
F.2.2     File-Structured Devices    F-4
F.2.3     Terminal Handlers          F-5
F.2.4     Multiple Input Files       F-5

APPENDIX G  ERROR MESSAGE SUMMARY     G-1
            G.1     SYSTEM HALTS          G-1
            G.2     ERROR MESSAGES      G-2

APPENDIX H  OS/78 MULTIFUNCTION OPERATION   H-1
            H.1     SYMBIONT COMMANDS  H-1
            H.2     SPOOLER COMMANDS  H-2
            H.3     CUSP CODE CONVENTION H-2
            H.4     WRITING A SYMBIONT H-3

APPENDIX I  FULL WORD COMMAND SWITCHES    I-1

APPENDIX J  DECstation HARDWARE CONFIGURATION SUMMARY J-1

FIGURES

FIGURE    2-1  Handling the Diskette        2-1
          2-2  Loading the Diskette             2-2
          2-3  Loading the RL01K Disk Pack      2-4
          7-1  Main Program and Subprogram for Calculating
                the Volume of a Regular Polyhedron  7-8
          7-2  Array Representation             7-35
          7-3  Array Storage                    7-36
          7-4  Equivalence of Array Storage     7-53
          7-5  Legal and Illegal Common Extensions 7-54
          7-6  Nesting of DO Loops              7-64
          7-7  Control Transfers and Extended Range 7-66

TABLES

TABLE   2-1  Terminal Operating Parameters  2-6
        2-2  OS/78 Permanent Logical Device Names 2-10
        2-3  OS/78 Extensions                     2-12
        2-4  General-Purpose Dash Options         2-17
        2-5  Storage Capacity of OS/78 File-Structured
             Devices                                2-23
        2-6  OS/78 Command Summary                 2-27
        3-1  COMPARE Options                       3-6
        3-2  COMPARE Error Messages                3-9
        3-3  COPY Options                          3-16
        3-4  COPY Error Messages                    3-17
        3-5  CREF Options                          3-19
        3-6  CREF Error Messages                    3-20
        3-7  DELETE Options                        3-24
        3-8  DELETE Error Messages                  3-24
        3-9  DIRECT Options                        3-25
        3-10 DIRECT Error Messages                  3-27
        3-11 DUPLICATE Options                     3-30
        3-12 DUPLICATE Error Messages               3-31
        3-13 FORMAT Error Messages                  3-36
        3-14 OS/78 Help Command Arguments           3-39
| 3-15 | HELP Error Messages                                      | 3-39 |
| 3-16 | LIST Options                                             | 3-41 |
| 3-17 | LIST Error Messages                                      | 3-41 |
| 3-18 | MAP Options                                              | 3-48 |
| 3-19 | MAP Error Messages                                       | 3-49 |
| 3-20 | RENAME Options                                           | 3-56 |
| 3-21 | Job Status Word                                          | 3-60 |
| 3-22 | Summary of SET Command Forms                            | 3-62 |
| 3-23 | OS/78 Device Handlers                                    | 3-67 |
| 3-24 | SET Error Messages                                       | 3-68 |
| 3-25 | SUBMIT Options                                           | 3-72 |
| 3-26 | TYPE Options                                             | 3-76 |
| 3-27 | TYPE Error Messages                                      | 3-77 |
| 4-1  | Editor Key Control Commands                              | 4-3  |
| 4-2  | Editor Input Commands                                    | 4-6  |
| 4-3  | Editor Listing Commands                                  | 4-6  |
| 4-4  | Editor Output Commands                                   | 4-7  |
| 4-5  | Editing Commands: Deletion and Alteration                | 4-9  |
| 4-6  | Editor Search Commands                                   | 4-10 |
| 4-7  | Editor Special Characters: Command Mode                  | 4-11 |
| 4-8  | Aborting Editor String Search Commands                   | 4-18 |
| 4-9  | Nonfatal Editor Error Messages                           | 4-22 |
| 4-10 | Major Editor Error Codes                                 | 4-22 |
| 4-11 | Editor Command and Special Characters                    | 4-23 |
| 5-1  | PAL8 Options                                             | 5-9  |
| 5-2  | Use of Arithmetic Operators                              | 5-19 |
| 5-3  | PAL8 Diagnostic Error Codes                              | 5-40 |
| 7-1  | FORTRAN IV Compiler Run-Time Options                     | 7-5  |
| 7-2  | Compiler Error Messages                                  | 7-5  |
| 7-3  | Loader Run-Time Options                                  | 7-10 |
| 7-4  | Loader Error Messages                                    | 7-11 |
| 7-5  | Run-Time System Options                                  | 7-16 |
| 7-6  | Run-Time System Error Messages                           | 7-17 |
| 7-7  | FORTRAN Statement Categories                             | 7-20 |
| 7-8  | FORTRAN Special Characters                               | 7-21 |
| 7-9  | Field Summary                                            | 7-25 |
| 7-10 | Classes of Symbolic Name                                 | 7-27 |
| 7-11 | FORTRAN Data Types                                       | 7-27 |
| 7-12 | Arithmetic Operators                                     | 7-39 |
| 7-13 | Base/Exponent Combinations                               | 7-39 |
| 7-14 | Binary Operator Evaluation Hierarchy                     | 7-40 |
| 7-15 | Relational Operators                                     | 7-41 |
| 7-16 | Logical Operators                                        | 7-42 |
| 7-17 | Logical Operator Hierarchy                              | 7-43 |
| 7-18 | Arithmetic IF Transfers                                  | 7-61 |
| 7-19 | Effect of Data Magnitude on G Format Conversions         | 7-93 |
| 7-20 | Character Storage                                        | 7-95 |
| 7-21 | Carriage Control Characters                              | 7-101|
| 8-1  | BATCH Run-Time Options                                   | 8-2  |
| 8-2  | BATCH Commands                                           | 8-3  |
| 8-3  | BATCH Error Messages                                     | 8-7  |
| 9-1  | ODT Command Summary                                      | 9-9  |
| C-1  | Summary of USR Functions                                 | C-2  |
CHAPTER 1

INTRODUCTION TO THE SYSTEM

The OS/78 operating system is a software product designed for the DECstation 78 and DECstation 88 Minicomputer systems. OS/78 permits you to use a variety of peripheral devices including: video and hard copy terminals, cartridge and flexible media disk drives, line and letter quality printers, and up to 32K words of memory.

OS/78 offers a versatile set of keyboard commands that supervise a comprehensive library of system programs. These programs allow you to develop your application programs using the PAL8 assembly language, or the higher level BASIC, and FORTRAN IV languages.

The system also has a batch stream processor, called BATCH, that provides for unattended (batch-mode) system operation.

1.1 DECSTATION 78/88 MINICOMPUTER SYSTEM HARDWARE

The DECstation 78/88 Minicomputer systems consist of four basic units:

- a video or hard copy console terminal
- a disk system (either flexible or cartridge media)
- a central processor
- main memory

In addition, the system can support a line printer (LA78, LA180, or LQP78), additional disk drives, and auxiliary terminals connected to the serial asynchronous interface ports. These ports are suitable for primary and secondary communications applications, terminals, and a variety of user-provided devices.

The console terminal provides the means for interacting with or "talking" to the computer. It serves as the primary source of OS/78 command inputs. There are three types of terminals: the VT78 and VT100 video terminals, and the LA36 hard-copy printer.

The disk system can consist of one or two dual disk drives that use flexible (floppy) media (RX01 and RX02), one or two disk drives that use removable cartridge media (RL01), or a combination of both types. The RX01 floppy disk drive uses a diskette formatted for single-density operation. The RX02 floppy disk drive uses a diskette formatted for either double-density or single-density operation.
INTRODUCTION TO THE SYSTEM

The main functions of the disk system are:

- To contain a prerecorded copy of the OS/78 software that runs the computer system.
- To provide space to store the programs and files that you create.

The central processing unit (CPU) executes instructions from main memory and controls all peripheral devices. On DECstation 78 systems, the CPU is built into the VT78 terminal. On DECstation 88 systems it resides in a cabinet with the disk system.

The main memory is the active storage area in the computer from which the CPU fetches and executes instructions and into which your programs store and manipulate data.

Appendix I summarizes the DECstation hardware configurations.

For more information on the DECstation Minicomputer System, refer to either the DECstation 78 User's Guide or the DECstation 88 User's Guide as applicable.

1.2 OS/78 SOFTWARE

The OS/78 software can run on DECsystem 78/88 configurations or on any equivalent PDP-8 configuration. The system allows you to write programs in three languages: PAL8 assembly language, BASIC, and FORTRAN IV. You manipulate program source and data files by entering OS/78 commands from the console terminal. The software to do this resides on the master disk media provided with your system.

DIGITAL supplies the OS/78 system software to you on one of three types of disk media. Which medium you get depends on your disk system. If your DECstation has an RX01 floppy disk drive, you will receive the software on two single-density diskettes. If your DECstation has an RX02 floppy disk drive, you will receive the software on one double-density floppy diskette. If your DECstation has an RL01 cartridge disk pack drive only, you will receive the software on one cartridge disk pack.
CHAPTER 2
GETTING ON THE AIR WITH OS/78

This chapter provides the basic information that you need to use the OS/78 operating system. You should read the entire chapter prior to using the system to become familiar with the features and capabilities of OS/78.

2.1 STARTING THE SYSTEM

This section tells you how to load and start your system and to make backup copies of the master disk media. Before proceeding, be sure that your DECstation is connected to an appropriate alternating current power source and the power switch is in the ON position (refer to the applicable hardware manual listed in the Preface).

2.1.1 Loading the Diskette

Since the diskette is thin and flexible, be careful when handling and storing it. Hold it by the edges and do not touch the exposed portions that show through the protective cover (see Figure 2-1). Also, use only felt-tip pens when writing on diskette labels. Store them in a dust-free area, and do not expose them to magnetic devices, direct sunlight, heat, or humidity. Refer to the RX8 and RX28 User’s Guides (see Preface) for further information on the care of diskettes.

Figure 2-1 Handling the Diskette
NOTE

Do not insert a double-density diskette into a single-density drive (RX01). The single-density drive cannot read a double-density diskette.

The following steps and Figure 2-2 describe how to load a diskette into an RX01 or RX02 disk drive:

![Figure 2-2 Loading the Diskette]

1. Open the door:
   a. Notice that the door latch is embedded in the horizontal bar that is part of the door assembly.
   b. Compress the latch between the thumb and index finger.
   c. The door is spring-loaded and will open when the latch disengages.

2. Remove the diskette from its storage jacket. (This is not the black protective cover in which the diskette is permanently sealed.)

3. Insert the diskette label side up (the jacket seams are on the bottom) into the horizontal slot in the drive.

4. Close the door. It will automatically lock when you push it down.

Ordinarily you should load the diskette that contains your system programs (system diskette) into Drive 0, which is the left-hand slot. You can use Drive 1 (right hand slot) if necessary, so long as there is no diskette in Drive 0 when you press the START button.
2.1.2 Loading the RL01K Cartridge Disk Pack

The following steps and Figure 2-3 describe how to load the RL01K cartridge disk pack into its drive. If you are loading the disk pack that contains the OS/78 system programs (system disk), it must reside on RL01 Drive 0. The drive number is embossed on the READY indicator lamp next to the LOAD button.

1. If the drive is not the top drive in the cabinet, you must first slide the drive out from the cabinet by grasping it by the horizontal slot in the front panel and gently pulling it toward you.

Release the drive's cover by sliding the latch on the right side of the cover away from you with your thumb; then raise the cover. If the slide will not operate, the drive currently has a disk pack loaded in it and is on line. Follow the procedures in Section 2.1.3 to unload the pack from the drive.

2. Separate the protective cover from the disk pack as follows:
   a. Lift the cartridge by grasping the handle with your right hand.
   b. Support the cartridge from underneath with your left hand.
   c. Lower the handle, then push the handle slide to its extreme position with your right thumb while raising the handle to its upright position.
   d. Lift the cartridge from the protective cover.

3. Place the cartridge into the drive shroud with the handle recess facing the rear of the machine.

4. Rotate the cartridge a few degrees clockwise and then counter-clockwise to ensure that it is properly seated within the shroud.

5. Gently lower the handle to a horizontal position to engage the drive spindle.

6. Place the protective cover on top of the cartridge.

7. Lower the drive's cover carefully and be sure that the latch engages with the case.

8. Press the LOAD button so that it engages in its innermost position. (The LOAD lamp will extinguish.) When the drive is up to speed and on line, the READY lamp will light.

CAUTION

The OS/78 system will not operate properly if you enable the write protect feature of RL01 Drive 0 (WRITE PROT button illuminated).
<table>
<thead>
<tr>
<th>TO READY DRIVE:</th>
<th>DRIVE INDICATORS:</th>
<th>TO LOAD RL01K CARTRIDGE:</th>
</tr>
</thead>
<tbody>
<tr>
<td>• RAISE CARTRIDGE ACCESS DOOR</td>
<td>LOAD:</td>
<td>• SUPPORT CARTRIDGE “A” WITH LEFT HAND HOLDING PROTECTION COVER “B”.</td>
</tr>
<tr>
<td>• LOAD RL01K DISK CARTRIDGE</td>
<td>LIGHTS TO INDICATE THAT CARTRIDGE MAY BE LOADED OR THAT SPINDLE IS STOPPED.</td>
<td>• PUSH HANDLE SLIDE “C” TO LEFT WITH THUMB OF RIGHT HAND</td>
</tr>
<tr>
<td>• DEPRESS RUN/STOP SWITCH (LOAD INDICATOR)</td>
<td>UNIT SELECT:</td>
<td>• RAISE COVER HANDLE “D” TO FULL UPRIGHT POSITION, RELEASING PROTECTION COVER “B”.</td>
</tr>
<tr>
<td>• NOTE THAT SPINDLE MOTOR STARTS TURNING</td>
<td>INDICATES LOGICAL DRIVE ADDRESS.</td>
<td>• LIFT CARTRIDGE “A” FROM PROTECTION COVER “B” AND CAREFULLY SEAT IT ON DRIVE SPINDLE WITH HANDLE RECESS FACING REAR OF DRIVE.</td>
</tr>
<tr>
<td>• AFTER 30 SECONDS, UNIT SELECT INDICATOR SHOULD LIGHT INDICATING DRIVE IS READY TO READ OR WRITE</td>
<td>WHEN LIT, INDICATES DRIVE IS READY TO READ, WRITE OR RECEIVE CONTROLLER COMMANDS.</td>
<td>• CAREFULLY ROTATE TOP COVER HANDLE “D” A FEW DEGREES CLOCKWISE AND COUNTER-CLOCKWISE TO ENSURE FIRM SEATING</td>
</tr>
<tr>
<td>• IF WRITE PROTECTION IS DESIRED, DEPRESS WRITE PROTECT SWITCH (PROTECT INDICATOR)</td>
<td>FAULT:</td>
<td>• GENTLY LOWER TOP COVER HANDLE “D” TO HORIZONTAL POSITION TO ENGAGE CARTRIDGE ON DRIVE SPINDLE.</td>
</tr>
<tr>
<td></td>
<td>WHEN LIT, INDICATES A DRIVE ERROR CONDITION. IF THIS CONDITION PERSISTS, SEEK ASSISTANCE.</td>
<td>• PLACE PROTECTION COVER “B” ON TOP OF CARTRIDGE.</td>
</tr>
<tr>
<td></td>
<td>WRITE PROTECT:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>WHEN LIT, INDICATES THAT CARTRIDGE CURRENTLY MOUNTED IS WRITE PROTECTED.</td>
<td></td>
</tr>
</tbody>
</table>

Figure 2-3  Loading the RL01K Disk Pack
2.1.3 Unloading the RL01K Cartridge Disk Pack

The following steps describe how to unload the RL01K cartridge disk pack from its drive.

1. If the drive is not the top drive in the cabinet, you must first slide the drive out from the cabinet by grasping it by the horizontal slot in the front panel and gently pulling it toward you.

Place the unit off line by pressing the LOAD button so that it releases to its outermost position. The READY indicator lamp will extinguish and in a few seconds the LOAD button will light. This indicates that the disk has stopped and you can open the cover.

2. Open the drive's cover by squeezing the latch on the right side of the cover between your thumb and index fingers; then raise the cover.

If the drive is not the top drive in the cabinet, you must first slide the drive out from the cabinet by grasping the sides of the front panel and gently pulling it toward you.

3. Remove the disk's protective cover from the drive.

4. Disengage the disk pack from the drive spindle as follows:
   a. Push the handle slide to its extreme position with your right thumb while raising the handle to a vertical position.
   b. Lift the cartridge from the drive shroud.

5. Place the disk pack in its protective cover as follows:
   a. Hold the cartridge by its handle in your right hand.
   b. Support the protective cover underneath with your left hand.
   c. Lower the cartridge into the protective cover until you hear the handle slide engage.

2.1.4 Starting the System

2.1.4.1 Setting Terminal Characteristics - Before using your terminal, read the instruction manual supplied with it. Each terminal has features that you can control either by setting switches or by typing special setup mode commands. Except for the VT100, you can use your terminal as supplied by DIGITAL once you have set its transmit and receive speeds. Table 2-1 lists the terminal operating parameters necessary for OS/78 operation.
2.1.4.2 Bootstrapping OS/78 - Once you have successfully loaded a system diskette or disk pack into Drive 0, you can start the system by calling the OS/78 Monitor. The monitor is the main control program that you interact with when you use OS/78.

Call the monitor by pressing the START or BOOT button on your DECstation. If you have a DECstation 78 series system, the START button is located on the right-hand side of the VT78 console terminal. If you have a DECstation 88 series system, the BOOT button is located on the front of the cabinet containing the disk(s) to the right of the DECstation 88 logo. After a brief pause the monitor will display the OS/78 command summary on the terminal and then a period (.) The period is a prompting symbol that means that the monitor is waiting for you to enter a command. You can now type any OS/78 command.

When OS/78 starts up, it displays a summary of the commands. This feature is controlled by the SET SYS INIT command (see Chapter 3). If your console terminal is a printer rather than a scope, type the following commands:

```
*SET TTY: NO SCOPE
*SET TTY: WIDTH 132
```

You may also wish to inhibit the command summary to save time and paper. Use the following command:

```
*SET SYS: NO INIT
```
2.1.5 Using the Terminal

The keyboard on the terminal allows you to enter data and OS/78 commands into the computer. Most of the keyboard is identical to a typewriter keyboard. Unlike a typewriter, pressing a key does not automatically display a character on the keyboard screen. Pressing a key sends a character selected to be sent to the central processing unit (CPU). The OS/78 program that is running would normally respond by transmitting this character back to the terminal, thereby displaying it. Consequently, typing a character will have no effect if OS/78 is not running or if OS/78 is busy running a program that is not waiting for input from the keyboard.

Certain keys have special functions when used in OS/78 commands. The keys and their functions are as follows:

**RETURN key**

Use this key to terminate OS/78 command lines, Editor command lines, program statements, and responses to program input queries.

**DELETE key**

Use this key to correct typing errors in text that is not yet terminated by a RETURN. Each time you press DELETE, the system deletes one character to the left until it reaches the beginning of the line.

**LINE FEED key**

Use this key while entering monitor command lines. It causes the monitor to display the command line up to the previous carriage return.

**CTRL key**

Use this key in combination with another key to enter a special control command. Hold the CTRL key down while striking the other key. In this manual, these combinations are shown with a slash (/) between CTRL and the other key, for example: CTRL/U, CTRL/C (do not type the slash).

- **CTRL/C** Terminates execution of a currently active program, and returns control to the monitor. It also returns control from a BASIC program to the BASIC editor.
- **CTRL/O** Stops the display of characters on the screen.
- **CTRL/S** Suspends terminal output of a program or monitor command but does not terminate the program or command.
- **CTRL/Q** Resumes the terminal output that was suspended by CTRL/S.
- **CTRL/U** Deletes a line typed to the monitor or the Editor.

The CTRL key commands require no terminator; the system performs the function as soon as you type the command.

**ESCAPE key**

This key performs special functions when using the Editor and running FORTRAN IV and PAL8 programs. The function of this key is described in those sections where its use is applicable.
2.2 MAKING A BACKUP COPY OF THE OS/78 SOFTWARE

It is important that you make at least one copy of the master disk media supplied with your DECstation so that a copy of the system will be available in case the media that contains your working copy fails. Use the master disk media only to make working copies of the system. Never use it as a working copy or alter it in any way!

The following sections describe the procedures to follow for each DECstation type.

2.2.1 DECstations With RX01 and RX02 Disk Systems Only

Perform the following steps to make a backup copy of your master diskette(s):

1. Place the master diskette in Drive 0, the left-hand slot, and a blank diskette in Drive 1.

2. Start the system by pressing the START button.

3. In response to the monitor's period (.), type the DUPLICATE command followed by a carriage return (pressing the RETURN key):

   .DUP RXA1:<RXA0;

   The system will duplicate the contents of the diskette in Drive 0 onto the diskette in Drive 1. When the operation is complete, control returns to the monitor, as indicated by the period (.) prompting symbol.

4. Remove both diskettes, and label the diskette taken from Drive 1 appropriately.

5. Repeat Steps 1 through 4 for master diskette 2 (RX01 systems) or to make additional backup copies.

6. Store the master diskettes in a safe place.

7. Place a copy of the system diskette in Drive 0 for use of the working system diskette.

8. Press the system's START button, and you are ready to use OS/78.

2.2.2 DECstations With RL01 Disk Systems Only

Perform the following steps to make a backup copy of your master disk pack:

1. Place the master disk pack in Drive 0 and a blank disk pack in Drive 1.

2. Start the system by pressing the BOOT button.
GETTING ON THE AIR WITH OS/78

3. Type the following commands in response to the monitor's period (.) prompting symbol:

```
.DEFORMAT RL01
.ZERO RL1A:/Y
.ZERO RL1B:
.ZERO RL1C:
.COPY RL1A:<RL0A:.*
```

The system will format the disk pack, zero each device and copy all the system files from unit A of Drive 0 to unit A of Drive 1. When the operation is complete, the monitor takes control as indicated by the period (.)

4. Remove both disk packs and label the disk pack taken from Drive 1 appropriately. This is the working system disk pack.

5. Store the master disk pack in a safe place.

6. Place the working system disk pack in Drive 0.

7. Press the system's BOOT button, and you are ready to use OS/78.

2.2.3 DECstations With RX02 and RL01 Disk Systems

The following steps describe how to make a backup copy of your master disk pack.

1. Place the master disk pack in RL01 Drive 0. Make sure that the drive is not write protected (WRITE PROT indicator lamp is extinguished).

2. Press the BOOT button.

3. Place a blank diskette in RX02 Drive 0.

4. Type the following command to initiate the backup operation:

```
.SUBMIT BUFLRX
```

5. When the monitor again displays the period (.), remove the disk pack from its drive, and place a blank disk pack in the drive.

6. Type the following command to complete the backup operation:

```
.SUBMIT BACKFL
```

7. When the monitor again displays the period (.), the disk pack in Drive 0 now contains a complete copy of the contents of the OS/78 system software that resides on Device A (RL0A) of your master disk pack. If you want to make an additional backup copy of the system disk, type the following command:

```
.RX/RX
```

then perform steps 5 through 7 again.

8. Store the master disk pack in a safe place.
2.3 DEMONSTRATION PROGRAMS

A series of programs are provided to demonstrate some of the capabilities of OS/78. These programs are run by a self-explanatory BATCH file called DEMO.BI. To initiate this demonstration, start the system and type the following command in response to the monitor's period (.):

```
,SUBMIT DEMO/H/T
```

The batch stream controlling the demonstration repeats it every three minutes until you type a CTRL/C.

Once you become familiar with the system, you may wish to remove the files comprising the demonstration to make room for other programs. To do so, use the DELETE command and type:

```
,DEL DEMO??,*
```

2.4 DEVICE AND FILE NAMES

2.4.1 Devices

The system recognizes each of its devices by one of the logical names listed in Table 2-2. You use these names in command strings to the system programs to specify the devices that you want to use. Although the system can recognize all the names, no one DECstation can have all listed physical devices. The SET HANDLER command, described in Chapter 3, allows you to specify which devices are available.

If you refer to a device that does not physically exist, the system will enter a loop condition and will not print any error message. You must restart the system by pressing the START or BOOT button to resume OS/78 operation.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSK</td>
<td>Default output device; usually same as SYS</td>
</tr>
<tr>
<td>SYS</td>
<td>The diskette or disk pack inserted in Drive 0 where the monitor and system programs reside.</td>
</tr>
<tr>
<td>RL0A</td>
<td>Unit A of RL01 Drive 0</td>
</tr>
<tr>
<td>RL0B</td>
<td>Unit B of RL01 Drive 0</td>
</tr>
<tr>
<td>RL0C</td>
<td>Unit C of RL01 Drive 0</td>
</tr>
<tr>
<td>RL1A</td>
<td>Unit A of RL01 Drive 1</td>
</tr>
<tr>
<td>RL1B</td>
<td>Unit B of RL01 Drive 1</td>
</tr>
<tr>
<td>RL1C</td>
<td>Unit C of RL01 Drive 1</td>
</tr>
<tr>
<td>RXA0</td>
<td>The diskette inserted in Drive 0</td>
</tr>
<tr>
<td>RXA1</td>
<td>The diskette inserted in Drive 1</td>
</tr>
<tr>
<td>RXA2</td>
<td>The diskette inserted in Drive 2</td>
</tr>
<tr>
<td>RXA3</td>
<td>The diskette inserted in Drive 3</td>
</tr>
<tr>
<td>TTY</td>
<td>Keyboard/screen</td>
</tr>
</tbody>
</table>

(continued on next page)
OS/78 recognizes each of its devices by a permanent name that is built into the system. You can temporarily assign a name of your choice, called a user-defined device, to a device name by using the ASSIGN command. For example,

```
ASSIGN RXA1 PLACE
```

causes all future references to PLACE to address the device RXA1. More than one user-defined device name can be active at a time. The DEASSIGN command causes OS/78 to eliminate any user-defined device names. Also, all such user-assigned names are lost when you restart OS/78 with the START or BOOT button or with the BOOT command.

### 2.4.2 File Names and Extensions

When referring to a file-structured device, the system ordinarily expects you to specify a file name. Files are referenced symbolically by a name of up to six alphanumeric characters followed, optionally, by a period and an extension of up to two alphanumeric characters. The extension to a file name is generally used as an aid for remembering the format of a file.

In most cases, you should use the standard OS/78 file name extensions listed in Table 2-3. If an extension is not specified for an output file, some system programs append assumed (default) extensions. Where an extension for an input file is not specified, the system searches for that file name with the default extension. Failing to find such a file, the system searches for the original file without an extension. For example, if PROG were specified as an input file to PAL8, the assembler for the OS/78 Operating System, the system would first look for the file PROG.PA (since .PA is the standard extension for PAL8 input files). If PROG.PA were not found, the system would try to find the file PROG (with no extension). Some system programs do not use default extensions; refer again to Table 2-3 and to the individual system program descriptions for additional details.

Some programs also accept the characters * and ? in file names. These characters, called wildcard characters, have special meaning to the programs involved. Section 2.5.6 describes how to use them.

Since files are the basic units of the OS/78 system, an understanding of files and their structure is helpful when using OS/78. A description of files and file-structured devices is given in Section 2.8. This section can be skipped since it is not necessary for properly operating the OS/78 system, but does give an insight into what is happening in the system.

2-11
## Table 2-3
### OS/78 Extensions

<table>
<thead>
<tr>
<th>Extension</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>.BA</td>
<td>BASIC source file (default extension for a BASIC input file).</td>
</tr>
<tr>
<td>.BI</td>
<td>Batch input file (input for BATCH).</td>
</tr>
<tr>
<td>.BN</td>
<td>Absolute binary file (default extension for ABSLDR and BITMAP input files; also used as default extension for PAL8 binary output file).</td>
</tr>
<tr>
<td>.CM</td>
<td>Command file.</td>
</tr>
<tr>
<td>.DI</td>
<td>Directory listing file.</td>
</tr>
<tr>
<td>.FT</td>
<td>FORTRAN language source file (default extension for FORTRAN input files).</td>
</tr>
<tr>
<td>.HN</td>
<td>I/O device handler save image.</td>
</tr>
<tr>
<td>.LD</td>
<td>FORTRAN load module file (default assumed by run-time system, FORTRAN IV loader).</td>
</tr>
<tr>
<td>.LS</td>
<td>Assembly listing output file (default extension for PAL8).</td>
</tr>
<tr>
<td>.MP</td>
<td>File containing a loading map (used by the Linking Loader, MAP command).</td>
</tr>
<tr>
<td>.PA</td>
<td>PAL8 source file.</td>
</tr>
<tr>
<td>.RA</td>
<td>RALF assembly language file (FORTRAN IV).</td>
</tr>
<tr>
<td>.RL</td>
<td>Relocatable binary file (default extension for a Linking Loader input file).</td>
</tr>
<tr>
<td>.SV</td>
<td>Memory image file (SAVE file); default for the R, RUN, SAVE, and GET commands.</td>
</tr>
<tr>
<td>.TM</td>
<td>Temporary file generated by system.</td>
</tr>
</tbody>
</table>

OS/78 commands and the abbreviated forms for each command are summarized at the end of this chapter. Greater detail on each command is given in Chapter 3.

### 2.5 ENTERING MONITOR COMMANDS

OS/78 operating system commands are entered at the terminal in response to the period (.) printed by the monitor. Pressing the RETURN key initiates execution of the entered command. Some commands require a particular format to distinguish between input and output files or to name certain devices. Any errors that are made while utilizing these commands result in an error message being displayed by the monitor. Error messages are summarized in Appendix G.
2.5.1 OS/78 Command Format

An OS/78 command is usually made up of the following parts:

1. a command word or words,
2. a list of output devices and files,
3. a left-angle bracket (<),
4. a list of input devices and files,
5. a series of options, and
6. the RETURN or ESCape key.

Parts 2 through 5 above are called the command parameters because they supply information concerning the devices and files that are affected by the command.

The general format for a command may thus be expressed as follows:

```
COMMAND outspec<inspec/options (terminator)
```

where:

- **COMMAND** is any OS/78 command.
- **outspec** is the output specification. This usually consists of a device name, a colon (:), a file name, and a file name extension (Example: RXA0:TEST.PA). Output specifications are optional. If you do not specify an output device, the system assumes DSK:. If you do not specify a file name, the system assumes you are referring to any or all the files on the designated device; the exact action depending on the command.
- `<` is the specification separator. Specifications to the left are output specifications. Specifications to the right are input specifications.
- **inspec** is the input specification. This usually consists of a device name, a colon (:), a file name, and a file name extension (Example: RXA1:TEST1.PA). For each input specification that you do not explicitly name a device, the system assumes the device associated with the previous input specification. If you do not specify a device name for the first input specification, the system assumes DSK:.
- `/options` is one or more option specifications that modify the command; these are discussed in Section 2.5.4 and in descriptions of each of the commands in Chapter 3.
- **terminator** is the command line terminator. This is usually a carriage return (pressing the RETURN key). This causes the system to begin execution of the command. In some cases, you must use the ESCape key. This is explained where applicable.
The command word and parameters in each command must conform to obey the following rules:

1. The command word or words may be abbreviated as shown in Table 2-5.

2. No command may have more than three output devices or files, and these must be separated by commas. Some commands, such as COPY and DIRECT, may have no more than one output specification.

3. You can omit the left-angle bracket (<) if you do not specify any output devices or files.

4. In general, no command may have more than nine input devices or files. The DIRECT and COPY commands are limited to five.

5. Multiple input and output device or file specifications must be separated by a comma (,).

6. The file name may not contain more than six characters. The extension may not contain more than two characters.

7. In a command line, the device name is followed by a colon (:), and is always separated from any file name by a colon.

For example, the command line

`.DIR RLOA: `/E`

when executed by pressing the RETURN key, displays the directory of RLOA on the terminal screen. The E option also causes the system to display any empty files in the order that they are located on the disk.

The command line

`.COPY RXA0:<RL1A:ARITH.PA,MULT.PA,DIVIDE.PA`

when executed, transfers the specified files ARITH.PA, MULT.PA and DIVIDE.PA from RL1A to RXA0.

Commands also can be simplified by the use of defaults as explained in Section 2.6.

2.5.2 Incorrect Commands

If you enter an incorrect command line and then press the RETURN key, one of the following results will occur:

1. If the command is not recognized, the system responds with a question mark followed by the monitor's period (.), and then waits for you to enter a correct command.
2. If the command is recognized, execution will begin immediately. If you see an error in the command line, press CTRL/C quickly to terminate the action and return control to the monitor. Whether this step prevents execution of the incorrect command depends on the type of command and the file(s) involved.

You can destroy important files if you incorrectly issue any of the following commands:

COPY    SAVE    SQUISH    DELETE    ZERO

In fact, the error may actually destroy part of the software. Therefore, be extremely careful when using these commands. Any specific cautions that should be exercised when using these commands are fully described in Chapter 3. Always be sure to make a back-up copy of the system and all other important files that you have created (see Section 2.2).

Section 2.5.3 describes how to correct any errors that have been made in the command line prior to executing the command.

2.5.3 Correcting Errors

Until the RETURN key is pressed, the system will not process the command line. Therefore, the command line is still available for correction. Always check the command lines before pressing the RETURN key since a line with errors may cause undesirable results.

To correct typing mistakes, use the DELETE key. This key erases the last character typed. Each use of the DELETE key causes one more character to be erased. The correct character or characters can then be typed.

A command line may be deleted completely before it is entered by typing CTRL/U (produced by holding down the CTRL key and pressing the U key). This echoes as ^U and returns control to the monitor without processing the current line. The monitor's period (.) indicates that the system is ready to accept a new command. You can then retype the command line. You can use the LINE FEED key to verify the contents of a command that is being entered. The system will display the portion of the command line that you have entered.

2.5.4 Using Input/Output Options

In addition to specifying output and input files in the command line, you can specify various options to perform select certain functions. Options are numbers, alphanumeric characters, or their full-word equivalents. Numbers used as options are generally contained in the command line with the equal sign (=) or square brackets ([ ]) construction. The alphanumeric option characters are set off from the I/O specifications by the slash (/) character for single character options and parentheses for a string of single characters. The usage of the slash, parentheses, equal sign, and square brackets is explained below.

You will find an explanation of the single character options in the command descriptions in Chapter 3. Appendix I lists the full-word equivalents to the single character options. A group of general-purpose options known as dash options that you can use in most command lines are described below.
2.5.4.1 Equal Sign Construction - An equal sign (=) followed by an octal number may occur only once in a command line and must be followed by a separator character (comma or left-angle bracket), other options, or a line terminator (RETURN or ESCape). For example,

```
>DIRECT SYSTS
```
will list the directory of the system device on the screen in three columns.

2.5.4.2 Slash Construction - A single alphanumeric character is preceded by a slash and can occur anywhere in the command line (even in the middle of a file name) although the usual position is at the end of the line. For example:

```
>COPY RXA:SECOND.EX<RXA:FIRST.EX/T
```
means the same as

```
>COPY RXA:<SECOND.EX<T<RXA:FIRST.EX
```
The option /T instructs the monitor to assign the current date to the output file.

2.5.4.3 Parentheses Construction - When two or more letter options are used, you can group them together inside parentheses. This construction is valid anywhere in the command line. For example,

```
>COPY RLIP:OUTPUT.EX<SYS:INPUT.EX(OT)
```
means the same as

```
>COPY RLIP:OUTPUT.EX<SYS:INPUT.EX/OT
```

2.5.4.4 Square Bracket Construction - The square bracket construction can only occur immediately after an output file name and consists of a left square bracket ([), a decimal number between 1 and 255, and a right square bracket ]). The square bracket construction allows you to optimize file storage by specifying an upper limit on the number of blocks required by your output file. For example:

```
>PAL UTILITY=LISTING LEVEL 2000 SOURCE
```
The output files are a file named BINARY on device DSK: having a maximum length of 19 blocks, and a file named LISTIN (only six characters are significant) on the device DSK: with a maximum length of 200 blocks. The input file is SOURCE on device DSK.

2.5.4.5 General-Purpose Dash Options - These options provide a shorthand method for specifying devices and files with certain commands. The form is:

```
-ex
```
where:

```
-ex
```
is one of the options specified in Table 2-4.
In the following example, typing

```
*PA.L TEST PA
```

will assemble TEST.PA, store the resulting binary program in TEST.BN
on DSK, and display the program listing on the terminal. The `-T`
option saves you from typing the specification:

```
TTY:<
```

<table>
<thead>
<tr>
<th>Option</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-PT</code></td>
<td>Selects the FORTRAN IV Compiler if you do not use the default file extension <code>.PT</code> (used with the COMPIL and EXECUTE commands).</td>
</tr>
<tr>
<td><code>-L</code></td>
<td>Send output to LPT. For example, typing <code>DIR-L</code> will list the director of the system device on the line printer (LPT).</td>
</tr>
<tr>
<td><code>-LS</code></td>
<td>Generate a listing file (used with the COMPIL, EXECUTE, and PAL commands). The listing file is written onto SYS: if no output device is specified and is given a <code>.LS</code> extension. The listing file name is the same as the file name that immediately preceded the <code>-LS</code> option in the command string.</td>
</tr>
<tr>
<td><code>-MP</code></td>
<td>Generate a memory map (used with the COMPIL, EXECUTE, and PAL commands).</td>
</tr>
<tr>
<td><code>-NB</code></td>
<td>Do not create a binary file (used with the COMPIL, EXECUTE, and PAL commands).</td>
</tr>
<tr>
<td><code>-PA</code></td>
<td>Selects the PAL8 Assembler if you do not use the default file extension <code>.PA</code> (used with the COMPIL and EXECUTE commands).</td>
</tr>
<tr>
<td><code>-T</code></td>
<td>Send output to terminal.</td>
</tr>
</tbody>
</table>

### 2.5.5 Remembering Previous Arguments

The system remembers the arguments that you use with the CREATE, LOAD, PAL, EDIT, and EXECUTE commands by storing them in a temporary file. Thereafter, when you use any of these commands without arguments, the system will use the arguments that it stored. When you again use these commands with arguments, the system replaces the currently stored arguments with the new arguments. The storage area for the EDIT command is separate from the others and its contents will not change unless you issue an EDIT command with arguments.
The following example illustrates this feature. If you entered the command:

```
.EXECUTE TEST1.PA
```

you could compile TEST.PA by issuing the command:

```
.COMPILE
```

**NOTE**

The system does not remember command arguments typed if you restart the system with the START button or the BOOT command.

---

2.5.6 Using Wildcards

The wildcard construction is a feature that allows you write generalized file specifications when manipulating files. Wildcards allow a file name or the extension specified in a command to be replaced totally with an asterisk or partially with a question mark. This allows you to designate classes of file names or extensions. The wild characters are particularly useful when doing multiple file transfers.

You can use the wildcard construction in the file name specifications of the COPY, DELETE, DIRECT, LIST, RENAME, and TYPE commands.

The asterisk (*) is used as a wild field to designate the entire file name or extension. This is illustrated in the following examples:

- `TEST1.*`: All files with the name TEST1 and any extension.
- `*.BN`: All files with a BN extension and any file name.
- `.*`: All files (except when used with the DELETE command).

The question mark (?) is used as a wildcard character to designate part of the file name or extension. A question mark is used for each unspecified character that is to be matched. For example, `PR??` matches all files beginning with PR that are two to four characters long. Other examples are as follows:

- `TEST2.B?`: All files with the name TEST2 and any extension beginning with B.
- `TES??..PA`: All files with a PA extension and any file name from three to five characters long beginning with TES.
- `??..??`: All files with file names of two characters or less.

The asterisk and the question mark can be specified together in the same command line:

- `???.*`: All files with file names of three characters or less.
A specification may not contain embedded asterisks (*). For example:

A*B.*

is an illegal specification and will produce the following message:

ILLEGAL *

If you include an * or ? in a command other than COPY, DELETE, DIRECT, LIST, RENAME, or TYPE, the following message appears:

ILLEGAL * OR ?

2.5.6.1 Wildcard Input File Specifications - This section describes how to use wildcards when specifying input files.

For example, to display the directory entries for all files on RXA1 with file names from two to four characters long and beginning with PR, type:

.DIR RXA1:PR??.*

To display entries for all files on RLOC with extensions ending with R, type:

.DIR RLOC:*.?R

The following wildcard specifications are illegal:

A*.BD The asterisk should replace the entire file name or ABC.*D
 or extension.

RXA?:C.BA Wildcards may not be used in device names.

2.5.6.2 Wildcard Output File Specifications - This section describes how to use wildcards when specifying output files.

You can use the asterisk in output file names, but the question mark is illegal. If no file name is specified, then *.* is assumed.

For example,

.DEL *.BN<*.PA

is legal. This command deletes any files with a .BN extension if the file and a .PA extension exist.

Another example would be:

.COPY RXA1:*.BK<SYS:*.PA

The system will copy all files with a .PA extension from the system device onto RXA1, giving them the same file name but assigning a .BK extension to each.

The question mark may not be used in output file specifications. For example:

.COPY RL1B:NEW.??<OLD.BA

is not allowed.
The wildcard construction, when used with the COPY and DELETE commands, is capable of destroying needed files if you do not use it carefully. We recommend that you follow these rules:

1. Keep a backup copy of the system and all other important files.

2. Use the /Q option when running COPY or DELETE with wildcards. This allows you to decide what to do with each file involved before it is acted upon by the system.

2.5.7 Indirect Commands (At Symbol (@) Construction)

The at symbol (@) construction allows you to include many file names and options in a single command, so that you do not have to type them each time you want to use them. You do this by creating a file with the Editor Program (EDIT command) that contains the desired file specifications and options. You then reference this file using the @ construction anywhere within the argument (file and option list) portion of a command. The form of the @ construction is:

@dev:file.exe

If dev: is omitted, DSK: is assumed. If the extension is omitted, .CM is assumed.

The system takes the text in the specified file and inserts it into the command string in place of the indirect command file reference. For example, if you create a file with the Editor called FLIST.CM that contains the string:

FILEB,FILEC/L,FILED

then the command:

+COMPILE FILEA,FILEB,FILEC/L,FILED,FILEZ

could be replaced by the command:

+COMPILE FILEA,@FLIST,FILEZ

You cannot use an indirect command file reference in arguments to the following commands:

<table>
<thead>
<tr>
<th>ASSIGN</th>
<th>GET</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOT</td>
<td>ODT</td>
</tr>
<tr>
<td>DATE</td>
<td>R</td>
</tr>
<tr>
<td>DEASSIGN</td>
<td>RUN</td>
</tr>
<tr>
<td>DUPLICATE</td>
<td>SAVE</td>
</tr>
<tr>
<td>FORMAT</td>
<td>START</td>
</tr>
</tbody>
</table>

Carriage returns and line feeds within the file are ignored, but nulls are not. A null signifies the end of a line. A command file may not be more than 512 characters in length.

You must separate an indirect command specification from other command specifications that follow on the same line with a non-alphanumeric character such as an apostrophe (').
2.5.8 Asterisk (*) Prompting Symbol

Under certain conditions, OS/78 may display an asterisk (*) as a prompting symbol. This display is a function of a special program called Command Decoder (described in Appendix D), and signifies a request to enter another list of files for the current system command. Except for commands where the appearance of the asterisk prompting symbol is specifically described, ignore it and return to the monitor by typing a CTRL/C.

2.6 USING DEFAULTS

You can reduce the amount of typing necessary for entering commands by taking advantage of the default ability of the system. A default condition is a command parameter that the system assumes if you do not explicitly specify one. You can use defaults to avoid typing device names repetitively.

1. For all output files and input files, the default device assumed by the monitor is DSK: and is equivalent to SYS: (that is, RXA0:, or RLOA, the diskette or disk pack in Drive 0). For example, the monitor interprets the following two file specifications identically:

   FILEA.AB,FILEB.CD<FILEC.EF

   DSK:FILEA,AB,DSK:FILEB.CD<DSK:FILEC.EF

   You can temporarily change this default with the ASSIGN command. The system will revert to the original default each time you restart the system with the START or BOOT button or with the BOOT command.

2. When a device name is explicitly given for an input file, any additional input files will default to the last explicitly specified device name. The following two file specifications are identical:

   TTY:<FILEA,FILEB,RL1B:FILEC,FILED

   TTY:<DSK:FILEA,DSK:FILEB,RL1B:FILEC,RL1B:FILED

   In this description,

   FILEA was assumed to be on DSK: because it was the first input file.

   FILEB was assumed to be on DSK: because the preceding file (FILEA) was on DSK: (same as RLOA:).

   FILEC was stated to be on RL1B::

   FILED was assumed to be on RL1B: because the preceding file (FILEC) was on RL1B:.
3. Some of the individual commands assume special default devices. The DIRECT and TYPE commands assume the terminal as the output device. The following command line will display the directory of the diskette inserted in Drive 1 on the terminal screen.

.DIRECT RXA1:

Note that the left-angle bracket is not needed in the above command line because no output files are specified. However, commands such as CREF (cross reference program) and LIST default to the line printer. Thus, if a line printer is not available, the output of the CREF command must be sent to a file or to the terminal (using the -T Dash option). Specific cases and use of default options are discussed in Chapter 3.

2.7 WHERE TO GET MORE INFORMATION

While you are running OS/78, you may want more information about the function of certain commands, their format, and options. You can use the HELP command to obtain this information. For example:

.,HELP COMPARE

will display the following information on the terminal:

SRCCOM.SV

CALLING COMMANDS:
.COMPAR outdev:file.PA<indev:file1.PA,indev:file2.PA
.COMPAR outfile.PA<infile.PA,infile2.PA -- -- -- (Files on DSK)

SWITCHES:
/B Compare blank lines
/C Don't compare (slashed) comments
/S Don't compare tabs and spaces
/T Convert tabs to spaces on output
/X Don't compare or print comments

ERRORS:
?0 Insufficient core
?1 Input error file 1 (or less than 2 input files)
?2 Input error file 2
?3 Output file too large
?4 Output error
?5 Can't open output file

If a line printer is attached to the system, you can get a hard copy printout by sending the desired HELP text to the line printer as follows:

.,HELP LPT:<PAL,COMPARE

or (by using the -L dash option)

.,HELP PAL,COMPARE-L

The HELP text for PAL and COMPARE will be printed on the line printer.
2.8 FILE-STRUCTURED DEVICES

A file-structured device is a random-access mass storage device that is logically divided into a number of blocks. This kind of device can read and write any desired block. A diskette and disk pack are file-structured devices, but a terminal is not.

All OS/78 file-structured devices are logically divided into 256-word blocks. Hence, blocks of 256 words are considered the standard size for OS/78. A file consists of one or more sequential (consecutively numbered) blocks. A minimum of one block per file is required, although a single file could occupy all of the blocks on a device.

By convention, OS/78 block numbers are specified in octal radix, while file lengths (number of blocks per file) are specified in decimal radix.

There are two types of file-structured devices, system devices and non-system (or files only) devices. A system device contains a copy of the OS/78 Monitor (also called the system head) on its medium and the OS/78 system programs; a non-system device does not.

The system device is also the disk drive that is accessed when you press the DECstation's START or BOOT button or use the BOOT command. If your system has an RX01 or RX02 diskette drive, the system device is Drive 0 (RXA0). If your system has RL01 cartridge disk drives only, the system device is device A of RL01 Drive 0 (RL0A).

Table 2-4 lists the OS/78 file-structured devices and their storage capacities.

<table>
<thead>
<tr>
<th>Device</th>
<th>Type</th>
<th>Size (in blocks)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RLxA</td>
<td>System</td>
<td>4018</td>
</tr>
<tr>
<td></td>
<td>Non-system</td>
<td>4074</td>
</tr>
<tr>
<td>RLxB</td>
<td>Non-system</td>
<td>4074</td>
</tr>
<tr>
<td>RLxC</td>
<td>Non-system</td>
<td>2018</td>
</tr>
<tr>
<td></td>
<td>System</td>
<td>431</td>
</tr>
<tr>
<td>RX01</td>
<td>Non-system</td>
<td>487</td>
</tr>
<tr>
<td></td>
<td>System</td>
<td>932</td>
</tr>
<tr>
<td>RX02</td>
<td>Non-system</td>
<td>981</td>
</tr>
<tr>
<td>VXA0*</td>
<td>Non-system</td>
<td>128</td>
</tr>
</tbody>
</table>

* DECstation 88/97 systems only
2.8.1 Using RX01 and RX02 Diskettes

The RX01 and RX02 diskettes are physically identical. The RX01 is a single-density diskette and the RX02 is a double-density diskette. This means that an RX02 diskette has approximately twice the storage capacity of an RX01 diskette (see Table 2-4).

The RX01 diskette is supplied already formatted for single-density operation. You must, however, create a directory area on it with the ZERO command before copying files to it. If you are creating backup copies with the DUPLICATE command, the ZERO command is not necessary.

The RX02 diskette is a single-density RX01 diskette that you reformat for dual-density operation with the DUPLICATE command's /D option. You must, however, create a directory area on it with the ZERO command before copying files to it. If you are creating backup copies with the DUPLICATE command, the ZERO command is not necessary.

You should observe the following rules when using diskettes:

- An RX01 disk drive can use single-density diskettes only.
- An RX02 disk drive can use either single-density or double-density diskettes. The system automatically determines the density of the diskette.
- A system diskette that contains an old version of OS/78 can run on an RX01 drive only. You can use it on an RX02 drive if you want to use it for file storage only.

2.8.2 Using the RL01 Cartridge Disk Pack

The RL01 cartridge disk pack is a high-density mass storage device that has a storage capacity that is approximately ten times that of the RX02 double-density diskette (see Table 2-4).

Each disk pack consists of three logical OS/78 devices: device A, device B, and device C. Device A and B are the same size. Transfers between these devices are faster than transfers between either one of them and device C. This is because the device C is smaller than the other two and consequently occupies less of each track on the disk than the others. This increases the search time required for device C blocks.

You must format and zero each new blank disk pack with the FORMAT and ZERO commands before using it with OS/78. If you do not do this, OS/78 cannot successfully use the disk pack.

Each disk pack incorporates a bad block mapping scheme that allows OS/78 software to ignore disk blocks that become defective. Whenever a bad block error occurs, you can use the FORMAT command to make the block unavailable to OS/78. This allows you to prolong the life of your disk packs.
2.8.3 Using the VXAO Extended-Memory Device

The VXAO device enables you to use the extended memory above 32K provided in DECstation 88/97 systems as though it were a separate file-structured device. The VXAO device provides high speed I/O performance similar to that of a fixed-head disk type of storage device. You refer to it in the same way as the other OS/78 devices. For example, this command

COPY VXAO:SAMPLE<RXAO:SAMPLE

copies a program called SAMPLE into the area of memory above 32K. The command

DIR VXAO:

produces a directory listing of the device.

2.9 FILES

A file is the fundamental storage unit of the OS/78 system. It is a uniquely named collection of data that is treated as a unit. The format of this data is unimportant; OS/78 can manipulate several standard formats, including ASCII files, binary files, and memory-image files. The important consideration is that the data forms a single and uniquely identifiable unit within the system.

2.9.1 File Directories

To maintain records of the files on a device, blocks 1 through 6 are reserved for the file directory of that device. Thus, file structured devices are also called directory devices. Entries in this directory inform the system of the name, size, and location of each file, including all empty files and the tentative output file, if one exists. Six blocks are always allocated, though all are not necessarily active at any given time. Block zero is unused, except on the system device, in which case it contains the system bootstrap, a program that reads in the OS/78 Monitor and the system device handler (SYS).

2.9.2 File Types

Three types of files exist in the OS/78 system:

1. Empty File - An empty file is a contiguous area of unused blocks. Empty files are created when permanent files are deleted or when the ZERO command is used.

2. Tentative File - A tentative file is a file that is open to accept output and has not yet been closed. Only one tentative file can be open on any single device at one time. Tentative files never appear in directory listings.

3. Permanent File - A permanent file is a file that has been given a fixed size and is no longer expandable. A tentative file becomes permanent when it is closed.
To further understand file types, consider what occurs when a file is created. Normally, in creating a tentative file, the system first locates the largest empty file available and creates a tentative file in that space. That establishes the maximum space into which the file can expand. The user program then writes data into the tentative file. At the end of the data, the program tells the system to close the tentative file, making it a permanent file. When the file is closed, whatever space remains at the end of the file is available to contain a new file.

2.9.3 ASCII File Format

ASCII files are packed as three 8-bit characters into two words as follows:

<table>
<thead>
<tr>
<th>WORD 1</th>
<th>CHARACTER 3</th>
<th>CHARACTER 1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(Bits 0-3)</td>
<td>(Bits 0-7)</td>
</tr>
<tr>
<td>WORD 2</td>
<td>CHARACTER 3</td>
<td>CHARACTER 2</td>
</tr>
<tr>
<td></td>
<td>(Bits 4-7)</td>
<td>(Bits 0-7)</td>
</tr>
</tbody>
</table>

| 0 | 3 | 4 | 11 |

2.10 OS/78 Command Summary

Table 2-5 is a summary of OS/78 commands. For each command, the table shows the command and its abbreviation printed in red ink, the format, and a brief functional description.
<table>
<thead>
<tr>
<th>Command</th>
<th>Format Example</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASSIGN</td>
<td>AS permdev user-name</td>
<td>Associates a new user-defined device name with the permanent logical device name.</td>
</tr>
<tr>
<td>BASIC</td>
<td>BAS</td>
<td>Requests execution of the BASIC language editor to allow for program creation and modification.</td>
</tr>
<tr>
<td>BOOT</td>
<td>BO/dv</td>
<td>Initializes the system residing on the specified system device RL or RX.</td>
</tr>
<tr>
<td>CANCEL</td>
<td>CA</td>
<td>Terminates the symbiont task (see Appendix H) that is currently running and returns OS/78 to 16K, single task operation (DECstation 78 systems only).</td>
</tr>
<tr>
<td>COMPARE</td>
<td>COMPA dev:output.ex&lt;dev:file1.ex,dev:file2.ex</td>
<td>Compares two source files line by line and displays all their differences.</td>
</tr>
<tr>
<td>COMPILE</td>
<td>COM file.ex&lt;file1.ex,.../options</td>
<td>Produces binary files and/or listings for specified program files. This command will chain to PAL8, BASIC, or FORTRAN IV depending on the file name extension. Under FORTRAN IV, only one input file can be specified.</td>
</tr>
<tr>
<td>COPY</td>
<td>COPY dev:file.ex&lt;dev:file.ex,...</td>
<td>Transfers files from one OS/78 device to another. Up to five input files can be specified.</td>
</tr>
<tr>
<td>CREATE</td>
<td>CREA file.ex</td>
<td>Opens a new file for editing.</td>
</tr>
<tr>
<td>CREF</td>
<td>CREP file.ex</td>
<td>Assembles and produces a cross-reference listing from the source file. Defaults to LPT, thus the -T option must be used to output to terminal.</td>
</tr>
</tbody>
</table>

(continued on next page)
<table>
<thead>
<tr>
<th>Command</th>
<th>Format Example</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATE</td>
<td>DA dd-mmm-yy</td>
<td>Enters the date into the system. If no argument is given, print current date on terminal or NONE if no date was specified.</td>
</tr>
<tr>
<td>DEASSIGN</td>
<td>DE</td>
<td>Eliminates all previous user-defined names.</td>
</tr>
<tr>
<td>DELETE</td>
<td>DEL dev:file.ex,.../options</td>
<td>Deletes one or more files. Up to five input files can be specified, separated by commas. All files must reside on the same device.</td>
</tr>
<tr>
<td>DIRECT</td>
<td>DIR dev:/options</td>
<td>Produces a listing of an OS/78 device directory.</td>
</tr>
<tr>
<td>DUPLICATE</td>
<td>DUP outdev:&lt;indev:/options</td>
<td>Reproduces the contents of the input diskette onto the output diskette.</td>
</tr>
<tr>
<td>EDIT</td>
<td>ED file or ED file&lt;file</td>
<td>Opens an already existing file for editing. Using the input/output construction allows a copy of the original file to be retained, while naming a new output file.</td>
</tr>
<tr>
<td>EXECUTE</td>
<td>EX file.ex, file.ex/options or EX file.ex&lt;file.ex</td>
<td>Produces binary files and/or listings for the specified program files, loads the binary file, and executes the program. This command will chain to PAL8, BASIC or FORTRAN IV depending on the file name extension. Under FORTRAN IV, only one file can be specified.</td>
</tr>
<tr>
<td>FORMAT</td>
<td>FORMAT RL01</td>
<td>Formats RL01 disk pack for OS/78 use.</td>
</tr>
</tbody>
</table>

(continued on next page)
<table>
<thead>
<tr>
<th>Command</th>
<th>Format Example</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>GE dev:file.ex</td>
<td>Loads memory image files (.SV format) into memory from a device.</td>
</tr>
<tr>
<td>HELP</td>
<td>HE command</td>
<td>Displays instructional information on the specified OS/78 command.</td>
</tr>
<tr>
<td>LIST</td>
<td>LI dev:file.ex,...</td>
<td>Lists the contents of the specified files on LPT.</td>
</tr>
<tr>
<td>LOAD</td>
<td>LO dev:file.ex</td>
<td>Runs one of the OS/78 loaders for either PAL8, BASIC or FORTRAN IV depending upon the extension of the filename.</td>
</tr>
<tr>
<td>MAP</td>
<td>MAP dev:file.ex&lt;dev:file.ex</td>
<td>Produces a memory map of the specified input file.</td>
</tr>
<tr>
<td>MEMORY</td>
<td>MEM n</td>
<td>Sets the number of memory fields available.</td>
</tr>
<tr>
<td></td>
<td>MEM</td>
<td>Displays both the number of fields actually being used by OS/78 and the number of fields available in the hardware.</td>
</tr>
<tr>
<td>ODT</td>
<td>OD</td>
<td>Loads and starts the ODT (Octal Debugging Technique) system.</td>
</tr>
<tr>
<td>PAL</td>
<td>PAL dev:binfile,dev:listfile, dev:creffile&lt;dev:infile,...</td>
<td>Runs the PAL8 assembler and assembles the specified source file.</td>
</tr>
<tr>
<td>QUEUE</td>
<td>Q filename-list/options</td>
<td>Line printer spooler symbiont command (see Appendix H) to queue files to be printed (DECstation 78 system only).</td>
</tr>
</tbody>
</table>

(continued on next page)
<table>
<thead>
<tr>
<th>Command</th>
<th>Format Example</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>R file.ex</td>
<td>Loads memory image files (except OS/78 system programs) from the system device (making it equivalent to a RUN SYS file.ex command except that the system scratch area is not affected) and starts it at the starting address. If .ex is omitted, .SV is assumed.</td>
</tr>
<tr>
<td>RENAME</td>
<td>REN dev:new.ex&lt;dev:old.ex/options</td>
<td>Changes the name of the file from the input name to the output name.</td>
</tr>
<tr>
<td>REQUEST</td>
<td>REQ symbiotic-name</td>
<td>Starts symbiotic task (DECstation 78 systems only. See Appendix H).</td>
</tr>
<tr>
<td>RUN</td>
<td>RU dev:file.ex</td>
<td>Loads a memory image file (except OS/78 system programs), moves its Core Control Block to the system scratch area, and starts the program at its starting address. This command is equivalent to a GET and START command, and must be used to run a program that does not reside on SYS:</td>
</tr>
<tr>
<td>SAVE</td>
<td>SA dev:file.ex options</td>
<td>Saves an image of the program currently in memory on the device specified. If .ex is omitted, .SV is assumed.</td>
</tr>
<tr>
<td>SET</td>
<td>SET any: [NO] READONLY</td>
<td>Changes device handler characteristics.</td>
</tr>
<tr>
<td></td>
<td>SET LPT: [NO] LC</td>
<td>Makes device read-only device.</td>
</tr>
</tbody>
</table>

(continued on next page)
<table>
<thead>
<tr>
<th>Command</th>
<th>Format Example</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>SET (Cont.)</td>
<td>SET LPT:WIDTH n</td>
<td>Sets column width to n.</td>
</tr>
<tr>
<td></td>
<td>SET LPT:COL n</td>
<td>Sets number of columns displayed by DIRECT (ter: also).</td>
</tr>
<tr>
<td></td>
<td>SET ter: [NO] ARROW</td>
<td>Sets terminal so that CTRL characters map as 'char rather than the actual code.</td>
</tr>
<tr>
<td></td>
<td>SET ter:COL n</td>
<td>Sets number of columns displayed by DIRECT (LPT: also).</td>
</tr>
<tr>
<td></td>
<td>SET ter: [NO] ECHO</td>
<td>Enables character echoing on input.</td>
</tr>
<tr>
<td></td>
<td>SET TTY: [NO] ESC</td>
<td>Sets console terminal so that ESC code maps to a $ rather than 033(8).</td>
</tr>
<tr>
<td></td>
<td>SET ter:HEIGHT m</td>
<td>Sets screen height to m.</td>
</tr>
<tr>
<td></td>
<td>SET SYS: [NO] INIT</td>
<td>Executes command in INIT.CM when system is bootstrapped.</td>
</tr>
<tr>
<td></td>
<td>SET SYS: [NO] INIT cmd</td>
<td>Executes command (cmd) when system is bootstrapped.</td>
</tr>
<tr>
<td></td>
<td>SET ter: [NO] PAGE</td>
<td>Enables CTRL/Q and CTRL/S.</td>
</tr>
<tr>
<td></td>
<td>SET ter: [NO] PAUSE n</td>
<td>Enables pause of n seconds during output.</td>
</tr>
<tr>
<td></td>
<td>SET ter: [NO] SCOPE</td>
<td>Deletes character from screen rather than printing a \ when DELETE key is used.</td>
</tr>
<tr>
<td></td>
<td>SET ter:WIDTH n</td>
<td>Sets column width to n.</td>
</tr>
<tr>
<td></td>
<td>SET HANDLER old&lt;new</td>
<td>Replaces handler currently in system (old) with new handler (new) called: handlername.HN.</td>
</tr>
</tbody>
</table>

(continued on next page)
<table>
<thead>
<tr>
<th>Command</th>
<th>Format Example</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>SET (Cont.)</td>
<td>SET HANDLER/L</td>
<td>Lists names of handlers currently in system.</td>
</tr>
<tr>
<td>SQUISH</td>
<td>SQ dev:&lt;dev:</td>
<td>Eliminates all embedded empty files on the device.</td>
</tr>
<tr>
<td>START</td>
<td>ST ffnnnn</td>
<td>Starts the program currently in memory at location nnnn in field ff.</td>
</tr>
<tr>
<td></td>
<td>ST</td>
<td>Starts the program currently in memory at the starting address specified in the Core Control Block.</td>
</tr>
<tr>
<td>SUBMIT</td>
<td>SU dev:file.ex&lt;dev:file.ex</td>
<td>Runs the BATCH program where the output is the optional spooling file and the input is the BATCH input file.</td>
</tr>
<tr>
<td>TERMINATE</td>
<td>TER</td>
<td>Causes OS/78 to terminate its operation and to run a user-created stand-alone program called TERMIN.SV.</td>
</tr>
<tr>
<td>TYPE</td>
<td>TY dev:file.ex,...</td>
<td>Displays the specified files on the terminal.</td>
</tr>
<tr>
<td>UA</td>
<td>UA (argument)</td>
<td>Remembers the argument where the argument must be a legal OS/78 command. UA with no argument executes the last remembered argument.</td>
</tr>
<tr>
<td>UB</td>
<td>UB (argument)</td>
<td>Similar to UA.</td>
</tr>
<tr>
<td>UC</td>
<td>UC (argument)</td>
<td>Similar to UA.</td>
</tr>
<tr>
<td>ZERO</td>
<td>ZERO dev:/options</td>
<td>Zeros the specified device directory, deleting any files that may exist or the device.</td>
</tr>
</tbody>
</table>
CHAPTER 3

OS/78 COMMANDS

OS/78 system software supports a set of commands that allow you to
direct computer operations. Each OS/78 command has a standard format
that you should be familiar with before using the command.

The format for each command is shown, including one or more examples.
Each command description is followed by any options that you can use
and the error messages that may result when using the command.

You must begin each command on a new line. Enter the command and its
arguments immediately to the right of the period (.), the monitor's
prompt symbol. Press the RETURN key to enter the command into the
system for execution. Characters that are system responses are shown
in black; characters that you type are shown in red.

Commands may or may not require arguments. Many of the commands have
default options that are automatically inserted in the command line by
the system if an argument is not explicitly specified. Default
options may also vary with each command. The use of defaults is
described in Section 2.6. The full-word equivalents for each option
are listed in Appendix I.
3.1 ASSIGN COMMAND

The ASSIGN command associates a user-defined name with one of the permanent logical device names listed in Table 2-1.

Format:

ASSIGN permdev user-name

Example:

.LAS RXA1 DEV2

In this example, device RXA1 is assigned the symbolic name DEV2 and can now be addressed by both RXA1: and DEV2:. Note that the colon is not used with both the permanent device name and the user-defined name in the ASSIGN command. However, when referenced in most other commands the colons must be used with the user-defined name.

If a user-defined name is not specified in the command line, any one that may have existed is removed and only the permanent device name is valid.

Example:

.LAS RXA1 DEV2  -Previous ASSIGN command
.LAS RXA1       -User-defined name becomes invalid

The user-defined name that is assigned to a permanent device can be from one to four alphanumeric characters long.

NOTE

A system error may occur if by coincidence the internal representation of a user-defined name matches some other internal code used by the system.

The ASSIGN command is particularly useful if an LQP line printer is attached to the system instead of an LA78 line printer. Since several of the OS/78 commands automatically send output to the LA78 line printer (LPT), assigning LPT to LQP allows output to be sent to the LQP78 line printer.

Example:

.LAS LQP LPT

ASSIGN processing is done by the monitor.
3.2 BASIC COMMAND

The BASIC command requests execution of the BASIC editor.

Format:

```
BASIC
```

Example:

```
.BASIC
NEW OR OLD--
```

For additional information on BASIC, refer to Chapter 6.

This command runs CCL.SV and BASIC.SV programs.
3.3 **BOOT COMMAND**

The BOOT command allows DECstation 88/80 users to start (bootstrap) the OS/78 system that resides on another disk device.

**Format:**

```
BOOT/dv
```

**where:**

```
dv is the first two characters of the name of the system device (RX for RX02 and RL for RL01) where you want to begin OS/78 operation.
```

**Example:**

```
,BOOT/RX
```

In this example your current system device is the RL01 disk (RL0A). Typing the above command causes the system to read in the monitor from the system diskette mounted in the RX02 disk drive and use it as the system device.

This command causes execution of the CCL.SV and the BOOT.SV programs.
3.4 CANCEL COMMAND

The CANCEL command terminates symbiont task operation and returns OS/78 to stand-alone operation with 16K of memory.

Format:

CANCEL

This command can be used on DECstation 78 series systems only. Refer to Appendix H for more information.
3.5 COMPARE COMMAND

The COMPARE command compares one source input file to another source input file to see if they are identical or different. Differences are sent to the output file.

Format:

COMPARE outdev:file.ex<inddev:file1.ex,inddev:file2.ex/options

where:

outdev:file.ex is the specification of the file containing the results of the comparison and the device (dev:) where you want reside.

inddev:file1.ex is the first input source file for comparison

inddev:file2.ex is the second input source file for comparison

/options is one or more of the options listed in Table 3-1

<table>
<thead>
<tr>
<th>Option</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>/B</td>
<td>Consider a blank line as a valid input line containing blanks instead of a carriage return.</td>
</tr>
<tr>
<td>/C</td>
<td>Ignore all comment fields during the comparison (assumes assembly language source files are being compared).</td>
</tr>
<tr>
<td>/S</td>
<td>Ignore all tabs and spaces during the comparison.</td>
</tr>
<tr>
<td>/T</td>
<td>Convert all tabs from the input file to spaces on the output device.</td>
</tr>
<tr>
<td>/X</td>
<td>Ignore all comment fields during the comparison and do not send any comments to the output device.</td>
</tr>
</tbody>
</table>

NOTE
When you use the /S and /B options together, blank lines are ignored.

Along with the necessary arguments for the command line are COMPARE options that can restrict or enhance the result of the COMPARE operation. The COMPARE program is commonly used to print all the editing changes between two different versions of a program. This makes it a useful documentation and debugging tool.
COMPARE makes the following assumptions:

- If you omit an input device, COMPARE assumes DSK
- If you omit the output specification altogether, COMPARE assumes TTY. (In most cases you will want to see the results on the terminal.)

Example:

```
.COMPA RLOB=DIFVER<RLOC:VERS1.FT,VERS2.FT
```

In this example, every source line in files VERS1.FT and VERS2.FT on RLOC is compared and the results of the comparison are sent to the file named DIFVER on device RLOB. The file extension .LS is automatically added to DIFVER.

Before the first two lines of input are compared, the current version number of the COMPARE command is sent to the output device followed by a header for each input file. The header, the first line in the file, is usually the title, and date, file name, or a description of that file.

Example:

```
SRCCOM V5A
1) /THIS IS THE FIRST SOURCE COMPARE FILE
2) /THIS THE SECOND SOURCE COMPARE FILE
```

The system performs the compare in two passes: the first compares file 1 to file 2 and the second compares file 2 to file 1.

When the COMPARE command is executed, lines from the first file that are either identical to or different from lines in the second file are sent to the output device until there are three consecutive lines that agree. The first line sent to the output device is preceded by the numeral one, a closing parenthesis, and a 3-digit decimal number. The numeral one represents the first file specified in the command line and the 3-digit decimal number is the page that the difference group is located on. Each succeeding line is also preceded by the numeral one and closing parenthesis. When three lines agree, only the first line is sent to the output device. The next line consists of a group of asterisks (*). This ends the first difference group for file 1.

Example:

```
1) 002  C
1)  D
1)  E
1)  F
1)  G
1)  H
1)  I
1)  J
1)  K
1)  L
1)  M
1)  N
****
```
Now the command proceeds to find the first difference group for file 2. It sends all lines from the second file that both agree and disagree with lines from the first file to the output device until there are three consecutive lines that agree.

NOTE

To change the number of consecutive lines that interrupt processing, use the =k option. By specifying the =k option, you can use any octal number to set the number of consecutive lines that separate one difference group from another.

The first line sent to the output device is preceded by the numeral two, a closing parenthesis, and a 3-digit decimal number. The numeral two represents the second file specified in the command line and the 3-digit decimal number is the page the difference group is located on. Each succeeding line is also preceded by the numeral two and closing parenthesis. When three lines agree, only the first line is sent to the output device. The next line consists of a group of asterisks (*). This ends the first difference group for file 2.

Example:

```
2) 002 4
2) 6
2) 2
2) X
2) B
2) Y
2) S
2) F
2) M
2) A
2) R
2) N
****
```

If all the lines on both files agree, a message "NO DIFFERENCES" is sent to the output device. (Header lines are disregarded.) (See Table 3-2 for error messages.)

3.5.1 Using COMPARE's Options

The /C option compares all source lines in one file with source lines in another file but disregards all assembly comment lines. In addition to disregarding comment lines in a comparison, you can restrict the output of the comment lines by using the /X option. When you specify the /X option, all comments are ignored while the comparison takes place and comment lines are not sent to the output device.

If the format of one file is different from that of another because of tabs and spaces, use the /S option. When you specify the /S option, the COMPARE command compares all source lines but disregards all tabs and differing spaces.
Blank lines are normally disregarded when comparing two files. If, in a comparison, a blank line is to be considered as a valid input line instead of a carriage return, use the /B option. The /B option causes a blank line to be considered a valid line containing all blanks.

This command runs the CCL.SV and SRCCOM.SV programs.

Table 3-2
COMPARE Error Messages

<table>
<thead>
<tr>
<th>Message</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>?0</td>
<td>The differences between the two files are too numerous for an effective compare. If the large number of differences is caused by comment lines, use the /X option to eliminate the comment lines.</td>
</tr>
<tr>
<td>?1</td>
<td>File 1 has an input error, or two input files are not specified in the command line.</td>
</tr>
<tr>
<td>?2</td>
<td>File 2 has an input error.</td>
</tr>
<tr>
<td>?3</td>
<td>The output device ran out of room.</td>
</tr>
<tr>
<td>?4</td>
<td>The output has an error.</td>
</tr>
<tr>
<td>?5</td>
<td>An output file cannot be created.</td>
</tr>
</tbody>
</table>
3.6 COMPILe COMMAND

The COMPILe command assembles PAL source input files to produce absolute binary output files, compiles FORTRAN source input files to produce relocatable binary output files and compiles, loads, and executes BASIC source input files.

Format:

COMPILe outdev:file.ex<indev:file.ex/options

Along with the necessary arguments for the command line are several PAL, BASIC, and FORTRAN options that can modify the result of the COMPILe operation. For detailed information on PAL, BASIC, and FORTRAN command formats and options, see the appropriate language chapter.

When the COMPILe command assembles a PAL input file, the absolute binary output file created usually has a .BN extension (the default for PAL8). When the COMPILe command compiles a FORTRAN input file, the relocatable binary output file created usually has an .RL extension (the default for FORTRAN IV). Both absolute and relocatable binary output files may be used as input files in the LOAD command.

An absolute binary file is an independent stand-alone program; a relocatable binary file is a dependent program that needs other system programs to aid it in its execution.

The extension of the first input file determines the compiler to be used. The .PA uses PAL8, .FT uses FORTRAN, .BA uses BASIC. If the extension is not specified as part of the file name in the command line, all assembler and compiler extensions are compared with the unspecified extension until a match is made. (The search order is PAL8 (.PA), FORTRAN (.FT), then BASIC (.BA).) The appropriate assembler or compiler is called and executed. If you use an extension other than the ones listed in Appendix E, specify a general-purpose dash option (see Section 2.5.4.5) to specify the appropriate compiler or assembler.

Example:

.COM RXA1:COMP<ACCTS.03-FT

In this example, the file TEST.FT resides on DSK (default device). Since no extension was given in the command, the system searches for any file named TEST.* (wildcard extension). When it finds TEST.FT, it runs the FORTRAN compiler, which produces the object file TEST.RL (default name and extension).

If no output file is specified, the first input file name specified in the command line is used as the default name of the output file.

Example:

.COMPILE TEST

The output file produced is named TEST.RL.
The COMPILE command can recall arguments from a previous COMPILE, LOAD, EXECUTE, or PAL command. Consequently the COMPILE command can be typed without any arguments until you restart the system with the START or BOOT button or with the BOOT command.

NOTE

Refer to the appropriate language chapter for any error messages received as a result of the COMPILE command.

This command causes execution of the CCL.SV program and the PAL8.SV, F4.SV, or BCOMP.SV programs and optionally the RALF.SV, LOAD.SV, and ABSLDR.SV programs.
3.7 COPY COMMAND

The COPY command transfers files from one device to another and allows you to change a file's name. (COPY command options are in Table 3-3; error messages are in Table 3-4.)

Format:

COPY outdev:file.ex<inputdevice:file1.ex...file5.ex/options

When the COPY command is executed, all specified input files are copied onto the output device. A message listing the name of each file copied is displayed until copying is done.

Example:

.COPY R1A:<PINE,SS,TEAK,ROSE,PA

Files Copied:
   TEAK,SS
   PINE,
   ROSE,PA

The three files are copied from the system device (SYS) to R1A.

3.7.1 File Transfer

When the COPY command is executed, each input file is copied in its exact format to the output device in the same order they are found on the input device. No changes are made during file transfer and as a result, any file, whether memory image, binary or ASCII, can be transferred. During the transfer, the original creation date is transferred and included in the directory. Along with the necessary arguments for the command line, COPY options can be specified that can change and enhance the result of a file transfer.

If input files are to be transferred to the output device in the exact order specified in the command line, use the /U option.

If the majority of the input files on the input device are to be transferred to the output device, use the /V option and specify the files that are not to be transferred. The COPY command then transfers all the files that are not specified in the command line. Wildcards (described in Section 2.5.6) can be used with the COPY command.

3.7.2 Examples of COPY Commands

The following are legal COPY command strings. When the operation is completed, control returns to the monitor. Some of the examples show the use of wildcards.

Example 1:

.COPY RX1:<A,B,C,D,E

This command string transfers the files A, B, C, D, and E from the system device (SYS) to RX1.
Example 2:

```
.COPY LPT:<*.FT,*.BA/U
```

This command string lists all FORTRAN files, then all BASIC files on
the line printer (same as LIST).

Example 3:

```
.COPY LPT:<RXA0:*.SV,*.BN/V
```

This command string copies from RXA0 to the line printer all files
other than memory image (.SV) and binary (.BN) files.

3.7.3 Predeletion

Before the COPY command copies the designated input files to the
output device, the output device is checked for files having the same
name as the input files. If an identical file name is found, that
file is deleted before file transfer is performed. This process is
known as predeletion and, by default, is automatically done each
time the COPY command is executed.

If there are no files with the same name on the output device, each
file is transferred to the smallest available empty space that it can
fit in.

Predeletion is advantageous because it creates space for a new file on
the output device by deleting the old file before file transfer. In
many cases, the output device has received as many files as it can
hold and therefore may not have enough room for a new tentative file.
By first deleting the old file, a sufficient amount of space could be
created for the new file. Predeletion normally places the new file in
the space occupied by the file being replaced. Therefore, if the
length of the input file is the same or shorter than the length of the
deleted file, that empty space is filled. Otherwise, the new
tentative file is placed somewhere else on the device, leaving a gap
where the old file was deleted. This alters the order of the files on
the output device after transfer.

If predeletion is not desired, use the /N option in the command line.
When the /N option is specified, postdeletion takes place.

3.7.4 Postdeletion

If the /N option is used with the COPY command, the designated input
files are copied to a tentative file on the output device without
checking for files having the same name. When the transfer operation
is completed, the tentative file is closed. Closing a tentative file
makes it a permanent file and, at the same time, deletes any old files
having the same name. This process, known as postdeletion, takes
place only when the /N option is specified.

Example:

```
.COPY RLOB:<RLOC:ACCT06.AC/N
```

A file named ACCT06.AC on device RLOC is copied to a tentative file on
device RLOB. After file transfer is completed, the file is closed and
any other file on device RLOB with a filename of ACCT06.AC is deleted.
The primary advantage of postdeletion is that if an input/output error (due, for example, to a faulty disk media) occurs on reading an input file, the corresponding output file will be preserved instead of deleted. Always use the /N option when copying important files to backup diskettes to avoid losing both file and backup file in the event of an input/output error.

3.7.5 Considering the Date on a Transfer

When files are transferred from one device to another, their creation dates are also transferred. These dates, along with their new or original file names, are included in the output directory. If it is desired to assign the current date to an output file and disregard the creation date, use the /T option. This option allows the COPY command to transfer files from one device to another and append the file names and the current date to the output directory.

To transfer all input files that have the current date, use the /C option. When the /C option is used, the COPY command copies only those files that were created or changed on the current date and appends the filenames and their creation dates to the output directory.

However, the transfer of all input files that have a creation date other than the current date is done by using the /O option. By specifying the /O option in the command line, the COPY command copies only those files with a date other than the current date from one device to another, and adds the file names and their creation dates to the output directory.

3.7.6 File Protection During a Transfer Operation

When transferring a large number of files, use the /F (failsafe) option. When the /F option is specified and an input file is encountered which will not fit on the output device, a message is printed on the terminal informing you to dismount the current disk pack or diskette and mount a new one on the same drive. To continue the COPY operation, type any character. If the /F option is not specified, then files which do not fit on the output device are not copied. Instead, a warning message is printed:

NO ROOM, SKIPPING filename

Example:

COPY RXA1:<RXA0:*.* FT/F

FilesCopied:
VERS09.FT
VERS12.FT
VERS10.FT
VERS11.FT
VERS7.FT

Mount Next Output Volume:
VERS5.FT
VERS6.FT
VERS8.FT
Before the /F option is used, all output directories on new devices that might be used for continued COPY operations should be emptied (zeroed). Empty devices by using the ZERO command before the COPY operation begins. You can continue COPY operations to a device with already existing files. However, if the length of the next file to be transferred exceeds the lengths of the empty spaces on the output device, another message for a new device to be mounted is printed on the terminal.

If many files are being transferred at one time, especially ones with similar names, each affected file specified in the command line can be verified as the correct file for the operation by using the /Q (query) option. When the /Q option is specified, the first input file name followed by a question mark (?) is printed on the terminal. At this time, decide whether or not the file is the correct one. If yes, type a Y, and that file will be transferred to the output device. If no, type any other character and that file will be ignored. After your file is either transferred or ignored, the next input file name followed by a question mark is printed on the terminal. This questioning continues until all affected files in the command line have been processed.

Example:

`.COPY SYS:<RL0B:>*.PA/Q

Files Copied:
MAIN.PA?/Y
SUB1.PA?/N
SUB2.PA?/Y
LIB1.PA?/N
LIB2.PA?/Y

In this example, the files named SUB1.PA and LIB1.PA are not transferred.

3.7.7 Terminating COPY Operations

The only control character used to abort COPY operations is CTRL/C.

When you type CTRL/C during a COPY operation, file transfer terminates immediately and control returns to the monitor. Because file transfer is terminated before a successful completion, some requested files may not have been transferred.

Example:

`.COPY RXA1:<RXA0:TABLE.FT
  "C
FILES COPIED:
.

This command executes the CCL.SV and FOTP.SV programs.
### OS/78 COMMANDS

#### Table 3-3
COPY Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>/C</td>
<td>Transfers all input files that have the current date to the output device.</td>
</tr>
<tr>
<td>/P</td>
<td>Displays the message: MOUNT NEW DEVICE: on the terminal when a file will not fit on the output device. Dismount current device and mount new device on same unit. Continue the COPY operation by typing any character. The new device must have a valid OS/78 directory on it.</td>
</tr>
<tr>
<td>/N</td>
<td>Transfers each file as a tentative file, then deletes any existing file having the same name and makes the tentative file permanent. This is postdeletion.</td>
</tr>
<tr>
<td>/O</td>
<td>Transfers only those input files with dates other than the current date.</td>
</tr>
<tr>
<td>/Q</td>
<td>Displays the name of each file and pauses to allow you to select (type Y) or reject (type any other character) the transfer operation for this file.</td>
</tr>
<tr>
<td>/T</td>
<td>Transfers all specified input files to the output device and at the same time, change the creation date on the output files to the today's date.</td>
</tr>
<tr>
<td>/U</td>
<td>Finds and transfers the designated input files in the exact order they are specified in the command line, not the order they are found on the input device.</td>
</tr>
<tr>
<td>/V</td>
<td>Transfers all files on the input device that are not specified in the command line to the output device.</td>
</tr>
</tbody>
</table>

#### NOTE
All output directories on devices that could be used for continued COPY operations (/P) must have valid OS/78 directories on them. Use the ZERO command to create as many blank disks or diskettes as you require for the operation.
### Table 3-4
COPY Error Messages

<table>
<thead>
<tr>
<th>Message</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>BAD INPUT DIRECTORY</td>
<td>The directory on the specified input device does not exist or is not valid.</td>
</tr>
<tr>
<td>BAD OUTPUT DEVICE</td>
<td>A non-file structured device is specified as the output device.</td>
</tr>
<tr>
<td>BAD OUTPUT DIRECTORY</td>
<td>The directory on the specified output device does not exist or is not valid.</td>
</tr>
<tr>
<td>ERROR ON INPUT DEVICE SKIPPING (file name)</td>
<td>The file specified is not transferred, due to a disk-reading error, but any previous or subsequent files are transferred and indicated in the new directory.</td>
</tr>
<tr>
<td>ERROR ON OUTPUT DEVICE SKIPPING (file name)</td>
<td>The file specified is not transferred, but any previous or subsequent files are transferred and indicated in the new directory.</td>
</tr>
<tr>
<td>ERROR READING INPUT DIRECTORY</td>
<td>A disk-reading error occurred on the input device. No files are transferred.</td>
</tr>
<tr>
<td>ERROR READING OUTPUT DIRECTORY</td>
<td>A disk-reading error occurred on the output device. No files are transferred.</td>
</tr>
<tr>
<td>ERROR WRITING OUTPUT DIRECTORY</td>
<td>A disk-writing error occurred on the output device. The device's directory is likely to be unusable.</td>
</tr>
<tr>
<td>ILLEGAL*</td>
<td>An * was entered as an embedded character in a file name, e.g., TMP*,BN (see Section 2.5.6 on wildcards).</td>
</tr>
<tr>
<td>ILLEGAL?</td>
<td>A ? was entered in an output specification (see Section 2.5.6 on wildcards).</td>
</tr>
<tr>
<td>NO FILES OF THE FORM xxxx</td>
<td>No files of the form (xxxx) specified were found on the current input device group.</td>
</tr>
<tr>
<td>NO ROOM, SKIPPING (file name)</td>
<td>No space is available on the output device to perform the transfer. Predeletion may already have occurred.</td>
</tr>
<tr>
<td>SYSTEM ERROR-CLOSING FILE</td>
<td>Bad directory format or system head has become corrupted.</td>
</tr>
</tbody>
</table>
3.8 CREATE COMMAND

The CREATE command calls and runs the OS/78 Editor to allow a new file to be opened. Once opened, the new file is ready to receive any text typed in. Only one output file specification is allowed.

Format:

CREATE outdev:file.ex/options

After you press the RETURN key, the Editor displays the number sign (#), its prompt symbol, on the terminal to indicate that it is ready to receive commands. (See Chapter 4 for a detailed explanation of the OS/78 Editor and its commands and options.)

Example:

    .CREATE RLOB:RUN1.PA
    $

In this example, the CREATE command opens a file named RUN1.PA on output device RLOB.

Each time a CREATE command is executed, the accompanying arguments (device and file name) are stored in a temporary area for subsequent use by the EDIT command. This allows you to type the EDIT command without any arguments to continue editing a file that you have previously created. These arguments are not available after you restart the system with the START or BOOT button or with the BOOT command. The EDIT command cannot recall file names that have no date assigned. Therefore, use the DATE command prior to the CREATE command.

This command causes the execution of both the CCL.SV and EDIT.SV programs.
3.9 CREF COMMAND

The CREF command assembles a PAL8 program and produces a cross-reference listing. (CREF commands are in Table 3-5; error messages are in Table 3-6.)

Format:

    CREF outdev:file.ex<indev:file.ex/options

Example:

    .CREF PROGA

The cross-reference listing contains the source program with the assembled instructions followed by the cross-reference table, which contains every user-defined symbol and literal. These listings aid in writing, debugging and maintaining assembly language programs. Using CREF is the same as using the PAL command with the /C option specified. If no output device is specified, LPT is the default output device.

This command executes the PAL8.SV, CCL.SV, and CREF.SV programs.

Table 3-5
CREF Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>/E</td>
<td>Do not eliminate the intermediate file CREFLS.TM that is output from the assembly and used as input to produce the CREF listing.</td>
</tr>
<tr>
<td>/M</td>
<td>Cross-reference mammoth files in two passes. Pass 1 processes the symbols from A through LGnnnn; pass 2 processes the symbols from LHnnnn through Z and literals. This permits very large files to be cross-referenced.</td>
</tr>
<tr>
<td>/P</td>
<td>Disable pass 1 listing output. The output is reenabled when $ is encountered. Thus the $ and symbol table are printed if the /P option is used.</td>
</tr>
<tr>
<td>/U</td>
<td>Disable pass 1 listing output and the symbol table.</td>
</tr>
<tr>
<td>/X</td>
<td>Do not process literals. For programs with too many symbols and literals for CREF, this option may create enough space for CREF to operate.</td>
</tr>
</tbody>
</table>
## OS/78 Commands

### Table 3-6
**CREF Error Messages**

<table>
<thead>
<tr>
<th>Message</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>2045 REFS</td>
<td>More than 2044 (decimal) references to one symbol were made.</td>
</tr>
<tr>
<td>CLOSE FAILED</td>
<td>CLOSE on output file failed.</td>
</tr>
<tr>
<td>ENTER FAILED</td>
<td>Entering an output file was unsuccessful - possibly output was specified to a read only device.</td>
</tr>
<tr>
<td>HANDLER FAIL</td>
<td>This is a fatal error on output.</td>
</tr>
<tr>
<td>INPUT ERROR</td>
<td>A read from the input device failed.</td>
</tr>
<tr>
<td>OUT DEV FULL</td>
<td>The output device is full (directory devices only).</td>
</tr>
<tr>
<td>SYM OVERFLOW</td>
<td>More than 896 (decimal) symbols and literals were encountered during a major pass. Try again using the /M option.</td>
</tr>
</tbody>
</table>
3.10 DATE COMMAND

The DATE command lets you set and inspect the current system date. You should always set the date wherever you start the system with the START or BOOT button or with the BOOT command. The system appends the current date to all files that you create, rename, or copy and prints it in directory listings and program output listings.

Format:

DATE dd-mmm-yy

where:

dd is a 2-digit decimal number representing the day,

mmm is a 3-character alphabetic string representing the month which must be the first three letters of the month, and

yy is a 2-digit decimal number representing the last two digits of the year.

NOTE

The only directory entry dates that the system can read are those for the current year and seven years preceding the current year. Any earlier dates will be printed incorrectly. If you want to examine directories with earlier dates, do not set the date when you start the system.

Example:

.DOATE 1-MAR-79

If the DATE command is typed without any arguments after the date has already been entered by a previous DATE command, the current day of the week and date are displayed.

Example:

.DOATE 1-JUN-77
.DOATE
.THURSDAY MARCH 1, 1979

If the date is specified incorrectly, an error message is displayed.

Example:

.DOATE 23-05-79
.BAD DATE

This command executes the CCL.SV program and is processed by the monitor.
DEASSIGN

3.11 DEASSIGN COMMAND

The DEASSIGN command removes any user-defined names that are currently assigned to the permanent devices.

Format:

DEASSIGN

Example:

.DEAS SYS DEV1
.DE

In this example, the system device can no longer be referenced as DEV1. Only SYS, DSK, RXA0, or RL0s are valid.

The monitor performs the DEASSIGN function.
3.12 **DELETE COMMAND**

The DELETE command deletes all specified files. (DELETE command options are in Table 3-8; error messages are in Table 3-9.)

**Format:**

```
DELETE indev:file1.ex...file5.ex/options
```

In response to the monitor's prompt (.), type the names of the files that you want to delete, then press the RETURN key. This will delete the files from the specified device. Wildcards are permitted. Only one device may be specified.

**Example:**

```
*DEL RLOB:REPORT.FT,SECUR.FT,ASSIST
```

Files Deleted:

| SECUR.FT |
| REPORT.FT |
| ASSIST |

A message followed by the name of each file deleted is displayed on the terminal.

**CAUTION**

If you do not specify any filenames, **.*.*** (meaning all files) is the default input specification.

This command deletes all copies of your system FORTRAN files that are on RXA1.

**Example:**

```
*DEL RXA1:*.*<SYS:*.*.FT
```

An advanced format that can be used with the DELETE command is:

```
DELETE outdev:file.ex<indev:file 1,.../options
```

If an output specification is given, then this command works differently. Instead of deleting the input files specified, the command lists those file names that match the given file specification, and that are on the input device. This list is then transformed to an output list by applying the wildcards specified in the output specification. The resulting list is then used to delete files from the output device specified.

This command executes the CCL.SV and FOTP.SV programs.
### Table 3-7
**DELETE Options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>/C</td>
<td>Delete only those specified files having the current date.</td>
</tr>
<tr>
<td>/N</td>
<td>Displays the names of all files on the output device that match the specification, but does not delete them.</td>
</tr>
<tr>
<td>/O</td>
<td>Delete only those specified files with dates other than the current date.</td>
</tr>
<tr>
<td>/Q</td>
<td>Display each file name on the terminal; type a Y (for yes) to delete the file or any other character (for no) if the file is not to be deleted.</td>
</tr>
<tr>
<td>/V</td>
<td>Delete all files that are not specified in the command line.</td>
</tr>
</tbody>
</table>

### Table 3-8
**DELETE Error Messages**

<table>
<thead>
<tr>
<th>Message</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>BAD OUTPUT DEVICE</td>
<td>This message usually appears when a non-file structured device is specified as the output device.</td>
</tr>
<tr>
<td>BAD OUTPUT DIRECTORY</td>
<td>The directory on the specified output device is not a valid OS/78 device directory.</td>
</tr>
<tr>
<td>DELETES PERFORMED ONLY ON INPUT</td>
<td>More than one input device was specified with the /D option when no output specification (device or file name) was included.</td>
</tr>
<tr>
<td>DEVICE GROUP 1 CAN'T HANDLE MULTIPLE DEVICE DELETES</td>
<td></td>
</tr>
<tr>
<td>ERROR READING OUTPUT DIRECTORY</td>
<td>Self-explanatory.</td>
</tr>
<tr>
<td>ERROR WRITING OUTPUT DIRECTORY</td>
<td>Self-explanatory.</td>
</tr>
<tr>
<td>ILLEGAL*</td>
<td>An * was entered as an embedded character in a file name, for example, TMP*.BN.</td>
</tr>
<tr>
<td>ILLEGAL?</td>
<td>A ? was entered in an output specification.</td>
</tr>
<tr>
<td>NO FILES OF THE FORM xxxxx</td>
<td>No files of the form (xxxx) specified were found on the current input device group.</td>
</tr>
</tbody>
</table>
3.13 DIRECT COMMAND

The DIRECT command sends the directory listing for file-structured devices to an output device. The terminal is the default output device. (DIRECT command options are in Table 3-9; error messages are in Table 3-10.)

Format:

```
DIRECT outdev:listfile<indev:filetype,.../options
```

Example:

```
.DIRECT RXA1:
```

Along with the necessary arguments for the command line are the DIRECT options (shown in Table 3-9) that control the DIRECT operation.

<table>
<thead>
<tr>
<th>Option</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>/B</td>
<td>Include the starting block numbers (octal) for each file in the directory listings.</td>
</tr>
<tr>
<td>/C</td>
<td>List only files with the current date, that is, the date entered with the most recent DATE command.</td>
</tr>
<tr>
<td>/E</td>
<td>Include empty file spaces in the directory listing.</td>
</tr>
<tr>
<td>/F</td>
<td>List only the file names and omit the lengths and dates.</td>
</tr>
<tr>
<td>/M</td>
<td>List only the empty files from the directory.</td>
</tr>
<tr>
<td>=n</td>
<td>List the directory in the number of columns specified by n. The n is a number ranging from 1-7.</td>
</tr>
<tr>
<td>/O</td>
<td>List only those file names from the directory with dates other than the current date.</td>
</tr>
<tr>
<td>/R</td>
<td>List the file name specified in the command line and the names of any files stored on disk following that specified file name.</td>
</tr>
<tr>
<td>/U</td>
<td>List each input specified as a separate directory listing with the file lengths and dates in the exact order specified in the command line.</td>
</tr>
<tr>
<td>/V</td>
<td>List all the files on the input device that are not specified in the command line.</td>
</tr>
</tbody>
</table>
The standard directory listing has the following format:

```
filename.ex    nnn    dd-mmm-yy
```

(filename with extension)   (number of blocks used in decimal)   (day, month, year)

NOTE

If the date is not entered into the system via the DATE command, directory listings will not contain file creation dates.

If no output device is specified, the output defaults to the terminal. However, if you want the directory printed on LPT, use the -L general-purpose dash option in the command line.

Example:

```
.DIR RLOC!-L
```

is equivalent to

```
.DIR LPT!<RLOC!
```

Wildcards are permitted in the input specification but are not allowed in the output specification.

Example:

```
.DIR SYS!*.*
```

As a result, only those files with FT extensions in the directory are displayed on the terminal.

Example:

```
.DIRECT RXA1!WN????.*
```

will display all the files from the directory on RXA1 that have a file name two to six characters long beginning with WN, and having any extension.

Example:

```
.DIR RXA1!DIRECT.DI<SYS!
```

saves the directory listing of the system device and stores it in a file named DIRECT.DI on device RXA1.

3.13.1 Directory Listing Selected by Date

Use the /C option to display the names of the files that have the current date on the terminal. To display only those file names from the directory with a date other than the current date, use the /O option in the command line.
3.13.2 Directory Format Selection

To display just the file names from the directory, use the /F option in the command line. The /F option displays only the file names and omits all block lengths and dates.

The /E option displays file names with their extensions, lengths, and dates, and empty files with their lengths in the exact order that reside on the device.

When the /M option is specified, each empty file and its file length are displayed. Do not mix the /M option with the other options.

The =n option displays the list of file names in the number of columns specified by n. The value of n must be in the range 1-7 inclusive. The values 1 through 7 can be substituted for n. For use on the terminal, values 1 through 3 are suggested. Values more than 3 are normally used for the line printer. Two columns is the default value. You can change this value with the SET command.

To display a specific file name and all file names from that point on, use the /R option in the command line. This option should not be mixed with other options.

If there are a large number of file names from the directory to be displayed, use the /V option. When you specify the /V option, all file names that are not specified by the command line are displayed.

The starting block number (octal) of each file in the directory is displayed by using the /B option. The starting block numbers are in the form of nnnn and displayed following the file name and extension.

This command causes the execution of both the CCL.SV and DIRECT.SV programs.

<table>
<thead>
<tr>
<th>Table 3-10</th>
<th>DIRECT Error Messages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message</td>
<td>Meaning</td>
</tr>
<tr>
<td>BAD INPUT DIRECTORY</td>
<td>This message occurs when the directory is non-existent or not in legal format.</td>
</tr>
<tr>
<td>DEVICE DOES NOT HAVE A DIRECTORY</td>
<td>This input device is a non-directory device, for example, TTY or LPT.</td>
</tr>
<tr>
<td>EQUALS OPTION BAD</td>
<td>The =n option is not in the range 1-7.</td>
</tr>
<tr>
<td>ERROR CLOSING FILE</td>
<td>System error.</td>
</tr>
<tr>
<td>ERROR READING INPUT DIRECTORY</td>
<td>A disk-read error occurred while trying to read the directory.</td>
</tr>
<tr>
<td>ERROR WRITING FILE</td>
<td>A disk-write error occurred while trying to write the output file.</td>
</tr>
<tr>
<td>ILLEGAL*</td>
<td>An asterisk (*) was included in the output file specification or an illegal * was included in the input file name.</td>
</tr>
<tr>
<td>ILLEGAL?</td>
<td>A question mark (?) was included in the output file specification.</td>
</tr>
<tr>
<td>NO ROOM FOR OUTPUT FILE</td>
<td>The output device does not have sufficient space for the directory listing file to be written.</td>
</tr>
</tbody>
</table>
3.14 DUPLICATE COMMAND

The DUPLICATE command copies the entire contents of one diskette to another diskette. If your system has an RX02 diskette drive, the command allows you to format diskettes for single or double-density operation. (DUPLICATE command options are in Table 3-11; error messages are in Table 3-12.)

NOTE
You cannot use this command to duplicate RL01 disk packs. Use the procedures described in Section 2.2.

Format:

DUPLICATE outdev:<indev>/options

where:

outdev: is the output device (RXA0 thru RXA3).
indev: is the input device (RXA0 thru RXA3).
/options is one or more nonconflicting options listed in Table 3-11.

If, in a transfer operation, the density of the output diskette is different from that of the input diskette, DUPLICATE automatically reformats the output diskette to the same density as the input diskette before proceeding. DUPLICATE can use up to 16K of memory for copy operations. To obtain the maximum copying speed, use the MEMORY command to provide the necessary memory space.

Example:

.DUPLICATE RXA1:<RXA0:
The contents of input device RXA0 is copied onto output device RXA1.

NOTE
If your system has both RX01 and RX02 drives, you can duplicate between them but only with single-density diskettes.

3.14.1 Changing Devices Before and After Executing the DUPLICATE Command

Since the monitor resides on the system device, the system device must remain on line when interacting with the monitor and any OS/78 system programs.
To duplicate the contents of a diskette containing only files (one that does not contain a system head), and your system has only two diskette drives, use the /P option. This causes the system to pause before and after the selected operation. A ready message followed by a question mark is displayed. This pause provides time to remove the system diskette and mount a new diskette for the actual duplication. To start the DUPLICATE operation, type a Y after the question mark and press the RETURN key. After the DUPLICATE operation is completed, a message is displayed asking if the diskette containing the monitor is remounted. The second pause provides time to remove the new copy and remount the system diskette so control can return to the monitor. After remounting the system diskette, type a Y after the question mark and press the RETURN key. Control returns to the monitor.

Example:

. DU RXA0:<RXA1:/P
   READY?Y
   IS MONITOR REMOUNTED?Y

3.14.2 Performing a Read Check

To check the integrity of a diskette, use the /R option and specify only the input device. When you specify the /R option, every block of the specified device is read and checked for bad sectors. If bad sectors exist, messages indicating the device, track number and sector number are displayed. If there are no bad sectors, control returns to the monitor.

Example:

. DU <RXA1:/R
   INPUT DEV READ ERROR TRACK:nn SEC:nn

   NOTE

   Bad sectors are usually a result of medium wear or mistreatment. Move your files onto a new diskette rather than using one with bad sectors.

3.14.3 Transfer Without Checking for Identical Contents

To transfer the contents of one device to another without performing any checks, use the /N option. When you specify the /N option, the contents of the input device is transferred or copied to the output device. If /N is not specified, each block on the output device is compared to that of the input device to assure accuracy.
3.14.4 Check for Identical Contents Without Transferring

To check whether the contents of diskettes are identical without transferring, use the /M option. The /M option reads and checks the contents of both devices for identical contents. If they do match, control returns to the monitor. However, if they differ in any way, a message, containing the device name, the track number, and the sector number of the sectors or blocks that do not match is displayed.

Example:

```
.DU RXA1:<RXA0:/M
COMPARE ERROR TRACK nn SECTOR nn
```

3.14.5 Formatting Diskettes (RX02 Drives Only)

The /S and /D options format diskettes for single-density or double-density use. To format a diskette, enter its name by itself in the command line, followed by the option. (RXCOPY considers a device name entered by itself to be an output device.)

For example, the following command sequence reformats the diskette in drive 1 from single density to double density.

```
.DU RXA1:/D
```

To change it back to single-density, type

```
.DU RXA1:/S
```

This command causes the execution of both the CCL.SV and the RXCOPY.SV programs.

<table>
<thead>
<tr>
<th>Option</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>/D</td>
<td>Format the output diskette for double-density use (RX02 drives only).</td>
</tr>
<tr>
<td>/M</td>
<td>Check both devices for identical contents and list the areas that do not match but do not perform a transfer unless otherwise specified.</td>
</tr>
<tr>
<td>/N</td>
<td>Copy the contents of one device to another but do not check them for identical contents unless otherwise specified.</td>
</tr>
<tr>
<td>/P</td>
<td>Pause and wait for user response before and after data transfers to/from diskette. To continue, type a Y. To abort, type any other character.</td>
</tr>
<tr>
<td>/R</td>
<td>Read every block on the input device and list the bad sectors but do not perform a transfer unless otherwise specified.</td>
</tr>
<tr>
<td>/S</td>
<td>Format the output diskette for single-density use (RX02 drives only).</td>
</tr>
<tr>
<td>Message</td>
<td>Meaning</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
<tr>
<td>CAN'T LOAD INPUT DEVICE</td>
<td>The name of the input device specified in the command line is not a permanent device name.</td>
</tr>
<tr>
<td>CAN'T LOAD OUTPUT DEVICE</td>
<td>The name of the output device specified in the command line is not a permanent device name.</td>
</tr>
<tr>
<td>COMPARE ERROR</td>
<td>When using the /M option all the areas that do not match are printed as COMPARE ERRORS.</td>
</tr>
<tr>
<td>DEVICE IS NOT RX</td>
<td>Either the input or output device specified is not one of the following:</td>
</tr>
<tr>
<td></td>
<td>RXA0: SYS;</td>
</tr>
<tr>
<td></td>
<td>RXA1: DSK;</td>
</tr>
<tr>
<td>INPUT DEVICE READ ERROR</td>
<td>Bad data on input device; bad sectors.</td>
</tr>
<tr>
<td>NO INPUT DEVICE</td>
<td>No input device is specified.</td>
</tr>
<tr>
<td>OUTPUT DEVICE READ ERROR</td>
<td>Bad data on output device; bad sectors.</td>
</tr>
<tr>
<td>OUTPUT DEVICE WRITE ERROR</td>
<td>Hardware disk-write error.</td>
</tr>
<tr>
<td>WRONG FLOPPY TYPE</td>
<td>Attempt to duplicate a double-density diskette between an RX02 drive and an RX01 drive.</td>
</tr>
</tbody>
</table>
3.15 EDIT COMMAND

The EDIT command calls the OS/78 Editor which allows the editing of already existing files. These files must contain ASCII source code.

Format:

```
EDIT outdev:file.ext<indev:file.ext/options
```

In response to the monitor's prompt (.), type the necessary command line and press the RETURN key. The file is now open and the output file is ready for modification. After you press the RETURN key, the number sign (#) is displayed on the terminal, informing you to proceed with an Editor input command. (See Chapter 4 for a detailed explanation of the OS/78 Editor and its options.)

Example:

```
.EDIT RLOB:FILE1.FT<RL1B:TABLE.FT
#
```

In this example, the EDIT command opens output file FILE1.FT on output device RL0B: and opens input file TABLE.FT on input device RL1B:.

**NOTE**

Since no output device is specified, the default option (DSK:) is used as the output device.

The EDIT command can recall arguments from a previous CREATE or EDIT command. This allows the EDIT command to be used without any arguments. When you restart the system with the START or BOOT button or with the BOOT command, the arguments are lost. However, if you specify both an input and an output file, only the arguments up to but not including the left-angle bracket (<) are remembered.

Example 1:

```
.DO 27-FEB-79
.DO
TUESDAY FEBRUARY 27, 1979
.DO CREATE MATCH.PA
.#A
.
.
.#E
.DO EDIT
.#R
(MATCH.PA is remembered by the EDIT command)
.
.
.#E
```
The above example shows a sequence of commands in creating and editing of file MATCH.PA. The EDIT command is given without any argument since it remembers the file MATCH.PA specified with the CREATE command.

If the date has changed and a command line is typed containing the EDIT command without any arguments, the error message BAD RECOLLECTION will be displayed.

Example 2:

```
.EDIT A2.FT<A1.FT
$R
. . . .
$E . . .
.EDIT
```

Since the second EDIT command had no arguments, the previous argument up to the left-angle bracket (<) is recalled. Thus, the second EDIT command is equivalent to:

```
.EDIT A2.FT
```

This command executes the CCL.SV and EDIT.SV programs.
EXECUTE

3.16 EXECUTE COMMAND

The EXECUTE command performs the following functions:

1. assemble or compile, link, load and execute a BASIC, FORTRAN, or PAL8 source program
2. link, load and execute an already assembled or compiled program
3. execute an already linked and loaded program

Format:

EXECUTE indev:file.ex

To use the EXECUTE command with a source file, the input device must be specified. Then that source file is assembled or compiled, linked and loaded into memory and executed. The file extension used determines the compiler or assembler called.

If the file name extension is not a standard extension, specify the correct assembler or compiler by using one of general-purpose dash options in the -ex portion of the command line (see Table 2-3).

Example:

.EXE RXA1:NABS.IN-PA

If the EXECUTE command is used with an already assembled or compiled program, the input device must be specified. When the EXECUTE command is processed, the binary file is loaded into memory and executed. If the file name extension is not specified in the command line but is specified in the directory, the correct extension is automatically added.

The EXECUTE command can remember arguments from a previous COMPILE, LOAD, PAL, or EXECUTE command. When you restart the system with the START or BOOT button or with the BOOT command, the arguments are lost.

NOTE

For options and expanded command strings acceptable to the EXECUTE command, refer to the appropriate language/loader section.

This command executes CCL.SV and one or more of the following programs: PAL8.SV and ABSLDR.SV, F4.SV, FRTS.SV and LOAD.SV, or BCOMP.SV and BLOAD.SV.
3.17 FORMAT COMMAND

The FORMAT command verifies and formats RL01 cartridge disk packs. You must use FORMAT on each new disk pack before you can store any OS/78 files on it. You should also use FORMAT to reformat any disk pack that has blocks that the system cannot access to make them invisible to the system. (After using FORMAT, you must also use the ZERO command to create a new directory.)

NOTE

FORMAT does not alter the contents of a disk pack that has no bad blocks. If there are bad blocks, the data residing in the good blocks that follow a bad block will become unusable after reformatting, so be sure to make copies of the files that you want to save before using FORMAT.

Format:

FORMAT RL01/n/P

where:

/n specifies the drive number, 0 or 1. The default value is 1.

/P causes the system to pause before and after the operation so you can change disk packs. If you select drive 0, the pause is automatic. Type a CTRL/C to return to the monitor. If you are running under BATCH, FORMAT does not pause after it completes the operation.

Example:

..FORMAT RL01

This command formats the disk pack on drive 1.

The FORMAT command executes the RLFRMT.SV program.
### Table 3-13
FORMAT Error Messages

<table>
<thead>
<tr>
<th>Message</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>BAD COMMAND LINE</td>
<td>The command line contains a syntax error. Retype the line correctly.</td>
</tr>
<tr>
<td>BAD DISK</td>
<td>Disk pack cannot be reformatted. There are more than 63 bad blocks or</td>
</tr>
<tr>
<td></td>
<td>the system area (blocks 0 - 70 octal) contains bad blocks.</td>
</tr>
<tr>
<td>CAUTION - THIS COMMAND DESTROYS CONTENTS OF SYSTEM DISK. CHANGE DISKS BEFORE PROCEEDING OR CTRL/C.</td>
<td>Drive 0 was selected for formatting. Be sure to replace your system disk</td>
</tr>
<tr>
<td></td>
<td>pack with the one that you want to format.</td>
</tr>
<tr>
<td>READY, STRIKE CARRIAGE RETURN TO CONTINUE</td>
<td>The /P option was selected. Mount the disk pack to be formatted and</td>
</tr>
<tr>
<td></td>
<td>type a carriage return to begin formatting.</td>
</tr>
</tbody>
</table>
3.18 GET COMMAND

The GET command loads memory image files with .SV extensions from an input device back into the same location in memory from which it was saved.

Format:

GET indev:file.ex

Example:

.GET RXA0:JOBCNT

If you specify a file name with no extension, an .SV extension is assumed.

Example:

.GET RLOB:FILE3

In this example, FILE3.SV is loaded into memory from device RLOB.

The specified input file and its Job Status Word are loaded into memory. The entire Core Control Block is sent to the system device where it can be referenced and maintained (see Section 3-31).

The GET command is commonly used to load save files from a device into memory. These save files can be executed either by the START or the EXECUTE command. In addition, you can debug them with octal debugging program (see ODT command).

NOTE

You cannot use the GET command to execute OS/78 system programs.

The monitor performs the GET function.
3.19 HELP COMMAND

The HELP command accesses the help command's text file called HELP.HL and displays its contents on the terminal. The file contains useful information about the OS/78 commands that you specify as an argument to the command (see Table 3-14). (See Table 3-15 for HELP error commands.)

Format:

```
HELP outdev:file.ex<arg
```

where:

```
outdev:file.ex is the output device and file name where the help information is being sent. If no output device is specified, the terminal (TTY) is assumed. If no output file extension is specified the .HL extension is assumed.
```

```
arg is usually an OS/78 command (see Table 3-14).
```

By typing the HELP command without any arguments, you can obtain a list of all OS/78 commands for which you can get help.

```
HELP
```

If you want details about a specific command, type the name of the command for which the information is desired, for example:

```
HELP PAL
```

To obtain a list of all legal arguments for HELP, type the HELP command followed by an asterisk or type the HELP command followed by 'HELP' as follows:

```
HELP *
```

or

```
HELP HELP
```

This command runs the HELP.SV and CCL.SV programs.
## Table 3-14
OS/78 Help Command Arguments

<table>
<thead>
<tr>
<th>Command</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASSIGN</td>
<td>LOAD</td>
</tr>
<tr>
<td>BASIC</td>
<td>MAP</td>
</tr>
<tr>
<td>BOOT</td>
<td>MEMORY</td>
</tr>
<tr>
<td>CANCEL</td>
<td>ODT</td>
</tr>
<tr>
<td>COMPARE</td>
<td>PAL</td>
</tr>
<tr>
<td>COPY</td>
<td>PALERR</td>
</tr>
<tr>
<td>CREATE</td>
<td>QUEUE</td>
</tr>
<tr>
<td>CREF</td>
<td>R</td>
</tr>
<tr>
<td>DATE</td>
<td>RENAME</td>
</tr>
<tr>
<td>DEASSIGN</td>
<td>REQUEST</td>
</tr>
<tr>
<td>DELETE</td>
<td>RUN</td>
</tr>
<tr>
<td>DIRECT</td>
<td>SAVE</td>
</tr>
<tr>
<td>DUPLICATE</td>
<td>SET</td>
</tr>
<tr>
<td>EDIT</td>
<td>SQUISH</td>
</tr>
<tr>
<td>EXECUTE</td>
<td>START</td>
</tr>
<tr>
<td>F4ERR</td>
<td>SUBMIT</td>
</tr>
<tr>
<td>FORMAT</td>
<td>TERMINATE</td>
</tr>
<tr>
<td>GET</td>
<td>TYPE</td>
</tr>
<tr>
<td>HELP</td>
<td>UA</td>
</tr>
<tr>
<td>LIST</td>
<td>ZERO</td>
</tr>
</tbody>
</table>

## Table 3-15
HELP Error Messages

<table>
<thead>
<tr>
<th>Message</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEVICE FULL</td>
<td>The output device becomes full when writing or is already full before output begins.</td>
</tr>
<tr>
<td>FETCH ERR</td>
<td>Cannot load handler for the specified device.</td>
</tr>
<tr>
<td>NO HELP</td>
<td>There is no file named HELP.HL on the system device.</td>
</tr>
<tr>
<td>NO HELP FILE</td>
<td>The input file specified does not exist as a HELP file.</td>
</tr>
<tr>
<td>NO TTY HAND</td>
<td>There is no handler in the system for the terminal.</td>
</tr>
<tr>
<td>READ ERR</td>
<td>An error occurred while reading the specified HELP file.</td>
</tr>
<tr>
<td>WRITE ERR</td>
<td>An error occurred while writing to the output file.</td>
</tr>
</tbody>
</table>
3.20 LIST COMMAND

The LIST command prints the contents of the specified input files on the line printer (LPT).

Format:

LIST indev:file1.ex...file5.ex/options

Example:

.LI PROD.BA

Options for the LIST command are shown in Table 3-16. Error messages are shown in Table 3-17.

When you use the LIST command, the contents of each input file are listed on LPT in the same order the file names are specified in the command line.

Example:

.LI RXA11FL1.FT,FL2.FT,MASSFIL.FT

Although the format specifies that only five files can be named in the command line, more than five files can be listed on the line printer through the use of the wildcard construction.

You can restrict the list of files you want with the command line options. When you specify the /C option, all the specified input files with the current date are listed. If the /O option is used, all specified input files with dates other than the current date are listed. If the /V option is used all files other than those you specified in the command line are listed. If you do not know which files you want, use the /Q option to go through the directory file by file. When you specify the /Q option, each file name followed by a question mark is printed on the terminal so you can decide whether or not you want that file in the list. If yes, type a Y and that file is listed. If no, type any other character and that file is ignored. This continues until all specified input files have been processed.

Example:

.LIST*.PA/Q
TOT.PA?Y
MASSFIL.PA?N
UPDT06.PA?N
FINTOT.PA?Y

As a result, the files named MASSFIL.PA and UPDT06.PA are not listed.

This command causes the execution of both the CCL.SV and FOTP.SV programs.

3-40
**OS/78 COMMANDS**

Table 3-16
LIST Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>/C</td>
<td>List only those files having the current date.</td>
</tr>
<tr>
<td>/O</td>
<td>List only those files having a date other than the current date.</td>
</tr>
<tr>
<td>/Q</td>
<td>Each time a file name followed by a question mark is printed on the terminal, type a Y (yes) to list the files or any other character (No) if the file is not to be listed.</td>
</tr>
<tr>
<td>/V</td>
<td>List all the files not specified in the command line.</td>
</tr>
</tbody>
</table>

Table 3-17
LIST Error Messages

<table>
<thead>
<tr>
<th>Message</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>BAD INPUT DIRECTORY</td>
<td>The directory on the specified input device is not valid OS/78 device directory.</td>
</tr>
<tr>
<td>BAD OUTPUT DEVICE</td>
<td>A non-file structured device was specified as the output device.</td>
</tr>
<tr>
<td>ERROR ON INPUT DEVICE, SKIPPING (file name)</td>
<td>The file specified is not transferred, but any previous or subsequent files are transferred and indicated in the new directory.</td>
</tr>
<tr>
<td>ERROR ON OUTPUT DEVICE, SKIPPING (file name)</td>
<td>The file specified is not transferred, but any previous or subsequent files are transferred and indicated in the new directory.</td>
</tr>
<tr>
<td>ERROR READING INPUT DIRECTORY</td>
<td>Self-explanatory.</td>
</tr>
<tr>
<td>ILLEGAL*</td>
<td>An * was entered as an embedded character in a file name, for example, TMP*.BN.</td>
</tr>
<tr>
<td>ILLEGAL?</td>
<td>A ? was entered in an output specification.</td>
</tr>
<tr>
<td>NO FILES OF THE FORM xxxx</td>
<td>No files of the form (xxxx) specified were found on the current input device group.</td>
</tr>
</tbody>
</table>
LOAD

3.21 LOAD COMMAND

The LOAD command loads into memory either an absolute PAL binary file (.BN extension) or a relocatable FORTRAN binary file (.RL extension).

When you use the LOAD command without arguments, it will use the arguments from the last COMPILE, LOAD, EXECUTE or PAL command. When you restart the system with the START or BOOT button or with the BOOT command, the arguments are lost.

3.21.1 PAL8 Absolute Binary Files

Absolute binary files are loaded into memory and are ready for execution.

Once a program file is loaded into memory, the following three things can be done.

1. Start the program, using the START command.
2. Save the program, using the SAVE command.
3. Examine or debug the program, using the ODT command.

Format:

LOAD indev:file1.ex...file9.ex/options

Along with the file specifications for the command line are the options described below.

Example:

LOAD FL3.BN, RUNSEQ.BN

NOTE

You can use the ODT command to examine and alter the contents of memory after your files are loaded.

3.21.1.1 Concatenated Programs (/S option) - The /S option loads into memory a group of programs that you have concatenated into a single file.

3.21.1.2 Loading Programs in Specified Areas (/ff and =ffnnnn options) - Absolute binary files can be loaded into a field other than the one they would normally load into by using /ff option, where ff is an octal number 0 through 37 that represents the field. By specifying the /n option, all specified input files are loaded beginning with the indicated field at a loading address equal to the address specified in the file.
Both the field and the starting address can be specified by using the 
=ffnnnn option

where:

ff is an octal number 0 through 37 that represents the field

nnnn is a 4-digit octal number 0000 through 7777 that
represents a starting address

NOTE

If ffnnnn is all zeros, the default option (field 0 starting address 0200) is used.

3.21.1.3 Programs not Using Fields 0 and 1 (/8 and /9 options) - If
locations 0-1777 in field 0 are not being used by the program, use the
/8 option in the command line. By specifying the /8 option, the
contents in these locations are not saved nor is the Command Decoder
(system program) reloaded. In effect, these unnecessary operations
are not performed.

If locations 0-1777 in field 1 are not being used by the program, use
the /9 option in the command line. When you specify the /9 option, the
contents in these locations are not saved and the User Service
Routine (system program) is not reloaded. In effect, these
unnecessary operations are not performed.

3.21.1.4 Editing or Patching a SAVE File (/I option) - Files that are
already assembled and saved on a device can be changed by using the /I
option in the command line. When you specify the /I option, SAVE
files are loaded and treated as absolute binary files. This allows
you to patch an existing save (.SV) file without reassembling the
entire program.

CAUTION

If you want to load more than one file, concatenate the files and use the /S
option. The command sequence:

LOAD PROG.SV/I
LOAD PROG1.SV/I

will not work the way you may think it
should because part of PROG.SV may be
destroyed when CCL.SV system program
runs to process the second LOAD command.

3.21.1.5 Execution After Loading (/G option) - To execute a program
immediately after loading it, use the /G option in the command line.
When you specify the /G option, execution takes place immediately
after loading is completed. Control is transferred to and remains
with the executing program until completion or until the program
transfers control back to the monitor.
3.21.1.6 Clearing Memory After Loading (/R option) - If you have already loaded a number of files and want to change some or all of the files loaded into memory, use the /R option in the command line. When you specify the /R option, the portion of memory where your files were loaded is reset (or cleared) to appear as though nothing had been loaded.

The LOAD command runs the CCL.SV and ABSDLR.SV programs.

3.21.2 FORTRAN Relocatable Binary Files

Since relocatable FORTRAN binary files are dependent programs, both files and system programs or subroutines are loaded and linked in memory.

Format:

    LOAD outdev:file.ex<inddev:file.ex/options-ex

The options summarized below and described in detail in Chapter 7.

After files are loaded, the system library FORLIB.RL is searched for programs or subroutines that are referenced by the program. When the subroutines are found, they are loaded and linked to the program.

NOTE

The COMPILE command with the /L and /G options lets you compile, link, and execute without using the LOAD and EXECUTE commands.

3.21.2.1 Specifying Additional Input Files (/C option) - If you have more than nine FORTRAN input files to load, use the /C (continue) option in the command line. When you specify the /C option, the remaining files are accepted as valid input and can be entered on the next command line. Terminate each continuation line (except the last) with a RETURN. Terminate the last line with an ESC.

3.21.2.2 Including System Symbols in the Symbol Map (/S option) - If a symbol map output file is specified in the command line with an .RL file and you want the system symbols also included in the map, use the /S option in the command line. The /S option includes all system symbols in the symbol map and marks them with a preceding number sign (#).
3.21.2.3 Execution After Loading (/G option) - To execute a program immediately after loading it, use the /G option in the command line. If you terminate the command line with a RETURN, execution takes place immediately after loading is completed. If you terminate the command line with an ESC, the FORTRAN run-time system prints an asterisk (*) to signify that you can enter I/O specifications (see Section 7.1.3). Terminate each assignment line except the last with a RETURN and the last with an ESC. Control is transferred to and remains with the executing program until completion or until the program transfers control back to the monitor.

When relocatable binary files are loaded and linked using the LOAD command, both the CCL.SV and LOAD.SV programs are executed.
3.22 MAP COMMAND

The MAP command produces a map of all the memory locations used by the specified absolute binary files. The LPT line printer is the default output device for this command. Terminal output may be requested through the use of the -T general-purpose dash option. Options for the MAP command are in Table 3-18; error messages are in Table 3-19.

Format:

MAP outdev:file.ex<inputdev:file1.ex...file9.ex/options-ex

MAP command options are described in Table 3-18.

NOTE

If a file name without an extension is specified as the output file, an .MP extension is automatically appended. Input extensions default as described for the LOAD command.

The output of the MAP operation is a series of lines, each of which is comprised of a string of digits. Each digit represents a single memory location, and can have the value 0, 1, 2 or 3. The value means the following:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>The location was not loaded into.</td>
</tr>
<tr>
<td>1</td>
<td>The location was loaded into once.</td>
</tr>
<tr>
<td>2</td>
<td>The location was loaded into twice.</td>
</tr>
<tr>
<td>3</td>
<td>The location was loaded into three or more times.</td>
</tr>
</tbody>
</table>

The appearance of a 2 or 3 may imply a programming error (for example, two separate routines are each trying to load values into the same location).

Each line of digits represents 100 (octal) memory locations. All lines are blocked in groups of two, whereby, each group represents a page. If the resulting map is sent to the terminal, it is bordered on top by a set of octal digits which range from 00-77. Each pair of octal digits corresponds to one memory location (see following example).

It is also bordered down the left hand column by a set of octal digits which are multiples of 64 (decimal). Each set of octal digits corresponds to the first memory location on that line. To determine the address (relative to zero) of a specific location in memory, find the line containing that location. Take the set of octal digits bordering that line and the pair of octal digits that the specific memory location falls under and add both values together.
The following is an example of output on terminal:

```
BITMAP V4 FIELD 0

00000 222222222111111100000000000000000000000000000000000000000
00100

00200 222222222222222222222222222222222222222222222222222222222
00300 222222222222222222222222222222222222222222222222222222222

00400 222222222222222222222222222222222222222222222222222222222
00500 222222222222222222222222222222222222222222222222222222222

00600 1111111111111111111111111111111111111111111111111111111111
00700 1111111111111111111111111111111111111111111111111111111111
01000
01000
01100
01200
01300

01400
01500

01600
01700
.
.
.
```

In this example, the location resulting from the addition is 322.

The following is an example of output directed to a line printer:

```
BITMAP V4 FIELD 0

00000 22222222 21111111 10000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00100

00200 22222222 22222222 22222222 22222222 22222222 22222222 22222222 22222222 22222222 22222222
00300 22222222 22222222 22111111 11111111 11111111 11111111 11111111 11111111 11111111 11111111

00400 22222222 22222222 22222222 22222222 22222222 22222222 22222222 22222222 22222222 22222222
00500 22222222 22222222 22222222 22222222 22222222 22222222 22222222 22222222 22222222 22222222

00600 11111111 11111111 11111111 11111111 11111111 11111111 11111111 11111111 11111111 11111111
00700 11111111 11111111 11111111 11111111 11111111 11111111 11111111 11111111 11111111 11111111
01000
01100

01200
01300

01400
01500

01600
01700
.
.
.
```
OS/78 COMMANDS

If any of the specified input files contain more than one program, only the first binary program in each file is used in the MAP command. The /S option in the command line allows MAP to use all binary programs in the specified input files. If more than one program is being mapped and you want all maps stored in one memory field, use the /n option, where n is an integer (0-37) that represents the field number. When you specify the /n option, all the maps of programs in the specified input file are constructed in the field you specified as n.

If more than nine input files are to be specified in the MAP command, specify only nine input files, then press the ESCape key. Pressing this key causes the Command Decoder to print an asterisk (*), which allows you to continue to specify more input file specifications. Each line, except the last, must be terminated by the RETURN key. If you specify a wrong input file, use the /R option at the end of that line. When you specify the /R option, memory is reset to look as though nothing had been read in. This is shown in the following example. Press the ESCape key to terminate the last line and to cause the command to be executed.

Example:

```
+MAP A,B,C,D,E,F,G,H,I  ESC
*J
*K
*L/R ESC
```

If only a terminal is available and no line printer, the style of output on the terminal can be changed to look as though it came from a line printer by using the /T option. Similarly, the style of output on the line printer can be changed to appear as though it came from the terminal.

This command executes the CCL.SV and BITMAP.SV programs.

<table>
<thead>
<tr>
<th>Option</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>/n</td>
<td>Confine the construction of all maps to the field you specify as n (n=0-37).</td>
</tr>
<tr>
<td>/R</td>
<td>Reset the map just constructed in memory to look as though nothing has been read in.</td>
</tr>
<tr>
<td>/S</td>
<td>Instead of mapping only the first binary file contained in the specified input files, map every absolute binary file in the specified input files.</td>
</tr>
<tr>
<td>/T</td>
<td>If output is sent to the terminal (default), change the format of the map to that of the line printer and if output is sent to the line printer, change the format of the map to that of the terminal.</td>
</tr>
</tbody>
</table>

Table 3-18
MAP Options
# OS/78 Commands

## Table 3-19
MAP Error Messages

<table>
<thead>
<tr>
<th>Message</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>BAD CHECKSUM, FILE #n</td>
<td>File number n of the input file list had a checksum error.</td>
</tr>
<tr>
<td>BAD INPUT, FILE #n</td>
<td>A physical end of file has been reached before a logical end of file, or extraneous characters have been found in binary file n.</td>
</tr>
<tr>
<td>ERROR ON OUTPUT DEVICE</td>
<td>Error occurred while writing on output device.</td>
</tr>
<tr>
<td>I/O ERROR FILE #n</td>
<td>An I/O error occurred in input file number n.</td>
</tr>
<tr>
<td>NO /I</td>
<td>Cannot produce a map of an image file.</td>
</tr>
<tr>
<td>NO INPUT</td>
<td>No binary file was found on the designated device.</td>
</tr>
</tbody>
</table>
3.23 MEMORY COMMAND

The MEMORY command displays the amount of memory that your DECstation has and sets a software limit on the amount of memory OS/78 will use.

Format:

MEMORY n

where:

n is an octal number representing the number of 4K memory fields available for use by OS/78 in the range: 0-7. If you do not specify a value, the command displays the amount of memory available in the hardware and any software limit that you specified previously up to 32K.

NOTE

Do not change the size of memory while a symbiont is running (see Appendix H).

The following table lists the values of n and their meanings.

<table>
<thead>
<tr>
<th>n</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>all available memory</td>
</tr>
<tr>
<td>1</td>
<td>8K</td>
</tr>
<tr>
<td>2</td>
<td>12K</td>
</tr>
<tr>
<td>3</td>
<td>16K</td>
</tr>
<tr>
<td>4</td>
<td>20K</td>
</tr>
<tr>
<td>5</td>
<td>24K</td>
</tr>
<tr>
<td>6</td>
<td>28K</td>
</tr>
<tr>
<td>7</td>
<td>32K</td>
</tr>
</tbody>
</table>

Example:

.MEMORY 3

16K MEMORY

To find the amount of memory actually being used by OS/78, type the command with no argument.

.MEM

12K OF 16K MEMORY

In this example, a 16K system has been restricted to only 12K of available memory. This was done by using a MEMORY 2 command.
OS/78 COMMANDS

If all available memory is being used, the total amount of memory is printed.

Example:

.MEM

16K MEMORY

This command executes the CCL.SV program.
3.24 ODT COMMAND

The ODT command is a debugging tool. It allows you to interpretively control execution and make alterations to a memory-resident program.

Format:

ODT

The program in memory is available for examination and modification in memory and controlled execution through the use of ODT instruction breakpoints. For more detailed information on ODT, see Chapter 9.

The monitor processes the ODT command.
3.25 PAL COMMAND

The PAL command assembles a PAL8 source file, producing an absolute binary file (.BN extension) and optional program listing and symbol cross reference listing.

Format:

PAL dev:binfile,dev:listfile,dev:creffile<dev:infil.../options

Example:

.PAL ASSEM1.PA

This command assembles a source file on SYS called ASSEM1.PA and produces a binary file on SYS called ASSEM1.BN.

If a file name is specified without an extension in the command line, .PA is assumed.

The PAL command can recall arguments from a previous COMPiLE, LOAD, or EXECUTE command.

Chapter 5 describes the PAL8 assembler and its command string options.
This command executes the PAL8.SV and CCL.SV programs.
The QUEUE command controls the demonstration symbiotic line printer spooler program (SPOOLR) described in Appendix H.
3.27 R COMMAND

The R command loads into memory the specified save file from the system device (SYS) and initiates execution.

NOTE

The difference between the R and RUN command is that the R command assumes the program is on device SYS, while the RUN command makes no such assumption. In addition, the RUN command writes on SYS during execution to save the program's Core Control Block for subsequent use in saving and/or starting the programs. The Core Control Block is not transferred to the system device when the R command is executed.

Format:

    R file.ex

Example:

    .R TEST

When a file name is specified without an extension, a .SV extension is assumed.

The R command is most commonly used on programs that are not going to be resaved. Use the RUN or GET commands with programs you want to update (using, for example, ODT) and save.

NOTE

You cannot use the R command to execute OS/78 system programs.

The monitor processes the R command.
RENAME

3.28 RENAME COMMAND

The RENAME command changes the name of the input file to the name specified as the output file. The same device must be specified in both the input and output specifications. Wildcards may be used with this command. Options for the RENAME command are in Table 3-20.

Format:

RENAME dev:file.ex<dev:file.ex/options

Execution of the RENAME command changes the file name, and a message followed by the old file name is displayed on the terminal.

Example:

.REN DSK:FILE.PA<DSK:RECORD.PA

Files Renamed:
RECORD.PA

This new file name replaces the old file name in the input directory. The creation date and the contents of the input file remain the same.

Example:

.RENAME *.*.FT
renames all files with .FT extensions to .BK extensions.

Table 3-20
RENAME Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>/C</td>
<td>Rename the input file only if it has the current date.</td>
</tr>
<tr>
<td>/O</td>
<td>Rename the input file only if it has a date other than the current date.</td>
</tr>
<tr>
<td>/T</td>
<td>After renaming the file, change the date of the new output file to today's date.</td>
</tr>
<tr>
<td>/V</td>
<td>Rename all files other than the one specified in the command line.</td>
</tr>
</tbody>
</table>

RENAME executes the CCL.SV and FOTP.SV programs.
3.29 REQUEST COMMAND

The REQUEST command initiates symbiont task operation by setting available memory to 12K words (MEM 2), then loading and starting the specified symbiont. The default extensions for a symbiont save image file are .SM and .SV.

Format:

REQUEST dev:symbiont-name.ex

This command can be used on DECstation 78 series systems only. Refer to Appendix H for more information.
OS/78 COMMANDS

RUN

3.30 RUN COMMAND

The RUN command loads the specified memory image (.SV) file in memory, transfers its Core Control Block onto the system device, and then initiates execution at the program's starting address. The RUN command is equivalent to a GET command followed by a START command.

Format:

RUN indev:file.ex

Example:

.RUN RXA1:PROG.

If a file name is specified without an extension, the extension .SV is assumed.

NOTE

You cannot use the RUN and R commands to execute the OS/78 system programs.

The monitor processes the RUN command.
3.31 SAVE COMMAND

The SAVE command saves the portion of memory being used as a memory-image (or save) file. The memory used is indicated by the contents of the Core Control Block saved on SYS in a monitor area or as explicitly specified by you. This new file is given the default extension .SV if you do not specify an extension.

Format:

SAVE input dev:filename.ex ffnnnn-ffmmmm,ffpppp,...;ffssss=cccc

where:

ffnnnn-ffmmmm ffnnnn is a 6-digit octal number representing the beginning address of a contiguous portion of memory to be saved. The beginning address consists of a field number (ff) in the range 0-37 followed by the memory location within the field (nnnn).

ffmmmm is a 6-digit octal number representing the ending address of the portion in memory. The ending address consists of the same field number as the beginning address (ff) in the range 0-37 followed by the memory location within the field (mmmm).

NOTE

The beginning and ending addresses must be the same field. Crossing field boundaries is not permitted.

ffpppp ffpppp is a 6-digit octal number representing the address of one location in memory. This represents saving the entire page in which the given location occurs. The address consists of a field number (ff) in the range 0-37 followed by the memory location within the field (pppp).

;ffssss ;ffssss is a 6-digit octal number preceded by a semicolon that represents the starting address of the program to be saved. The starting address consists of a field number (ff) in the range 0-37 followed by the memory location within the field (ssss).

=cccc =cccc is a 4-digit octal number preceded by an equal sign that represents the contents of the Job Status Word. The bits in the Job Status Word shown in Table 3-21 control memory allocation for the same file.
### OS/78 Commands

#### Table 3-21
Job Status Word

<table>
<thead>
<tr>
<th>Bit Condition</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit 0=1</td>
<td>File does not load into locations 0-1777 in field 0, (0000-1777).</td>
</tr>
<tr>
<td>Bit 1=1</td>
<td>File does not load into locations 0-1777 in field 1, (10000-11777).</td>
</tr>
<tr>
<td>Bit 2=1</td>
<td>Program must be reloaded before it can be restarted because it modifies itself during execution.</td>
</tr>
<tr>
<td>Bit 3=1</td>
<td>Program being run will not destroy the BATCH monitor.</td>
</tr>
<tr>
<td>Bit 4=1</td>
<td>A memory image file that was generated through the LINKER contains overlays.</td>
</tr>
<tr>
<td>Bit 5</td>
<td>Reserved for OS/78 system programs.</td>
</tr>
<tr>
<td>Bits 6-9</td>
<td>Unused, and reserved for future expansion.</td>
</tr>
<tr>
<td>Bit 10-1</td>
<td>Locations 0-1777 in field 0 need not be saved when calling the Command Decoder overlays.</td>
</tr>
<tr>
<td>Bit 11=1</td>
<td>Locations 0-1777 in field 1 need not be saved when calling the USR.</td>
</tr>
</tbody>
</table>

If you use SAVE with no arguments, the necessary information is automatically taken from the current Core Control Block (stored by the last GET, RUN, LOAD or EXECUTE command).

#### 3.31.1 Restrictions on Arguments of the SAVE Command

The arguments used with the SAVE command have several restrictions. They are as follows:

1. The output device must be explicitly specified in the command line. It does not default to DSK.

   `.SAVE DSK:FILE1`

2. The beginning and ending addresses of an area in memory (fnnn-fmmmm) must both be located in the same field. Crossing field boundaries is not permitted.

   Example:

   `.SAVE SYS: EXAMPL.BN 20055-20643`

3. Once an area on a page is specified, that entire page is saved. If another area on the same page is also specified, an error message is sent to the terminal.

   Example:

   `.SAVE RXA1:FL2 10077-10122, FL2 10156-10177
SAVE ERROR`
OS/78 COMMANDS

4. If a field number is not specified in the starting and ending addresses of an area in memory, field zero is used. Otherwise, the field number must be specified.

Example:

.SAVE SYS: FILE1 0-177,201-377

5. Saving the area between locations 7600-7777 in any field is permitted. However, locations 7600-7777 in fields 0 and 1 should not be saved because the resident portion of the monitor resides there. Therefore, if the area between 7600-7777 is going to be saved, limit it to fields 2 and 3 and even then only if necessary (due to BATCH).

6. If you specify an address on a page having an odd page number, that page will be saved with the previous page. This is automatically done by the system.

Example:

.SAVE DSK: PROG1 2434

In this example, the location is on page 12 which is even. Thus, the contents of page 12 are saved (2400-2577).

Example:

.SAVE DSK: PROG2 2634

In this example, the address 2634 is on page 13 which is odd. Therefore, in order to save page 13, page 12 is also saved. This means that the effective area being saved is 2400-2777.

The monitor processes the SAVE command.
SET

3.32 SET COMMAND

The SET command modifies the operating characteristics of OS/78 and its devices, according to options that are specified. SET command device attributes are in Table 3-21. OS/78 device handlers are listed in Table 3-22; error messages are in Table 3-23.

Format:

```
SET device [ NO ] parameter argument
```

where:

- **device** indicates the OS/78 device handler that is to be modified or replaced
- **NO** is an optional modifier that indicates that the feature specified is to be disabled. NO cannot be used with every attribute
- **parameter** is the feature to be modified (see Table 3-22 and the following paragraphs.)
- **argument** is a optional variable (numeric or alphabetic) that is supplied by you

<table>
<thead>
<tr>
<th>TTY,SLUX,VLUX</th>
<th>SYS</th>
<th>LPT</th>
<th>HANDLER</th>
<th>Any I/O Device</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARROW</td>
<td>INIT xxxxx</td>
<td>COL n LC WIDTH = n</td>
<td>See Section 3.32.1.13</td>
<td>READONLY</td>
</tr>
<tr>
<td>COL n</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ECHO</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ESC</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HEIGHT m</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PAGE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*PAUSE n</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>READONLY</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SCOPE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WIDTH = n</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

3.32.1 SET Command Forms

3.32.1.1 **ARROW** - This parameter causes the handler to map any CTRL character sent to the terminal into the ASCII codes for uparrow (') and the graphic symbol that is associated with the control code.

* Applies to TTY only.
OS/78 COMMANDS

Format:

SET ter: ARROW

where:

ter: is TTY, SLU2, SLU3, VLU2, or VLU3 (default condition)

For example, the code for CTRL/X (030) would be sent to the terminal as two codes 136 ("7) and 130 (X).

To remove this feature, use the NO modifier in the command line. This causes the true control codes to be sent to the terminal.

Example:

.SET TTY: NO ARROW

3.32.1.2 COL n - This parameter changes the number of columns used to print the directory listing (DIRECT command) from two (its default) to the decimal number you specify as n.

Format:

SET TTY COL n

where:

n is a decimal value in the range 0-7.

To remove the value you specified as n and return to its default value of 2, you must specify 2 as the argument n.

Example:

.SET TTY COL 2

3.32.1.3 ECHO - This parameter enables character echoing from the TTY SLUX and VLUx handlers (default system condition).

Format:

SET ter: ECHO

where:

ter: is TTY, SLU2, SLU3, VLU2 or VLU3

To inhibit character echoing at the terminal, use the NO modifier with the command.

Example:

.SET TTY: NO ECHO

3-63
OS/78 COMMANDS

3.32.1.4 ESC - This parameter causes the handler to map the code for ESC (033) into the code for the dollar sign ($) character (044) before sending it to the terminal.

Format:

SET TTY ESC

To remove this feature, use the NO modifier with the ESC attribute.

Example:

.SET TTY NO ESC

3.32.1.5 HEIGHT m - This parameter changes the number of lines (m) that are displayed on the terminal between pauses (see Section 3.29.8). (The default value of m is 24 (decimal).)

Format:

SET ter: HEIGHT m

where:

   ter: is TTY, SLU2, SLU3, VLU2, or VLU3

   m is a decimal number in the range 0-255.

3.32.1.6 INIT cmd - This parameter causes the system device to execute the command you specify as cmd when you press the START or BOOT button or use the BOOT command. This command name can contain a maximum of five characters excluding a carriage return.

Examples:

.SET SYS INIT HELP

causes the system to run the HELP.SV program and display the summary of OS/78 commands for which there is help available.

To execute the command contained in the file INIT.CM, type:

.SET SYS INIT

To inhibit this feature, use the NO modifier in the command line.

.SET SYS NO INIT

3.32.1.7 LC - This parameter causes the line printer (LPT) handler to accept lower-case characters on input.

Format:

SET LPT: LC

To remove this feature, use the NO modifier in the command line.

Example:

.SET LPT: NO LC
3.32.1.8 PAGE – This parameter enables the CTRL/S and CTRL/Q features described in Chapter 2 (default system condition).

Format:

```
SET ter: PAGE
```

where:

```
ter: is TTY, SLU2, SLU3, VLU2, or VLU3
```

To inhibit the CTRL/S and CTRL/Q features, use the NO modifier in the command line.

Example:

```
.SET TTY NO PAGE
```

3.32.1.9 PAUSE – This parameter changes the pause time between each screenful of data output to the console terminal from 3 seconds (default) to the decimal number you specify as n.

Format:

```
SET TTY: PAUSE n
```

where:

```
n is a decimal value in the range 0-85. If you specify zero or use the NO modifier, no pause takes place.
```

3.32.1.10 READONLY – This parameter causes the device specified to become a read-only device. Therefore, any output sent to this device will cause an error message informing you that the output device is a read-only device.

Format:

```
SET dev: READO
```

where:

```
dev: is any OS/78 I/O device
```

To remove the READONLY feature, use the NO modifier in the command line.

Example:

```
.SET TTY NO READO
```

NOTE

The READONLY attribute remains in effect until you restart the system with the START or BOOT button or with the BOOT command.
OS/78 COMMANDS

3.32.1.11 SCOPE - When an input line of characters is typed on a video terminal and echoed on the screen, this parameter causes the last character echoed on the screen to be erased each time you press the DELETE key.

Format:

    SET ter: SCOPE

where:

    ter: is TTY, VLU2 or VLU3

When using printing terminals remove this feature with the NO modifier. The system will echo a backslash (\) each time you press the DELETE key. (System defaults: TTY, VLU2 and VLU3 = SCOPE; SLU2 and SLU3 = NO SCOPE.)

Example:

    .SET TTY NO SCOPE

3.32.1.12 WIDTH = n - This parameter changes the column width of the terminal or line printer to the number of columns you specify as n.

Format:

    SET ter: WIDTH = n

where:

    ter: is TTY, SLU2, SLU3, VLU2, VLU3, or LPT

    n is a decimal number and a multiple of 8 in the range 001-255, but not 128. The default value is 80 for TTY, VLU2, and VLU3, and 120 for SLU2 and SLU3.

Example:

    .SET TTY WIDTH = 72

3.32.1.13 HANDLER - The SET HANDLER command allows you to replace a device handler that resides in the OS/78 Monitor (system head) with another device handler. A device handler is a system program that operates an input/output device. The Monitor calls a device handler via the USR routine (described in Appendix C), when you use a logical device name in an OS/78 command. Each OS/78 handler may control one or two logical devices depending on the device. The number of device handlers in OS/78 is fixed; you can replace them, but you cannot increase or decrease their number.

The OS/78 Monitor is supplied to you with a standard set of device handlers that allows you to use the devices provided with most DECstation configurations. In addition, a complete set of device handlers (see Table 3-23) is distributed to you on the system disk or diskettes that contain the OS/78 system programs. If your DECstation has a device for which there is no handler in the standard set, you can use the SET HANDLER command to replace one of the existing handlers with the one supplied for the device.
The command has two forms. The first form allows you to replace a device handler, and the second form prints a list of the device handlers residing in the system head. You cannot combine these forms in a single command.

Format 1 - Replace device handlers:

    SET HANDLER old<new

where:

old
    is the name of a handler currently residing in the system head that you want to replace.

new
    is the name of the handler that you want to insert into the system head. The handler must reside on the system device (SYS) as a save file with a .HN extension.

Example:

    .SET HANDLER LPT<LQP

This command causes the line printer handler LPT to be replaced by the letter quality printer handler LQP that resides on SYS as LQP.HN.

Example:

    .SET HANDLER RX78C<SLU3

This command causes the handler for the RXA2 and RXA3 devices used on DECstation 78 systems to be replaced by the SLU3 device handler.

Format 2 - List on the terminal the device handlers currently in the system head:

    SET HANDLER/L

---

Table 3-23
OS/78 Device Handlers

<table>
<thead>
<tr>
<th>Handler Name</th>
<th>Logical Device Names</th>
<th>Device</th>
</tr>
</thead>
<tbody>
<tr>
<td>BAT *</td>
<td>BAT</td>
<td>Batch stream handler</td>
</tr>
<tr>
<td>LPT</td>
<td>LPT</td>
<td>LA78 and LA180 Printers</td>
</tr>
<tr>
<td>LQP *</td>
<td>LQP</td>
<td>Letter Quality Printer</td>
</tr>
<tr>
<td>RLO</td>
<td>RLOA, RLOB</td>
<td>RL01 Cartridge Disk Drive 0</td>
</tr>
<tr>
<td>RLI</td>
<td>RLIA, RLIB</td>
<td>RL01 Cartridge Disk Drive 1</td>
</tr>
<tr>
<td>RLC</td>
<td>RLOC, RLCB</td>
<td>RL01 Cartridge Disk Drive 1/0</td>
</tr>
</tbody>
</table>

(continued on next page)

* Not part of the standard configuration supplied in the system head.
### Table 3-23 (Cont.)
#### OS/78 Device Handlers

<table>
<thead>
<tr>
<th>Handler Name</th>
<th>Logical Device Names</th>
<th>Device</th>
</tr>
</thead>
<tbody>
<tr>
<td>RX28C *</td>
<td>RXA2, RXA3</td>
<td>RX01 and RX02 Diskette Drives 2 and 3 on DECstation 88 systems</td>
</tr>
<tr>
<td>RX78C</td>
<td>RXA2, RXA3</td>
<td>RX01 and RX02 Diskette Drives 2 and 3 on DECstation 78 systems</td>
</tr>
<tr>
<td>RXNS</td>
<td>RXA0, RXA1</td>
<td>RX01 and RX02 Diskette Drives 0 and 1 on DECstation 78 and 88 systems</td>
</tr>
<tr>
<td>SLU2</td>
<td>SLU2</td>
<td>Serial line handler for hard copy ASCII terminals</td>
</tr>
<tr>
<td>SLU3 *</td>
<td>SLU3</td>
<td>Serial line handler for hard copy ASCII terminals</td>
</tr>
<tr>
<td>VLU2 *</td>
<td>VLU2</td>
<td>Serial line handler for ASCII video terminals</td>
</tr>
<tr>
<td>VLU3 *</td>
<td>VLU3</td>
<td>Serial line handler for ASCII video terminals</td>
</tr>
<tr>
<td>VXA0 *</td>
<td>VXA0</td>
<td>Extended Memory Device on DECstation 88/97 systems only</td>
</tr>
</tbody>
</table>

* Not part of the standard configuration supplied in the system head.

### Table 3-24
#### SET Error Messages

<table>
<thead>
<tr>
<th>Message</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>?CAN'T - DEVICE DOESN'T EXIST</td>
<td>A nonexistent device was referenced.</td>
</tr>
<tr>
<td>?CAN'T - DEVICE IS RESIDENT</td>
<td>No modifications are allowed to the system handler.</td>
</tr>
<tr>
<td>CAN'T - HANDLER ALREADY RESIDENT</td>
<td>Attempt to replace a handler with one that was already in the system head.</td>
</tr>
<tr>
<td>?CAN'T - OBSOLETE HANDLER</td>
<td>The handler has an old version number.</td>
</tr>
</tbody>
</table>

(continued on next page)
<table>
<thead>
<tr>
<th>Message</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAN'T - TOO MANY LOGICAL DEVICES</td>
<td>The system does not allow you to have more than 12 logical devices (excluding SYS, DSK, and TTY). The handler that you are attempting to insert will cause the number of logical devices to exceed 12.</td>
</tr>
<tr>
<td>?CAN'T - UNKNOWN VERSION OF THIS HANDLER</td>
<td>The version of the handler is not one recognized, possibly because it is a newer version.</td>
</tr>
<tr>
<td>DONE</td>
<td>The requested operation was successful.</td>
</tr>
<tr>
<td>?ILLEGAL WIDTH</td>
<td>A WIDTH argument of 0 or greater than 255 was specified. For the TTY, a width of 128 or one not a multiple of 8 was specified.</td>
</tr>
<tr>
<td>INCOMPATIBLE MONITOR - OS/78 V3.0 EXPECTED</td>
<td>The system head contains an old version of the OS/78 Monitor. You cannot change handlers in a system head that has an old version of the monitor.</td>
</tr>
<tr>
<td>?I/O ERROR ON SYS:</td>
<td>An I/O error occurred while trying to read or rewrite the handler.</td>
</tr>
<tr>
<td>I/O READ ERROR</td>
<td>A hardware error occurred while reading from the system device.</td>
</tr>
<tr>
<td>I/O WRITE ERROR</td>
<td>A hardware error occurred while writing to the system device.</td>
</tr>
<tr>
<td>?NUMBER TOO BIG</td>
<td>The number specified was out of range.</td>
</tr>
<tr>
<td>OLD HANDLER NOT FOUND IN MONITOR</td>
<td>You attempted to replace a handler that is not in the system head.</td>
</tr>
<tr>
<td>?SYNTAX ERROR</td>
<td>Incorrect format used in SET command or NO specified when not allowed.</td>
</tr>
<tr>
<td>?UNKNOWN ATTRIBUTE FOR DEVICE dev</td>
<td>An illegal attribute was specified for the given device.</td>
</tr>
<tr>
<td>xxxxxxxx.HN NOT FOUND ON SYS</td>
<td>The handler that you want to insert into the system head does not reside on the system device (SYS) or does not have a .HN extension.</td>
</tr>
</tbody>
</table>
3.33 SQUISH COMMAND

The SQUISH command eliminates any embedded empty files on the specified device.

Format:

    SQUISH outdev:<indev:

If you specify an input device only, the operation is performed on that device. If you specify both input and output devices, the files are copied to the output device from the input device. If the input device is a system device, the system head is preserved during the transfer. Before the operation begins, the system asks you to verify that you specified the correct device.

Example:

    SQ RXA1:
    ARE YOU SURE?Y

If yes, type a Y and the files are compressed.

    SQ SYS:
    ARE YOU SURE?N

If no, type any other character and the operation is ignored.

Example:

    SQ SYS:
    ARE YOU SURE?Y

In this example, all embedded empty files on the system device are eliminated.

CAUTION

A device error during a SQUISH operation will leave the entire contents of the device corrupted in a non-obvious way. Therefore, do not use this command unless you have a backup copy of both the system and other files available.

This command executes the CCL.SV and PIP.SV programs.
3.34 START COMMAND

The START command initiates execution of the program that is in memory either at the specified starting address in the command line or the starting address specified in the current Core Control Block.

Format:

START ffnnnn

where:

ffnnnn is a 6-digit octal number representing the starting address of the program. The starting address consists of a field number (ff) in the range 0-37 followed by the memory location (nnnn) within the field. If the starting address is not specified in the command line, the program is started at the address specified in the Core Control Block.

Example:

START 10555

In this example, this command line will execute a program starting in field 1 at location 555.

The monitor processes the START command.
3.35 SUBMIT COMMAND

The SUBMIT command provides batch processing with the option of spooling to output files. Options for the SUBMIT command are listed in Table 3-24.

Format:

```
SUBMIT spooldev:<batchfile.ext/options
```

Example:

```
.SU RLOB:BTHCHIN
```

If no input file extension is specified, `.BI` is assumed.

Along with the necessary arguments for the command line are SUBMIT options described in Table 3-24.

<table>
<thead>
<tr>
<th>Option</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>/E</td>
<td>Treat all errors as non-fatal errors so batch processing continues uninterrupted.</td>
</tr>
<tr>
<td>/H</td>
<td>Process the batch input file without echoing and without sending the $JOB and $END batch commands to both the terminal and batch log.</td>
</tr>
<tr>
<td>/Q</td>
<td>Send only the $JOB and $MSG batch commands, and all results from batch processing to the output device.</td>
</tr>
<tr>
<td>/T</td>
<td>Send all output from batch processing to the terminal only.</td>
</tr>
<tr>
<td>/U</td>
<td>Do not interrupt batch processing and ignore anything typed at the keyboard with the exception of CTRL/C.</td>
</tr>
</tbody>
</table>

The SUBMIT command is commonly used to run multiple programs and sequences of system commands that require little or no interaction with the user or operator.

3.35.1 Processing and Terminating a Batch Input File

The SUBMIT command starts processing your batch input file by printing a job header. The job header is printed on the LA78 line printer if it is available. (The LQP78 line printer is not supported in batch processing.) If an LA78 line printer is not available, the terminal is used as the output device to print a log of batch operations. Processing continues with the OS/78 commands and BATCH commands following the $JOB batch command and terminates with the $END batch command. Termination can also occur from an error in the OS/78
OS/78 COMMANDS

commands. However, batch processing is allowed to continue in spite of errors if the /E option is used in the command line. When you specify the /E option, all errors are treated as non-fatal errors and batch processing continues without interruption.

If the ESCape key is used to terminate an OS/78 command in the batch file, it will cause your batch run to terminate unless the /E option is specified. To terminate all the commands in a batch file, use the RETURN key. If CTRL/C is typed during batch processing, the batch run is terminated, and control returns to the monitor.

If you want to log the $JOB and $MSG commands, and batch results on the output device, and disregard the OS/78 commands and comment lines, use the /Q option. If you have an LA78 and want all batch output sent to the terminal instead, use the /T option in the command line. When you specify the /T option, batch output is sent to the terminal only.

If you want your batch input file to be processed without interruption, use the /U option in the command line. If you specify the /U option, any errors are non-fatal, and your batch input file is processed without interruption. Anything you type on the terminal except CTRL/C during batch processing is ignored.

If you want your batch input file to be processed without echoing its contents on the terminal and without sending the $JOB and $END batch monitor commands to the terminal and BATCH log, use the /H option. When you specify the /H option, echoing, $JOB, and $END batch monitor commands are suppressed. Statements that are not suppressed by the /H option are as follows:

1. Any OS/78 command messages and user program messages sent to BATCH
2. User program messages sent to the terminal
3. Batch run-time error messages
4. $MSG commands

NOTE

Do not use the /Q and /H options in the same command line.

3.35.2 Spooling

The SUBMIT command also provides optional spooling of output files. Spooling is a process whereby output intended for unit-record devices (such as a terminal or line printer) is redirected to a file on disk or diskette. This provides a convenient way for producing multiple copies of a listing. It also allows an installation that has a slow printer to perform computations rapidly and to review the results later when processing time is not at a premium.

To use the spooling option during batch processing, specify an output device, the system disk as the input device and the batch file in the command line. The output device specified as the spool device must be a file-structured device (disk or diskette). If no output device is specified, spooling does not take place.
Example:

.SU RXA1:<RXA0:BATCHX

NOTE
Your batch input file must be on the system device (SYS).

When spooling is specified, output ordinarily directed to a non-file structured device is intercepted and stored in temporary files on the spool device. The system automatically assigns a name to each file. The first file is called BTCHA1. The second is called BTCHA2 and so on up to and including BTCHA9. If you have more than nine input files, the names continue with BTCHB0 up to and including BTCHZ9.

See Chapter 8 for a detailed explanation of OS/78 Batch.
This command executes the CCL.SV and BATCH.SV programs.
3.36 TERMINATE COMMAND

The TERMINATE command causes the system to terminate OS/78 operation and load and execute a user-written program. The program must reside on SYS in save file (memory image) format and have the name TERMIN.SV. You must press the START or BOOT button to resume OS/78 operation after using this command.

Format:

TER

This command executes the CCL.SV program.
3.37 TYPE COMMAND

The TYPE command displays the contents of the specified input files on the terminal screen. Options for the TYPE command are in Table 3-25; error messages are in Table 3-26.

Format:

TYPE ter:<indev:file1.ex...file5.ex/options

where:

ter: is TTY, SLU2, SLU3, VLU2, or VLU3. (default is TTY)

indev:file1.ex is the device and file specification. You can enter up to five different specifications, each separated by a comma.

Along with the necessary arguments for the command line are the TYPE options described below and summarized in Table 3-26.

Table 3-26
TYPE Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>/C</td>
<td>Type those files with the current date.</td>
</tr>
<tr>
<td>/O</td>
<td>Type those files with a date other than the current date.</td>
</tr>
<tr>
<td>/Q</td>
<td>Each time a file name followed by a question mark is printed on the terminal, type a Y (yes) to type the file or any other character (no) if the file is not to be typed.</td>
</tr>
<tr>
<td>/V</td>
<td>Type all files that were not specified in the command line.</td>
</tr>
</tbody>
</table>

The TYPE command displays the contents of each input file on the terminal in the same order the file names are specified in the command line.

Example:

.TY SYS:MASFL

Although the format specifies that only five file names can be used in the command line, more than five files can be typed on the terminal by using wildcards.
3.37.1 Using TYPE Options

Use the /C or /O option if the file's date is to determine what is to be displayed. When you specify the /C option, all the specified input files with the current date are typed. If the /O option is used, all specified input files with dates other than the current date are typed. The /V option displays all files other than the ones that were specified. The /Q option displays each file name followed by a question mark. If you want to display that file, type a Y. If not, type any other character and that file is ignored. This continues until all specified input files have been processed.

NOTE

Use the LIST command to print files on the LA78 line printer (LPT).

This command executes the CCL.SV and FOTP.SV programs.

Table 3-27
TYPE Error Messages

<table>
<thead>
<tr>
<th>Message</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>BAD INPUT DIRECTORY</td>
<td>The directory on the specified input device is not valid OS/78 device directory.</td>
</tr>
<tr>
<td>BAD OUTPUT DEVICE</td>
<td>A non-file structured device is inappropriately specified as the output device.</td>
</tr>
<tr>
<td>ERROR ON INPUT DEVICE, SKIPPING</td>
<td>The file specified is not transferred, but any previous or subsequent files are transferred and indicated in the new directory.</td>
</tr>
<tr>
<td>(file name)</td>
<td></td>
</tr>
<tr>
<td>ERROR ON OUTPUT DEVICE, SKIPPING</td>
<td>The file specified is not transferred, but any previous or subsequent files are transferred and indicated in the new directory.</td>
</tr>
<tr>
<td>(file name)</td>
<td></td>
</tr>
<tr>
<td>ERROR READING INPUT DIRECTORY</td>
<td>Self-explanatory.</td>
</tr>
<tr>
<td>ILLEGAL*</td>
<td>The wildcard character * was entered as an embedded character in a file name, e.g., TMP*.BN.</td>
</tr>
<tr>
<td>ILLEGAL?</td>
<td>The wildcard character ? was entered in an output specification.</td>
</tr>
<tr>
<td>NO FILES OF THE FORM xxxx</td>
<td>No files of the form (xxxx) specified were found on the current input device group.</td>
</tr>
</tbody>
</table>
3.38 UA, UB, UC Commands

The UA, UB, and UC commands are three separate commands that save their specified arguments in a temporary area. Only a single OS/78 command can be used as an argument. These three commands can recall their arguments when the command that saved the OS/78 command is typed without any arguments. Once recalled, the OS/78 command is executed. When you restart the system with the START or BOOT button or with the BOOT command, the arguments are lost.

Format:

UA
UB
UC

(any OS/78 command line)

Type the command followed by the command line you want to save, then press RETURN. This saves the specified command line in a temporary area. There are three areas, one for each command. To recall and execute the stored command, type the command that saved it, but do not specify any arguments.

Example:

.UA COPY RXA1:<DSK:RECALL

In this example, the system stores the COPY command and its arguments in a temporary file.

Example:

.UA

In this example the OS/78 command previously specified with the UA command and stored in a temporary file is recalled and executed.

These commands are most commonly used with OS/78 commands that are repeated throughout a batch file. The use of these instructions decreases the amount of input required.

The monitor processes the UA, UB, and UC commands.
3.39 ZERO COMMAND

CAUTION

Use this command with extreme care, since it can erase the contents of a system device, including the system head.

The ZERO command clears the directory on a device and deletes all files stored on that device. Use it only for devices that contain your programs and data files. If used on a system device, it will destroy the system head and convert it to a files-only device.

Format:

ZERO dev:

The command executes the CCL.SV and PIP.SV programs.

Example:

.ZERO RXA1:

When you use the DIRECT command on the device just zeroed, a blank directory will appear showing all blocks are available.

If the specified device is SYS, the system asks you if this is the correct device. If yes, type a Y and the directory is zeroed. If no, type any other character, and the operation is ignored. Remember that if you type a Y, the system on the specified device will be destroyed.
CHAPTER 4

THE SYMBOLIC EDITOR

4.1 INTRODUCTION

The Editor allows you to create and modify ASCII text files. These files may contain assembly language source programs, FORTRAN and BASIC source programs, or any other information that has the format of character strings.

The Editor is a helpful tool; however, it must be told precisely what to do. You direct its operation by typing commands in the form of a single letter or a letter with arguments and, in most cases, pressing the RETURN key directly after the command line.

This chapter describes the procedures you follow to create a file and the commands you use to modify it.

4.2 CALLING THE EDITOR

The CREATE and EDIT commands call and run the Editor.

The Editor operates in two modes: command mode and text mode. In command mode, the Editor displays a number sign as a prompt (#) and interprets the characters you type on the terminal as instructions to do some job on the text in its buffer. In text mode, the Editor interprets the characters you type as text. The two modes -- and the controls you use to request them -- are described in Section 4.3.

4.2.1 Creating a New File - The CREATE Command

The CREATE command summons the Editor to let you open and write a new file. The format is

CREATE outdev:file

CREATE accepts no input specifications and only one file name and device for output. You provide the input by typing in text at the terminal.

After you press the RETURN key to execute the command, the system Editor displays a number sign (#) on the screen to indicate that it is ready to receive your first instruction.
Thus,

```
.CREATE RXA1:RUN1.PA
#
```

opens a file named RUN1.PA on output device RXA1.

Before you can type in text from the terminal, you must put the Editor into text mode. For details on text mode, see Section 4.3.1.

### 4.2.2 Editing an Existing File - The EDIT Command

The EDIT command summons the OS/8 Editor to let you retrieve and work on a source program previously stored as a file. The format is:

```
EDIT outdev:file=indev:file1,...indev:file9
```

The Editor signals with a number sign (#) as soon as it is ready to accept your first instruction.

To work on a source program that you have created and stored as a file (or sequence of files), enter the file or files as input in the EDIT command line. EDIT will accept up to nine input files in a line.

The Editor allows only one output file in a command line. You must specify an output file to receive the modified version of your source program.

For example, the following command opens input file TABLE.PT on RXA0 and a file called FILE1.PT on RXA1 for output (The Editor signals when ready.):

```
.EDIT RXA1:FILE1.PT<RXA0:TABLE.PT
#
```

To cause the Editor to read in the first page of the input file, type R in response to the number sign. (For details on Editor commands, see Section 4.3.2.)

### 4.3 Modes of Operation

The OS/8 Editor operates in two modes: the command mode and the text mode.

In the command mode, the Editor prints a # on the terminal to indicate that it is waiting for you to type a command on the keyboard.

In text mode, the Editor accepts anything you type at the keyboard as part of the file you are creating or modifying.

The key commands in Table 4-1 enable you to transfer between modes or return control to the monitor.
Table 4-1
Editor Key Control Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Mode in Which Used</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTRL/C</td>
<td>Text and Command Modes</td>
<td>Returns control to the monitor. All text that has been edited is lost. CTRL/C should be used with utmost caution, since no output file will be stored.</td>
</tr>
<tr>
<td>CTRL/O</td>
<td>Command Mode</td>
<td>Stops the listing of text. Returns control to Command Mode.</td>
</tr>
<tr>
<td>CTRL/L</td>
<td>Text Mode</td>
<td>Returns the Editor to Command Mode.</td>
</tr>
</tbody>
</table>

4.3.1 Text Mode

To put the Editor in text mode so that you can enter a new file -- or modify or add to one that you have already created -- type the Insert or Append command.

Format:

I RETURN

or

A RETURN

These commands cause the Editor to place the text that you enter at the terminal into its text buffer. If you use the Insert command, the Editor stores the text before the first line of any existing material in the buffer. The Append command instructs the Editor to place the text you enter after the last line of existing text in the buffer.

The Editor accepts text in both upper and lower case.

To enter a line of text that you have typed on the terminal, press the RETURN key.

For example:

$1
HEAD OF THE BUFFER

or

$A
BOTTOM OF THE BUFFER

Before you type RETURN, read the line over for errors. Make corrections with the DELETE key or the CTRL/U key command. DELETE erases the last character you typed. CTRL/U deletes the entire line. (CTRL/U is equivalent to typing DELETE back to the beginning of the line.)
THE SYMBOLIC EDITOR

To correct a line that you have sent to the buffer with RETURN, you must put the Editor in command mode with CTRL/L and use the appropriate editing commands (see Section 4.3.2.4).

The buffer holds approximately 5600 characters (decimal). When 256 locations remain, the Editor rings the warning bell on the terminal. From this point until the buffer is full, typing RETURN causes the Editor to enter a line of text, then switch to command mode and ring the terminal bell. You may continue to enter text by this method one line at a time until the Editor detects the absolute end of its buffer.

To continue, you must first empty the buffer. The Page command enables you to send the contents of the buffer -- or any part of it -- to an output device. To use the Page command, return to command mode with CTRL/L.

Format:

P

or

nP

or

m,nP

where:

n is a line you want to send to an output device.

n,m is a sequence of lines (n through m) that you wish to send to an output device

The P command automatically appends a form feed to the output, thus producing a page of text. This allows you to paginate the contents of your file.

Before you start typing in the next page, make sure that no text remains in the buffer. To do this, use the Kill command (see Section 4.3.2.3), which clears the buffer. Then type the Append command (to put the Editor back in text mode) and continue entering your source program.

To return to command mode at any point, type CTRL/L.

To end the session -- that is, to place all remaining text in the output file, close the file, and return control to the monitor -- use the Exit command.

Format:

E <RET>
4.3.2 Command Mode

In command mode, the Editor performs the operations you specify on the text in the buffer.

For example, to enter text into the buffer from your input device, use the Read command.

Format:

\[ \text{R (RET)} \]

The Read command instructs the Editor to read a page of text from an input device into the buffer -- that is, to read text until it encounters a form feed character. If the buffer contains text already, the Editor adds the new page to it.

The Editor provides five types of command: Input, Listing, Output, Editing, and Search.

Each command consists of a single letter, preceded optionally by one or two numeric arguments. The letter indicates the operation; the arguments in most cases tell the Editor which lines to act upon.

Enter the commands after the number sign prompt in upper case only.

General format:

\[ \text{X (RET)} \]

or

\[ \text{nX (RET)} \]

or

\[ \text{m,nX (RET)} \]

where:

\[ \text{X} \]

is a command

\[ \text{m,n} \]

are line numbers (m must be less than n.)

Except for noted exceptions, you terminate the command with the RETURN key.

4.3.2.1 Input Commands - Input commands (Table 4-2) instruct the Editor to accept text from the terminal (text that you type in) or from an input device (text that you have stored as a file). To execute the commands, type RETURN key.

Special characters, including lower-case letters may be input to the file. The ESCape character is echoed as a dollar sign ($) for readability.

In these commands, the Editor ignores ASCII codes 340 through 376. These codes include the codes for the lower-case alphabet (ASCII 341-372). The Editor returns to the command mode only after encountering a form feed or when the text buffer becomes full.
## The Symbolic Editor

### Table 4-2
Editor Input Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Append the text being typed at the keyboard until a form feed (ASCII 214 or CTRL/L) is encountered. The form feed returns control to command mode. Text input following the A command is appended to whatever is currently in the text buffer.</td>
</tr>
<tr>
<td>I</td>
<td>Insert whatever text is typed before line 1 of the text buffer. The form feed (CTRL/L) terminates the insertion process and returns control to the command mode.</td>
</tr>
<tr>
<td>nI</td>
<td>Insert whatever text is typed (until a form feed is typed) before line n of the text buffer.</td>
</tr>
<tr>
<td>R</td>
<td>Read one page from the input device specified to the EDIT or CREATE commands, and append the new text to the current contents of the buffer. If no input file was indicated or if no input remains, a question mark (?) is printed and the Editor returns to the command mode.</td>
</tr>
</tbody>
</table>

### 4.3.2.2 Listing Commands
Listing commands (Table 4-3) display on the terminal all or part of the contents of the text buffer. Type RETURN key to execute.

### Table 4-3
Editor Listing Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td>List entire contents of the text buffer on the terminal.</td>
</tr>
<tr>
<td>nL</td>
<td>List line n of the text buffer on the terminal.</td>
</tr>
<tr>
<td>m,nL</td>
<td>List lines m through n of the text buffer on the terminal.</td>
</tr>
<tr>
<td>G</td>
<td>Get and list the next line that has a label associated with it. A label in this context is any line of text that does not begin with one of the following: space (ASCII 240) / (ASCII 257) TAB (ASCII 211) RETURN (ASCII 215) At the termination of a G command, control returns to the command mode with the current line counter equal to the line just listed.</td>
</tr>
</tbody>
</table>

(continued on next page)
THE SYMBOLIC EDITOR

Table 4-3 (Cont.)
Editor Listing Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>nG</td>
<td>Get and list the first line that begins with a label, starting the search at line n.</td>
</tr>
<tr>
<td>B</td>
<td>Print the number of available memory locations in the text buffer. The Editor returns the number of locations on the next line. To estimate the number of characters that can be accommodated in this area, multiply the number of free locations by 1.7.</td>
</tr>
</tbody>
</table>

The Editor remains in command mode after a list command and updates the value of the current line counter to be equal to the number of the last line printed.

4.3.2.3 Output Commands - Output commands (Table 4-4) send text from the buffer to a device you specify for output. Type RETURN key to execute.

Table 4-4
Editor Output Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>Output the current buffer and transfer all remaining pages of input to the output file; close the output file and enter it in the directory. When this buffer is full, the text is output to the indicated output file. The E command automatically outputs a form feed after the last line of output, and returns control to the monitor.</td>
</tr>
</tbody>
</table>

NOTE

If you do not use the E command to close a file after editing, any changes, additions, or corrections will not appear in the output file. Thus, the E command should usually be the last command that you enter in an editing session (also see Q command).

<table>
<thead>
<tr>
<th>Command</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>Write the entire text buffer to the output file.</td>
</tr>
<tr>
<td>nP</td>
<td>Write line n of the text buffer to the output file.</td>
</tr>
</tbody>
</table>

(continued on next page)
<table>
<thead>
<tr>
<th>Command</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>m,nP</td>
<td>Write lines m through n, inclusive, to the output file.</td>
</tr>
</tbody>
</table>

**NOTE**

The P command automatically appends a form feed to its output, thus producing a page of text. This command allows you to paginate your listing. However, if the K command is not used after a P command, the text remains in the buffer and is again output with the new text read in before the next P command.

| K       | Kill the buffer. All text is deleted from the text buffer. |

**NOTE**

The Editor ignores the commands nK or m,nK, with the result that you cannot destroy the buffer by mistyping a List command (m, nL).

| Q       | Immediate end-of-file. The Q command causes the text buffer to be output. The file is then closed (entered into the directory with the current date as its creation date), and control returns to the monitor. |

| N       | Write the current buffer to the indicated output file and read the next logical page. The N command is equivalent to a P, K, R command sequence. |

| nN      | Write the current buffer to the output file, kill the buffer, and read the next logical page. This is done n times until the nth logical page is in the text buffer. Control then returns to command mode. (The N command cannot be used with an empty text buffer, since there is no text to be written. If the buffer is empty when the N command is attempted, a question mark (?) is printed.) For example, to read in the fourth page of a file, give the commands

```
#R (to read the first page)
```

and

```
#3N (to read three more pages)
```

(continued on next page)
TABLE 4-4 (Cont.)
Editor Output Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>V</td>
<td>The V command causes the entire text buffer to be listed on the line printer. The V command only works with the LA78 line printer. It does not work with the LQP78 line printer.</td>
</tr>
<tr>
<td>nV</td>
<td>List line n of the buffer on the line printer.</td>
</tr>
<tr>
<td>m,nV</td>
<td>List lines m through n, inclusive, on the line printer.</td>
</tr>
</tbody>
</table>

During an editing session, the device you specify for output may become full before you have written out the entire file.

If this happens and you attempt to write more text on the device, an error occurs. The Editor immediately closes the file and prints the following message and prompt.

**FULL**

* To continue, you must specify a new output device and file name. Since the Editor retains the contents of its buffer, no text is lost.

When you have finished the editing job, combine your output files.

4.3.2.4 Editing Commands - Editing commands (Table 4-5) permit deletion or alteration of text in the buffer. Press the RETURN key to execute.

**Table 4-5**
Editing Commands: Deletion and Alteration

<table>
<thead>
<tr>
<th>Command</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>nC</td>
<td>Change the text of line n to the line(s) typed after the command is entered (typing a form feed terminates the text input). The C command is equivalent to a D command followed by an I command.</td>
</tr>
<tr>
<td>m,nC</td>
<td>Delete lines m through n, and replace with the text line(s) typed after the command is entered. (Typing CTRL/L indicates the end of the changed lines.) The C command utilizes the text collector in altering text.</td>
</tr>
<tr>
<td>nD</td>
<td>Delete line n from the buffer.</td>
</tr>
<tr>
<td>m,nD</td>
<td>Delete lines m through n from the buffer.</td>
</tr>
</tbody>
</table>

(continued on next page)
THE SYMBOLIC EDITOR

Table 4-5 (Cont.)
Editing Commands: Deletion and Alteration

<table>
<thead>
<tr>
<th>Command</th>
<th>Meaning</th>
</tr>
</thead>
</table>
| nY      | Yank (read) in n pages from the input file into the text buffer, without writing any output. For example, 

#SY

reads through four logical pages of input, deleting them without producing output. The fifth page is read into the text buffer, and control automatically returns to command mode. |

**NOTE**

Use this command with caution; it irrevocably deletes the contents of the text buffer. |

m,n$plM | Move lines m through n directly before line p in the text buffer. The $ character means that you type the dollar sign key, not ESCape, ALTMODE, or other possibilities. The old occurrence of the moved text is then removed. This command can move one line, but it needs three arguments. You can provide three arguments by specifying the same line number twice. For example, 

#6,6$21M |

moves line 6 in front of line 21. |

4.3.2.5 Search Commands - Search commands (Table 4-6) cause the Editor to search a text for occurrences of characters and strings that you specify. The Editor sets the current line pointer at the line containing the characters you want to find.

Search commands are discussed in detail in Section 4.4.

Table 4-6
Editor Search Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>Perform a character search (Section 4.4.1).</td>
</tr>
<tr>
<td>J</td>
<td>Perform an interbuffer search for character strings (Section 4.4.2.2).</td>
</tr>
<tr>
<td>F</td>
<td>Look for next occurrence of the string currently being sought.</td>
</tr>
<tr>
<td>ESC($)</td>
<td>Perform an intrabuffer character string search.</td>
</tr>
</tbody>
</table>
THE SYMBOLIC EDITOR

4.3.2.6 Special Command Mode Characters - Special characters recognized by the Editor in the command mode are listed in Table 4-7.

Table 4-7
Editor Special Characters: Command Mode

<table>
<thead>
<tr>
<th>Character</th>
<th>Function</th>
</tr>
</thead>
</table>
| Period (.)    | The Editor assigns an implicit decimal number to the line on which it is currently operating. At any given time the period, which represents this decimal number, may be used as an argument to a command. In the following example, the L command is used since it allows text to be listed. Typing
|               | #$L                                                                  |
|               | means list the current line. Typing                                  |
|               | #$-1,.+1L                                                            |
|               | means list the line preceding the current line, the current line, and the line following it, and then update the current line counter to the decimal number of the last line printed. The Editor updates the current line counter, represented by the period, as follows: |
|               | • After an R (Read page) or A (Append) command, the period is equal to the number of the last line in the buffer. |
|               | • After an I (Insert) or C (Change) command, the period is equal to the number of the last line entered. |
|               | • After an L (List) or S (Search) command, the period is equal to the number of the last line listed. |
|               | • After a D (Delete) command, the period is equal to the number of the line immediately after the deletion. |
|               | • After a K (Kill) command, the period is equal to 0.                |
|               | • After a G (Get and list) command, the period is equal to the number of the line displayed by the G. |
|               | • After an M (Move) command, the period is not updated and remains whatever it was before the command. |

(continued on next page)
<table>
<thead>
<tr>
<th>Character</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slash (/)</td>
<td>The symbol slash (/) has a value equal to the decimal number of the last line in the buffer. It may also be used as an argument to a command. For example,</td>
</tr>
<tr>
<td></td>
<td>#10, /L</td>
</tr>
<tr>
<td></td>
<td>means list from line 10 to the end of the buffer.</td>
</tr>
<tr>
<td></td>
<td>When the Editor is in command mode, pressing the LINE FEED key has the same effect as</td>
</tr>
<tr>
<td></td>
<td>.+1L</td>
</tr>
<tr>
<td></td>
<td>which causes the Editor to display the line following the current one and to increment the value of the current line counter (period symbol) by one. LINE FEED does not perform this function while in the text mode.</td>
</tr>
<tr>
<td>Right-Angle Bracket (&gt;))</td>
<td>Typing the right-angle bracket (&gt;) while in command mode is equivalent to typing</td>
</tr>
<tr>
<td></td>
<td>.+1L</td>
</tr>
<tr>
<td></td>
<td>and causes the Editor to echo &gt; and then display the line following the current line. The value of the current line counter is increased by one so that it refers to the last line displayed.</td>
</tr>
<tr>
<td>Left-Angle Bracket (&lt;))</td>
<td>In command mode, typing the left-angle bracket (&lt;) is equivalent to typing</td>
</tr>
<tr>
<td></td>
<td>.-1L</td>
</tr>
<tr>
<td></td>
<td>and causes the Editor to echo &lt; and then print the line preceding the current line. The value of the current line counter is decreased by one so that it refers to the last line printed.</td>
</tr>
<tr>
<td>Equal Sign (=)</td>
<td>In the command mode, using the equal sign in conjunction with either the line indicator period (.) or slash (/) causes the Editor to display the decimal value of the argument preceding it. You can find by this method the number of the current line (.=nnnn) or the total number of lines in the buffer (/=nnnn).</td>
</tr>
</tbody>
</table>

(continued on next page)
THE SYMBOLIC EDITOR

Table 4-7 (Cont.)
Editor Special Characters: Command Mode

<table>
<thead>
<tr>
<th>Character</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Colon (:)</td>
<td>The colon performs exactly the same function as the equal sign (=).</td>
</tr>
<tr>
<td>ESCape Key</td>
<td>When the Editor is in command mode, pressing the ESCape key signals an intrabuffer character search. It echoes as a dollar sign ($) on the terminal screen. When the Editor is in text mode, the Escape key echoes as a dollar sign, but it is stored in the file as an ESCape character (033).</td>
</tr>
</tbody>
</table>

4.4 SEARCHING A TEXT

The following search commands enable you to make additions and corrections in your text. The Editor searches for occurrences of the single character or character string that you specify.

4.4.1 Single-Character Search -- the S Command

The format of a single-character search is:

```
S
x
```

where:

```
x
```

is the alphanumeric character you want to search for.

To specify a line or a sequence of lines that you want to search, use the following format:

```
nS
c
```

or

```
n,mS
c
```

where:

```
  n is a line number
  n,m are line numbers indicating a segment of text
  c is any single character you wish to search for
```

For example, the following command causes the Editor to search lines 20 to 40 for an occurrence of the character B:

```
#20, 40S
B
```
THE SYMBOLIC EDITOR

The Editor displays the character it is searching for and everything preceding it in the line. At this point you can perform the following operations.

- Delete the entire portion of the line not yet displayed and terminate the line and the search by pressing the RETURN key.
- Delete characters from right to left by typing the DELETE key.
- Insert characters after the last one printed simply by typing them.
- Insert a carriage return/line feed, thus dividing the line into two, by pressing the LINE FEED key followed by CTRL/L.
- Continue searching the line to the next occurrence of the search character by typing CTRL/L.
- Change the search character in the line and continue searching by typing CTRL/G(BELL) followed by the new search character. This allows all editing to be done in one pass.
- Type CTRL/G(BELL) twice to terminate the command.

The usual form of the character search command is $.S, followed by the RETURN key and the character to be located. Use this form of the command to modify the current line.

4.4.2 The Character String Search

The Editor can search the buffer for any unique combination of characters. In a character string search, the Editor sets the current line pointer at the line containing the first occurrence of the string.

Two types of character string search are available: intrabuffer and interbuffer.

4.4.2.1 Intrabuffer String Search - In an intrabuffer search, the Editor scans the text in the buffer for the string you specify. If it fails to find an occurrence of the string, it prints a question mark and returns to command mode.

To initiate an intrabuffer search, type the ESCape key in response to the Editor's prompt and enter the string. (ESCAPE echoes as a dollar sign.) The string must occur in one line.

If you wish to begin the search at line 1 of the buffer, terminate the string with a single quotation mark ('). If you wish to begin the search at the current line + 1, use a double quotation mark (") to terminate the string.
THE SYMBOLIC EDITOR

The format of an intrabuffer search command is:

$string'

or

$string"

where:

$ is the character the Editor echoes when you type ESCape

string is a group of up to 20 ASCII characters

' (single quote) causes the Editor to begin searching at line 1 of the buffer

" (double quote) causes the Editor to begin searching at current line +1

NOTE

Do not include single or double quotation marks in a string because the Editor recognizes them as instructions.

The Editor places the number of the first line containing the search string in the current line indicator and displays the prompt sign ($). To display the number on the terminal, type a period (.) followed by an equal sign. The format is:

.=

You can use a line number you obtain this way as an argument in any Editor command.

For example, the following command causes the Editor to search for the first occurrence of the string CDF10, beginning at line 1 of the buffer:

$CDF10'

The response to this command is revealed as line 35:

=.35

Command lines can include more than one instruction. For example, assume that the buffer contains the following text:

ABC DEF GJO
1A2B3C4D5E6
.STRINGABCD
:

To list the line that contains ABC, type

$ABC'L

4-15
THE SYMBOLIC EDITOR

The search begins with line 1 and continues until the Editor finds the string. The Editor sets the current line counter equal to the line in which the string ABC occurred. The L (List) command causes the line to be printed as follows:

ABC DEF GJO

The Editor returns to command mode, awaiting further commands. If you want to find the next reference to ABC, type:

'*L

In this case, the quotation marks ('') cause the last string the Editor searched for to be used again, with the search beginning at the current line +1. It is not necessary to enter the search string again. The command may be used several times in succession. For example, if you want to find the fourth occurrence of a string containing the characters FEWMET, type:

'FEWMET' '*'L

This command will list the line which contains the fourth occurrence of that string. The L command (or any other command code) can follow either ' ' or '". The L command causes the line to be listed if the Editor finds the string.

To clear the text string buffer, type:

'*

The Editor responds with a question mark and clears the text string buffer.

The properties of the commands ' and '" allow for easy and useful editing, as the following example illustrates. To change the CIF 20 to CIF 10, enter the following commands:

'NEW FIELD

The above set of instructions first causes the Editor to start at line 1 and search for the line beginning with DUM, . Then it searches for CIF 20, starting from the line after the line containing DUM, . The line number of the line containing the string CIF 20 becomes the current line number. The C command applies the instructions of the command line to what is typed in the next line -- that is, the string CIF 10.

Since this search feature produces a line number as a result, any operations which require a line number will accept a string instead. For example:

'4L

lists the fourth line after the first occurrence of the text STRING in the text buffer.

'LABEL1,'*'LABEL2,'L

lists all lines between the two labels, inclusive.

'PFLUG'S

performs a character search on the line which contains PFLUG. (Type the search character after typing the RETURN key that enters the line.)
THE SYMBOLIC EDITOR

In commands that include both strings and explicit numbers, strings should appear first. For example, the following commands:

\$1+$BAD!'+L

will not list the next line after the string BAD! occurs. The correct syntax is:

\$BAD!'+L

4.4.2.2 Interbuffer String Search -- J Command -- In an interbuffer search, the Editor scans the contents of the text buffer for the character string you specify. If it fails to find an occurrence of the string, it sends the buffer to an output file, clears the buffer, and reads in the next page of text from the input file. The Editor then resumes the search at line 1 of the new buffer. When the input file is exhausted, the Editor prints the number sign prompt (#) and awaits your next instruction.

If the search is successful, the Editor sets the current line indicator equal to the number of the line containing the first occurrence of the string.

The format for an interbuffer search is:

\$string'

where:

$ is a prompt character printed by the Editor
string is a group of up to 20 ASCII characters
' (single quote) causes the Editor to begin searching at line 1 of the buffer.

To display the number of the line containing the string, type a period (.) after the Editor's number sign prompt (#), followed by an equal sign. The format is:

.=

For example, the following command instructs the Editor to make an interbuffer search for the string WRITE, beginning at line 1 of the current buffer. The .= construction reveals that line 4 of the current buffer contains the string.

\$J
$WRITE'
#.=0004

To find further occurrences of the string WRITE, type the F command. The F command searches the buffer for the last character string entered, starting from the current line count + 1. The displayed line following the F command line contains a number prompt sign (#), the format you type to obtain a line number (.=), and the line number. The result is:

\$F
#.=0008
THE SYMBOLIC EDITOR

This example causes a search for the string WRITE, starting at the current line + 1. If you have specified no output file, the J or F command reads the next input buffer without attempting to produce any output.

NOTE

Use the J command for interbuffer searches only. After the J or F command has processed the entire input file, execute either an E or Q command to close the output file.

Table 4-8 lists the commands used to abort the string search command.

Table 4-8
Abortting Editor String Search Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTRL/U</td>
<td>A CTRL/U will return control to the Editor command mode if you type it while entering text in a string search command.</td>
</tr>
<tr>
<td>DELETE</td>
<td>Pressing the DELETE key while entering text for a string search causes the text so far entered to be ignored and allows a new string to be inserted. The Editor displays a dollar sign ($) in response.</td>
</tr>
</tbody>
</table>

4.5 A SAMPLE EDITING JOB

You will often find it necessary to edit a closed file to change the code, correct overlooked errors, or insert additional information.

This section shows you how to use editing commands to edit an existing assembly language program. The example program, shown below, is written in PAL8 assembly language. When executed it displays HELLO!

Note that it contains deliberate errors.

```assembly
*200
MONADR=7600
START, CLA CLL /CLEAR ACCUMULATOR AND LINK
TLS /CLEAR TERMINAL FLAG
TAD BUFADR /SET UP POINTER
DCA PNTR /FOR GETTING CHARACTERS
TFF /SKIP IF TERMINAL FLAG SET
JMP , -1 /NO! CHECK AGAIN
TAD I PNTR /GET A CHARACTER
TLS /PRINT A CHARACTER
ISZ PNTR /DONE YET?
CLA CLL /CLEAR ACCUMYATOR AND LINK
TAD I PNTR /GET ANOTHER CHARACTER
SZA CLA /JUMP ON ZERO AND CLEAR
JMP I MON /RETURN TO MONITOR
JMP NEXT /GET READY TO PRINT ANOTHER
BUFADR BUFF /BUFFER ADDRESS
PNTR, BUFF /POINTER
BUFF, 215212;"H";"E";"L";"O";0 |
MON, MONADR /MONITOR ENTRY POINT
```
THE SYMBOLIC EDITOR

You have checked the above program and want to make the following changes:

- Insert a comment line at beginning of program.
- Correct TFF to TSF in line with label NEXT.
- Correct the spelling of accumulator where it appears the second time.
- Transpose lines JMP I MON and JMP NEXT.
- Add a comma after label BUFADR.

To summon the Editor and open the file, type EDIT and the file name, SAMPLE/PA. The Editor will prompt with the number sign (#) to indicate that it is ready to receive the editing commands:

`EDIT SAMPLE.PA
#

If you are changing a part of a program but want to save your original file, type the following commands:

`RENAME SAMPLE.BK<SAMPLE.PA
`EDIT SAMPLE.PA<SAMPLE.BK

The Editor will assign the name SAMPLE.PA to the modified version. SAMPLE.BK will remain unchanged.

At this point, the text buffer is empty. Type the R (Read) command to read in the first page of the file, which in the case of this example, is the entire program. The Editor prompts with the number sign (#) when it has executed the command. Use the L command to display the program.

First you want to insert a comment line at the top of the program to serve as a title. To do this, type the I (Insert) command, followed by your text.

`I
/ROUTINE TO TYPE A MESSAGE

This command inserts whatever text you type before line 1 of the text buffer. The RETURN key is typed after the line to allow the Editor to recognize this text as a separate line. Now type CTRL/L to return to the Editor command mode. Verify that this is now line 1 by typing the command

`1L

which causes the Editor to display line 1 of the buffer.

/ROUTINE TO TYPE A MESSAGE

Your next job is to change TFF to TSF.

Use the string search ESCape command to find and display the line

`$TFF'L

4-19
THE SYMBOLIC EDITOR

Use the single character search option and the current line pointer to make the correction in the line by typing.

$.S

followed by the RETURN key and the search character, F.

The Editor will display the line up to the F. Delete the F by pressing the DELETE key and type in the correct character S. Since this completes all changes in this line, type CTRL/G (BELL) followed by another CTRL/G (BELL) to return to command mode.

Now change ACCUMYLATOR to ACCUMULATOR. This example will illustrate the Editor's ability to accomplish repetitive searches for identical character strings. The intrabuffer search commands are shown here to illustrate their use.

The intrabuffer search uses the ESCape key and the single quote (') and double quote (") constructions.

Typing

$!CL A'"L

(where $ is the echo for ESCape) will list the line containing the second occurrence of CLA, that is,

CLA CLL /CLEAR ACCUMYLATOR AND LINK

To correct the word, use the single character search command S with the period construction to indicate the current line.

Type

$.S

Now type the RETURN key and Y, the character that is the object of the search. Use the DELETE key to erase this character and type the correct character U.

This concludes the corrections on this line. To return to the Editor command mode type CTRL/G twice. Note that if you typed a carriage return after making the correction, control would return to the command mode but the remainder of the line after the corrected character to the end of the line would be deleted.

Verify correction by typing

$.L

to list the corrected line

CLA CLL /CLEAR ACCUMULATOR AND LINK

The next correction transposes the two lines JMP I MON and JMP NEXT.

Use the character string search command J, which will return the number of the line to be moved. Type

$J
$JMP I MON'
$.=0016
THE SYMBOLIC EDITOR

Use the M (Move) command to transpose the line by typing

16,16$18M

This command moves line 16 (JMP IMOV) before line 18, or after line 17 (JMP NEXT), which in effect performs the transposition. It also
deletes the initial occurrence of the moved line, now making JMP IMON
the 17th line. Type 17L to verify this.

Next add a comma after the label BUFADR. The G (Get) command allows
you to get the next line that has a label associated with it after the
line that is equal to the current line counter. Thus, typing

$G

will get the line starting with BUFADR and display it on the terminal
as follows:

BUFADR  BUFF  /BUFFER ADDRESS

Then make the correction by adding a Comma after the label, using the
change or character search commands previously described.

When you have completed all the changes, display the corrected file on
the terminal. Type

$L

and the entire contents of the text buffer will appear.

Remember to close out the file and return to the monitor by typing the
E command.

4.6 EDITOR OPTIONS

The Editor provides the following options:

- /B The Editor converts two or more spaces to a TAB when
  reading from an input device.

- /D The Editor deletes the old copy of the output file (if
  one exists) before opening the new output file on the
  device. If you do not specify /D, the Editor does not
delete the old copy of the output file until you have
  transferred all data to the new file with the E or Q
  command.

4.7 EDITOR ERROR MESSAGES

Two types of error messages, nonfatal and fatal, are generated when an
error is made while running the Editor.

Nonfatal errors, such as an incorrect format in a command string or a
search for nonexistent information, cause the Editor to display a
question mark. For example, if a command requires two arguments and
only one is provided, the Editor will display a question mark (?),
perform a carriage return/line feed, and ignore the command as typed.
Similarly, if you type an illegal or unrecognized command character,
the error message ? will be displayed, followed by a carriage
return/line feed, the command will be ignored. However, if you
provide an argument for a command that does not require one, the
argument may be ignored and the normal function of the command
performed. Table 4-9 lists nonfatal errors that you may encounter
while using the Editor.

Table 4-9
Nonfatal Editor Error Messages

<table>
<thead>
<tr>
<th>Condition/Message</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>L ?</td>
<td>The buffer is empty. Nonexistent information is requested.</td>
</tr>
<tr>
<td>7,5L ?</td>
<td>The arguments are in the wrong order. The Editor cannot list backward.</td>
</tr>
<tr>
<td>17$10M ?</td>
<td>This command requires two arguments before the $; only one was provided.</td>
</tr>
<tr>
<td>H ?</td>
<td>Nonexistent command letter.</td>
</tr>
</tbody>
</table>

Major errors (Table 4-10) cause control to return to the monitor.
These errors cause a message to be printed in the form:

?n "C

where:

n is an error code listed in the table
"C indicates that control has passed to the monitor.

These errors generally result in complete loss of the output file.

Table 4-10
Major Editor Error Codes

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Editor failed in reading a device. Error occurred in device handler; most likely a hardware malfunction.</td>
</tr>
<tr>
<td>1</td>
<td>Editor failed in writing onto a device; generally a hardware malfunction.</td>
</tr>
<tr>
<td>2</td>
<td>File close error occurred. For some reason the output file could not be closed; the file does not exist on that device.</td>
</tr>
<tr>
<td>3</td>
<td>File open error occurred. This error occurs if the output device is a read-only device or if no output file name is specified on a file-oriented output device.</td>
</tr>
<tr>
<td>4</td>
<td>Device handler error occurred. The Editor could not load the device handler for the specified device. This error should not normally occur.</td>
</tr>
</tbody>
</table>
4.8 SUMMARY OF EDITOR COMMANDS AND SPECIAL CHARACTERS

The command and special characters discussed in this chapter are summarized in Table 4-11.

Table 4-11
Editor Command and Special Characters

<table>
<thead>
<tr>
<th>Command</th>
<th>Format</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A</td>
<td>Append the following text being typed at the keyboard until a CTRL/L (form feed) is typed. The form feed returns control to the command mode. Text input following the A command is appended to whatever is present in the text buffer.</td>
</tr>
<tr>
<td>B</td>
<td>B</td>
<td>List the number of available memory locations in the text buffer. The Editor returns the number of locations on the next line. To estimate the number of characters that can be accommodated in this area, multiply the number of free locations by 1.7.</td>
</tr>
<tr>
<td>C</td>
<td>nC</td>
<td>Change the text of line n to the line(s) typed after the command is entered. (Typing a CTRL/L terminates the input.)</td>
</tr>
<tr>
<td></td>
<td>m,nC</td>
<td>Delete lines m through n and replace with the text line(s) typed after the command is entered. (Typing CTRL/L indicates the end of the inserted lines.)</td>
</tr>
<tr>
<td>D</td>
<td>nD</td>
<td>Delete line n from the buffer.</td>
</tr>
<tr>
<td></td>
<td>m,nD</td>
<td>Delete lines m through n from the buffer.</td>
</tr>
<tr>
<td>E</td>
<td>E</td>
<td>Output the text buffer and transfer all remaining pages of the input file to the output file, closing the output file and returning to the monitor.</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>Follows a string search. Look for next occurrence of the string currently being sought (by the J command).</td>
</tr>
</tbody>
</table>

NOTE

If the search fails while you are using the F command, further commands cause the system to prompt with a ?. The file must be closed and then reopened.

(continued on next page)
## THE SYMBOLIC EDITOR

### Table 4-11 (Cont.)
Editor Command and Special Characters

<table>
<thead>
<tr>
<th>Command</th>
<th>Format</th>
<th>Meaning</th>
</tr>
</thead>
</table>
| G       | G      | Get and list the next line that has a label associated with it. A label in this context is any line of text that does not begin with one of the following:

  - space (ASCII 240)
  - / (ASCII 257)
  - TAB (ASCII 211)
  - RETURN (ASCII 215)

At the termination of a G command, control goes to the command mode with the current line indicator (.) equal to the line just listed.

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>nG</td>
<td></td>
<td>Get and list the first line that begins with a label, starting the search at line n.</td>
</tr>
<tr>
<td>I</td>
<td>I</td>
<td>Insert whatever text is typed before line 1 of the text buffer. (Typing CTRL/L terminates the entering process and returns control to the Editor command mode.)</td>
</tr>
<tr>
<td>nI</td>
<td></td>
<td>Insert whatever text is typed (until a CTRL/L is typed) before line n of the text buffer.</td>
</tr>
<tr>
<td>J</td>
<td>J</td>
<td>Interbuffer search command for character strings (see Section 4.4.2.2 describing the InterBuffer Character String Search).</td>
</tr>
</tbody>
</table>

### NOTE

If the search fails while you are using the J command, further commands cause the system to prompt with a ?. The file must be closed and then reopened.

| K      | K      | Kill the buffer. Delete all text from the text buffer. |

### NOTE

The Editor ignores the commands nK and m,nK with the result that you cannot destroy the buffer by mistyping a List command (m,nL).

(continued on next page)
THE SYMBOLIC EDITOR

Table 4-11 (Cont.)
Editor Command and Special Characters

<table>
<thead>
<tr>
<th>Command</th>
<th>Format</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td>L</td>
<td>List entire contents of the text buffer on the terminal.</td>
</tr>
<tr>
<td></td>
<td>nL</td>
<td>List line n of the text buffer on the terminal.</td>
</tr>
<tr>
<td></td>
<td>m,nL</td>
<td>List lines m through n of the text buffer on the terminal. Control then returns to command mode.</td>
</tr>
<tr>
<td>M</td>
<td>m,n$PM</td>
<td>Move lines m through n directly before line p in the text buffer. The $ character represents typing the dollar sign key, and not other possible keys. The old occurrence of the moved text is removed.</td>
</tr>
<tr>
<td>N</td>
<td>N</td>
<td>Write the current buffer to the output file and read the next page.</td>
</tr>
<tr>
<td></td>
<td>nN</td>
<td>Write the current buffer to the output file, kill the buffer, and read the next page. This action is repeated n times until the nth page is in the text buffer. Control then returns to command mode. You may not use the N command with an empty text buffer. A question mark (?) is printed if you attempt to do this.</td>
</tr>
<tr>
<td>P</td>
<td>P</td>
<td>Write the entire text buffer to the output file. The P command automatically outputs a FORM character (214) after the last line of output.</td>
</tr>
<tr>
<td></td>
<td>nP</td>
<td>Write line n of the text buffer to the output file and a FORM character.</td>
</tr>
<tr>
<td></td>
<td>m,nP</td>
<td>Write lines m through n, inclusive, to the output file and a FORM character.</td>
</tr>
<tr>
<td>Q</td>
<td>Q</td>
<td>Immediate end-of-file. Q causes the text buffer to be output and the file closed.</td>
</tr>
<tr>
<td>R</td>
<td>R</td>
<td>Read one page from the input device and append the new text to the current contents of the text buffer. If no input file was indicated or if no input remains, a question mark (?) is displayed and control returns to the command mode.</td>
</tr>
<tr>
<td>S</td>
<td>S</td>
<td>Character search command (see Section 4.4.1).</td>
</tr>
</tbody>
</table>

(continued on next page)
### THE SYMBOLIC EDITOR

Table 4-11 (Cont.)
**Editor Command and Special Characters**

<table>
<thead>
<tr>
<th>Command</th>
<th>Format</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>V</td>
<td>V</td>
<td>List the entire text buffer on the line printer.</td>
</tr>
<tr>
<td></td>
<td>nV</td>
<td>List line n of the text buffer on the line printer.</td>
</tr>
<tr>
<td></td>
<td>m,nV</td>
<td>List lines m through n, inclusive, on the line printer.</td>
</tr>
</tbody>
</table>
| Y       | nY     | Yank (read) in a logical page from the input file, without writing any output. For example, 
|         |        | #5Y reads through four logical pages of input, deleting them without producing output. The fifth page is read into the text buffer, and control automatically returns to the command mode. |
| $(ESC)  | TEXT'  | Perform a character string search for the string TEXT. Following a string search, "#" causes a search for the next occurrence of the string (see Section 4.4.2.1 describing the Intrabuffer Character String Search). |
| .= or .: |        | Typing these characters obtains the current line number (.=) and the last line number (/=) in the text buffer. The number is printed by the Editor immediately after you type the equal sign. (The colon character is equivalent to the equal sign.) |
| /= or /: |        |        |
| >       | >      | Equivalent to .+1L, list the next line in the text buffer. |
| <       | <      | Equivalent to .-1L, list the preceding line in the text buffer. |
| LINE FEED Key |        | Equivalent to .+1L, list the next line in the text buffer. |
| #       | #      | Print the current Editor version number. |
CHAPTER 5

THE PAL8 ASSEMBLER

5.1 INTRODUCTION

PAL8, the OS/78 Operating System assembler, generates binary object files from source (ASCII) programs written in the PAL8 assembly language.

PAL8 is a two-pass assembler. During pass 1, the source program is read and an internal symbol table is produced that contains the PAL8 permanent symbols and any new symbols that you define. During pass 2, the assembler reads the source file again, generates the binary code using the symbol table definitions created during pass 1, and continues defining symbols as well. The binary file that is output may be loaded into memory as the "current" executable program by the LOAD command. Absolute binary format consists of 8-bit bytes, containing field setting commands, address setting commands, and sequential data words. An optional third pass will produce a program listing if one is desired. During pass 3, the assembler reads the source file a final time and generates the assembly listing as an ASCII (character string) file. The assembly listing consists of the source statement together with its current location counter and the generated code in octal. The first 40 (decimal) characters of the first line of each page of the listing contain a title, the assembler version number, the date and the listing page number.

Use the OS/78 command PAL to call the assembler. You can also use the commands CREF and EXECUTE as explained in this chapter.

The PAL command specifies the binary and listing output devices and file names, the input devices and file names, and any options that you select. From one to nine input files may be specified. The typical way to assemble, load, and then run a program called PROG is as follows:

- PAL PROG - Assemble the program
- LOAD - Load the program into memory
- SAVE SYS PROG - Save the program
- R PROG - Run the program

The long form of the command string is

PAL dev:binary,dev:listing,dev:crefls<dev:input,.../options

If the extension to the file name is omitted, the following extensions are assumed.

.PA for input files.
.BN for binary output file.
.LS for listing output file.
.TM for intermediate CREF file.

5-1
THE PAL8 ASSEMBLER

If an assembly or CREP listing is not desired, omit the listing file or CREP file, respectively.

For example, to assemble, load, and run a PAL8 program named SAMPLE, which is stored on diskette unit 1, type

```
.PAL RXA1:SAMPLE/G-T
```

After assembly the program is loaded and run (since the /G was specified) with the starting address assumed to be location 0200 in field 0; the binary file is stored on the DSK: device as SAMPLE.BN. The -T General-Purpose Dash Option displays the assembled program listing on the terminal (see Table 2-3).

If a binary file is not desired, specify the -NB option at the end of the command line (NB stands for No Binary). For example, to get a listing only, type

```
.PAL SAMPLE-LS-NB
```

The -LS option indicates that a listing should be produced.

The assembler displays any error messages encountered in the program on the terminal, even when a listing is not produced. Typing CTRL/O at the keyboard during an assembly suppresses the display of error messages. However, messages are still printed in the listing file (if any) and occur immediately before the line that is in error.

For example, the command line

```
.PAL SAMPLE/S-LS
```

causes PAL8 to assemble SAMPLE.PA (or SAMPLE), generating DSK: SAMPLE.BN and putting the listing into the file SAMPLE.LS on the default device DSK. The /S option suppresses listing of the symbol table.

The command line

```
.PAL BIN<SAMPLE.PA/G=600
```

assembles SAMPLE.PA, creates a binary output file named BIN.BN, loads the file BIN.BN, and starts it at location 600. The construction =600 is an option that specifies the starting address.

Assembly can be terminated at any time by typing CTRL/C on the keyboard, and any output files being stored will be deleted. Otherwise, PAL8 always returns to the monitor upon completion of assembly.

A source program may consist of a number of source file modules to be assembled together. You do this by specifying a string of input device and file names separated by commas. For example,

```
.PAL PART1,RXA1:PART2,RL0A:PART3
```

assembles a three-part program. This technique is useful when it is desired to assemble two programs that are identical except for a few lines at the beginning of the programs. Different lines can be broken
out into a "prefix file". For example, two different file assemblies may be generated by

.PAL PRFX1,FILE

and

.PAL PRFX2,FILE

You can enter up to nine input files to be treated as one source input in a command line.

If more than one input file is specified, and output files are desired but not explicitly specified, the name of the first input file is used for the output file names. For example,

.PAL A,B

produces the binary file A.BN.

If a file name other than the first input file is desired for the binary name, use the -NB General-Purpose Dash Option after the last input file name not desired as the binary file name. For example,

.PAL A-NB,B

produces DSK:B.BN and

.PAL A,B,C-NB,D,E,F

produces DSK:D.BN.

If a -LS option is specified, it must appear immediately after an input file name. This is the name that will be used for the name of the listing file. For example,

.PAL A,B-LS

produces DSK:B.LS while

.PAL A-LS,B

produces DSK:A.LS

The -L or -T General-Purpose Dash Options used with a PAL or COMPIL<command> send the listing output file to the line printer and terminal respectively.

Note that the PAL command normally produces a binary file even when a name is not given. Thus, typing

.PAL LPT:<file

produces a binary file.

If you do not specify an extension, PAL assumes that the input file extension is .PA. Thus, the command

.PAL TEST

causes the assembler to search for a file name DSK:TEST.PA. If no file with .PA is found on DSK:, the assembler then searches for a file named TEST with no extension. It is good practice when creating a PAL8 source file to include a .PA extension to remind you what type of source file it is.
The COMPILER and EXECUTE commands may also be used to invoke PAL8. These commands search the directory of the specified device for the file given with the command, and if one is found with a .PA extension, PAL8 is invoked. For example,

`*COMPILE TEST`

will run PAL8 if TEST.PA is found. An unusual extension may be explicitly specified by typing

`*PAL TEST.XX`

which will assemble DSK:TEST.XX. To specify PAL8 as the processor in the COMPILER command, use the -PA General-Purpose Dash Option in the command line as follows:

`*COMPILE TEST.XX-PA`

The EXECUTE command is similar to the COMPILER command except that the EXECUTE command is supported by the /G option.

If an argument is not given with a PAL or COMPILER or EXECUTE command, the argument used with the last such command is assumed when that command is used again.

5.2 CREATING AND RUNNING A PAL8 PROGRAM

The following steps demonstrate the procedure for creating and running a PAL8 program.

5.2.1 Creating a Program

Create the assembly language source file by calling the Editor as follows:

`*CREATE SAMPLE.PA`

Since a new program is being created, only a single file name need be specified. The OS/78 Editor will then display a number sign (#) to indicate it is ready to accept a command. (See Chapter 4 for a detailed discussion of the OS/78 Editor.)

Type the A (Append) command to allow the Editor to accept text. Then type in the program, one line at a time. Press the RETURN key after each line.

```
#A
/ROUTINE TO TYPE A MESSAGE
*200
MONADR=7600
START, CLA CLL /CLEAR ACCUMULATOR AND LINK
   TLS /CLEAR TERMINAL FLAG
   TAD BUFADR /SET UP POINTER
   DCA PNTR /FOR GETTING CHARACTERS
```
THE PAL8 ASSEMBLER

NEXT, TSF /SKIP IF TERMINAL FLAG SET
JMP ,1 /NO! CHECK AGAIN
TAD I PNTR /GET A CHARACTER
TLS /PRINT A CHARACTER
ISZ PNTR /DONE YET?
CLA CLL /CLEAR ACCUMULATOR AND LINK
TAD I PNTR /GET ANOTHER CHARACTER
SZA CLA /JUMP ON ZERO AND CLEAR
JMP NEXT /GET READY TO PRINT ANOTHER
JMP I MON /RETURN TO MONITOR

BUFADR, BUFF /BUFFER ADDRESS
PNTR, BUFF /POINTER
BUFF, 215#212;#H;#E;#L;#O;#O
MON, MONADR /MONITOR ENTRY POINT

Now type a CTRL/L to terminate input. This command returns you to the Editor command mode.

Type the L (List) command in response to the Editor's number sign (#) to list the text that was inserted into the text buffer.

When you are satisfied that the input is correct, type the E (Exit) command to store the file and return to the monitor.

5.2.2 Assembling a Program

Now assemble the source program just created. Use the command:

PAL SAMPLE-LS

This command creates two files, a binary file called SAMPLE.BN, and a listing file (-LS option) called SAMPLE.LS. Use the TYPE command to display the listing on the terminal or the LIST command to print the listing on a line printer.

The assembly listing produced by PAL appears as follows:

/Routine to Type a Message

PAL8-V13A 14-MARCH-79 PAGE 1

/Routine to Type a Message

0200 *200
7600 MONADR=7600

000200 7300 START, CLA CLL /CLEAR ACCUMULATOR AND LINK
000201 6046 TLS /CLEAR TERMINAL FLAG
000202 1216 TAD BUFADR /SET UP POINTER
000203 3217 DCA PNTR /FOR GETTING CHARACTERS
000204 6041 NEXT, TSF /SKIP IF TERMINAL FLAG SET
000205 5204 JMP ,1 /NO! CHECK AGAIN
000206 1617 TAD I PNTR /GET A CHARACTER
000207 6046 TLS /PRINT A CHARACTER
000210 2217 ISZ PNTR /DONE YET?
000211 7300 CLA CLL /CLEAR ACCUMULATOR AND LINK
000212 1617 TAD I PNTR /GET ANOTHER CHARACTER
000213 7640 SZA CLA /JUMP ON ZERO AND CLEAR
000214 5204 JMP NEXT /GET READY TO PRINT ANOTHER
000215 5631 JMP I MON /RETURN TO MONITOR
000216 0220 BUFADR, BUFF /BUFFER ADDRESS
000217 0220 PNTR, BUFF /POINTER
000220 0215 BUFF, 215#212;#H;#E;#L;#O;#O

5-5
THE PAL8 ASSEMBLER

000221 0212
000222 0310
000223 0305
000224 0314
000225 0314
000226 0317
000227 0241
000230 0000
000231 7600 MON, MONADR /MONITOR ENTRY POINT

/ROUTINE TO TYPE A MESSAGE PAL8-V13A 14-MARCH-79 PAGE 2

BUFADR 0216
BUFF 0220
MON 0231
MONADR 7600
NEXT 0204
PNTR 0217
START 0200

ERRORS DETECTED: 0
LINKS GENERATED: 0

The first column of the listing gives the field number and octal address. The second column is the assembled object code. The symbol table is printed at the end followed by the number of errors detected and number of links generated. Link generation is described in Section 5.12. Each error generates an error message (see Section 5.14).

If errors have been detected, the program has been written or typed incorrectly. Check it again.

The COMPILE command may also be used to assemble the program by typing

."COMPILE SAMPLE

Several options are available with the PAL command. The options are described in Section 5.3.

5.2.3 Loading and Saving a Program

Load the binary file generated by assembling SAMPLE.PA into memory by typing

."LOAD SAMPLE

The SAMPLE program is now the "current" memory image.

Since programs in memory image format can be executed directly, it is desirable to save this format of your program. Do this with the SAVE command by typing

."SAVE SYS SAMPLE

The memory image format of SAMPLE is now both in memory and on the system device as a new file called SAMPLE.SV.
5.2.4 Executing the Program

Since the program now resides on SYS and in main memory, you can execute it by typing

.START

Otherwise, you can load the file SAMPLE.SV into memory from SYS: and run by typing

.R SAMPLE

As the program runs, it displays the message HELLO!

You can also use the EXECUTE command to assemble, load and run the program.

.EXECUTE SAMPLE

This command produces the binary file SAMPLE.BN, loads it into memory, and starts it running.

Another load-and-go method that is available with the PAL command is the /G option. Typing

.PAL SAMPLE/G

assembles the input file SAMPLE.PA, loads the binary file, and executes the program. Also, the command

.LOAD SAMPLE/G

will load the binary file SAMPLE.BN and execute it.

5.2.5 Getting and Using a Cross-Reference Listing

The Cross-Reference Program (CREF) aids in debugging assembly language programs by pinpointing all references to a particular symbol.

Generate the CREF listing by using the CREF command or the PAL command with the /C option. Typing

.PAL SAMPLE/C-LS

will produce a binary file and the CREF listing as a file called SAMPLE.LS. Using the TYPE or LIST command will display the listing on the terminal or print it on a line printer, respectively. Further information on CREF is given with the discussion of the CREF command in Chapter 3.

The output of CREF is identical to the PAL8 assembler output except that the CREF program numbers each line in decimal and generates after the listing a cross-reference table that has the following format:

| BUFADR | 6 | 18# |
| BUFF   | 18| 19  | 20# |
| MON    | 17| 29# |
| MONADR | 3#| 29  |
| NEXT   | 8#| 16  |
| PNTR   | 7 | 10  | 12 | 14 | 19# |
| START  | 4#|

5-7
The cross-reference table contains every user-defined symbol and literal, sorted alphabetically. If literals are used, each literal is indicated by an underline followed by the field and address at which it occurs. For each symbol and literal there appears a list of numbers that specify the lines in which each is referenced. The symbol # follows the number of the line where the symbol is defined.

5.2.6 Obtaining a Memory Map

Many times it is desirable to obtain a map of a program showing memory locations used by the given binary file. Generate the map by using the MAP command. Typing

```
*MAP SAMPLE-L
```

will print the map on a line printer from SAMPLE.BN, and typing

```
*MAP SAMPLE
```

will display the map on the terminal (in effect, equivalent to the command MAP SAMPLE-T).

The input file must always be a binary file (.BN extension). Use the command

```
*MAP MAPFIL<SAMPLE
```

to place the map in the file MAPFIL.MP. Display the map on the terminal or print it on a line printer by using the commands TYPE and LIST, respectively. The map for the program SAMPLE is shown below.

```
BITMAP V4 FIELD 0

000000000111111222222233333344444455555555556666666677777777
0123456701234567012345670123456701234567012345670123456701234567
00000
00100
00200 1111111111111111111111111111110000000000000000000000000000000000
00300
00400
00500
00600
00700
```

The output is a series of lines that are made up of a string of digits. Each digit, which represents a single memory location, can have a value of 0 to 3. A 0 means that the location is empty while a 1 means that the location was loaded into once. The appearance of a 2 means that a location was loaded into two times. A 3 means that the location was loaded into three or more times. The appearance of a 2 or 3 may imply a programming error in that two or more separate routines are each trying to load values into the same location. The example program shows memory locations 0200 through 0231 being loaded into once which is correct. Further information on the MAP command is given in Chapter 3.
5.3 PAL8 OPTIONS

The command string typed for the PAL command may include several options. The options are listed in Table 5-1.

<table>
<thead>
<tr>
<th>Option</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>/B</td>
<td>This option makes the operator ! a 6-bit left shift instead of an inclusive OR (A!B equals A^100 B). This allows you to pack two 6-bit ASCII characters into a 12-bit word. This effect applies for the entire assembly.</td>
</tr>
<tr>
<td>/C</td>
<td>Create a symbol cross-reference listing (runs CREF.SV program) after assembly. The third output file specified (optional) is the temporary output file passed to CREF. The second output file is the listing file to be produced. If no third output file is given, SYS:CREPS.TM is assumed and will be deleted after use. The /C option supersedes the /G and /L options if specified in the same command string.</td>
</tr>
<tr>
<td>/E</td>
<td>Enable error messages if a link is generated. The LG error message is generated as well as the link being flagged.</td>
</tr>
<tr>
<td>/F</td>
<td>Disable extra zero fill in TEXT pseudo-op. If the text in the TEXT pseudo-op contains an even number of characters, no word of zeros will be added to the end.</td>
</tr>
<tr>
<td>/G</td>
<td>Load the binary file into memory and begin execution at the indicated starting address. If no starting address is indicated, start at 200.</td>
</tr>
<tr>
<td>/H</td>
<td>Generate nonpaginated output. Headers (including page numbers and page format) are suppressed.</td>
</tr>
<tr>
<td>/J</td>
<td>Do not list lines of unassembled conditional source code.</td>
</tr>
<tr>
<td>/K</td>
<td>Used in assembling very large programs; allows more space in field 1 to be used for symbol table storage.</td>
</tr>
<tr>
<td>/L</td>
<td>Load the resulting binary file into memory but do not start it.</td>
</tr>
</tbody>
</table>

(continued on next page)
### Table 5-1 (Cont.)
**PAL8 Options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>/N</td>
<td>Generate the symbol table but not the rest of the listing.</td>
</tr>
<tr>
<td>/O</td>
<td>Disable output of default (200) current location counter (CLC) setting after a FIELD pseudo-op. The CLC remains unchanged.</td>
</tr>
<tr>
<td>/S</td>
<td>Omit the symbol table normally generated with the listing.</td>
</tr>
<tr>
<td>/T</td>
<td>Output a carriage return/line feed in place of form feed character(s) in the program listing.</td>
</tr>
<tr>
<td>/W</td>
<td>Do not remember the number of literals that were previously stored on a page after changing the current location counter to an off page value and then back again.</td>
</tr>
</tbody>
</table>

When the /L or /G option is specified, you can also include any option in the command line for the LOAD command, such as = starting address option. If no address is specified, 00200 is assumed. If no binary output file is specified (by using the -NB option) with a /L or /G, a temporary file SYS:PAL8BN.TM is created and loaded.

### 5.4 CHARACTER SET

PAL8 programs are composed of physical lines containing assembly language mnemonics that indicate processor instructions, user-defined symbols, comments, listing control characters and pseudo-operators (assembler directives). The following characters (see Appendix A also) are used to specify these components.

1. The alphabetic characters A through Z
2. The numeric characters 0 through 9
3. The characters special characters and operators described below
4. Characters that are ignored during assembly, such as LINE FEED and FORM FEED

All other characters are illegal (except when used in a comment) and cause the following error message to be printed during passes 1 and 2:

```
IC nnnn
```

where:

nnnn represents the octal location at which the illegal character occurred.
As assembly proceeds, each instruction is assigned a location determined by the current location counter. When an illegal character or any other error is encountered during assembly, the value of the current location counter is displayed in the error message.

NOTE
You cannot use lower case characters in labels, instruction mnemonics and operands.

5.5 STATEMENTS

A PAL8 source program is prepared at the terminal using the Editor program (EDIT command) to enter a sequence of statements. You must enter each statement on a single line and terminate with a carriage return. PAL8 statements have four elements. They are identified by the order of their appearance in the statement and by the separating (or delimiting) character that follows or precedes the element. These elements are:

1. label
2. instruction
3. operand
4. comment

A statement must contain at least one of these elements and may contain all four. The assembler interprets and processes the statements, generating one or more binary instructions or data words, or performing an assembly process.

5.5.1 Labels

A label is the symbolic name created by the programmer to identify the location of a statement in the program. If present, the label is written first in a statement. It must begin with an alphabetic character, contain only alphanumeric characters, and be terminated by a comma. There must be no intervening spaces between any of the characters and the comma. A label may be of any length, but only the first six characters are significant. If a label is the only element on a line, it identifies the location of the next program location.

For example,

```
*200 A,
B,0
```

Defines A as 00200
Defines B as 00200 and stores 0000 at location 00200

5.5.2 Instructions

An instruction may be one or more of the mnemonic machine instructions or a pseudo-operation that directs assembly processing. (Assembly pseudo-ops are described in Section 5.11.) Instructions are terminated with zero or more spaces (or tabs) followed by a semicolon, slash, or the end of the line.
5.5.3 Operands

Operands are the octal or symbolic addresses of an assembly language instruction or the argument of a pseudo-operator, and can be any legal expression. In each case, interpretation of an operand depends on the instruction or the pseudo-op. Operands are terminated by a semicolon, slash, or the end of the line.

5.5.4 Comments

Comments are arbitrary strings of any character in the ASCII set (see Appendix A) that begin with a slash (/). Comments do not affect assembly processing or program execution but are useful in the program listing to record information for later analysis or debugging. The assembler ignores all characters between the slash and the next carriage return.

It is possible to have only a carriage return on a line, resulting in a blank line in the final listing. Such a line is ignored, and the current location counter is not incremented.

5.6 FORMAT CHARACTERS

The following characters are useful in controlling the format of an assembly listing to improve readability. They allow a neat readable listing to be produced by providing a means of spacing through the program.

5.6.1 Form Feed

The form feed character causes the assembler to output blank lines (or a form feed character if listing on the line printer) in order to skip to a new page in the output listing during pass 3; this feature is useful in creating a page-by-page listing. The form feed is generated by the Editor P (Page) command. The pseudo-op EJECT may also be used to form pages in the assembly listing (see Section 5.11.7).

5.6.2 Tab

Tabs are used in the body of a source program to separate fields into columns. For example, a line written

    GO, TAD TOTAL/MAIN LOOP

is much easier to read if tabs are inserted to form

    GO, TAD TOTAL /MAIN LOOP

Each occurrence of a tab character causes PAL8 to output enough spaces to move to the next text column. Each text column is 8 characters wide.
5.6.3 Statement Terminators

Each statement is terminated by the carriage return/line feed character combination produced by the Editor when the RETURN key was pressed during the Insert or Append modes. The semicolon (;) may also be used as a statement terminator and is considered identical to a carriage return except that it will not terminate a comment. For example,

```
TAD A /THIS IS A COMMENT; TAD B
```

The entire expression between the slash and the end of the line is considered a comment. Thus in this case the assembler ignores the TAD B. If, for example, a sequence of instructions to rotate the contents of the accumulator and link six places to the right is desired, it can be written as follows:

```
RTR
RTR
RTR
```

However, as an alternative, all three instructions can be placed on a single line by separating them with the special character semicolon and terminating the entire line with a carriage return. The above sequence of instructions can then be written:

```
RTR;RTR;RTR
```

These multistatement lines are particularly useful when setting aside a section of data storage. For example, a 4-word block of data could be reserved by specifying either of the following:

```
LIST, 1;2;3;4
```

or

```
LIST, 1
2
3
4
```

5.7 NUMBERS

Any sequence of digits delimited by a SPACE, TAB, semicolon, or the end of a line forms a number. PAL8 initially interprets numbers in octal (base 8). This can be changed to decimal using the pseudo-op DECIMAL (Section 5.11.10). Numbers are used in expressions.

5.8 SYMBOLS

A symbol is a string of alphanumerics characters beginning with a letter and delimited by a nonalphanumerics character. Although a symbol may be any length, only the first six characters are significant. Since additional characters are ignored, symbols which are identical in their first six characters are considered identical.

5-13
5.8.1 Permanent Symbols

The assembler symbol table initially contains definitions of the symbols for all computer instructions and PAL8 pseudo-ops. These symbols are permanently defined by PAL8 and need no further definition by the user; they are summarized in Section 5.15. For example,

HLT  This is a symbolic instruction assigned the value 7402 in its permanent symbol table.

5.8.2 User-Defined Symbols

All desired symbols not defined by the assembler (in its permanent symbol table) must be defined within the source program. User symbols may be defined in two ways:

1. As a statement label. Labels are assigned a value equal to the current location counter.
2. As an explicitly defined symbolic value (for example, A = 33).

Permanent symbols (instructions, special characters, and pseudo-ops) may not be redefined as a label or symbolic value. The following examples are legal labels:

ADDR,
TOTAL,
SUM,
AL,

The following labels are illegal:

AD>M,  (contains an illegal character)
7ABC,  (first character not alphabetic)
LA_BEL,  (contains embedded spaces)
D+TAG,  (contains a legal but non-alphanumeric character)
LABEL  (a comma does not follow immediately after)
TAD,  (instruction mnemonic)

5.8.3 Current Location Counter

As source statements are processed, PAL8 assigns consecutive memory addresses to the instructions and data words of the object program (binary and listing) being produced.

The current location counter contains the address in which the next word of object code will be assembled and is automatically incremented each time a memory location is assigned. A statement that generates a single object program storage word increments the location counter by one. Another statement might generate six storage words, incrementing the location counter by six.

The location counter is set or reset by typing an asterisk followed by an expression giving the address in which the next program word is to be stored. The expression may include symbols, but every such symbol must have been defined at some previous point in the current source file(s) being assembled. If the origin is not set by the user, PAL8 begins assigning addresses at location 200.
The symbol TAG in the following example is assigned a value of 0300, the symbol B a value of 0302, and the symbol A a value of 0303.

```
*300 /SET CURRENT LOCATION COUNTER TO 300
TAG, CLA
JMP A
B, 0
A, DCA B
```

If a symbol is defined more than once as a label, the assembler will display the "illegal definition" error message:

```
ID address
where:

address is the octal value of the location counter at the second occurrence of the symbol definition. The symbol is not redefined. PAL8 error conditions are described in Section 5.14.
```

Example:

```
*300
START, TAD A
DCA COUNTER
CONTIN, JMS LEAVE
JMP START
A, -74
COUNTER, 0
START, CLA CLL
```

The symbol START would have a value of 0300; the symbol CONTIN would have a value of 0302; the symbol A would have a value of 0304; and the symbol COUNTER (considered COUNTET by the assembler, because the assembler uses only the first six characters of a symbol) would have a value of 0305. When the assembler processes the next line, it will display the error message:

```
ID COUNTET+0001
```

PAL8 will also display an error message if you refer to an undefined symbol. For example,

```
*7170
A, TAD C
CLA CMA
HLT
JMP A1
C, 0
```

This would produce the "undefined symbol" error message

```
US A+0003
```

since the symbol A1 has not been defined.
5.8.4 Symbol Table

Initially, the assembler's symbol table contains the definitions of the computer instructions and PAL8 pseudo-ops; these are PAL8's permanent symbols. As the source program is processed, user-defined symbols and their 12-bit binary values are added to the symbol table. Entries in the symbol table are listed in alphabetic order at the end of the assembly listing file.

During pass 1, if PAL8 detects that the symbol table is full (in other words, there is no more memory space in which to store symbols and their values), the "symbol table exceeded" error message is displayed as follows:

```
SE address
```

and control returns to the monitor. The number of symbols defined in the program may be reduced by using relative addressing for example, JMP START+3).

You can also segment a program and assemble the segments separately, taking care to define correct links between the segments. PAL8's symbol capacity is 2621 (decimal) symbols of which 96 are permanent. Where PAL8 is run under BATCH, 2357 symbols can be defined. (Use of the /K option expands these to 2971 and 2719 respectively, but assembly time is slower.)

Instructions for altering the permanent symbol table are in Section 5.11.8.

5.8.5 Direct Assignment Statements

New symbols and their assigned values may be inserted directly into the symbol table by using a direct assignment statement.

Format:

```
  SYMBOL=value
```

where:

```
  value      is a number or an expression.
```

No spaces or tabs may appear between the symbol to the left of the equal sign and the equal sign itself but they may appear (and are ignored) after the equal sign. The following are examples of direct assignment statements:

```
A=6
EXIT=JMP I 0
C=A+B
CO=JMS I [...] 
```

All symbols to the right of the equal sign must already be defined, except that symbols are allowed to be undefined during pass 1. For example,

```
A=B
...                          
B=3
```
During pass 1, A will equal 0, since it is undefined thus far. During pass 2, A will equal 3, since B is given the value 3 at the end of pass 1. The use of the equal sign does not increment the location counter; it is an instruction to the assembler itself rather than a data value.

A direct assignment statement may also equate a new symbol to the value assigned to a previously defined symbol. For example,

```
BETA=17
GAMMA=BETA
```

The new symbol GAMMA is entered into the user's symbol table with the value 17. The value assigned to a symbol may be changed as follows:

```
ALPHA=5
ALPHA=7
```

The second line of code shown changes the value assigned to ALPHA from 5 to 7.

Symbols defined by use of the equal sign may be used in any valid expression. For example,

```
*200
A=100 /DOES NOT UPDATE CURRENT LOCATION COUNTER
B=400 /DOES NOT UPDATE CURRENT LOCATION COUNTER
A+B /THE VALUE 500 IS ASSEMBLED AT LOC 200
TAD A /THE VALUE 1100 IS ASSEMBLED AT LOC 201
```

If the symbol to the left of the equal sign is in the permanent symbol table, the "redefinition" diagnostic

```
RD address
```

will be displayed as a warning (address is the value of the location counter at the point of redefinition). The new value will be stored in the symbol table. For example,

```
CLA=7600
```

will cause the diagnostic

```
RD+200
```

Whenever CLA is used after this point, it will have the value 7600.

5.8.6 Symbolic Instructions

Symbols used as instructions must be predefined by the assembler or defined in the assembly by the programmer. If a statement has no label, the instructions may appear first in the statement and must be terminated by a space, tab, semicolon, slash, or carriage return. The following are examples of legal instructions:

```
TAD (a mnemonic machine instruction)
PAGE (an assembler pseudo-op)
ZIP (an instruction defined by the user)
```
5.8.7 Symbolic Operands

Symbols used as operands normally have a value defined by the user. The assembler allows symbolic references to instructions or data defined elsewhere in the program. Operands may be numbers or expressions. For example,

TOTAL, TAD ACI+TAG

The values of the two symbols ACI and TAG (previously defined in the program) are combined by a two's complement add. (See Section 5.9.1 on Operators.) This value is then used as the operand address.

5.9 EXPRESSIONS

Expressions are formed by the combination of symbols, numbers, and certain characters called operators, which cause specific arithmetic operations to be performed. An expression is terminated by either a comma, carriage return, or semicolon. Expressions are evaluated by a left-to-right scan.

5.9.1 Operators

Seven characters in PAL8 act as operators:

+ Two's complement addition
- Two's complement subtraction
~ Multiplication (unsigned, 12-bit integer)
% Division (unsigned, 12-bit integer)
! Boolean inclusive OR
& Boolean AND
Space Treated as a Boolean inclusive OR except
(or TAB) in a memory reference instruction

No checks for arithmetic overflow are made during assembly, and any overflow bits are lost from the high-order end. For example,

7755+24

will give a result of 1.

The operators plus (+) and minus (-) may be used freely as unary (prefix) operators.

Multiplication is accomplished by repeated addition. No checks for sign or overflow are made. All 12 bits of each factor are considered as magnitude. For example,

3000^2

will give a result of 6000.
THE PAL8 ASSEMBLER

Division is accomplished by repeated subtraction. The quotient is the number of subtractions that are performed. The remainder is not saved and no checks are made for sign. Division by 0 will arbitrarily yield a result of 0. For example,

7000%1000

will yield a result of 7. This example could be written as:

-1000%1000

The answer might be expected to be -1 (7777), but all 12 bits are considered as magnitude and the result is still 7.

Use of the multiplication and division operators requires more attention to sign (on the part of the programmer) than is required for simple addition and subtraction. Table 5-2 contains examples of expressions using arithmetic operators.

Table 5-2
Use of Arithmetic Operators

<table>
<thead>
<tr>
<th>Expression</th>
<th>Also Written as</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>7777+2</td>
<td>-1+2</td>
<td>+1</td>
</tr>
<tr>
<td>7776-3</td>
<td>-2-3</td>
<td>7773 or -5</td>
</tr>
<tr>
<td>0^2</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>2^0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>1000^7</td>
<td></td>
<td>7000 or -1000</td>
</tr>
<tr>
<td>0%12</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>12%0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>7777%1</td>
<td>-1%1</td>
<td>7777 or -1</td>
</tr>
<tr>
<td>7000%1000</td>
<td>-1000%1000</td>
<td>7</td>
</tr>
<tr>
<td>1%2</td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

The ! operator causes a Boolean inclusive OR to be performed bit by bit between the left-hand term and the right-hand term. Giving the /B option changes the memory of "!" throughout the assembly to become a 6-bit left shift of the left term prior to the inclusive OR of the right. According to this interpretation,

If A=1 and B=2

then

A!B=0102

Under normal conditions A!B would be 0003.

The & operator causes a Boolean AND to be performed bit by bit between the left and right values.
SPACE is an operator that has special significance depending on the context in which it is used. When the symbol preceding the space is not a memory reference instruction as in the following example

SMA CLA

it causes an inclusive OR to be performed between them. In this case, SMA=7500 and CLA=7600. The expression SMA CLA is assembled as 7700. When SPACE is used following pseudo-operators, it merely delimits the symbol. When it is used after memory reference operators, it has a special function explained below.

User-defined symbols are treated as non-memory reference instructions. For example,

\[
\begin{align*}
A &= 1234 \\
B &= 77 \\
A \ B &
\end{align*}
\]

stores a data value of 1277 (octal), the same as A!B.

If data values are generated, the current location counter is incremented. For example,

\[
B-7;A+4;A-B
\]

produces three words of information; the current location counter is incremented after each expression. The statement

\[
\text{HLTCLA=HLT CLA}
\]

produces no information to be loaded (it produces a value for "HLTCLA" in the symbol table) and hence does not increment the current location counter.

In the program

\[
\begin{align*}
*4271 \\
\text{TEMP,} \\
\text{TEM2,} \\
0
\end{align*}
\]

the location counter is not incremented after the line TEMP,; the two symbols TEMP and TEM2 are assigned the same value, in this case 4721.

Since a CPU instruction has an operation code of three bits as well as one indirect bit, one page bit, and seven address bits, the assembler must combine memory reference instructions in a manner somewhat differently from the way in which it combines operate or IOT instructions. The assembler differentiates between memory reference instructions and user-defined symbols. The following symbols are the memory reference instructions:

\[
\begin{align*}
\text{AND} & \quad 0000 \quad \text{Logical AND} \\
\text{TAD} & \quad 1000 \quad \text{Two's complement addition} \\
\text{ISZ} & \quad 2000 \quad \text{Increment and skip if zero} \\
\text{DCA} & \quad 3000 \quad \text{Deposit and clear accumulator} \\
\text{JMS} & \quad 4000 \quad \text{Jump to subroutine} \\
\text{JMP} & \quad 5000 \quad \text{Jump}
\end{align*}
\]

When the assembler has processed one of these symbols, the space or tab following it acts as an address field delimiter. In the example,

\[
\begin{align*}
*4100 \\
\text{JMP A} \\
\text{A,} \\
\text{CLA}
\end{align*}
\]
A has the value 4101, JMP has the value 5000, and the space acts as a field delimiter. These symbols are represented in binary as follows:

A: 100 001 000 001
JMP: 101 000 000 000

The seven address bits of A are taken, for example,

000 001 000 001

The remaining bits of the address are tested to see if they are zeros (page zero reference); if they are not, the current page bit is set:

000 011 000 001

The operation code is then ORed into the JMP value to form

101 011 000 001

or, in octal

5301

In addition to the above tests, the page bits of the address field are compared with the page bits of the current location counter. If the page bits of the address field are nonzero and do not equal the page bits of the current location counter, an out-of-page reference is being attempted, and the assembler will take action as described in Section 5.12, on Link Generation and Storage.

5.9.2 Special Characters

In addition to the operators described in the previous section, PAL8 recognizes several special characters that serve specific functions in the assembly process:

- = equal sign
- , comma
- * asterisk
- . period
- " double quote
- ( parentheses
- [ square brackets
- / slash
- ; semicolon
- <> angle brackets
- $ dollar sign

The equal sign, comma, asterisk, slash, and semicolon have been previously described. The remaining special characters are described in the following sections.

5.9.2.1 Period (.) - The period character (.) represents the value contained in current location counter. It may be used in any expression (except to the left of an equal sign). It must be separated from other symbols by a space or other operator. For example,

*200
JMP .+2
is equivalent to JMP 0202. Also,

*300
*+2400

will produce in location 0300 the quantity 2700. Consider

*140
PRINT=JMS I.
2200

The second line (PRINT=JMS I.) does not increment the current location
counter; therefore, 2200 is placed in location 140 and PRINT is
placed in the user's symbol table with an associated value of 4540
(the octal equivalent of JMS I.). This technique is useful in
creating "global" subroutine calls.

Large buffers may be defined by using a format such as the following:

*1200 /BUFFER LOCATION
BUFFER, 0 /FIRST WORD OF BUFFER
*+400 /DEFINE A 401 WORD BUFFER
NEXT, 0 /PROGRAM CONTINUES

5.9.2.2 Double Quote (") - When a double quote (") precedes an ASCII
character, PAL8 assembles the 8-bit ASCII equivalent of the character.
(ASCII codes are listed in Appendix A.) For example,

CLA
TAD "A"

The constant 0301 is placed in the accumulator when these two
instructions are eventually executed. The character must not be a
carriage return or one of the characters that are ignored on input
(discussed at the end of this section).

5.9.2.3 Parentheses () and Brackets[] - Left and right parentheses,
(), enclose a current page literal. The right parenthesis is
optional.

*200

.

CLAC
TAD INDEX
TAD (2)
DCA INDEX
.
.

The left parenthesis is a signal to the assembler that the expression
that follows is to be evaluated and assigned a word in the literal
area of the current page. This is the same area in which the indirect
address linkages are stored. In the above example, the quantity 2 is
stored in a word in the literal area beginning at the end of the
current memory page. The instruction in which the literal appears is
given the address of the literal. A literal is assigned to storage
the first time it is encountered; subsequent references to the same
literal from the current page are made to the same location. The use
THE PAL8 ASSEMBLER

of literals frees symbol storage table and makes programs much more readable. Literal allocation starts with the last location on the page and works towards the first location. If the literal area reaches the instruction/data area, a PE (Page Exceeded) error message is generated and assembly continues.

If square brackets ([ and ]) are used in place of parentheses, the literal is assigned to page zero rather than the current page. This enables a value to be referenced from any address within the field. For example,

*200
  TAD[2]
  .
  .

*500
  TAD[2]
  .
  .

The closing member is optional. Literals may contain any expression.

NOTE

Literals can be nested, for example:

*200
  TAD (TAD (30))

This type of nesting may be continued to as many as six levels, depending on the number of other literals on the page and the complexity of the expressions within the next. If the limits of the assembler are reached, the error message BE (too many levels of nesting) or PE (too many literals) will result.

5.9.2.4 Angle Brackets (<> ) - Angle brackets (<> ) are used as conditional delimiters. The code enclosed in the angle brackets is assembled or ignored, depending on the definition of the symbol or value of the expression preceding the angle brackets. (The IFDEF, IFNDEF, IFZERO, and IFNZRO pseudo-operators are used with angle brackets and are described in Section 5.11.9.)

NOTE

Programs that use conditionals should avoid angle brackets in comments as they will be interpreted as beginning or terminating the conditional.
5.9.2.5 Dollar Sign($) - The dollar sign ($) character is optional at the end of a program and is interpreted as an unconditional end-of-pass. It may, however, occur in a text string, comment, or double quote (") term, in which case it is interpreted in the same manner as any other character. This feature is provided for compatibility with older PDF-8 assemblers, and its use is not recommended.

5.9.3 Other Characters

The following characters are handled by the assembler for the pass 3 program listing but are otherwise ignored:

- **FORM** FEED: Skips to a new page
- **LINE** FEED: Creates a line spacing without causing a carriage return
- **SPACE**: Spaces to next character position
- **TAB**: Spaces to next "tab column" of 8 characters
- **RETURN**: Terminates each line
- **BEL** (CTRL/G): Sounds the terminal buzzer

5.10 INSTRUCTION SET

The instruction set for PAL8 includes processor instructions, input/output (I/O) instructions, and assembler instructions (pseudo-ops). The processor instructions are further divided into two basic groups of instructions: memory reference and microinstructions. (See Section 5.15 for detailed listing of instructions.)

5.10.1 Memory Reference Instructions

Memory reference instructions have the following format:

![Memory Reference Bit Instructions Diagram]

Bits 0 through 2 contain the operation code of the instruction to be performed. Bit 3 indicates whether the memory reference is indirect. Bit 4 indicates whether the instruction is referencing the current page rather than page zero. Bits 5 through 11 (7 bits) specify an address. Using these seven bits, 200 octal (128 decimal) locations can be directly specified; the page bit increases accessible locations to 400 octal or 256 decimal. A list of the memory reference instructions and their codes is given at the end of this chapter.
In PAL8, a memory reference instruction must be followed by one or more spaces and tabs, an optional I and/or Z designation, and any valid expression.

When the character I appears in a statement between a memory reference instruction and an operand, the operand, is interpreted as the address (or location) containing the address of the operand to be used in the current instruction. Consider:

```
TAD 40
```

which is a direct address statement, where 40 is interpreted as the location on page zero containing the quantity to be added to the accumulator. References to locations on the current page and page zero may be done directly. For compatibility with older PDP-8 assemblers, the symbol Z is also accepted as a way of indicating a page zero reference, as follows:

```
TAD Z 40
```

This is an optional notation, not differing in effect from the previous example. Thus, if location 40 contains 0432, then 0432 is added to the accumulator when the code is executed. Now consider:

```
TAD I 40
```

which is an indirect address statement, where 40 is interpreted as the address containing the address of the quantity to be added to the accumulator. Thus, if location 40 contains 0432, and location 432 contains 0456, then 456 is added to the accumulator when the instruction is eventually executed.

NOTE

Because the letter I is used to indicate indirect addressing, it may not be redefined as a label or variable. Likewise the letter Z, which is sometimes used to indicate a page zero reference, may not be redefined.

5.10.2 Microinstructions

Microinstructions are divided into two classes: operate and Input/Output Transfer (IOT) microinstructions. Operate microinstructions are further subdivided into Group 1, Group 2, and Group 3.

NOTE

If an illegal combination of microinstructions is specified, the assembler will perform an inclusive OR between them, resulting in an unexpected operation. For example,

```
CLL SKP is interpreted as SPA
(7100)(7410) (7510)
```

5-25
5.10.2.1 **Operate Microinstructions** - Operate instructions are divided into three groups of micro instructions. Although it is possible to combine instructions within a group, it is not logically possible to combine instructions from different groups. Group 1 microinstructions perform clear, complement, rotate and increment operations on the Accumulator and Link registers, and are designated by the presence of a 0 in bit 3 of the machine instruction word.

![Binary diagram](image)

- **Rotate AC and L Right**
- **Rotate AC and L Left**
- **Rotate 1 Position if A 0, 2 Positions if A 1**
- (BSW if Bits 8, 9 are 0)

**Logical Sequence:**
1. CLA, CLL
2. CMA, CML
3. IAC
4. RAR, RAL, RTL, BSW

**Group 1 Operate Microinstruction Bit Assignments**

The following constants can be produced in the accumulator by a single instruction:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CLA</td>
</tr>
<tr>
<td>1</td>
<td>CLA IAC</td>
</tr>
<tr>
<td>2</td>
<td>CLA CLL CML RTL</td>
</tr>
<tr>
<td>3*</td>
<td>CLA CLL CML IAC RAL</td>
</tr>
<tr>
<td>4*</td>
<td>CLA CLL IAC RTL</td>
</tr>
<tr>
<td>6*</td>
<td>CLA CLL CML IAC RTL</td>
</tr>
<tr>
<td>100*</td>
<td>CLA IAC BSW</td>
</tr>
<tr>
<td>2000</td>
<td>CLA CLL CML RTR</td>
</tr>
<tr>
<td>3777</td>
<td>CLA CLL CMA RAR</td>
</tr>
<tr>
<td>4000</td>
<td>CLA CLL CML RAR</td>
</tr>
<tr>
<td>5777</td>
<td>CLA CLL CMA RTR</td>
</tr>
<tr>
<td>6000*</td>
<td>CLA CLL CML IAC RTR</td>
</tr>
<tr>
<td>7775</td>
<td>CLA CLL CMA RTL</td>
</tr>
<tr>
<td>7776</td>
<td>CLA CLL CMA RAL</td>
</tr>
<tr>
<td>7777</td>
<td>STA (=CLA CMA)</td>
</tr>
</tbody>
</table>

Instructions that are starred (*) must not be used on software to be transported onto old (non-omnibus) PDP-8 computers.

Group 2 microinstructions check the contents of the Accumulator and Link and, based on the check, continue to or skip the next instruction. Group 2 microinstructions are identified by the presence of a 1 in bit 3 and a 0 in bit 11 of the machine instruction word.
THE PAL8 ASSEMBLER

REVERSE SKIP SENSING OF BITS 5, 6, 7 IF SET

LOGICAL SEQUENCE:  1. (BIT 8 IS 0) - SMA OR SZA OR SNL
                    (BIT 8 IS 1) - SPA AND SNA AND SZL
                    2. CLA
                    3. OSR, HLT

Group 2 Operate Microinstruction Bit Assignments

Group 3 microinstructions reference the MQ register. They are differentiated from Group 2 instructions by the presence of a 1 in bits 3 and 11 of the machine instruction word.

Group 3 Operate Microinstruction Bit Assignments

Group 1 and Group 2 microinstructions cannot be combined since bit 3 determines either one or the other. Group 2 has two groups of skip instructions. They can be referred to as the OR group and the AND group.

<table>
<thead>
<tr>
<th>OR Group</th>
<th>AND Group</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMA</td>
<td>SPA</td>
</tr>
<tr>
<td>SZA</td>
<td>SNA</td>
</tr>
<tr>
<td>SNL</td>
<td>SZL</td>
</tr>
</tbody>
</table>

The OR group is designated by a 0 in bit 8, and the AND group by a 1 in bit 8. OR and AND group instructions cannot be combined with each other since bit 8 determines either one or the other.

If skip instructions are combined, it is important to note the conditions under which a skip may occur.

1. OR Group -- If these skips are combined in a statement, the inclusive OR of the conditions determines the skip. For example:

   SZA SNL

   The next statement is skipped if the Accumulator contains 0000 or the link is a 1 or both.
2. AND Group -- If the skips are combined in a statement, the logical AND of the conditions determines the skip. For example:

SNA SZL

The next statement is skipped only if the accumulator differs from 0000 and the link is 0.

5.10.2.2 Input/Output Transfer Microinstructions - Input/output transfer microinstructions initiate operation of peripheral equipment and effect an information transfer between the central processor and the input/output device(s).

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

DEVICE CODE SELECTION

CONTROL BITS

IOT Instruction Bit Assignments

5.10.3 Autoindexing

Consecutive address references are often necessary for obtaining data values when processing large amounts of data. Autoindex registers (locations 10-17 of each memory field) are used for this purpose. When one of the absolute locations from 10 through 17 (octal) is indirectly addressed, the contents of the location are incremented and then used as an indirect operand address. This allows consecutive memory locations to be addressed, using a minimum of instructions. It must be remembered that initially these locations (10 through 17 on page 0 of each field) must be set to one less than the first desired address. No incrementation takes place when locations 10 to 17 are addressed directly. For example, if the instruction to be executed next is in location 300 and the data to be referenced is on the page starting at location 5000, autoindex register 10 can be used to address the data as follows:

0276 1377 TAD (4777) /=5000-1
0277 3010 DCA 10 /SET UP AUTO INDEX
0300 1410 TAD I 10 /INCREMENT TO 5000
. . . /BEFORE USE OF AN INDIRECT
. . . /ADDRESS
. . .
0377 4777 (Literal Area of Page 1)

Note that the Data Field must be set to the field of the data being referenced, in this case, Field 0.

When the instruction in location 300 is executed, the contents of location 10 will be incremented to 5000, and the contents of location 5000 will be added to the contents of the accumulator. If the instruction TAD I 10 is executed again, the contents of location 5001 will be added to the accumulator, and so on.
5.11 PSEUDO-OPERATORS

Pseudo-operators are used to direct the assembler to perform certain processing operations or to interpret subsequent coding in a certain manner. Some pseudo-ops generate storage words in the object program, other pseudo-ops direct the assembler on how to proceed with the assembly. The pseudo-ops are defined in the permanent symbol table.

5.11.1 Indirect and Page Zero Addressing

The pseudo-operators I and Z specify the type of addressing to be performed. These are discussed in Section 5.10.1.

5.11.2 Extended Memory

5.11.2.1 FIELD Pseudo-Operator - The pseudo-op FIELD instructs the assembler to output a field setting so that it may assemble code into more than one memory field. This field setting is output during pass 2 in the object binary file and is recognized by the LOAD command which in turn causes all subsequent information to be loaded into the field specified by the expression.

Format:

FIELD ff

where:

ff is an integer, a previously defined symbol, or an expression whose terms have been defined. The value must be in the range 0 to 37.

This field setting is output to the binary file during pass 2 along with a default current location counter setting of 200. These settings are read by the LOAD command when it is executed to begin loading information into the new field.

The field setting is never remembered by the assembler except as the high-order digit of the Location Counter on the listing. A binary file produced without field settings will be loaded into field 0 when using the LOAD command.

A symbol in one field may be used to reference the same location in any other field. The field to which it refers is determined by the use of the CDF and CIF instructions. CDF and CIF instructions must be used prior to any instruction referencing a location outside the current field, as shown in the following example:

```
*200
TAD P301
CDF 00
CIF 10
JMS PRINT
CIF 10
JMP NEXT
P301, 301
FIELD 1
*200
```
When FIELD is used, the assembler follows the new FIELD setting with an origin at location 200. For this reason, to assemble code at location 400 in field 1, it would be necessary to write

```
FIELD1       /CORRECT EXAMPLE
*400
```

The following is incorrect and will not generate the desired code:

```
*400         /INCORRECT
FIELD1
```

Specifying the /O option to PAL8 inhibits the output of the default current location counter setting of 200 after a FIELD pseudo-op. This leaves the current location counter at its previous value.

5.11.2.2 Specifying Data and Instruction Fields - The PDP-8's memory addresses are specified by the contents of the Memory Reference Instruction modified by the Data Field and Instruction Field Registers. Direct addressing, specified by bit 3=0, causes reference to the address given in bits 5-11 in page 0 of the current field, if bit 4=0, or to the current page, if bit 4=1. Indirect addressing, specified by bit 3=1, causes reference to the indirect address contained in the location specified by bits 4-11, used as above. The indirect address for AND, TAD, ISZ, and DCA refers not to the current field, but to the field specified in the Data Field Register. The JMP and JMS instructions refer to locations in the field specified in the Instruction Field Register.

The Data Field Register and Instruction Field Register can be set under program control by means of the CIF and CDF instructions. The CIF instruction causes the Instruction Field Buffer to be set to the specified field. The CDF instruction causes the Data Field Register to be changed immediately. Other instructions allow the program to read, save, and restore the Data Field and Instruction Field Registers. Completion of execution of a JMP or JMS instruction causes the Instruction Field Register to be set to the contents of the Instruction Field Buffer. This procedure permits a program to choose a new field, then execute a jump from the current field to a chosen address in the new field.
The CDF and CIF instructions let you specify fields 0 to 37 as data and instruction fields. Entering the argument requires knowledge of the bit arrangement of these two instructions.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>IOT</th>
<th>Bit Arrangement</th>
</tr>
</thead>
<tbody>
<tr>
<td>CDF</td>
<td>6201</td>
<td>110 01a cdeb01</td>
</tr>
<tr>
<td>CIF</td>
<td>6202</td>
<td>110 01a cdeb10</td>
</tr>
</tbody>
</table>

Bits a cde b indicate the data or instruction field. (The positioning of the bits is eccentric as to maintain compatibility with older PDP-8 systems.)

To specify a field from 0 to 7, use bits c, d, and e only. The format of the instructions are:

CDF n0
CIF n0

where:

n0 is an octal number that PAL8 ORs with the instruction code
n is an octal digit from 0 to 7 (bits cde)

For example, this instruction

CDF 60

specifies field 6 by causing PAL8 to do the following OR.

<table>
<thead>
<tr>
<th>Instruction code</th>
<th>Bit Arrangement</th>
</tr>
</thead>
<tbody>
<tr>
<td>6201</td>
<td>110 010 000 001</td>
</tr>
<tr>
<td>60</td>
<td>000 000 110 000</td>
</tr>
<tr>
<td>6261</td>
<td>110 010 110 001</td>
</tr>
</tbody>
</table>

Keep in mind that to call for fields above field 7 (above 32K) with CDF and CIF, you must first load the KT8A Extended Mode Register with the LXM instruction (see the KT8A Memory Management Control User's Guide). For example, the following code deposits 7777 in field 12, location 1000.

LXM
CDF 24
TAD (7777)
DCA I (1000)

KT8A users must also ensure that their programs and device handlers do not contain the following combination of instruction steps.

CIF /Change instruction field
IOT /Any PDP8 IOT instruction
JMP I /The instruction that does the CIF

If you enable the KT8A and turn on the interrupts, the KT8A hardware will return to the wrong place on traps between the CIF and JMP I instructions.
THE PAL8 ASSEMBLER

To specify a field from 10 to 17, use bits cde and set bit b. The format of the instructions are:

CDF n4  
CIF n4

where:

n4 is an octal number that PAL8 ORs with the instruction code

n is an octal value from 0 to 7 (bits cde)

4 is an octal value indicating a field range of 10 to 17 (sets bit b)

For example, this instruction

CDF 64

indicates field 16.

To specify a field from 20 to 27, use bits cde and set bit a. The formats are:

CDF ln0  
CIF ln0

where:

ln0 is an octal number that PAL8 ORs with the instruction code

1 is an octal value indicating field range 20 to 27 (sets A)

n is a value from 0 to 7 (bits CDE)

For example, this instruction

CDF 160

indicates field 26.

To specify a field from 30 to 37, use bits CDE and set bit A and B. The formats are:

CDF ln4  
CIF ln4

where:

ln4 is an octal number that PAL8 ORs with the instruction code

1...4 are octal values indicating a field range of 30 to 37 (set bits A and B)

n is an octal digit in the range 0 to 7 (bits CDE)
For example, this instruction

CDF 164

specifies field 36

One way to avoid confusion with this unusual bit configuration is to define high fields with convenient mnemonics. For example:

F36=164
CDF F36

5.11.3 Resetting the Location Counter

The PAGE n pseudo-op resets the location counter to the first address of page n, where n is an integer, a defined symbol, or a symbolic expression whose terms have been defined previously and whose value is from 0 to 37 inclusive. If n is not specified, the location counter is reset to the beginning of the next page of memory. For example,

PAGE 2 sets the location counter to 00400
PAGE 6 sets the location counter to 01400

If the pseudo-op is used without an argument and the current location counter is at the first location of a page, the current location counter will not be reset. In the following example, the code TAD B is assembled into location 00400:

*377
JMP .-3
PAGE
TAD B

If several consecutive PAGE pseudo-ops are given, the first will cause the current location counter to be reset as specified. The rest of the PAGE pseudo-ops will be ignored.

5.11.4 Reserving Memory

ZBLOCK instructs the assembler to reserve n words of memory containing zeros, starting at the address indicated by the current location counter. It is of the form

ZBLOCK n

For example,

ZBLOCK 40

causes the assembler to reserve 40 (octal) words and store zeros in them. The n may be an expression. If n=0, no locations are reserved. A ZBLOCK statement may have a label.
5.11.5 Relocation Pseudo-Operator

It is sometimes desirable to assemble code at a given location and then move it at run time to another location for execution. This may result in errors unless the relocated code is assembled in such a way that the assembler assigns symbols their execution-time addresses rather than their load-time addresses. The RELOC pseudo-op establishes a virtual location counter without altering the actual location counter. The line

```
RELOC expr
```

sets the virtual location counter to expr. The line

```
RELOC
```

resets the virtual location counter back to the actual location counter value and terminates the relocation section.

For example, the following program causes the assembler to load the word at CODE into location 204, but assembles it as if it were loaded into 1371. The asterisks after the location values indicate that the virtual and the actual location counters differ for that line of code. Only the virtual location counter is listed. Do not use current page literals in code that is affected by a RELOC.

```
000200  0200  *200
000201  1205  TAD CHAR
         7402  HLT
         1367  RELOC 1367
001367* 1371  TAD CODE
001370* 7402  HLT
001371* 0000  CODE,  0
         0205  RELOC
000205  0000  CHAR,  0
```

5.11.6 Suppressing the Listing

The portions of the source program enclosed by XLIST pseudo-ops will not appear in the listing file; the assembled binary will be output, however.

Two XLIST pseudo-ops may be used to enclose the code to be suppressed in which case the first XLIST with no argument will suppress the listing, and the second will allow it again. XLIST may also be used with an expression as an argument. The listing will be inhibited if the expression is not equal to zero, or allowed if the expression is equal to zero. XLIST pseudo-ops never appear in the assembly listing.

5.11.7 Controlling Page Format

The EJECT pseudo-op causes the listing to skip to the top of the next page. A page eject is done automatically every 55 lines; EJECT is useful if more frequent paging is desired. If this pseudo-op is followed by a string of characters, the first 40 (decimal) characters of that string will be used as a new title at the top of each page of the listing.
5.11.8 Altering the Permanent Symbol Table

If your DECSYSTEM includes one or more optional devices that you want to program directly whose instruction sets are not defined in the permanent symbol table, then you would have to alter the symbol table to include the IOT instructions for these devices.

In another situation, programmed-defined symbols might require more space than is available in the symbol table. Again, the symbol table would have to be altered by removing all definitions not needed in the program being assembled. PAL8 has three pseudo-ops that can be used to alter the permanent symbol table. These pseudo-ops are recognized by the assembler only during pass 1. During either pass 2 or pass 3, the assembler ignores them, and they have no effect.

EXPUNGE deletes the entire permanent symbol table except pseudo-ops.

FIXTAB appends all currently defined symbols to the permanent symbol table. All symbols defined before the occurrence of FIXTAB are made part of the permanent symbol table for the current assembly.

To append the following instructions to the symbol table, you have to generate an ASCII file called SYM.PA. This file contains the following:

    CLSK=6131 /SKIP ON CLOCK INTERRUPT
    FIXTAB
    /SO THAT THIS WON'T BE
    /PRINTED IN THE SYMBOL TABLE

The ASCII file is then entered in the PAL8 input designation. By placing the definitions at the beginning of the source file, you can avoid loading an extra file. Each time the assembler is loaded, the PAL8's initial permanent symbol table is restored.

The third pseudo-op used to alter the permanent symbol table in PAL8 is FIXMRI. FIXMRI defines a memory reference instruction and is of the form:

    FIXMRI name=value

The letters FIXMRI must be followed by one space, the symbol for the instruction to be defined, an equal sign, and the value of the symbol. The symbol will be defined and stored in the symbol table as a memory reference instruction. The pseudo-op must be repeated for each memory reference instruction to be defined. For example,

    EXPUNGE
    FIXMRI TAD=1000
    FIXMRI DCA=3000
    CLA=7200
    FIXTAB

When the preceding program segment is read by the assembler during pass 1, all symbol definitions are deleted, and the three symbols listed are added to the permanent symbol table. Notice that CLA is not a memory reference instruction. This process can be performed to alter the assembler's symbol table so that it contains only the symbols used at a given installation or by a given program.
5.11.9 Conditional Assembly Pseudo-Operators

The IFDEF pseudo-op takes the form

```
IFDEF symbol <source code>
```

If the symbol indicated is previously defined, the code contained in the angle brackets is assembled; if the symbol is undefined, this code is ignored. Any number of statements or lines of code may be contained in the angle brackets. The format of the IFDEF statement requires a single space before and after the symbol.

Example:

```
IFDEF A <TAD A
    DCA B>
```

The IFNDEF pseudo-op is similar in form to IFDEF and is expressed as:

```
IFNDEF symbol <source code>
```

If the symbol indicated has not been previously defined, the source code in angle brackets is assembled. If the symbol is defined, the code in the angle brackets is ignored.

The IFZERO pseudo-op is of the form

```
IFZERO expression <source code>
```

If the evaluated expression is equal to zero, the code within the angle brackets is assembled; if the expression is nonzero, the code is ignored. Any number of statements or lines of code may be contained in the angle brackets. The expression may not contain any embedded spaces and must have a single space preceding and following it.

IFNZRO is similar in form to the IFZERO pseudo-op and is expressed as

```
IFNZRO expression <source code>
```

If the evaluated expression is not equal to zero, the source code within the angle brackets is assembled; if the expression is equal to zero, this code is ignored.

Pseudo-ops can be nested. For example,

```
IFDEF SYM <IFNZRO X2<...>>
```

The evaluation and subsequent inclusion or deletion of statements are done by evaluating the outermost pseudo-op first.

Conditional code that is not assembled can be deleted from the listing output by the /J option.

5.11.10 Radix Control

Numbers used in a source program are initially considered to be octal numbers. However, the program may change the radix interpretation by the use of the pseudo-operators DECIMAL and OCTAL. The DECIMAL pseudo-op interprets all following numbers as decimal until the occurrence of the pseudo-op OCTAL. The OCTAL pseudo-op resets the radix to octal.
5.11.11 Entering Text Strings

The TEXT pseudo-op allows a string of text characters to be entered as data and stored in 6-bit ASCII. The format required is the pseudo-op TEXT followed by one or more spaces or tabs, a delimiting character (must be a printing character), the string of text, and the same delimiting character. If the number of characters in a specified string is odd, the last word will contain zero in its right half. If the number of characters is even, a final word of zero will be appended. Either way, a final 6-bit character of zeros is generated providing a convenient "end-of-string" indication. Note that the /F option prevents the extra word of zero when the number of characters is even. Note also that six bits are sufficient to encode only the printing characters. For example:

    TAG, TEXT'123*'

The string would be stored as

    6162
    6352
    0000

The /F option inhibits the generation of the extra 6-bit zero word. Alternatively, the statement "*.l" may be used to eliminate the extra zero word (when the number of characters is even).

5.11.12 End-of-File Signal

PAUSE signals the assembler to stop processing the file being read. The current pass is not terminated, and processing continues with the next file. The PAUSE pseudo-op is present only for compatibility with paper tape assemblers, and its use is not recommended.

5.11.13 Use of DEVICE and FILENAME Pseudo-Operators

The pseudo-operators DEVICE and FILENAME may be used by calls to the User Service Routine (see Appendix C), or may be used for other purposes. They store 6-bit ASCII strings at the current location. The form for these pseudo-ops is

    DEVICE name
    FILENAME name.extension

The name used with DEVICE can be from 1 to 4 alphanumeric characters. These are trimmed to 6-bit ASCII and packed into two words, filled-in with zeros on the right if necessary. With FILENAME (FILENA is also acceptable), the name (or name.extension) may be from 1 to 6 alphanumeric characters and the optional extension may be 1 or 2 characters. The characters are trimmed to 6-bit ASCII and packed two to a word. Three words are allocated for the file name, filled with zeros on the right if fewer than 6 characters are specified, followed by one word for the extension. The example

    L, FILENAME ABC.DA
is equivalent to the following coding:

L, 0102
0300
0000
0401

The symbols DEVICE and FILENAME may not be used as labels since they are predefined pseudo-ops.

5.12 LINK GENERATION AND STORAGE

In addition to handling symbolic addressing on the current page of memory, PAL8 automatically generates "links" for off-page references. If a direct memory reference is made to an address not on the page where an instruction is located or on page 0, the assembler sets the indirect bit (bit 3), and an indirect address literal (a "link") will be stored on the current memory page. If the specified reference is already an indirect one, the error diagnostic II (Illegal Indirect) will be generated. In the following example,

*2117
A,
CLA
.
.
.
*2600
JMP A

the assembler will recognize that the register labeled A is not on the current page and will generate a link to it as follows:

1. In location 2600 the assembler will place the word 5777 (equivalent to JMP I 2777).

2. In address 2777 (the last available location on the current page), the assembler will place the word 2117 (the actual address of A).

In the listing, the octal code for the instruction will be followed by a single quote (') to indicate that a link was generated.

Although the assembler will recognize and generate an indirect address linkage when necessary, the program may indicate an explicit indirect address by the pseudo-op I, for example:

*2117
A,
CLA
.
.
.
*2600
JMP I (A)
THE PAL8 ASSEMBLER

The assembler cannot generate a link for an instruction that is already specified as being an indirect reference, since the computer supports no such instruction format. For the example shown below, the assembler will print the error message II (Illegal Indirect).

```
$2117
A,
CLA
.
.
.
$2600
JMP I A
```

NOTE

The option /E makes link generation a condition that produces the LG (Link Generated) error message.

The above coding will fail because A is not defined on the page where JMP I A is attempted, and the indirect bit is already set.

Literals and links are stored on each page starting at page address 177 (relative) and extending toward page address 0 (relative). Whenever the origin is then set to another page, the literal area for the current page is output. There is room for 160 (octal) literals and links on page zero and 100 (octal) literals on each other page of memory. Literals and links are stored only as far down as the highest instruction on the page. Further attempts to define literals will result in a PE (Page Exceeded) or ZE (page zero Exceeded) error message.

5.13 TERMINATING ASSEMBLY

PAL8 will terminate assembly and return to the monitor under any of the following conditions:

1. Normal exit: The end of the source program was reached on pass 2 (or pass 3 if a listing is being generated).

2. Fatal error: One of the following error conditions was found and flagged (Section 5.14):

   BE  DE  DF  PH  SE

3. CTRL/C: If typed, control returns to the monitor. Any partial output files are deleted.

5.14 DIAGNOSTIC ERROR MESSAGES

PAL8 will detect and flag error conditions and display error messages both on the console terminal and in the program listing. The format of the error message is:

code address
THE PAL8 ASSEMBLER

where:

code is a 2-letter code that specifies the type of error.

address is either the absolute octal address where the error occurred or the address of the error relative to the last label (if there was one) on the current page.

For example, the instruction sequence:

BEG, TAD LBL
%TAD LBL

would produce the error message

IC BEG+0001

since % is an illegal character because of its placement.

In the listing, error messages are output as 2-character messages on the line just prior to the line in which the error occurred. Table 5-3 lists the PAL8 error codes. Fatal errors cause PAL8 to terminate the assembly immediately (deleting any output files produced so far) and return to the monitor.

Table 5-3
PAL8 Diagnostic Error Codes

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>BE</td>
<td>A PAL8 internal table has overflowed. This situation can usually be corrected by decreasing the level of literal nesting or the number of current page literals used prior to this point on the page. Fatal error: assembly cannot continue.</td>
</tr>
<tr>
<td>CF</td>
<td>Chain to CREF error. CREF.SV was not found on SYS.</td>
</tr>
<tr>
<td>DE</td>
<td>Device error. An error was detected when trying to read or write a device. Fatal error: assembly cannot continue.</td>
</tr>
<tr>
<td>DF</td>
<td>Device full. Fatal error: assembly cannot continue.</td>
</tr>
<tr>
<td>IC</td>
<td>Illegal character. The character is ignored and the assembly is continued.</td>
</tr>
<tr>
<td>ID</td>
<td>Illegal redefinition of a symbol. An attempt was made to give a previous symbol a new value by means other than the equal sign. The symbol is not redefined.</td>
</tr>
<tr>
<td>IE</td>
<td>Illegal equals. An attempt was made to equate a variable to an expression containing an undefined term. The variable remains undefined.</td>
</tr>
<tr>
<td>II</td>
<td>Illegal indirect. An off-page reference was made; a link could not be generated because the indirect bit was already specified.</td>
</tr>
</tbody>
</table>

(continued on next page)
### Table 5-3 (Cont.)
PAL8 Diagnostic Error Codes

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>IP</td>
<td>Illegal pseudo-op. A pseudo-op was used in the wrong context or with incorrect syntax.</td>
</tr>
<tr>
<td>IZ</td>
<td>Illegal page zero reference. The pseudo-op Z was found in an instruction which did not refer to page zero. The Z is ignored.</td>
</tr>
<tr>
<td>LD</td>
<td>The /L or /G options have been specified but the Absolute Loader system program is not present.</td>
</tr>
<tr>
<td>LG</td>
<td>Link generated. This code is displayed only if the /E option was specified to PAL8.</td>
</tr>
<tr>
<td>PE</td>
<td>Current nonzero page exceeded. An attempt was made to</td>
</tr>
<tr>
<td></td>
<td>1. Override a literal with an instruction.</td>
</tr>
<tr>
<td></td>
<td>2. Override an instruction with a literal.</td>
</tr>
<tr>
<td></td>
<td>3. Use more literals than the assembler allows on that page.</td>
</tr>
<tr>
<td></td>
<td>This can be corrected by decreasing either the number of literal's on the page or the number of instructions on the page.</td>
</tr>
<tr>
<td>PH</td>
<td>A conditional assembly bracket is still in effect at the end of the input stream. This is caused by nonmatching &lt; and &gt; characters in the source file.</td>
</tr>
<tr>
<td>RD</td>
<td>Redefinition. A permanent symbol has been defined with =. The new and old definitions do not match. The redefinition is allowed.</td>
</tr>
<tr>
<td>SE</td>
<td>Symbol table exceeded. Too many symbols have been defined for the amount of memory available. Fatal error: assembly cannot continue.</td>
</tr>
<tr>
<td>UO</td>
<td>Undefined origin. An undefined symbol has occurred in an origin statement.</td>
</tr>
<tr>
<td>US</td>
<td>Undefined symbol. A symbol has been processed during pass 2 that was not defined before the end of pass 1.</td>
</tr>
<tr>
<td>ZE</td>
<td>Page 0 exceeded. This is the same as PE except occurs in page 0.</td>
</tr>
</tbody>
</table>

### 5.15 PAL8 PERMANENT SYMBOL TABLE

The following mnemonics represent the central processor's instruction set found in the permanent symbol table within the PAL8 Assembler. For additional information on these instructions, refer to the DECstation 78 User's Guide.
## THE PAL8 ASSEMBLER

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Code</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>0000</td>
<td>Logical AND</td>
</tr>
<tr>
<td>TAD</td>
<td>1000</td>
<td>Two's complement add</td>
</tr>
<tr>
<td>ISZ</td>
<td>2000</td>
<td>Increment and skip if zero</td>
</tr>
<tr>
<td>DCA</td>
<td>3000</td>
<td>Deposit and clear AC</td>
</tr>
<tr>
<td>JMS</td>
<td>4000</td>
<td>Jump to subroutine</td>
</tr>
<tr>
<td>JMP</td>
<td>5000</td>
<td>Jump</td>
</tr>
<tr>
<td>IOT</td>
<td>6000</td>
<td>In/Out transfer</td>
</tr>
<tr>
<td>OPR</td>
<td>7000</td>
<td>Operate</td>
</tr>
</tbody>
</table>

### Group 1 Operate Microinstructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Code</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOP</td>
<td>7000</td>
<td>No operation</td>
</tr>
<tr>
<td>IAC</td>
<td>7001</td>
<td>Increment AC</td>
</tr>
<tr>
<td>BSW</td>
<td>7002</td>
<td>Byte swap</td>
</tr>
<tr>
<td>RAL</td>
<td>7004</td>
<td>Rotate AC and Link left one</td>
</tr>
<tr>
<td>RTL</td>
<td>7006</td>
<td>Rotate AC and Link left two</td>
</tr>
<tr>
<td>RAR</td>
<td>7010</td>
<td>Rotate AC and Link right one</td>
</tr>
<tr>
<td>RTR</td>
<td>7012</td>
<td>Rotate AC and Link right two</td>
</tr>
<tr>
<td>CML</td>
<td>7020</td>
<td>Complement the link</td>
</tr>
<tr>
<td>CMA</td>
<td>7040</td>
<td>Complement the AC</td>
</tr>
<tr>
<td>CLL</td>
<td>7100</td>
<td>Clear Link</td>
</tr>
<tr>
<td>CLA</td>
<td>7200</td>
<td>Clear AC</td>
</tr>
</tbody>
</table>

### Group 2 Operate Microinstructions (1 cycle)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Code</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>HLT</td>
<td>7402</td>
<td>Halts the computer</td>
</tr>
<tr>
<td>SKP</td>
<td>7410</td>
<td>Skip unconditionally</td>
</tr>
<tr>
<td>SNL</td>
<td>7420</td>
<td>Skip on nonzero Link</td>
</tr>
<tr>
<td>SLZ</td>
<td>7430</td>
<td>Skip on zero Link</td>
</tr>
<tr>
<td>SZA</td>
<td>7440</td>
<td>Skip on zero AC</td>
</tr>
<tr>
<td>SNA</td>
<td>7450</td>
<td>Skip on nonzero AC</td>
</tr>
<tr>
<td>SMA</td>
<td>7500</td>
<td>Skip on minus AC</td>
</tr>
<tr>
<td>SPA</td>
<td>7510</td>
<td>Skip on positive AC (zero is positive)</td>
</tr>
</tbody>
</table>

### Group 3 Operate Microinstructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Code</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>MQL</td>
<td>7421</td>
<td>Load MQ, clear AC</td>
</tr>
<tr>
<td>MQA</td>
<td>7501</td>
<td>MQ OR into AC</td>
</tr>
<tr>
<td>SWP</td>
<td>7521</td>
<td>Swap AC and MQ</td>
</tr>
</tbody>
</table>

### Combined Operate Microinstructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Code</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>CIA</td>
<td>7041</td>
<td>Complement and increment AC</td>
</tr>
<tr>
<td>STL</td>
<td>7120</td>
<td>Set Link to 1</td>
</tr>
<tr>
<td>GLK</td>
<td>7204</td>
<td>Get Link (put Link in AC, bit 11)</td>
</tr>
<tr>
<td>STA</td>
<td>7240</td>
<td>SET AC to -1 (7777)</td>
</tr>
</tbody>
</table>

### Internal IOT Microinstructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Code</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>SKON</td>
<td>6000</td>
<td>Skip with interrupts on and turn them off</td>
</tr>
<tr>
<td>ION</td>
<td>6001</td>
<td>Turn interrupt facility on</td>
</tr>
<tr>
<td>IOF</td>
<td>6002</td>
<td>Turn interrupt facility off</td>
</tr>
<tr>
<td>GTF</td>
<td>6004</td>
<td>Get flags</td>
</tr>
<tr>
<td>RTF</td>
<td>6005</td>
<td>Restore flag, ION</td>
</tr>
<tr>
<td>CAF</td>
<td>6007</td>
<td>Clear all flags</td>
</tr>
</tbody>
</table>

5-42
Memory Extension IOT Instructions

CDF 62n1 Change to Data Field n (n=00 to 07)
     62n5 Change to Data Field n (n=10 to 17)
     63n1 Change to Data Field n (n=20 to 27)
     63n5 Change to Data Field n (n=30 to 37)

CIF * 62n2 Change to Instruction Field n (n=00 to 07)
       62n6 Change to Instruction Field n (n=10 to 17)
       62n2 Change to Instruction Field n (n=20 to 27)
       63n6 Change to Instruction Field n (n=30 to 37)

CDI 62n3 Change to Data and Instruction Fields n
     (n=00 to 07)
     62n7 Change to Data and Instruction Fields n
     (n=10 to 17)
     63n3 Change to Data and Instruction Fields n
     (n=20 to 27)
     63n7 Change to Data and Instruction Fields n
     (n=30 to 37)

CDF CIF 62n3 Combine CDF and CIF

Bit assignments for n are as follows.

Field Number (0-37) = abcde

\[ \begin{array}{c|c|c|c|c}
\text{bit 0} & \text{bit 11} \\
\hline
\text{CDF} & 110 & 01a & cde & b01 \\
\text{CIF} & 110 & 01a & cde & b10 \\
\text{CDF CIF} & 110 & 01a & cde & b11 \\
\end{array} \]

RDF 6214 Read (OR) the Data Field into bits 6-8 of AC

RIF 6224 Read (OR) the Instruction Field into bits 6-8 of AC

RSB 6234 Read Instruction Save Field into bits 7-8 and Data Save Field into bits 10-11 of AC

RMF 6244 Restore memory fields to state prior to last interrupt by loading the Data Save Field into DF register and the Instruction Save Field into the IB register and inhibiting interrupts. At the next JMP or JMS, IB is loaded into the IF register and the interrupt inhibit is removed.

* Executed at next JMP or JMS instruction.
### Keyboard (1 cycle)

<table>
<thead>
<tr>
<th>Code</th>
<th>Decimal</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>KCF</td>
<td>6030</td>
<td>Clear keyboard flag</td>
</tr>
<tr>
<td>KSF</td>
<td>6031</td>
<td>Skip on keyboard flag</td>
</tr>
<tr>
<td>KCC</td>
<td>6032</td>
<td>Clear keyboard flag and AC</td>
</tr>
<tr>
<td>KRS</td>
<td>6034</td>
<td>OR keyboard buffer into AC</td>
</tr>
<tr>
<td>KIE</td>
<td>6035</td>
<td>Set/clear interrupt enable</td>
</tr>
<tr>
<td>KRB</td>
<td>6036</td>
<td>Clear AC, read keyboard buffer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Clear keyboard flag</td>
</tr>
</tbody>
</table>

### Terminal (1 cycle)

<table>
<thead>
<tr>
<th>Code</th>
<th>Decimal</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TSF</td>
<td>6041</td>
<td>Skip on terminal flag</td>
</tr>
<tr>
<td>TCF</td>
<td>6042</td>
<td>Clear terminal flag</td>
</tr>
<tr>
<td>TSK</td>
<td>6045</td>
<td>Skip on keyboard or terminal flag</td>
</tr>
<tr>
<td>TLS</td>
<td>6046</td>
<td>Load terminal, display character, and clear terminal flag</td>
</tr>
</tbody>
</table>
CHAPTER 6

BASIC

6.1 INTRODUCTION

OS/78 BASIC* is an interactive programming language used in scientific and business environments to solve mathematical problems with a minimum of programming effort. It also is used by educators and students as a problem-solving tool and as an aid to learning through programmed instruction and simulation.

In many respects the BASIC language is similar to other programming languages (such as FORTRAN IV), but BASIC is aimed at facilitating communication between the user and the computer. BASIC simply requires that you type in the computational procedure as a series of numbered statements, making use of common English words and familiar mathematical notations. Because of the small number of commands necessary and its easy application in solving problems, BASIC is an easy computer language to learn. With experience, the advanced techniques available can be added in the language to perform more intricate manipulations or to express a problem more efficiently and concisely.

6.2 MAJOR COMPONENTS OF OS/78 BASIC

The BASIC subsystem has four major components.

1. BASIC Editor (BASIC.SV)
2. Compiler (BCOMP.SV)
3. Loader (BLOAD.SV)
4. BASIC Run Time System (BRTS.SV, BASIC.AF, BASIC.SF, BASIC.FF)

The BASIC editor is used to create and edit program source files. During this process, the editor creates a file called BASIC.WS containing the current program.

Once the program has been prepared for execution, entering a RUN command causes the editor to chain to the BASIC compiler. The compiler converts the statements in BASIC.WS into relocatable binary instructions.

Following compilation, the BASIC loader is automatically requested. The loader converts the relocatable binary data output by the compiler into executable form and loads the result into memory.

The BASIC loader then chains to the BASIC Run Time System (BRTS) which executes the program. The modules BASIC.AF, BASIC.FF, and BASIC.SF are overlays to BRTS.SV.

6.3 BASIC INSTRUCTION REPETOIRE

BASIC instructions and commands can be grouped in three categories as follows:

1. BASIC Editor commands that allow you to
   a. Create or modify a program,
   b. Execute a program,
   c. Retrieve a program from diskette, and
   d. Save a program.

*BASIC is a registered trademark of the trustees of Dartmouth College.
2. BASIC statements, comprising the BASIC language, that are the building blocks used to create and structure BASIC programs.
3. BASIC Functions, represented by subroutines, that are built into BASIC primarily to facilitate problem solving activities.

6.4 CALLING BASIC
To enter the BASIC subsystem, type

```
  BASIC
```

in response to the Monitor dot. This command invokes the BASIC editor.

6.5 BASIC EDITOR COMMANDS

6.5.1 Using the BASIC Editor
The BASIC editor incorporates all the tools and capabilities necessary to create, correct, modify, execute, save and retrieve BASIC programs. After calling BASIC, the BASIC editor responds with the displayed query

```
NEW OR OLD--
```

Editing is now continued in one of two ways:

1. Typing NEW with a file name instructs the system to initiate the creation of a new file.
2. Typing OLD with a file name instructs the system to retrieve an old file containing a previously-generated program.

Figure 6-1 summarizes user actions implemented by the BASIC editor. This figure shows two arbitrarily selected file names (MAKER/ALTER). File names may contain no more than six alphanumeric symbols.

The left side of Figure 6-1 shows the types and patterns of activities that you ordinarily pursue after typing the NEW command. The right side of the illustration shows activities frequently carried out after typing the OLD command. Note that most of the BASIC editor commands appear on both sides of the figure. Only the sequence in which they are used differs.

6.5.2 BASIC Editor Commands
This section summarizes the BASIC editor commands. Letters that are not required in naming the command are shown as lower case in the Command/Parameter description. For example, only NE is required to be recognized by the BASIC editor as the NEW command.

NOTE
The RETURN key must be pressed following each BASIC editor command.

6.5.2.1 NEW Command — The NEW command clears the memory workspace and specifies the name of the program that is to be input.

<table>
<thead>
<tr>
<th>Command</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEw</td>
<td>file[.ex]</td>
</tr>
<tr>
<td>file.ex</td>
<td>is the new file name and extension of the program about to be typed in. If the extension is omitted, the editor assigns .BA.</td>
</tr>
</tbody>
</table>
BASIC

NEW OR OLD

NEW

FILE NAME

MAKER

READY

CREATE

Your Program
Using BASIC

STATEMENTS
FUNCTIONS

MODIFY

Your Program Using

Correction Keys

DELETE
CTRL/U
RETURN
RESEQ

REVIEW

Your Program Using

LIST
LISTNH
CTRL/S
CTRL/Q Keys

PRESERVE

Your Program Using

SAVE

EXECUTE

Your Program Using

RUN
RUNNH

ERROR MESSAGE

END OF BASIC SESSION

1. *Keyboard Monitor Command
2. *Shaded areas indicate user action at the console
An alternate method is to type NEW without a file name, followed by the RETURN key. BASIC displays

    FILE NAME --

in response to which the file name and extension is typed.

For example, to clear the workspace and name the new program "TEST.BA", type

    NEW TEST

or

    NE TEST.BA

6.5.2.2 OLD Command — The OLD command clears the memory workspace, and causes the editor to find a program on a diskette and place it into the workspace.

<table>
<thead>
<tr>
<th>Command</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>OLD</td>
<td>dev:file.[ex]</td>
</tr>
<tr>
<td>dev:file.ex</td>
<td>is the device, file name, and extension of the program on the disk. If the extension is omitted, BASIC assumes &quot;BA&quot;.</td>
</tr>
</tbody>
</table>

Another method is to type OLD without a file name. BASIC displays

    FILE NAME --

in response to which the device, file name, and extension is typed. When no device is specified, the BASIC editor defaults to DSK (usually = SYS).

For example, to bring TEST.BA into the workspace from RXA1, type

    OLD RXA1:TEST.BA

or

    OL RXA1:TEST

6.5.2.3 LIST/LISTNH Commands — The LIST command displays the current program along with a header line, containing the program name, date, and the revision number of BASIC. The date is displayed only if the current date has been entered into the system.

<table>
<thead>
<tr>
<th>Command</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>List</td>
<td>[n]</td>
</tr>
</tbody>
</table>

If n is omitted all program statements in the workspace are displayed. When n is specified, line n and all subsequent lines are displayed. Type CTRL/O to terminate a listing.
For example, typing LIST (or LI) displays the program PROG.BA:

```
LIST
PROG BA 5A 26-JUL-77
10 FOR A=1 TO 5
20 PRINT A
30 NEXT A
40 END
READY
```

Typing LI 30 displays line 30 and all subsequent lines:

```
LI 30
PROG BA 5A 26-JUL-77
30 NEXT A
40 END
READY
```

The LISTNH command also displays the program statements in the workspace but without the header.

<table>
<thead>
<tr>
<th>Command</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>LISTNH</td>
<td>[n]</td>
</tr>
</tbody>
</table>

n has the same effects as specified for the LIST command, but does not display the header.

6.5.2.4 SAVE Command — The SAVE command writes the program in the workspace onto the diskette as a permanently saved file. Do not confuse this command with the Monitor SAVE command.

<table>
<thead>
<tr>
<th>Command</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAVE</td>
<td>[dev:file.ex]</td>
</tr>
</tbody>
</table>

dev is the device on which you want to store your program.

file.ex is the file name and extension that the program will have on the diskette. If both are left out, BASIC will use the current file name and extension of the program in the workspace. If only the extension is omitted BASIC assigns .BA. If DEV: is omitted, BASIC assumes the device is DSK.

In the following example, the program “TEST.BA” is in the workspace. To store it on RXA1 under the same file name and extension, type

```
SAVE RXA1:TEST.BA
```

or

```
SA RXA1:TEST
```
6.5.2.5 RUN/RUNNH Commands — The RUN command executes the program in the workspace. Note that this command differs from the monitor RUN command.

<table>
<thead>
<tr>
<th>Command</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>RUN</td>
<td>(none)</td>
</tr>
</tbody>
</table>

Prior to program execution, this command displays a heading consisting of the file name and extension, BASIC version number and system date, assuming that the current date has been entered in the system. When program execution is completed, BASIC displays READY.

The following example shows the program PROG.BA in the workspace displaying the result of a calculation:

```
RUN
PROG BA  5A  26-JAN-76
3.179
```

The RUNNH command executes the program in the workspace, but does not display the header line.

<table>
<thead>
<tr>
<th>Command</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>RUNNH</td>
<td>(none)</td>
</tr>
</tbody>
</table>

Thus, the only difference between the RUNNH and RUN commands is that RUN prints the header and RUNNH does not display the header.

6.5.2.6 NAME Command — The NAME command allows the user to rename the program in the workspace.

<table>
<thead>
<tr>
<th>Command</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAME</td>
<td>newfil[.ex]</td>
</tr>
</tbody>
</table>

newfil.ex is the new file name and extension of the program in the workspace. If the extension is omitted, the editor assigns .BA.

To change the name of the program in the workspace to PROG.BA, type

```
NAME PROG.BA
```

or

```
NA PROG
```

6.5.2.7 SCRATCH Command — The SCRATCH command erases all statements from the workspace, that is, it clears the workspace.

<table>
<thead>
<tr>
<th>Command</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCRATCH</td>
<td>(none)</td>
</tr>
</tbody>
</table>

6.5.2.8 BYE Command — The BYE command exits from BASIC and returns control to the Monitor from BASIC.
Command Parameters

BYE (none)

Typing BYE before SAVEing a newly created program will delete the program. Whenever the BASIC editor is waiting for user input, typing CTRL/C performs the same function as BYE.

6.5.3 BASIC Control Keys
This section describes the control keys that are used to correct errors, eliminate and substitute program links, and control program listings.

6.5.3.1 Correcting Typing and Format Errors (DELETE, CTRL/U) — Errors made while typing programs at the terminal are easily corrected. Pressing the DELETE key causes deletion of the last character typed. One character is deleted each time the key is pressed.

Sometimes it is easier to delete a line being typed and retype the line rather than attempt a correction using a series of DELETEds. Typing CTRL/U will delete the entire line currently being worked on and echoe “DELETED” and a carriage return-line feed. Use of the CTRL/U key is equivalent to typing DELETEds back to the beginning of the line.

6.5.3.2 Eliminating and Substituting Program Lines (RETURN) — To delete a program line that has already been entered into the computer, simply type the line number and then press the RETURN key. Both the line number and the statement(s) are removed from the program.

Change individual lines by simply retyping them in again. Whenever a line is entered, it replaces any existing line having the same line number. New lines may be inserted anywhere in the program by giving them unique line numbers.

6.5.3.3 Interrupting Program Execution — Program execution may be terminated by typing CTRL/C. BASIC responds by displaying the READY message allowing you to correct or add statements to the program.

NOTE
BASIC responds to CTRL/C with a “READY” message only if you have already given the RUN command. Typing CTRL/C when the BASIC Editor is operational causes a return to the Monitor.

6.5.3.4 Controlling Program Listings at the Terminal Console (CTRL/S, CTRL/C and CTRL/O) — For programs exceeding a single display frame (24 lines) the user may wish to stop the scrolling effect that occurs after typing the LIST/LISTNH command. Three sets of control keys are provided to do this. They are as follows:

1. CTRL/S keys. Simultaneously pressing these keys suspends listing (scrolling) of the program. However it leaves the program in a state where you may resume listing.
2. CTRL/Q keys. Simultaneously pressing these keys (after having suspended scrolling with CTRL/S), resumes the listing process.
3. CTRL/O keys. Simultaneously pressing these keys aborts the listing process and causes the BASIC editor to display READY.

6.5.4 Resequencing Programs (RESEQ)
If a program is extensively modified, you may find that some portions of the program have line numbers spaced so closely together that they do not permit any further addition of statements. Reranging the lines in the program to provide a practical increment between line numbers can be done by using the RESEQ program. Note that the RESEQ program modifies the line numbers in GOSUB and IF-THEN statements to agree with the new line numbers assigned to program statements by RESEQ. Line lengths must not exceed 80 characters and programs may not exceed 350 lines.
Typically, the program would be used as follows:

SAVE DSK: SAMPLE BA User saves program SAMPLE which requires renumbering.

READY BASIC is ready for next command.

OLD DSK: RESEQ User calls for program RESEQ.

READY BASIC is ready for next command.

RUNNH User runs RESEQ program.

FILE: DSK: SAMPLE.BA Program asks for filename. User responds with device, name, and extension of program to be renumbered.

START, STEP: 100, 10 Program asks for a starting line number (START) and for the increment between line numbers (STEP). User requests that SAMPLE start with line number 100 and each line be incremented by 10.

READY Renumbering is accomplished. BASIC ready for next command.

OLD DSK: SAMPLE.BA User calls back his program.

READY BASIC ready for next command.

LISTNH User gets listing of program SAMPLE for further modification.

### 6.6 DATA FORMATS ACCEPTABLE TO BASIC

#### 6.6.1 Numeric Information

#### 6.6.2 Numbers

Numbers are expressed in decimal or E (exponential) format. Examples of numbers in both categories are as follows:

<table>
<thead>
<tr>
<th>Decimal</th>
<th>E Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10.23E27</td>
</tr>
<tr>
<td>7</td>
<td>6.21E+27</td>
</tr>
<tr>
<td>+69</td>
<td>-7.232E6</td>
</tr>
<tr>
<td>-52</td>
<td>2.211E-3</td>
</tr>
<tr>
<td>-3.9265</td>
<td>-2.2114E-4</td>
</tr>
<tr>
<td>0.123</td>
<td></td>
</tr>
<tr>
<td>-0.769</td>
<td></td>
</tr>
</tbody>
</table>

In the decimal format, the decimal point is optional for integers. That is, BASIC assumes a decimal point after the rightmost digit of an integer. In E type format, BASIC assumes a positive exponent when no plus (+) or minus (-) sign follows the E. Substitute the words "times ten to the power of" for the letter E when reading E type format numbers.
Numeric data may be input in either format. Results of computations with an absolute value outside the range +.000001<N<999999 are always output in E type format. BASIC handles six significant digits as shown by the following examples:

<table>
<thead>
<tr>
<th>Value Typed In</th>
<th>Value Output by BASIC</th>
</tr>
</thead>
<tbody>
<tr>
<td>.01</td>
<td>0.0099999</td>
</tr>
<tr>
<td>.0099</td>
<td>0.0099</td>
</tr>
<tr>
<td>999999</td>
<td>999999</td>
</tr>
<tr>
<td>100000</td>
<td>1.000000E+007</td>
</tr>
<tr>
<td>.0000009</td>
<td>.899999E−006</td>
</tr>
</tbody>
</table>

Note in the above examples that the nature of the binary numbering system does not permit a completely accurate representation of certain decimal numbers. Hence they are output as close to the true value as the internal logic of the computer permits.

BASIC automatically suppresses the printing of leading and trailing zeros in integer numbers and all but one leading zero in decimal numbers. As can be seen from the preceding examples, BASIC formats all exponential numbers in the form:

$$\text{sign \ xxxxxxxE(+or−)n}$$

where x represents the number carried to six decimal places, E stands for “times 10 to the power of”, and n represents the exponential value.

For example,

- .347021E+009 is equal to 347.021,000, and
- .726000E−003 is equal to 0.000726

All numbers used in BASIC must have an absolute value (N) in the range:

$$10^{-616}<N<10^{+616}$$

### 6.6.3 Simple Variables

A simple variable in BASIC is an algebraic symbol representing a number, and is formed by a single letter or a letter followed by a digit. For example,

**Acceptable Variables**

- J
- B3

**Unacceptable Variables**

- 2C (a digit cannot begin a variable)
- AB (two or more letters cannot form a variable)

Values may be assigned to variables either by indicating the values in LET statements, or by inputting the values as data via INPUT and DATA statements.
Examples:

10 LET I=53721
20 LET B3=456.9
30 LET X=20E9
40 INPUT Q
50 DATA 5.6,7

6.6.4 Subscripted Variables

In addition to simple variables, BASIC accepts another class of variables called subscripted variables. Subscripted variables provide additional computing capabilities for handling arrays, lists, tables, matrices, or any set of related variables. Variables are allowed one or two subscripts. A single letter or a letter followed by a digit forms the name of the variable. The subscript is formed by one or two integers enclosed in parentheses and separated by commas. Up to 31 arrays are possible in any program, subject only to the amount of memory available for data storage. For example, an array might be described as A(J) where J goes from 1 to 5, as follows:

A(1),A(2),A(3),A(4),A(5)

This allows reference to be made to each of the five elements in the array A. A two-dimensional array A(J,K) can be defined in a similar manner, but the subscripted variable A must always have the same number of subscripts (that is, A(J) and A(J,K) cannot be used in the same program). It is possible, however, to use the same variable name as both a subscripted and an unsubscripted variable. Both A and A(J) are valid variable names and can be used in the same program. For more information on arrays and the use of subscripts, see the description of the DIM statement (Section 6.7.10.1).

6.6.5 Arithmetic Operations

6.6.5.1 Operators — BASIC performs addition, subtraction, multiplication, division and exponentiation, as well as more complicated operations (explained in detail later in the manual). The five operators used in writing most formulas are:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Operator</th>
<th>Meaning</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Addition</td>
<td>A+B</td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>Subtraction</td>
<td>A-B</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
<td>A*B</td>
<td></td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
<td>A/B</td>
<td></td>
</tr>
<tr>
<td>^ (or **)</td>
<td>Exponentiation</td>
<td>A^B or (A**B)</td>
<td></td>
</tr>
</tbody>
</table>

(Raises A to the B Power)

6.6.5.2 Priority — In any given mathematical formula, BASIC performs the arithmetic operations in the following order:

1. Parentheses receive top priority. Any expression within parentheses is evaluated before an unparenthesized expression.
2. In the absence of parentheses, the order of priority is:
   a. Exponentiation
   b. Multiplication and Division (of equal priority)
   c. Addition and Subtraction (of equal priority)
3. If either sequence 1 or 2 above does not clearly designate the order of priority, then the evaluation of expressions proceeds from left to right.
The expression \( A+B-C \) is evaluated from left to right as follows:

1. \( A+B \) = step 1
2. (result of step 1)-C = answer

The expression \( A/B*C \) is also evaluated from left to right since multiplication and division are of equal priority:

1. \( A/B \) = step 1
2. (result of step 1)*C = answer

6.6.5.3 Parentheses — Parentheses may be used to change the order or priority because expressions within parenthesis are always evaluated first. Thus, by enclosing expressions appropriately, the order of evaluation can be controlled. Parentheses may be nested, that is, enclosed by one or more sets of parentheses. In this case, the expression within the innermost parentheses is evaluated first, and then the next innermost, and so on, until all have been evaluated. Consider the following example:

\[
A = 7*((B^2+4)/X)
\]

The order of priority is:

1. \( B^2 \) = step 1
2. (result of step 1)+4 = step 2
3. (result of step 2)/X = step 3
4. (result of step 3)*7 = answer

Parentheses also prevent any confusion or doubt as to how the expression is evaluated. For example,

\[
A*B^2/7+B/C+D^2
\]

\[
((A*B^2)/7)+((B/C)+D^2)
\]

Both of these formulas will be executed in the same way. However, the second example may be easier to understand. Spaces may also be used to increase readability. Since the BASIC compiler ignores spaces, the two statements:

10 LET B=D^2+1
10 LET B=D^2+1

are identical, but spaces in the first statement provide ease in reading.

6.6.5.4 Rules for Exponentiation — The following rules apply in evaluating the expression \( A^B \).

1. If \( B=0 \), then \( A^B=1 \)
2. If \( A=0 \) and \( B>0 \), then \( A^B=0 \)
3. If \( A=0 \) and \( B<0 \), then \( A^B=0 \) and a DIV error message is displayed
4. If \( B \) is an integer \( \geq 9 \), then \( A^B = A_1 * A_2 * A_3 * \ldots * A_n \), where \( n = B \)
5. If \( B \) is an integer \( < 0 \) then \( A^B = 1/A_1 * A_2 * A_3 * \ldots * A_n \), where \( n = B \)
6. If \( B \) is a decimal (non-integer) and \( A>0 \), then \( A^B = EXP(B*LOG(A)) \)
7. If \( B \) is a positive or negative decimal (non-integer) and \( A>0 \), the program halts and an EM error is displayed.

\[
3^0=1 \\
0^{-2}=0 \\
3^{-5}=3*3*3*3*3=243 \\
3^{-5}=1/243 \\
2^{-3.6}=e^{B1nA}=e^{3.61n2} \\
-3^2.6 \text{ is illegal.}
\]
6.6.5.5 Relational Operators — A program may require that two values be compared at some point to discover their relation to one another. To accomplish this, BASIC makes use of the following relational operators:

- =  equal
- <  less than
- \( \leq \) or \( \geq \)  less than or equal to
- >  greater than
- \( \geq \) or \( \leq \)  greater than or equal to
- \(<\) or \(\neq\)  not equal to

Depending upon the result of the comparison, control of program execution may be directed to another part of the program. Relational operators are used in conjunction with the IF-THEN statement.

The meaning of the (\(=\)) sign should be clarified. In algebraic notation, the formula \(X=X+1\) is meaningless. However, in BASIC (and most computer languages), the equal sign designates replacement rather than equality. Thus, this formula is actually translated "add one to the current value of \(X\) and store the new result back in the same variable \(X\)". Whatever value has previously been assigned to \(X\) will be combined with the value 1. An expression such as \(A=B+C\) instructs the computer to add the values of \(B\) and \(C\) and store the result in a third variable \(A\). The variable \(A\) is not being evaluated in terms of any previously assigned value, but only in terms of \(B\) and \(C\). Therefore, if \(A\) has been assigned any value prior to its use in this statement, the old value is lost; it is replaced instead by the value of \(B+C\).

6.6.6 String Information
The previous sections dealt only with numerical information. However, BASIC also processes alphanumerical information called strings. A string is a sequence of characters, each of which is a letter, a digit, a space, or some character other than a statement terminator (backslash or carriage return).

6.6.6.1 String Character Set — The character set recognized by BASIC is as shown below. The decimal code for each character is also shown.

<table>
<thead>
<tr>
<th>Code Number</th>
<th>Code Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 @</td>
<td>18 R</td>
</tr>
<tr>
<td>1 A</td>
<td>19 S</td>
</tr>
<tr>
<td>2 B</td>
<td>20 T</td>
</tr>
<tr>
<td>3 C</td>
<td>21 U</td>
</tr>
<tr>
<td>4 D</td>
<td>22 V</td>
</tr>
<tr>
<td>5 E</td>
<td>23 W</td>
</tr>
<tr>
<td>6 F</td>
<td>24 X</td>
</tr>
<tr>
<td>7 G</td>
<td>25 Y</td>
</tr>
<tr>
<td>8 H</td>
<td>26 Z</td>
</tr>
<tr>
<td>9 I</td>
<td>27 [</td>
</tr>
<tr>
<td>10 J</td>
<td>28 \</td>
</tr>
<tr>
<td>11 K</td>
<td>29 ]</td>
</tr>
<tr>
<td>12 L</td>
<td>30 ^</td>
</tr>
<tr>
<td>13 M</td>
<td>31 _</td>
</tr>
<tr>
<td>14 N</td>
<td>32 (</td>
</tr>
<tr>
<td>15 O</td>
<td>33 !</td>
</tr>
<tr>
<td>16 P</td>
<td>34 &quot;</td>
</tr>
<tr>
<td>17 Q</td>
<td>35 #</td>
</tr>
</tbody>
</table>
6.6.6.2 String Conventions — Strings may be used as constants or variables. String constants are enclosed in quotes. For example,

"THIS IS A STRING CONSTANT"

A string variable consists of a letter followed by a dollar sign ($ or a letter and a single digit followed by a dollar sign. A$ and A1$ are both legitimate string variable names while 2A$ and A2$ are not.

A string variable may contain at most eight characters unless it has been dimensioned with the DIM statement. Quotation marks may be included in strings by indicating two quotation marks in succession. For example, the string A"B would appear in a program as:

10 LET A$="A""B"

The following lines:

10 LET A$=" "QUOTE" "
20 PRINT A$
99 END

will result in this display:

"QUOTE"

It is important to recognize the real, structural difference between strings and numerical data. This number 2

is not identical to this string:

"2"

Numerical data may not be used where strings are required, and vice-versa.
6.6.6.3 String Concatenation — Strings can be concatenated, that is, connected like links in a chain, by using the ampersand (&). For example, as a result of the following lines, the next statement executed will be line 460:

\[
\begin{align*}
400 & \text{ LET } A1$ = "AB" \\
410 & \text{ LET } B1$ = "CD" \\
420 & \text{ LET } C1$ = "ABCD" \\
430 & \text{ IF } C1$ = A1$ & B1$ GO TO 460
\end{align*}
\]

The ampersand can be used to concatenate string expressions wherever a string expression is legal, with the exception that information cannot be stored by means of a LET statement in concatenated string variables. That is, concatenated string variables cannot appear to the left of the equal sign in a LET statement. For example, this statement is legal

\[
\text{LET } A$ = B$ & C$
\]

while this statement is not:

\[
\text{LET } A$ & B$ = C$
\]

6.6.7 Format Control Characters

In OS/78 BASIC, a terminal line is formatted into five fixed zones (called print zones) of 14 columns each. A program such as:

\[
\begin{align*}
1 & \text{ LET } A = 2.3 \\
2 & \text{ LET } B = 21 \\
3 & \text{ LET } C = 156.75 \\
4 & \text{ LET } D = 1.134 \\
5 & \text{ LET } E = 23.4 \\
10 & \text{ PRINT } A, B, C, D, E \\
15 & \text{ END}
\end{align*}
\]

where the control character comma (,) is used to separate the variables in the PRINT statement, will cause the values of the variables to be displayed, using all five zones. For example,

\[
\begin{align*}
2.3 & \quad 21 & \quad 156.75 & \quad 1.134 & \quad 23.4
\end{align*}
\]

\text{READY}

It is not necessary to use the standard five zone format for output. The control character semicolon (;) causes the text or data to be output immediately after the last character printed.

The following example program illustrates the use of the control characters in PRINT statements.

\[
\begin{align*}
5 & \text{ READ } A, B, C, D, E, F \\
10 & \text{ PRINT } A, B, C, D, E, F \\
15 & \text{ PRINT} \\
20 & \text{ PRINT } A; B; C; D; E; F \\
25 & \text{ DATA } 4, 5, 6 \\
30 & \text{ DATA } 16, 25, 36
\end{align*}
\]
BASIC

RUNNH

\[
\begin{array}{cccccc}
4 & 5 & 6 & 16 & 25 & 36
\end{array}
\]

READY

As this example illustrates, when more than five variables are listed in the PRINT statement, OS/78 BASIC automatically moves the sixth number to the beginning of the next line.

6.6.8 Files
Files are referenced symbolically by a name of up to six alphanumeric characters followed, optionally, by a period and an extension of two alphanumeric characters. The extension to a file name is generally used as an aid for remembering the format of a file.

A fixed length file is one which is already in existence. That is, it has been created and CLOSED. The length of a fixed length file is equal to the number of blocks in the file and cannot be changed.

A variable length file is a newly created file. Until the file is CLOSED, it is equal in length to the largest free space on the device. When the file is CLOSED it becomes a fixed length file equal in length to the actual number of blocks it occupies. Unless the file is CLOSED, the CHAIN, STOP or END statements will cause a loss of the file.

6.7 BASIC STATEMENTS
BASIC statements are the principal components of BASIC programs. The general format of BASIC statements is:

\[ xxx \quad \text{Word Type Parameters} \]

where

\[ xxx \] is the line number;

\[ \text{Word Type} \] is the statement (instruction) type; and

\[ \text{Parameters} \] are the variables used in conjunction with the statement type.

Each statement starts with a line number followed by the word statement type. Spaces have no significance in BASIC language statements except in messages or literal strings which are displayed or printed out. Thus, spaces may, but need not, be used to modify a program and make it more readable.

Multiple statements may be placed on a single line by separating each statement from the preceding statement with a backslash. For example:

\[ 10 \text{ LET A}=5\backslash \text{LET B}=.2\backslash \text{LET C}=3\backslash \text{PRINT "ENTER DATA"} \]

All of the statements in line 10 will be executed before BASIC continues to the next line. Only one statement number at the beginning of the entire line is necessary. However, it should be remembered that program control cannot be transferred to a statement within a line, only to the first statement of the line in which it is contained. This consideration is important when using control statements or loop statements (see related descriptions).

6.7.1 Statement Line Numbers (Sequencing)
Failure to assign a line number results in the message:

WHAT
Each line of the program begins with a line number of 1 to 5 digits that serves to identify the line as a statement. The largest allowable line number is 99999.

A common programming practice is to number lines by fives or tens, so that additional lines may be inserted in a program without the necessity of renumbering lines already present. Renumbering a program can be accomplished by using the RESEQ program (Section 6.5.4).

6.7.2 The PRINT Statement
The PRINT statement is used to perform calculations and display results. It is also used to display alphanumeric (string) messages.

PRINT Statement Form

<table>
<thead>
<tr>
<th>Line Number</th>
<th>Statement</th>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxx</td>
<td>PRINT</td>
<td>expressions</td>
</tr>
</tbody>
</table>

Where expressions may be numbers, variables, strings or arithmetic expressions, separated by commas or semicolons. When used without an expression, a blank line will be output on the terminal.

In BASIC, a line is formatted into five fixed zones (called print zones) of 14 columns each. If the expressions in a PRINT statement are separated by commas, each will begin at a 14-column interval. That is, the first expression will be displayed starting at the first position, the second at the fifteenth position, and so forth. If more than five variables are involved, the display will automatically continue at the beginning of the next line.

If this format is not desired, you may separate expressions with the semicolon (;), causing the text or data to be output immediately after the last character printed (see Example 3).

If the last expression in a PRINT statement is followed by a comma or semicolon, the next display called for by the program will begin on the same line (if there is room). This also applies to the question mark displayed by the INPUT statement (see Examples 2 and 4).

Any algebraic expression in a PRINT statement will be evaluated using the current value of the variables.

Regardless of format (integer, decimal, or E-type), BASIC prints numbers in the form:

```
    sign number space
```

where the sign is either minus (–) or blank (for plus) and a blank space always follows the number (see Example 3).

Strings and numeric expressions may be combined in a single PRINT statement (see Example 3).

To output an alphanumeric message, enclose the expression in quotation marks. To print a quotation mark (") put two quotation marks ("" ) in the expression (see Example 5).

The following examples illustrate the use of the PRINT statement.
Example 1:

The following lines

```
40 LET A=1
50 LET B=2
55 LET C=3
60 PRINT A,B,C
99 END
```

will cause each variable to begin at a 14-column interval as follows:

```
1  2  3
```

Example 2:

The following lines

```
110 LET A$="PRINTING"
120 LET B$= "STRINGS"
130 PRINT A$, A$;
140 PRINT B$
199 END
```

will display the following:

```
PRINTING STRING
```

Example 3:

The following lines:

```
10 A=5
20 PRINT "NUMBER";A;"AND";6
99 END
```

will cause this display:

```
NUMBER 5 AND 6
```

Blanks appear before the 5 and 6 because they are positive. The blank following the 5 accounts for the blank that BASIC always generates after a number.

Example 4:

The following lines

```
60 PRINT "NUMBER OF YEARS";
70 INPUT N
99 END
```

will display (the number 9 is typed in by the user) the following:

```
NUMBER OF YEARS
9
```
Example 5:

The following lines

```
80 PRINT *MESSAGE*
90 PRINT *A**R*
100 PRINT ***QUOTE***
199 END
```

will cause the following display:

```
MESSAGE
A*R
***QUOTE***
```

6.7.3 Information Entry Statements (DATA, READ)
The DATA and READ statements are always used together. The DATA statement is used to set up a list of numeric or string values. These values are accessed by the READ statement which assigns those values to variables in a program.

6.7.3.1 DATA Statement Format — The DATA statement sets up a list of values to be used by the READ statement.

**DATA Statement Form**

<table>
<thead>
<tr>
<th>Line Number</th>
<th>Statement</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxx</td>
<td>DATA</td>
<td>Values</td>
</tr>
</tbody>
</table>

where values are numeric and/or string entries separated by commas. The DATA statement serves as a source of input variables for the program. The variables are accessed for processing by the READ statement.

There may be any number of DATA statements in a program. They are not executed and may be placed anywhere within the program. However, BASIC treats them all together as a single list.

Both string data and numeric data may be intermixed in a single DATA statement.

String data in a DATA list must always be enclosed by quotation marks.

For example, the three DATA statements

```
30 DATA "FIRST"
120 DATA 2, "SECOND"
240 DATA 3, "THIRD", 4
```

are equivalent to the following statement:

```
240 DATA "FIRST", 2, "SECOND", 3, "THIRD", 4
```

6.7.3.2 READ Statement Format — The READ statement accesses variables defined by the DATA statement and assigns those values to variables in a program.
READ Statement Form

<table>
<thead>
<tr>
<th>Line Number</th>
<th>Statement</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxx</td>
<td>READ</td>
<td>variables</td>
</tr>
</tbody>
</table>

where variables are names corresponding to values contained in DATA statements. The following examples illustrate the use of the READ statement.

Variable names must be separated by commas.

A READ statement may contain numeric and string variable names intermixed, to correspond to numeric and alphanumeric values in the DATA list. However, since values are taken from the DATA list in sequential order, you must ensure that the values in the list are in the correct sequence to correspond to the variable names of the same format (numeric or string).

A READ statement may have more or fewer variables than there are values in any one DATA statement. The READ statement causes BASIC to search all available DATA statements in the order of their line numbers until values are found for each variable in the READ. A second READ statement will begin reading values where the first one stopped.

The READ statement is always used in combination with the DATA statement.

Example 1:

The following lines

```
10 READ A, B, C
20 DATA 1, 2, 3
```

will set variable A equal to 1, B equal to 2, and C equal to 3.

Example 2:

These statements

```
50 READ C, D$, E, F$(K)
60 DATA 5, "AAA", 12, "WORD"
```

will set:

- C=5
- D$="AAA"
- E=12
- F$="WORD"

Example 3:

The following program will display the first and third variables in a DATA list:

```
50 READ A$, B$, C$
60 PRINT A$, C$
70 DATA *DIS*, *XXX*, *PLAY*
99 END
```
The screen will show:

**DISPLAY**

Example 4:

The following program uses DATA statements to supply both a variable number of scores and variable score values to an average calculation routine.

```
100 PRINT "NUMBER"
110 PRINT "OF SCORES", "AVERAGE"
120 PRINT
125 READ N1
127 FOR I = 1 TO N1
130 READ N
135 IF N = 0 GOTO 999
140 LET S = 0
150 FOR K = 1 TO N
160 READ T
170 LET S = S + T
180 NEXT K
190 PRINT N, S/N
200 NEXT I
890 DATA 5
900 DATA 3, 82, 88, 97
910 DATA 5, 66, 78, 71, 82, 75
920 DATA 4, 82, 86, 100, 91
930 DATA 4, 72, 82, 73, 82
940 DATA 6, 61, 73, 67, 80, 84, 79
999 END
```

RUNNH

<table>
<thead>
<tr>
<th>NUMBER</th>
</tr>
</thead>
<tbody>
<tr>
<td>OF SCORES AVERAGE</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>data</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>89</td>
</tr>
<tr>
<td>5</td>
<td>74.4</td>
</tr>
<tr>
<td>4</td>
<td>89.75</td>
</tr>
<tr>
<td>4</td>
<td>77.25</td>
</tr>
<tr>
<td>6</td>
<td>74</td>
</tr>
</tbody>
</table>

**READY**

6.7.4 LET Statement

The LET statement assigns the value of an algebraic expression to a variable. Use of the word LET is optional.

**LET Statement Form**

<table>
<thead>
<tr>
<th>Line Number</th>
<th>Statement</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxx</td>
<td>LET</td>
<td>v = expression</td>
</tr>
</tbody>
</table>
where

v is a variable; and

expression is a number, another variable, a string (enclosed in quotes), or an arithmetic expression.

The following examples illustrate the use of the LET statement.

Example 1:
The numeric variable A is set equal to 5 by the statement

10 LET A = 5

Example 2:
The string variable A$ is set equal to "XYZ" by the statement

10 A$ = "XYZ"

Example 3:
An element (3,2) in array A is set equal to an element (1,4) in array B by the statement

10 LET A(3,2) = B(1,4)

NOTE
The LET statement does not necessarily imply an equality. LET means "evaluate the expression to the right of the equal sign and assign this value to the variable on the left". Thus, the statement

L=L+1

means "set L equal to a value one greater than it was before".

6.7.5 Loops (FOR and NEXT Statement)

6.7.5.1 FOR Statement Format — The FOR statement is used in combination with the NEXT statement to specify program loops.

FOR Statement Form

<table>
<thead>
<tr>
<th>Line Number</th>
<th>Statement</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxx</td>
<td>FOR</td>
<td>V = x to y [STEP z]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LOOP INDEX INDEX INDEX</td>
</tr>
<tr>
<td></td>
<td></td>
<td>INDEX INITIAL TERMINAL STEP</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VALUE VALUE VALUE</td>
</tr>
</tbody>
</table>

where

V is a variable that serves as the INDEX for the loop. The index value is incremented (or decremented) each time the loop is executed.
x is an expression (numerical value, variable name, or mathematical expression) indicating the initial value of the index, that is, the value it will have before the loop is executed the first time;

y is the terminal value of the index. When \( y \) is "beyond" \( y \), the loop will not be repeated; and

z is the step value, that is, it is the value that is added to the index each time the loop is executed. Variable \( z \) may be assigned a minus value in cases where it is desired to decrement the index. If "STEP \( z \)" is omitted, a value +1 is assumed.

The \( x \), \( y \), and \( z \) values are expressions: these expressions are evaluated upon first encountering the loop. This includes setting the index equal to the initial value. Therefore, the program can later jump back to the same FOR statement any number of times to re-start the loop. If the \( x \), \( y \), and \( z \) values are unchanged, the loop will be repeated the same number of times each time the program executes the FOR statement.

A variable used as an index in a FOR statement must not be subscripted.

The block of instructions to be executed repeatedly will immediately follow the FOR statement. After the last of these instructions there must be a NEXT statement whose parameter is the same as the index of the loop.

If the initial value is "beyond" the terminal value, the loop will never execute because an initial check is made of the starting and terminal values before the loop is executed. "Beyond", as used here, means that the initial value is not equal to the terminal value and adding the step value will only increase the difference. If the step value is negative "beyond" means "less than". If the step value is positive, "beyond" means "greater than".

The value of the loop index can be changed within the loop, thus influencing the number of times the loop is executed.

A program can have one or more loops within a loop. This is called "nesting loops", and is often used with subscripted variables.

It is possible to exit from a loop without the index reaching the terminal value by using an IF statement. Control may transfer into a loop only if that loop was left earlier without being completed.

The following examples illustrate the use of the FOR statement.

Example 1:

The following program

```
10 DIM C(2,3)
20 FOR A=1 TO 3
30 FOR B=1 TO 2
40 READ C(B,A)
50 PRINT C(B,A), B*A
60 NEXT B
70 NEXT A
80 DATA 1,2,3,4,5,6
90 END
```
will display:

```
1  1  1
2  2  1
3  1  2
4  2  2
5  1  3
6  2  3
```

Lines 30 through 60 of the above program represent a loop nested within another loop (line 20 through 70). The FOR statement on line 30 will be executed three times. Each time, its loop statements (lines 40 through 60) will be repeated twice.

Example 2:

The following lines

```
10 LET B=2\C=3
20 FOR A=C-B TO B^C STEP C
30 PRINT A
40 NEXT A
50 PRINT A
99 END
```

will display:

```
1
4
7
7
```

When the FOR statement was encountered, it interpreted the initial value as 1 (3 minus 2), the terminal value as 8 (2 to the third power) and the step value as 3.

Example 3:

The following loop

```
10 FOR I=10 TO 1 STEP -1
20 NEXT I
30 PRINT I
99 END
```

will display:

```
1
```

Example 4:

The following loop

```
10 FOR D=1 TO 5
20 LET D=D+4
30 NEXT D
99 END
```
will only be executed once.

Example 5:

The loops in the following program will never be executed:

```
10 FOR D=5 TO 1
20 PRINT D
30 NEXT D
10 FOR D=1 TO 5 STEP -1
20 PRINT D
30 NEXT D
99 END
```

6.7.5.2 NEXT Statement Format — The NEXT statement defines the end of a program loop.

**NEXT Statement Form**

<table>
<thead>
<tr>
<th>Line Number</th>
<th>Statement</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxx</td>
<td>NEXT</td>
<td>v (variable)</td>
</tr>
</tbody>
</table>

where v represents the index value used in the corresponding FOR statement.

A NEXT statement is never used without a FOR statement. The variable value must be identical to the index value (v) used in the corresponding FOR statement. The variable parameter may not be subscripted.

6.7.6 Control Statements (GOTO, IF-THEN, GOSUB, RETURN)

Normally the statements in a BASIC program are executed in the sequence they appear in a program. Control statements allow this execution sequence to be redirected.

6.7.6.1 Unconditional Branch (GOTO Statement) — The GOTO statement is used to transfer control to another line statement in a program. BASIC then continues execution at the line number referred to in the GOTO statement.

**GOTO Statement Form**

<table>
<thead>
<tr>
<th>Line Number</th>
<th>Statement</th>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxx</td>
<td>GOTO</td>
<td>n</td>
</tr>
</tbody>
</table>

where n is the line number of the statement to which control should be transferred.

For example, the following program employs two separate GOTO statements to redirect program control.

```
10 GOTO 40
20 PRINT "SECOND"
30 STOP
40 PRINT "FIRST"
50 GOTO 20
99 END
```

When executed, the program displays:

```
FIRST
SECOND
```
NOTE
When the program reaches the GOTO statement, the statements immediately following will not be executed; instead, execution is transferred to the statement beginning with the line number indicated.

6.7.6.2 Conditional Branch (IF-THEN Statement) — The IF-THEN statement tests a condition and redirects program control if that condition is true. That is, if the condition specified is true, the IF-THEN statement effectively executes a GOTO statement for the line number specified. When the condition is false, the next statement is executed.

<table>
<thead>
<tr>
<th>Line Number</th>
<th>Statement</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxx</td>
<td>IF</td>
<td>v1 relation v2 THEN n</td>
</tr>
</tbody>
</table>

where

v1, v2 are variables, numbers, strings, or expressions to be compared;
relation is the relational operator to be used in comparing v1 and v2 (See Section 6.6.5.4);
THEN may be replaced with GOTO if desired (either THEN or GOTO is acceptable, but one of the two must be present); and
n is the number of the statement to which the program will jump if the relationship described is true.

The following examples illustrate the IF-THEN statement.

Example 1:
After executing the first two instructions

```
20 LET A=5
30 IF A =>2 GOTO 100
40 PRINT "NO"
100 PRINT "HERE"
```

control is transferred to line 100.

Example 2:
After executing the following instructions

```
20 LET A=5
30 IF A=2 GOTO 100
40 PRINT "NO"
99 END
```

the console will display:

```
NO
```
After executing the following instructions

```
10 LET A$ = "*"
20 IF A$ = "Z" GOTO 99
30 PRINT "YES"
99 END
```

the program will display:

**YES**

because the ASCII code for "*" is not equal to the ASCII code ASCII for a "Z".

Strings may be used in IF-THEN statements, but comparisons are based on the positions of the characters in the string sequence. The question mark (?) is the highest alphanumeric character and the at sign (@) is the lowest.

The two strings described in the IF-THEN statement are compared one character at a time, from left to right, until the ends of the strings are reached or until an inequality is found.

If the strings are of unequal length, BASIC lengthens the shorter string by adding spaces to the right until both are of equal length. If "AB" is compared to a four-character string, it will be treated as "AB(space) (space)".

When using numerical expressions in the IF-THEN statement, the test for equality may not always work due to the nature of the arithmetic used by the computer. One way to get around this is to compare the absolute value of the difference between the operands to a very small number. For example, instead of

```
20 IF A = B THEN 50
```

use

```
20 IF ABS(B) < 0001 THEN 50
```

6.7.6.3 **Branch to Subroutine (GOSUB Statements)** — A subroutine is a group of statements that perform a processing operation at more than one point in a program.

The GOSUB statement is used to branch to the subroutine and the RETURN statement redirects control back to the main body of the program.

GOSUB Statement Form

<table>
<thead>
<tr>
<th>Line Number</th>
<th>Statement</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxx</td>
<td>GOSUB</td>
<td>n</td>
</tr>
</tbody>
</table>

where n is the line number of the first line of the subroutine.

When the program encounters a GOSUB statement, the following action occurs:

1. BASIC internally records the number of the statement following the GOSUB statement.
2. Control is transferred to statement number n.

GOSUB is always used with the RETURN statement.

The RETURN statement is the last statement in a subroutine. When the program encounters the RETURN statement, control transfers back to the statement following the GOSUB.
A subroutine can call another subroutine. This is called "nesting" (see Example 2). Programs may be written to transfer control from one statement to another in the same subroutine, or to a statement in a different subroutine. When a RETURN is encountered, control returns to the statement following the last GOSUB that was executed. Subroutines may not be nested more than ten levels deep (Example 2 shows two levels of nesting).

The following examples illustrate the GOSUB statement.

Example 1:

The following program

```
100 LET A$="HI"
110 LET B$="THERE"
120 LET C$=""
130 LET V$=A$
140 GOSUB 1000
150 LET V$=A$&C$&B$
160 GOSUB 1000
170 STOP
1000 REM PRINT V$
1010 PRINT V$
1020 RETURN
9999 END
```

will display:

```
HI
HI THERE
```

Example 2:

The following program, showing subroutine nesting,

```
100 FOR I=1 TO 3
110 LET V=I
120 GOSUB 1000
130 LET X=2*I
140 GOSUB 500
150 NEXT I
200 STOP

500 REM CALCULATE
510 LET V=X+2
520 GOSUB 1000
530 RETURN

1000 REM PRINT VALUE
1010 PRINT "THE VALUE IS";
1020 PRINT V
1030 RETURN
9999 END
```
BASIC

will display:

THE VALUE IS 1
THE VALUE IS 4
THE VALUE IS 2
THE VALUE IS 6
THE VALUE IS 3
THE VALUE IS 8

6.7.6.4 Return from Subroutine (RETURN Statement) — The RETURN statement is used to redirect control from a subroutine.

RETURN Statement Form

<table>
<thead>
<tr>
<th>Line Number</th>
<th>Statement</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxx</td>
<td>RETURN</td>
<td>(none)</td>
</tr>
</tbody>
</table>

RETURN is always used with the GOSUB statement.

When the RETURN is encountered, it causes control to transfer to the statement following the last GOSUB executed.

6.7.7 Program Termination Statements (END, STOP)

BASIC is equipped with two statements, the END and STOP statements, that can be used to terminate program execution.

6.7.7.1 END Statement Format — The END statement terminates execution of the program. It informs the BASIC compiler that it has reached the last line of the program.

END Statement Form

<table>
<thead>
<tr>
<th>Line Number</th>
<th>Statement</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxx</td>
<td>END</td>
<td>None</td>
</tr>
</tbody>
</table>

**NOTE**

Only one END statement may appear in a program and it must be the last statement in the program.

6.7.7.2 STOP Statement Format — The STOP statement is used to terminate the execution of a program.

STOP Statement Form

<table>
<thead>
<tr>
<th>Line Number</th>
<th>Statement</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxx</td>
<td>STOP</td>
<td>None</td>
</tr>
</tbody>
</table>

Note that there may be several STOP statements in a program.

6.7.8 The INPUT Statement

The INPUT statement permits the operator to specify data during execution of a program.
INPUT Statement Form

<table>
<thead>
<tr>
<th>Line Number</th>
<th>Statement</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>XXX</td>
<td>INPUT</td>
<td>Variable List</td>
</tr>
</tbody>
</table>

where variables can be a single variable or a list of variables.

The INPUT Statement will cause the program to pause during execution, display a question mark, and wait for you to enter a value and press the RETURN key. If there are several variables involved, the program will expect you to type in a value corresponding to each variable. If you press the RETURN without having done this, the system will display another question mark and wait for the rest of the data. When the values have all been typed in, the program will continue with the variable names now equivalent to the values typed in. The first variable will equal the first entry, the second will equal the second entry, etc. (See Example 1).

The following values are recognized as acceptable when inputting numeric data:

+ or - sign
digits 0 through 9
the letter E
leading spaces (ignored)
(first decimal point)

All other characters are treated as delimiters for separating numeric data. That is, when the system encounters a character other than those specified, it will consider that it has come to the end of the entry relating to the variable it is currently processing and will apply any characters typed in after that to the following variable, if any (see Example 1).

When inputting numeric data, two delimiters read in succession imply that the data between delimiters is 0 (see Example 2).

In response to an INPUT statement, you can provide more values than are requested by the INPUT statement. The remaining or unused values are saved for subsequent use by the next INPUT statement. The question mark is not displayed until the program is out of data.

When inputting string data all characters are recognized as part of the string. Quotation marks are not typed in unless they are deliberately meant to be part of the string.

Each string requested by an INPUT statement must be terminated by a carriage return which acts as the data delimiter. This is necessary since all characters except for the carriage return are recognized as part of the data string.

String variables are assumed to be eight characters long unless otherwise described in a DIM statement.

The following examples illustrate the use of the INPUT statement.

Example 1:

If, in response to this statement:

```
100 INPUT A,B,C,D,E
```

you type

```
?23.7A4E3 9<+1
```
the variables will have the following values:

A: -2
B: 3.7
C: 4000 (4E3=4x10^3=4000)
D: 9
E: 1 (numbers are assumed to be positive unless they are specified to be negative).

The delimiters in the above line are the comma, the letter "A", the space after the number 3, the left-angle bracket (<), and the RETURN.

Example 2:

If, in response to the same statement, you type

? -2, 3.7, 4E3, 1

The results will be identical except that variable "D" will have the value 0.

Example 3:

If, in response to the statement:

50 INPUT R$

you type

? "A,C>=+7

the string variable R$ will have the value:

"A,C>=+7

Example 4:

If, in response to:

40 LET A=5
50 INPUT B(A)

you type

? 7

B(5) will now have the value of 7.

6.7.9 The REMark Statement
The REM statement is a nonexecutable statement used to insert comments into the source program.

REMARK Statement Form

<table>
<thead>
<tr>
<th>Line Number</th>
<th>Statement</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxx</td>
<td>REM</td>
<td>comments</td>
</tr>
</tbody>
</table>
6.7.10 Ancillary Statements (DIMension, RESTORE, DEFine, RANDOMIZE and CHAIN)
BASIC has five additional statements that are used as helping statements and fall in no particular category. They are described in the following paragraphs.

6.7.10.1 DIMension Statement Format — The DIMension statement is used to describe subscripted variables and to define the length of strings.

**DIMension Statement Form**

<table>
<thead>
<tr>
<th>Line Number</th>
<th>Statement</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxx</td>
<td>DIM</td>
<td>v(n[,m])</td>
</tr>
</tbody>
</table>

where v is the name of the subscripted variable.

If the variable name (v) is numeric, and

1. m is omitted, then n+1 is the number of elements in an array or vector (see Example 1).
2. m is specified, then n+1 is the number of rows in a two-dimensional array (see Example 2).
3. if v is numeric, then m+1 is the number of columns in a two-dimensional array (see Example 2);

If the variable name (v) is alphanumeric, and

1. m is omitted, then n is the length of the string. This describes a single string, not a list, and cannot exceed 80 (see Example 3).
2. m is specified, then n+1 is the number of strings in the list. This is a one dimensional array (see Example 4).
3. if v is alphanumeric, then m is the length of each string in a one-dimensional list. It cannot exceed 80 (see Example 4).

The parameters n and m must be integer constants. They are limited in size only by the amount of available memory.

If a numeric variable is used in the program with a subscript but is not defined in a DIM statement, BASIC assigns it an array size of ten.

BASIC assumes a maximum string length of 8 characters unless the variable appears in a DIM statement.

Two-dimensional string variables are not permitted.

When the variable is used in other statements, it is not permitted to have subscripts whose values are higher than those in the DIM statement.

The first element of every array is automatically assumed to have a subscript of zero. Therefore, the number of boxes in a one dimensional array is n+1. The number of boxes in a two-dimensional array is (n+1)*(m+1). The first element in an array is v(0) or v(0,0) (see Example 2). However, the “zero” elements can be disregarded in programming unless the user wishes to conserve memory.

More than one array can be defined in a single DIM statement (see Example 7).

In general, wherever you can use a single variable name in a statement, you can use a subscripted one. Exceptions are noted in descriptions of the individual statements (such as the index of the FOR statement).

The following examples illustrate the use of DIMension statement.
Example 1:

The following statement

10 DIM A(5)

describes six numeric elements as follows:

| A(0) | A(1) | A(2) | A(3) | A(4) | A(5) |

To store a 5 in A(3) and then display it, type

20 LET A(3)=5
30 PRINT A(3)

Example 2:

The following statement

10 DIM A(3,5)

describes 24 numeric elements \((4 \times 6 = 24)\) as follows:

<table>
<thead>
<tr>
<th>SIX COLUMNS</th>
</tr>
</thead>
<tbody>
<tr>
<td>A(0,0) A(0,1) A(0,2) A(0,3) A(0,4) A(0,5)</td>
</tr>
<tr>
<td>A(1,0) A(1,1) A(1,2) A(1,3) A(1,4) A(1,5)</td>
</tr>
<tr>
<td>A(2,0) A(2,1) A(2,2) A(2,3) A(2,4) A(2,5)</td>
</tr>
<tr>
<td>A(3,0) A(3,1) A(3,2) A(3,3) A(3,4) A(3,5)</td>
</tr>
</tbody>
</table>

Example 3:

The following statement

10 DIM C$(12)

describes one string, 12 characters long:

C$

Example 4:

The following statement

10 DIM D$(3,20)

describes 4 strings, each 20 characters long:

D$(0) D$(1) D$(2) D$(3)
Example 5:

The following program will fill the array in Example 4 from a DATA list:

```
10 DIM D$(3,20)
20 FOR Y=0 TO 3
30 READ D$(Y)
40 NEXT Y
50 FOR Z=0 TO 3
60 PRINT D$(Z)
70 NEXT Z
80 DATA "ZERO","ONE","TWO","THREE"
99 END
```

and display each element:

```
ZERO
ONE
TWO
THREE
```

Example 6:

After these statements:

```
10 DIM B(3,5)
20 FOR G=1 TO 3
30 LET B(G,0)=G
40 NEXT G
50 FOR H=2 TO 5
60 LET B(0,H)=H
70 NEXT H
99 END
```

The area diagrammed in Example 2 would appear as follows:

<table>
<thead>
<tr>
<th></th>
<th>B(0,2)</th>
<th>B(0,3)</th>
<th>B(0,4)</th>
<th>B(0,5)</th>
</tr>
</thead>
<tbody>
<tr>
<td>B(1,0)</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B(2,0)</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B(3,0)</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Example 7:

The following statement dimensions both the one-dimensional array A and the two-dimensional array B:

```
10 DIM A(20), B(4,7)
```

6.7.10.2 RESTORE Statement — The RESTORE statement allows the program to go back to the beginning of a DATA list (paragraph 6.7.3.1) after using the list in a READ statement (paragraph 6.7.3.2).
RESTORE Statement Form

<table>
<thead>
<tr>
<th>Line Number</th>
<th>Statement</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxx</td>
<td>RESTORE</td>
<td>None</td>
</tr>
</tbody>
</table>

If it is desired to use the same data more than once in a program, RESTORE makes it possible to recycle through the DATA list beginning with the first value in the first DATA statement.

The RESTORE statement may be used in programs where DATA statements convey numeric or string data to READ statements.

The same variable names may be used the second time through the data since the values are being read as though for the first time.

The following example illustrates the use of the RESTORE statement.

The following lines

```
10 READ A,B,C,D
20 PRINT A;B;C;D
30 RESTORE
40 READ E,F,G,H
50 PRINT E;F;G;H
60 DATA 1,2,3,4
99 END
```

will cause this display:

```
1 2 3 4
1 2 3 4
```

6.7.10.3 DEFine Statement — This statement allows you to add functions to a program.

DEFine Statement Form

<table>
<thead>
<tr>
<th>Line Number</th>
<th>Statement</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxx</td>
<td>DEF</td>
<td>FNa(x)=expression</td>
</tr>
</tbody>
</table>

where

- a is the capital letter used for identifying the function;
- x is a dummy variable and must be the same on both sides of the equal (=) sign; and
- expression defines the function by indicating the calculation process (function) involved.

There must be a DEF statement for each function used in the program.

The DEF statement must appear before the first use of the function it defines.

If there is more than one variable involved in the function, BASIC will identify them by their position.

Up to 14 different arguments may be used.
Up to 26 FN functions may be defined in a single program (FNA, FNB . . . FNZ).

The following examples illustrate the use of the DEFine statement.

**Example 1:**

The statement

```
10 DEF FND(S)=S^2
```

will cause the later statement

```
50 LET R=FND(4)
```

to be evaluated as R=16. BASIC locates the DEFine statement for the function FND, substitutes the 4 for the variable (S) in the expression (S^2) and calculates the value of FND(4) to be 4^2=16.

**NOTE**

A variable that is used as a dummy argument in a DEFine statement can also be used elsewhere in the program.

**Example 2:**

This program:

```
10 DEF FNH(N,P)=2*P+N
20 LET X=4\LET Y=5
30 PRINT FNH(X,Y)
40 END
```

will display:

```
14
```

BASIC takes the first value in the function (4) as “N”, because “N” appears first in the DEFine statement. It takes the second value (5) as “P”, because “P” is in the second position.

```
DEF FNH(N,P)=2*P+N
```

first position    second position

```
PRINT FNH(X,Y)
```

**6.7.10.4 RANDOMIZE Statement** — The RANDOMIZE statement is used with the RND function to generate a different set of numbers each time the program is run.

**RANDOMIZE Statement Form**

<table>
<thead>
<tr>
<th>Line Number</th>
<th>Statement</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxx</td>
<td>RANDOMIZE</td>
<td>None</td>
</tr>
</tbody>
</table>

6-35
6.7.10.5 CHAIN Statement — The CHAIN statement allows one program to execute another program. It can be used to divide large programs into a number of smaller programs that are to be written and stored separately.

CHAIN Statement Form

<table>
<thead>
<tr>
<th>Line Number</th>
<th>Statement</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxx</td>
<td>CHAIN</td>
<td>&quot;dev:file.ex&quot;</td>
</tr>
</tbody>
</table>

where "dev:file.ex" is the device and the file name of the program to be executed.

When BASIC encounters a CHAIN statement in a program, it stops execution of that program, retrieves the program named in the CHAIN statement from the specified device and file, compiles the CHAINed program (if necessary) and begins execution of that program.

If the program was started in the editor, when execution of all programs in the chain is complete, the workspace will contain the original program.

All output files that are opened in the original program must be closed before the CHAIN statement is encountered.

A BASIC language program may only CHAIN to another BASIC language program, and the program it CHAINs to may not have an extension of ".*" (see the following example).

A compiled program may also execute CHAIN statements, but it can only CHAIN to other compiled programs which have ".*" extensions.

In the following example, the program "SECOND" will CHAIN to the program "FIRST".

```
NEW FIRST

READY
10 PRINT "TARGET"
20 END
SAVE SYS:FIRST

READY
NEW SECOND

READY
10 PRINT "ORIGINAL"
20 CHAIN "SYS:FIRST.BA"
30 END
SAVE SYS:SECOND

READY
RUNNH
ORIGINAL
TARGET
```
Enter FIRST and store it.

Enter SECOND and store it.

Execute the programs.

Displayed by SECOND.

Displayed by FIRST.

6.7.11 File Handling Statements
The file capability provided by OS/78 BASIC allows writing to or reading from the peripheral devices of the system.

6.7.11.1 FILE# Statement — The FILE# statement defines and opens a file.
FILE# Statement Form

<table>
<thead>
<tr>
<th>Line Number</th>
<th>Statement</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxx</td>
<td>FILE</td>
<td>t # n: “dev:file.ex”</td>
</tr>
</tbody>
</table>

where

t must be one of the following:

(blank) – for an input string file
V – for an output string file
N – for an input numeric file
VN – for an output numeric file;

n is the number you are assigning to the file (It must be 1, 2, 3, or 4 and can be a numeric variable); and

“dev:file.ex” is the standard device, file name and extension; It must be either a string variable or the string itself in quotation marks.

A file must be opened before it can be used. The only exception is the terminal, which is always available for use.

The n in this statement is the number you must use in all FILE# statements that refer to the file. The terminal is always FILE#0.

The following examples illustrate the use of the FILE# statement.

Example 1:

The following statement describes file number 1 to be the string file HPRDAT.AS on RXA1 and opens it for output.

10 FILEV#1:’RXA1:HPRDAT.AS’

Example 2:

The following statement describes file number 2 to be the numeric file DATA.NU on RXA1 and opens it for output.

10 FILEVN#2:’RXA1:DATA.NU’

Example 3:

The following statement describes file number 3 to be the string file TEST.AB on RXA1 and opens it for input.

10 FILE#3:’RXA1:TEST.AB’

Example 4:

The following statement describes file number 4 to be the numeric file Fila.CD on RXA1 and opens it for input.

10 FILEN#4:’RXA1:FILA.CD’

6.7.11.2 PRINT# Statement — The PRINT# statement writes data into files.
PRINT# Statement Form

<table>
<thead>
<tr>
<th>Line Number</th>
<th>Statement</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxx</td>
<td>PRINT#</td>
<td>n:expressions</td>
</tr>
</tbody>
</table>

where

n is the file number (It may be a numeric variable); and
expressions depends on file type, numeric or string as discussed below.

As long as PRINT# is used for only numbers or numeric variables separated by commas or semicolons (or RETURN at the end of a PRINT# line), BASIC converts commas to spaces and does not write the carriage return and line feed to numeric files. The only thing written out will be a “list” of numbers separated by spaces. Each time INPUT# is used, it will read another number from the list (see Example 1).

When dealing with string files, symbols such as RETURNS, semicolons, and so forth, are used in the PRINT# statement in the same way they are used in the PRINT statement. The PRINT# statement works exactly the same way that the PRINT statement does, except that the line goes to the file designated instead of to the terminal.

The important difference here is that string files involve lines, while numeric files involve individual numbers. That is, each INPUT# will read a line (see Example 2).

If PRINT# is used for numerics to a string file, BASIC will convert them to strings. If an attempt is then made to INPUT# them into numeric variables, BASIC will convert them back to numerics. However, the RETURN and line feed at the end of a line will be converted to zeros. This can be dealt with by adding two extra variables to the INPUT# statement.

The following examples illustrate the use of the PRINT# statement.

Example 1:

The following lines

```
10 FILEVN#1:*SYS:TST.XX*
20 PRINT#1:1,2
30 PRINT#1:3,4,
40 PRINT#1:5,6
50 CLOSE#1
60 FILEN#1:*SYS:TST.XX*
70 FOR X=1 TO 6
80 INPUT#1:Z
90 PRINT Z
100 NEXT X
199 END
```

will display

```
1
2
3
4
5
6
```
Example 2:

The following lines

```plaintext
10 PRINT "A", "B"
20 PRINT "C", "D"
30 PRINT "E"
40 END
```

will display

```
A  B
C  D  E
```

The same display will also be caused by

```plaintext
5 DIM J$(30)
10 FILE#2:"RXA1:PROG.XX"
20 PRINT #2:"A", "B"
30 PRINT #2:"C", "D"
40 PRINT #2:"E"
50 CLOSE#2
60 FILE#2:"RXA1:PROG.XX"
70 INPUT#2:J$
80 PRINT J$
90 INPUT#2:J$
100 PRINT J$
199 END
```

6.7.11.3 INPUT# Statement  — The INPUT# statement reads data from a file.

**INPUT# Statement Form**

<table>
<thead>
<tr>
<th>Line Number</th>
<th>Statement</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxx</td>
<td>INPUT#</td>
<td>n: variables</td>
</tr>
</tbody>
</table>

where

- n is the file number of the file being read (it may be a variable); and
- variables is the list of variables into which data will be read. Each variable is separated by a comma.

Normally, data from numeric files is read into numeric variables and data from string files is read into string variables.

It is possible however to write numerics into a string file and then read them into either numeric or string variables, depending on how it is desired to use them. If numbers are read from a string file into string variables, they will be in string form and subject to the same rules as other strings.

Numbers read from a string file into numeric variables will be converted to numerics. Line feeds are ignored in this case.

The following examples illustrate the use of the INPUT# statement.
Example 1:

To read two strings from RXA1:FIL.DA, enter

```
10 FILE#1:"RXA1:FIL.DA"
20 INPUT#1:A$,B$
```

A$ will contain the first string, B$ the second string.

Example 2:

To read five numbers from RXA1:TST.XX, enter

```
10 FILEN#3:"RXA1:TST.XX"
20 INPUT#3:A,B,C,D,E
```

Example 3:

The following program writes numerics to a string file and reads them back as numerics:

```
10 FILEV#1:"SYS:FILA.ZZ"
20 FOR I=1 TO 5
30 PRINT#1:I
40 NEXT I
50 CLOSE#1
60 FILE#1:"SYS:FILA.ZZ"
70 FOR I=1 TO 5
80 INPUT#1:J,K,L
90 PRINT J
100 NEXT I
110 END
```

It will display:

```
1
2
3
4
5
```

6.7.11.4 RESTORE# Statement — The RESTORE statement resets the file data printer back to the beginning of the file, that is, RESTORE effectively performs a file close followed immediately by a file open so that the first data element in the file can be reread.

RESTORE# Statement Form

```
Line Number    Statement        Parameters
       xxx         RESTORE#         n
```

where

- `n` is the number of the file to be reset. It may be a number or a numeric variable.
For example, if RXA1: FILB.LM is a numeric file containing the numbers 1 through 9, the instructions

```
100 FILEN#3:*RXA1: FILB.LM*
110 FOR I=1 TO 3
120 INPUT #3:Z
130 PRINT Z
140 NEXT I
150 RESTORE#3
160 INPUT #3:Z
170 PRINT Z
199 END
```

will display:

```
1
2
3
1
```

**NOTE**

If n is zero, the DATA list in the program is reset.

**6.7.11.5 CLOSE# Statement** — The CLOSE# statement finishes the processing of a file and allows it number to be assigned to another file in a FILE# statement.

**FILE CLOSE# Statement Form**

<table>
<thead>
<tr>
<th>Line Number</th>
<th>Statement</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxx</td>
<td>CLOSE#</td>
<td>n</td>
</tr>
</tbody>
</table>

where

n is the number of the file to be closed. It may be a variable.

For example, in the following program

```
50 FILEV #1:*SYS:TEST.XX*
60 PRINT #1:*A*,*B*,*C*,*D*
70 CLOSE #1
80 FILE #1:*RXA1: FILD.DA*
90 INPUT #1:J$
99 END
```

The CLOSE# statement at line 70 finished the processing of file SYS:TEST.XX and allowed its number, 1, to be assigned to RXA1: FILD.DA in line 80.

**NOTE**

All output files must be CLOSED before any of the following is executed.

CHAIN
STOP
END
CTRL/C

Failure to do so results in loss of file.
6.7.11.6 **IF END# Statement** — The IF END# statement determines if the End of File (EOF) marker has been read from a file, and if so, branches to the line specified.

**IF END# Statement Form**

<table>
<thead>
<tr>
<th>Line Number</th>
<th>Statement</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxx</td>
<td>IF END#</td>
<td>n THEN m</td>
</tr>
</tbody>
</table>

where:

- **n** is the file number of the file in question (it may be a variable); and
- **m** is the line number to which control passes when the end of the file is detected.

This statement works only for string files.

The IF END# statement should come immediately after the PRINT# or INPUT# statement for that file. If, as a result of the IF END# statement, control passes to line m, it means that the last PRINT# or INPUT# was not successful. That is, nothing was actually read from or written to the file as a result of the last INPUT# or PRINT# statement.

For Example,

The following lines

```
10 FILE#1: "SYS:PROGA.BB"
20 PRINT#1: "A"
30 PRINT#1: "B"
40 CLOSE#1
50 FILE#1: "SYS:PROGA.BB"
60 INPUT#1:A$
70 IF END#1 THEN 100
80 PRINT A$
90 GOTO 60
100 PRINT "END OF FILE"
110 CLOSE#1
199 END
```

will display

```
A
B
END OF FILE
```

6.8 **BASIC FUNCTIONS**

BASIC functions are standard subroutines incorporated into the BASIC Run Time System (BRTS) to aid computations and text handling.

Function calls consist of a three letter (all capitals) name followed by an argument in parentheses. The argument may be a number, variable, expression, or another function. Generally, functions may be used anywhere a number or variable is legal in a mathematical expression.

Most functions compute a value based on the value of the argument or arguments involved. They are said to return this value. For example, SQR(A) "returns" the square root of A.
Functions may return either strings or numbers. Functions that return strings have names ending in a dollar sign (STR$, SEG$), while functions returning numbers have names that do not end in a dollar sign (SGN, VAL).

6.8.1 Arithmetic Functions (ABS, INT, EXP, RND, SGN, SQR)

6.8.1.1 BASIC ABS Function — The ABS function returns the absolute value of an expression.

Format

    ABS(X)

where

    X is a number, numeric variable, or numeric expression

6.8.1.2 BASIC EXP Function — The EXP function calculates the value of e raised to the X power, where e is equal to 2.71828. That is, EXP(X) is equivalent to 2.71828X.

Format

    EXP(X)

where

    X is a number, numeric variable, expression, or another function.

6.8.1.3 BASIC INT Function — The INT function returns the value of the largest integer not greater than the argument.

Format

    INT(X)

where

    X is a number, numeric variable, expression, or another function.

**NOTE**

This function can be used to round numbers to the nearest integer by specifying INT(X+.5).

For example, the function INT(34.67) has the value 34; the functions INT(34.67+.5) and INT(34.37+.5) have these values of 35 and 34, respectively, and these functions INT(−23) and INT(−14.39) have these values of −23 and −15, respectively.

6.8.1.4 BASIC RND Function — The RND function produces random numbers between (but not including 0 and 1).

Format

    RND(X)

where

    X is a dummy variable in this function.
Every time this function is encountered in a statement, it will produce a different set of decimal numbers. However, the program is RUN again, the same set of numbers will be produced (see Example 1).

If this repetition is undesirable, it can be changed with the RANDOMIZE statement.

For example, the following program is run twice with identical results:

```
10 FOR A = 1 TO 5
20 PRINT RND(X)
30 NEXT A
40 END
```

```
RUNNH
0. 361572
0. 332764
0. 633057
0. 350342
0. 670166

READY
RUNNH
0. 361572
0. 332764
0. 633057
0. 350342
0. 670166

READY
```

**6.8.1.5 BASIC SGN Function** — The SGN function creates a value based on the sign of the argument.

Format

```
SGN(X)
```

where

X is a number, numeric variable, numeric expression, or another function.

**NOTE**

The value of the SGN function will be 1 if the argument is any positive number, 0 if the argument is zero, and −1 if the argument is negative.

**6.8.1.6 BASIC SQR Function** — The SQR function computes the positive square root of an expression.

Format

```
SQR(X)
```

where

X is a number, numeric variable, numeric expression, or another function.
NOTE
If the argument is negative, the absolute value of the argument is used.

6.8.2 Trigonometric Functions (ATN, COS, LOG, SIN)

6.8.2.1 BASIC ATN Function — The ATN function calculates the angle (in radians) whose tangent is given as the argument of the function.

Format

\[ \text{ATN}(X) \]

where

\[ X \]

is a number, numeric variable, expression, or another function, representing the tangent of an angle.

6.8.2.2 BASIC COS Function — The COS function is used to calculate the cosine of an angle specified in radians.

Format

\[ \text{COS}(X) \]

where

\[ X \]

is a number, numeric variable, expression, or another function, representing the size of an angle in radians.

6.8.2.3 BASIC LOG Function — The LOG function calculates the natural logarithm of \( X \) (to the base e).

Format

\[ \text{LOG}(X) \]

where

\[ X \]

is a number, numeric variable, expression, or another function.

6.8.2.4 BASIC SIN Function — The SIN function is used to calculate the sine of an angle specified in radians.

Format

\[ \text{SIN}(X) \]

where

\[ X \]

is a number, numeric variable, expression or another function, representing the size of an angle in radians.

6.8.3 String Handling Functions (ASC, CHRS, DATS, LEN, POS, SEG$, STR$, TAB, VAL)

6.8.3.1 BASIC ASC Function — The ASC function converts a one character string to its code number (see CHR$).

6-45
Format

ASC(X)

where

X is a one character string.

To find what will be returned for any character, look for the character in the "CHARACTER" column of Table 6-1. The number to the left of it is the decimal equivalent.

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Character</th>
<th>Decimal</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>@</td>
<td>32</td>
<td>(space)</td>
</tr>
<tr>
<td>1</td>
<td>A</td>
<td>33</td>
<td>!</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td>34</td>
<td>&quot;</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td>35</td>
<td>#</td>
</tr>
<tr>
<td>4</td>
<td>D</td>
<td>36</td>
<td>$</td>
</tr>
<tr>
<td>5</td>
<td>E</td>
<td>37</td>
<td>%</td>
</tr>
<tr>
<td>6</td>
<td>F</td>
<td>38</td>
<td>&amp;</td>
</tr>
<tr>
<td>7</td>
<td>G</td>
<td>39</td>
<td>'</td>
</tr>
<tr>
<td>8</td>
<td>H</td>
<td>40</td>
<td>(</td>
</tr>
<tr>
<td>9</td>
<td>I</td>
<td>41</td>
<td>)</td>
</tr>
<tr>
<td>10</td>
<td>J</td>
<td>42</td>
<td>*</td>
</tr>
<tr>
<td>11</td>
<td>K</td>
<td>43</td>
<td>+</td>
</tr>
<tr>
<td>12</td>
<td>L</td>
<td>44</td>
<td>,</td>
</tr>
<tr>
<td>13</td>
<td>M</td>
<td>45</td>
<td>-</td>
</tr>
<tr>
<td>14</td>
<td>N</td>
<td>46</td>
<td>.</td>
</tr>
<tr>
<td>15</td>
<td>O</td>
<td>47</td>
<td>/</td>
</tr>
<tr>
<td>16</td>
<td>P</td>
<td>48</td>
<td>0</td>
</tr>
<tr>
<td>17</td>
<td>Q</td>
<td>49</td>
<td>1</td>
</tr>
<tr>
<td>18</td>
<td>R</td>
<td>50</td>
<td>2</td>
</tr>
<tr>
<td>19</td>
<td>S</td>
<td>51</td>
<td>3</td>
</tr>
<tr>
<td>20</td>
<td>T</td>
<td>52</td>
<td>4</td>
</tr>
<tr>
<td>21</td>
<td>U</td>
<td>53</td>
<td>5</td>
</tr>
<tr>
<td>22</td>
<td>V</td>
<td>54</td>
<td>6</td>
</tr>
<tr>
<td>23</td>
<td>W</td>
<td>55</td>
<td>7</td>
</tr>
<tr>
<td>24</td>
<td>X</td>
<td>56</td>
<td>8</td>
</tr>
<tr>
<td>25</td>
<td>Y</td>
<td>57</td>
<td>9</td>
</tr>
<tr>
<td>26</td>
<td>Z</td>
<td>58</td>
<td>;</td>
</tr>
<tr>
<td>27</td>
<td>[</td>
<td>59</td>
<td>,</td>
</tr>
<tr>
<td>28</td>
<td>\</td>
<td>60</td>
<td>&lt;</td>
</tr>
<tr>
<td>29</td>
<td>]</td>
<td>61</td>
<td>=</td>
</tr>
<tr>
<td>30</td>
<td>_</td>
<td>62</td>
<td>&gt;</td>
</tr>
<tr>
<td>31</td>
<td></td>
<td>63</td>
<td>?</td>
</tr>
</tbody>
</table>

6-46
For example,

The following program

```
10 LET A$="*"
20 PRINT ASC("P"), ASC(A$), ASC("9")
30 END
```

will display

```
16 42 57
```

6.8.3.2 BASIC CHRS Function — The CHRS function converts a code number (modulo 64) to its equivalent character in the 64 character set.

Format

`CHRS(X)`

where

`X` is a number, numeric expression, or numeric variable (0 < X < 63).

To find what will be returned for any number, look for the number in the “DECIMAL” column of the preceding conversion table. The equivalent character will be to the right of the number.

For example, the following line

```
10 PRINT CHRS(1), CHRS(40)
```

will display:

```
A (a
```

6.8.3.3 BASIC DATS Function — The DATS function returns the current system date.

Format

`DATS(X)`

where

`X` is a dummy variable in this function.

The date is returned as an eight-character string of the form:

`MM/DD/YY`

If the date has not been specified with the Monitor DATE command, no characters will be returned.

For example, the following lines

```
10 LET D$ = DATS(X)
20 PRINT D$
```
will display:

\[
07/20/77
\]

if that date was entered with the Monitor DATE command.

6.8.3.4 BASIC LEN Function — The LEN function returns the number of characters in a string.

Format

\[
\text{LEN(X$)}
\]

where

\[
\text{X$} \quad \text{is a string or string variable. It may be several concentrated strings and/or variables.}
\]

6.8.3.5 BASIC POS Function — The POS function returns the location of a specified group of characters in a string.

Format

\[
\text{POS(X$, Y$, Z)}
\]

where

\[
\begin{align*}
\text{X$} & \quad \text{is the string to be searched (it may be a string variable or string constant);} \\
\text{Y$} & \quad \text{is the series of characters you are searching for (it may be a string variable or a string constant); and} \\
\text{Z} & \quad \text{is the position in the string at which you want to begin the search.}
\end{align*}
\]

This function searches X$ for the first occurrence of Y$. The search begins with the Zth character in X$.

Z may not be less than zero or greater than the length of string X$.

If Y$ contains no characters, the function returns a one.

If X$ contains no characters, it returns a zero.

If Y$ is not found, it returns a zero.

For example, the following lines

\[
10 \text{ DIM B2$(12)} \\
20 \text{ B2$ = "ABCD\text{EFGHI}D\text{E}"} \\
30 \text{ PRINT POS (B2$, "DE\text{F}\text{E}", 7)}
\]

will display:

\[
10
\]

6.8.3.6 BASIC SEGS Function — The SEGS function returns the sequence of characters between two positions in a string.
**BASIC**

Format

```
SEG$(X$,Y,Z)
```

where

- `X$` is the string containing the characters to be returned (it may be a string variable or a string constant);
- `Y` is the position of the first character to be returned; and
- `Z` is the position of the last character to be returned.

If `Y` is less than 1, it is set to 1.

If `Y` is greater than the length of `X$`, no characters are returned.

If `Z` is less than 1, no characters are returned.

If `Z` is greater than the length of `X$`, it is set equal to the length of `X$`.

If `Z` is smaller than `Y`, no characters are returned.

For example, the following lines

```
10 DIM B2$(12)
20 B2$ = "ABCDEFGHIDEF"
30 PRINT SEG$(B2$,3,5)
```

will display:

```
CDE
```

6.8.3.7 **BASIC STR$ Function** — The STR$ function converts numbers to strings.

Format

```
STR$(X)
```

where

- `X` is a numeric expression

**NOTE**

The string that is returned is in the form in which numbers are output in BASIC without leading or trailing blanks.

6.8.3.8 **BASIC TAB Function** — The TAB function allows you to position characters anywhere on the terminal line.
Format

    TAB(X)

where

    X  is the position (from 1 to 80) in which the next character will be displayed.

This function may only be used in a PRINT or PRINT# statement.

Positions on the line are considered by BASIC to be numbered from 1 to 80 across the screen from left to right.

Each time the TAB function is used, positions are counted from the beginning of the line, not from the current position of the cursor.

If X is less than the current position of the cursor, the display starts at the current position.

If X is greater than 80, the display will begin at the first position of the next line.

In order to keep track of the cursor, BASIC maintains a “column count”, which represents the position of the cursor at any given time. As the cursor moves across the screen, BASIC adds to the column count. When the cursor returns to the first position, the column count is reset to 0. The activity of the TAB function is based on this count.

There are circumstances in which the column count does not coincide with the position of the cursor. For example, the PNT(07) function will add 1 to the count without moving the cursor. Also, the PNT(13) function will return the cursor to the first position on the screen without setting the column count to zero. The user, therefore, must take this into account when using the TAB function after PNT functions. The column count will be corrected the next time there is a “normal” return to column one.

For examples, the following lines

    60 LET B=5
    70 PRINT "A";TAB(B);"C"

will cause this display:

    A________C

6.8.3.9 BASIC VAL Function  —  The VAL function converts a string to numeric data.

Format

    VAL(X$)

where

    X$  is a string constant or string variable made up of those values that BASIC recognizes as acceptable when inputting numeric data:

    + or - sign  
    digits 0 through 9    
    the letter E    
    leading spaces (ignored)    
    . (first decimal point)
6.8.4 Display Console Control Function (PNT)

**BASIC PNT Function** — The PNT function is used to perform special actions on the terminal, such as sounding the buzzer, erasing the screen, and moving the cursor.

**Format**

```
PNT(X)
```

where

```
X    is the value of the character to be output.
```

Special actions that can be performed by the PNT function are as follows:

- **PNT(07)** sounds buzzer
- **PNT(08)** moves cursor one space to left
- **PNT(09)** moves to next tab stop (Tab stops are set every 8 spaces.)
- **PNT(10)** moves cursor down one line and scrolls if required
- **PNT(13)** moves cursor to left margin of current line
- **PNT(27); “A”** moves cursor up one line
- **PNT(27); “C”** moves cursor right one position
- **PNT(27); “H”** moves cursor to upper lefthand corner of screen (“home” position)
- **PNT(27); “J”** erases from cursor position to end of screen
- **PNT(27); “K”** erases line from cursor to right margin
- **PNT(27); “[”** stops display of any new lines when screen is full. Each time the operator presses the SCROLL key, another line will be displayed. If he presses SHIFT and SCROLL, twelve new lines will be displayed. This is called Hold Screen Mode.
- **PNT(27); CHRS(28)** turns off Hold Screen Mode.

**NOTE**
The PNT function may only be used in a PRINT or PRINT# Statement.
6.8.5 Trace Function

**BASIC TRC Function** — The TRC function causes BASIC to print the line number of each statement in the program as it is executed.

Format

\[ v = \text{TRC}(x) \]

where

v can be any letter (it is a dummy argument and has no purpose except to occupy that position on the line); and

x is 1 to turn the function on and 0 to turn it off.

This function is used to follow the progress of a program and help in tracking down errors.

When BASIC encounters TRC(1) in a program, it displays the line number of each line in the program as it is executed. The line numbers are displayed between percent signs.

When TRC(0) is encountered, the function is turned off and normal program operation resumes.

Certain types of statements are not recorded by TRC: namely, DATA, DEF, DIM, END, GOTO, NEXT, RANDOMIZE, REM, and STOP.

For example, the following program

```
60 T=TRC(1)
70 GOSUB 90
80 GOTO 140
90 PRINT "IN OUTER SUB"
100 GOSUB 120
110 RETURN
120 PRINT "IN INNER SUB"
130 RETURN
140 T= TRC(0)
150 END
```

will display:

```
% 70 %
% 90 %
IN OUTER SUB
% 100 %
% 120 %
IN INNER SUB
% 130 %
% 110 %
% 140 %
```
6.9 SUMMARY OF BASIC EDITOR COMMANDS

<table>
<thead>
<tr>
<th>Command</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>BYE</td>
<td>Exits from the editor and returns control to the monitor</td>
</tr>
<tr>
<td>List</td>
<td>Displays the program statements in the workspace with a header</td>
</tr>
<tr>
<td>LISTNH</td>
<td>Displays the program statements in the workspace, without a header</td>
</tr>
<tr>
<td>Name</td>
<td>Renames the program in the workspace</td>
</tr>
<tr>
<td>New</td>
<td>Clears the workspace and tells the editor the name of the program the user is about to type</td>
</tr>
<tr>
<td>Old</td>
<td>Clears the workspace, finds a program on the disk, and puts it into the workspace</td>
</tr>
<tr>
<td>Run</td>
<td>Executes the program in the workspace, after displaying a header</td>
</tr>
<tr>
<td>RUNNH</td>
<td>Executes the program in the workspace, without displaying a header</td>
</tr>
<tr>
<td>Save</td>
<td>Puts the program in the workspace on a disk</td>
</tr>
<tr>
<td>Scratch</td>
<td>Erases all statements from the workspace</td>
</tr>
</tbody>
</table>

6.10 SUMMARY OF BASIC STATEMENTS

<table>
<thead>
<tr>
<th>Statement</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chain</td>
<td>Executes another program</td>
</tr>
<tr>
<td>Example: 40 Chain “SYS:PROG.BA”</td>
<td></td>
</tr>
<tr>
<td>Close#</td>
<td>Closes a file</td>
</tr>
<tr>
<td>Example: 100 Close#1</td>
<td></td>
</tr>
<tr>
<td>Data</td>
<td>Sets up a list of values to be used by the READ statement</td>
</tr>
<tr>
<td>Example: 240 Data “FIRST”, 2, 3</td>
<td></td>
</tr>
<tr>
<td>Def</td>
<td>Defines functions</td>
</tr>
<tr>
<td>Example: 10 Def FND(S) = S + 5</td>
<td></td>
</tr>
<tr>
<td>Dim</td>
<td>Describes a string and/or any subscripted variables</td>
</tr>
<tr>
<td>Example: 50 Dim B(3, 5), D$_{s}(3, 72)$</td>
<td></td>
</tr>
<tr>
<td>End</td>
<td>Terminates program compilation and execution</td>
</tr>
<tr>
<td>Example: 100 End</td>
<td></td>
</tr>
<tr>
<td>File#</td>
<td>Defines and opens a file</td>
</tr>
<tr>
<td>Example: 20 File#2: “RXA1:DATA.NV”</td>
<td></td>
</tr>
</tbody>
</table>
Statement     Function

FOR           Describes program loops (used with NEXT)
               Example:  60 FOR X=1 TO 10 STEP 2

GOSUB         Transfers control to a subroutine (used with RETURN)
               Example:  50 GOSUB 100

GOTO          Transfers control to another statement
               Example:  100 GOTO 50

IF            Tests the relationship between two variables, numbers, or expressions
               Example:  20 IF A=0 THEN 50

IF END#       Tests for the end of a string file
               Example:  60 IF END#3 THEN 100

INPUT         Accepts data from the terminal
               Example:  80 INPUT A,B,C

INPUT#        Reads data from a file
               Example:  50 INPUT#1:A$

LET           Assigns a value to a variable
               Example:  90 LET A$="XYZ"

NEXT          Indicates the end of a program loop (used with FOR)
               Example:  140 NEXT I

PRINT         Displays data on the screen
               Example:  200 PRINT A,"X".6

PRINT#        Writes data to a file
               Example:  180 PRINT#1:J

RANDOMIZE     Causes the RND function to produce a different set of numbers each time the program
               in run
               Example:  10 RANDOMIZE

READ          Sets variables equal to the values in DATA statements
               Example:  50 READ A$,B

REM           Inserts comments into the program
               Example:  30 REM COMPUTE EARNINGS

RESTORE       Sets program READ statements back to the beginning of the DATA list
               Example:  85 RESTORE
BASIC

Statement  Function

RESTORE#  Resets a file pointer back to the beginning of that file
Example: 130 RESTORE#3

RETURN  Returns control from a subroutine (used with GOSUB)
Example: 115 RETURN

STOP  Terminates program execution
Example: 40 STOP

6.11 SUMMARY OF BASIC FUNCTIONS

Command  Function

ABS(X)  Returns the absolute value of an expression
Example: 10 LET X=ABS(-66)
will assign X a value of 66

ASC(X$)  Converts a one character string to its code number
Example: 20 PRINT ASC("B") will display 2

ATN(X)  Calculates the angle (in radians) whose tangent is given as the argument
Example: 30 LET X=ATN(.57735)
will assign X a value of 0.523598

CHR$(X)  Converts a code number to its equivalent character
Example: 40 PRINT CHR$(1) will display A

COS(X)  Returns the cosine of an angle specified in radians
Example: 50 LET Y=COS(45*3.14159)/180
will assign Y a value of 0.707108

DAYS(X)  Returns the current system date
Example: 60 PRINT DAYS(X)
will display the system date, such as 07/20/77

EXP(X)  Calculates the value of e raised to a power, where e is equal to 2.71828
Example: 30 IF Y>EXP(1.5) GOTO 70
will go to line 70 if Y is greater than 4.48169

INT(X)  Returns the value of the nearest integer not greater than the argument
Example: 60 LET X=INT(34.67)
will assign X the value 34
**BASIC**

<table>
<thead>
<tr>
<th>Command</th>
<th>Function</th>
</tr>
</thead>
</table>
| LEN(X$) | Returns the number of characters in a string  
Example: 10 PRINT LEN ("DOG") will display 3 |
| LOG(X)  | Calculates the natural logarithm of the argument  
Example: 10 PRINT LOG(959) will display 6.86589 |
| PNT(X)  | Outputs non-printing characters for terminal control  
Example: 50 PRINT PNT(13) will move the cursor to the left margin of the current line |
| POS(X$, Y$, Z) | Returns the location of a specified group of characters (Y$) in a string (X$) starting at a character position (Z)  
Example: 60 LET V = POS("ABCDBC", "BC", 4) will assign V a value of 5 |
| RND(X)  | Returns a random number between (but not including) 0 and 1  
Example: 70 PRINT RND(X) will display a decimal number, such as 0.361572 |
| SEG$(X$, Y$, Z) | Returns the sequence of characters in a string (X$) between two positions in the string (X, Y)  
Example: 30 LET R$ = SEG$("ABCDEF", 2, 4) will assign R$ a value of BCD |
| SGN(X)  | Returns 1 if the argument is positive, 0 if it is zero, and -1 if it is negative  
Example: 200 PRINT 5 * SGN(-6) will display -5 |
| SIN(X)  | Returns the sine of an angle specified in radians  
Example: 30 LET B = SIN(30 * 3.14159 / 180) will assign B a value of 0.5 |
| SQR(X)  | Returns the positive square root of an expression  
Example: 40 PRINT SQR(16) will display 4 |
| STR$(X) | Converts a number into a string  
Example: 120 PRINT STR$(1.7611124) will display the string 1.76111 |

6-56
### 6.12 BASIC ERROR MESSAGES

#### 6.12.1 Compiler Error Messages

The following error messages are generated by the BASIC compiler:

<table>
<thead>
<tr>
<th>CH</th>
<th>ERROR IN CHAIN STATEMENT</th>
<th>NM</th>
<th>MISSING LINE NUMBER</th>
</tr>
</thead>
<tbody>
<tr>
<td>DE</td>
<td>ERROR IN DEF STATEMENT</td>
<td>OF</td>
<td>OUTPUT FILE ERROR</td>
</tr>
<tr>
<td>DI</td>
<td>ERROR IN DIM STATEMENT</td>
<td>PD</td>
<td>PUSHDOWN STACK OVERFLOW</td>
</tr>
<tr>
<td>FN</td>
<td>ERROR IN FILE NUMBER OR NAME</td>
<td>QS</td>
<td>STRING LITERAL TOO LONG</td>
</tr>
<tr>
<td>FP</td>
<td>INCORRECT FOR STATEMENT</td>
<td>SS</td>
<td>BAD SUBSCRIPT OR FUNCTION ARG</td>
</tr>
<tr>
<td>FR</td>
<td>ERROR IN FUNCTION ARGS</td>
<td>ST</td>
<td>SYMBOL TABLE OVERFLOW</td>
</tr>
<tr>
<td>IF</td>
<td>ERROR IN IF STATEMENT</td>
<td>SY</td>
<td>SYSTEM INCOMPLETE</td>
</tr>
<tr>
<td>IC</td>
<td>I/C ERROR</td>
<td>TB</td>
<td>PROGRAM TOO BIG</td>
</tr>
<tr>
<td>LS</td>
<td>MISSING EQUALS SIGN IN LET</td>
<td>TD</td>
<td>TOO MUCH DATA IN PROGRAM</td>
</tr>
<tr>
<td>LT</td>
<td>STATEMENT TOO LONG</td>
<td>TS</td>
<td>TOO MANY CHARS IN STRING</td>
</tr>
<tr>
<td>MD</td>
<td>MULTIPLY DEFINED LINE NUMBER</td>
<td>UD</td>
<td>ERROR IN UDEF STATEMENT</td>
</tr>
<tr>
<td>ME</td>
<td>MISSING END STATEMENT</td>
<td>UF</td>
<td>FOR STATEMENT WITHOUT NEXT</td>
</tr>
<tr>
<td>MO</td>
<td>OPERAND EXPECTED, NOT FOUND</td>
<td>US</td>
<td>UNDEFINED STATEMENT NUMBER</td>
</tr>
<tr>
<td>MP</td>
<td>PARENTHESIS ERROR</td>
<td>UU</td>
<td>USE STATEMENT ERROR</td>
</tr>
<tr>
<td>MT</td>
<td>OPERAND OF MIXED TYPE</td>
<td>XC</td>
<td>CHARs AFTER END OF LINE</td>
</tr>
<tr>
<td>NF</td>
<td>NEXT STATEMENT WITHOUT FOR</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### 6.12.2 Run-Time System Error Messages

The following error messages are generated by the BASIC run-time system:

<p>| BO       | NO MORE BUFFERS AVAILABLE    | GS | TOO MANY NESTED GOSUBS |
| CI       | IN CHAIN, DEVICE NOT FOUND   | IA | ILLEGAL ARG IN UDEF   |
| CL       | IN CHAIN, FILE NOT FOUND     | IF | ILLEGAL DEV:FILENAME |
| CX       | CHAIN ERROR                  | IN | INQUIRE FAILURE       |
| DA       | READING PAST END OF DATA     | IO | TTY INPUT BUFFER OVERFLOW |
| DE       | DEVICE DRIVER ERROR          | LM | TAKING LOG OF NEGATIVE NUMBER |
| DC       | NO MORE ROOM FOR DRIVERS     | OE | DRIVER ERROR WHILE OVERLAYING |
| DV       | ATTEMPT TO DIVIDE BY ZERO    | OV | NUMERIC OR INPUT OVERFLOW |
| EF       | LOGICAL END OF FILE          | PA | ILLEGAL ARG IN POS    |
| EM       | NEGATIVE NUMBER TO REAL POWER| RE | READING PAST END OF FILE |</p>
<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EN</td>
<td>ENTER ERROR</td>
<td>SC</td>
<td>CONCATENATED STRING TOO LONG</td>
</tr>
<tr>
<td>FB</td>
<td>USING FILE ALREADY IN USE</td>
<td>SL</td>
<td>STRING TOO LONG OR UNDEFINED</td>
</tr>
<tr>
<td>FC</td>
<td>CLOSE ERROR</td>
<td>SR</td>
<td>READING STRING FROM NUMERIC FILE</td>
</tr>
<tr>
<td>FF</td>
<td>FETCH ERROR</td>
<td>ST</td>
<td>STRING TRUNCATION ON INPUT</td>
</tr>
<tr>
<td>FI</td>
<td>CLOSING OR USING UNOPENED FILE</td>
<td>SO</td>
<td>SUBSCRIPT OUT OF RANGE</td>
</tr>
<tr>
<td>FM</td>
<td>FIXING NEGATIVE NUMBER</td>
<td>SW</td>
<td>WRITING STRING INTO NUMERIC FILE</td>
</tr>
<tr>
<td>FN</td>
<td>ILLEGAL FILE NUMBER</td>
<td>VR</td>
<td>READING VARIABLE LENGTH FILE</td>
</tr>
<tr>
<td>FO</td>
<td>FIXING NUMBER &gt; 4095</td>
<td>WE</td>
<td>WRITING PAST END OF FILE</td>
</tr>
<tr>
<td>GR</td>
<td>RETURN WITHOUT GOSUB</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
CHAPTER 7

FORTRAN IV

7.1 OVERVIEW

The FORTRAN (or FORMula TRANSLator) programming language enables you to express mathematical operations in a form similar to standard mathematical notation as well as perform a variety of such applications as process control, information retrieval, and commercial data processing.

This chapter describes the components of the FORTRAN system—the Compiler, Loader, Run-Time System, and Library—and the elements of the language.

OS/78 FORTRAN IV conforms generally to the specifications for American National Standard FORTRAN. Several enhancements have been added, and these are described in Section 7.2.

To create and run a FORTRAN program, use the following procedure.

1. Write the program in source code, using the statements and other features of the FORTRAN language as they are described in this chapter. Divide the program into logical units—one for the main program and one for each subroutine you need. Use the OS/78 Editor to create a file for each program unit.

2. Use the COMPIL command to compile and assemble the FORTRAN source program. The Compiler (which chains automatically to the assembler) accepts one program unit—a main program or a subroutine—and produces one module of relocatable code; that is, assembled code which has not yet been assigned permanent addresses in memory. The Compiler also produces an optional listing file. (For complete details on the FORTRAN Compiler, see Section 7.1.1.)

3. Use the LOAD command to link the program units and assign permanent memory locations to the complete program. The FORTRAN Loader accepts up to 128 modules of relocatable code. It links subroutines to the main program, assigns permanent addresses, and produces a loader-image file, which contains the complete program in linked and relocated form. In addition, it determines if the program requires any functions from the FORTRAN library and copies them in relocated form into the loader-image file. The Loader also produces an optional loader symbol map, showing the areas in memory that the program will use. (For details on the Loader, see Section 7.1.2.)
4. Load the program into memory and run it with the EXECUTE command. The EXECUTE command summons the Run-Time System, which accepts a loader-image file, places it in memory, determines the I/O requirements of the program, and starts execution. (For details on the Run-Time System, see Section 7.1.3.)

The Compiler and Loader provide options that enable you to compile, load, and run a program with a single command. In addition, the EXECUTE command accepts source code or relocatable code, performs the necessary operations, and causes execution.

For example, assume you have written a short program called POWER that you want to enter as a file on your system device and run. To enter the file, summon the Editor with the CREATE command, naming the file in the command line and adding the extension FT, the standard OS/78 extension for a file containing a FORTRAN source program.

```
.CREAME POWER.FT
```

As soon as the Editor displays its prompt ($) to indicate that it is ready to receive your first instruction, type I (to put the Editor into text mode) and a carriage return and enter the program.

```
C FORTRAN DEMONSTRATION
C COMPUTE AND PRINT POWERS OF TWO
DIMENSION A(16)
WRITE(4,15)
15 FORMAT(1H ',POWER OF TWO')
   DO 20 N=1,16
      A(N)=2.*N
   20 CONTINUE
WRITE (4,25) (N,A(N),N=1,16)
25 FORMAT(1H ',2** ',I2,=' ',F10.1)
STOP
END
```

Now type CTRL/L to put the Editor in command mode and issue the EXIT instruction. The Editor will write file POWER.FT on your system device and return control to the monitor.

To compile and assemble the source program, use the COMPIL command, entering POWER.FT as the input file. If you wish to generate an annotated listing file and send it to the lineprinter, enter LPT: as the second output specification.

```
.COMPIL POWER,LPT:<POWERS.FT
```

The Compiler will send the binary file to SYS: (which it uses as a default device), adding an RL extension to indicate that the file contains relocatable FORTRAN code, and produce the following annotated listing on the lineprinter.

```
C FORTRAN DEMONSTRATION
0002 C COMPUTE AND PRINT POWERS OF TWO
0003 C DIMENSION A(16)
0004 15 WRITE (4,15)
0005 15 FORMAT(1H ',POWER OF TWO')
0005 DO 20 N=1,16
0006 20 A(N)=2.*N
0007 20 CONTINUE
0008 20 WRITE (4,25) (N,A(N),N=1,16)
0009 25 FORMAT(1H ',2** ',I2,=' ',F10.1)
0010 25 STOP
0011 25 END
```

7-2
FORTAN IV

To create a loader-image file—containing the program in absolute binary form and a copy of the exponentiation library function—use the LOAD command, specifying the RL file as input. To send a loader-symbol map—showing the areas of memory the program uses—to the lineprinter, enter LPT: as the second output specification.

*.LOAD POWER>LPT: <POWER.RL

The Load command creates a loader-image file (adding the LD extension) on SYS and the following map.

LOADER V24A 17-APR-79

SYMBOL VALUE LVL OVLY

ARGERR 00204 0 00
EXIT 00223 0 00
$MAIN 10000 0 00
11000 = 1ST FREE LOCATION

LVL OVLY LENGTH

0 00 10601

The optional loader symbol map lists all symbols defined in the loader image file. The LVL and OVLY entries apply only to OS/8, a superset of OS/78 FORTAN.

Following the alphabetical list of symbols, the loader prints the address of the first free memory location and the length, in octal words. This information is useful for optimizing memory requirements.

To execute this program, call the Run-Time System with the EXECUTE command, specifying POWER.LD for input.

*.EXECUTE POWER.LD

The program output will appear on the terminal screen.

POWER OF TWO

2** 1= 2.0
2** 2= 4.0
2** 3= 8.0
2** 4= 16.0
2** 5= 32.0
2** 6= 64.0
2** 7= 128.0
2** 8= 256.0
2** 9= 512.0
2** 10= 1024.0
2** 11= 2048.0
2** 12= 4096.0
2** 13= 8192.0
2** 14= 16384.0
2** 15= 32768.0
2** 16= 65536.0
FORTRAN programs are usually saved as loader-image files and then run with the EXECUTE command. To produce a loader-image file with a single command—that is, to instruct the Compiler/Assembler to chain automatically to the Loader—use the Compiler /L option.

```
COMPILE POWER.FT/L.
```

The FORTRAN system consists of the following components: the Compiler (plus Assembler), the Loader, the Run-Time system, and the FORTRAN library of functions. These components are described fully in Sections 7.1.1 through 7.1.4.

### 7.1.1 The COMPILER

The OS/78 FORTRAN IV Compiler/Assembler accepts one FORTRAN source language program or subroutine as input, examines each FORTRAN statement for validity, and produces a list of error messages plus a relocatable assembly-language version of the source program, along with an optional annotated source listing, as output.

If your program contains one or more subroutines, compile and assemble the main program and each subroutine separately, then use the Loader to link them together.

The FORTRAN Compiler terminates a compilation by chaining automatically to the Assembler.

To summon the FORTRAN compiler, use the COMPILE command. The Compiler accepts 1 to 3 output file specifications and 1 to 9 files for input, along with several run-time options. If you omit the device name, the Compiler assumes SYS. If you omit the extensions on the output filenames, the Compiler adds .RL and .LS to the binary file and the listing.

The format is

```
```

where:

- `file.RL` is the relocatable binary code
- `file.LS` is an optional listing file
- `file.MP` is an optional loader symbol map (to obtain it, you must chain to the Loader with the /L option)
- `file1.FT...9` is a single program unit—a main program or a subroutine—written as 1 to 9 files

The Compiler assigns an internal statement number (ISN) to each FORTRAN IV statement sequentially, in octal notation, starting with ISN 2 at the first FORTRAN statement. When the Compiler encounters an error during compilation, it prints a 2-character error code, followed by the ISN of the offending statement, on the terminal. An extended error message is printed below every erroneous statement in the annotated listing. Certain errors causes the Compiler to return immediately to the monitor, thereby preventing the output of a listing file.
7.1.1.1 Compiler Options - Compiler options are described in Table 7-1. If you chain automatically to the Loader (with the /L option) you can also add Loader options to the command line.

<table>
<thead>
<tr>
<th>Option</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>/G</td>
<td>Chain to the loader when assembly is complete and chain to the run-time system following creation of a loader image file (equivalent to the EXECUTE command) (see Section 7.1.3).</td>
</tr>
<tr>
<td>/L</td>
<td>Chain to the loader when assembly is complete to create a loader-image file. If the /L option is not specified, the system will return to the monitor upon completion.</td>
</tr>
<tr>
<td>/N</td>
<td>Suppress compilation of ISNs. This option reduces program memory requirements by two words per executable statement; however, it also prevents full error traceback at run time.</td>
</tr>
<tr>
<td>/Q</td>
<td>Optimize cross-statement subscripting during compilation. This option should not be requested when any variable that appears in a subscript is modified either by referencing a variable equivalent to it or via a SUBROUTINE or FUNCTION call (whether as an argument or through COMMON).</td>
</tr>
</tbody>
</table>

7.1.1.2 Compiler Error Messages - During pass 2, the Compiler displays error messages on the terminal as a 2-character message followed by the ISN of the erroneous statement. To suppress the messages, type CTRL/O at the terminal. If you request a listing, it will contain an extended error message following the erroneous statement. Except as noted, errors located by the Compiler do not halt processing. Error messages are described in Table 7-2.

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>AA</td>
<td>More than six subroutine arguments are arrays.</td>
</tr>
<tr>
<td>AS</td>
<td>Bad ASSIGN statement.</td>
</tr>
<tr>
<td>BD</td>
<td>Bad dimensions (too big or syntax) in DIMENSION, COMMON or type declaration.</td>
</tr>
<tr>
<td>BS</td>
<td>Illegal in BLOCK DATA Program.</td>
</tr>
<tr>
<td>CL</td>
<td>Bad COMPLEX literal.</td>
</tr>
<tr>
<td>CO</td>
<td>Syntax error in COMMON statement.</td>
</tr>
<tr>
<td>DA</td>
<td>Bad syntax in DATA statement.</td>
</tr>
<tr>
<td>DE</td>
<td>This type of statement illegal as end of DO loop (that is, GO TO, another DO).</td>
</tr>
</tbody>
</table>

(continued on next page)
<table>
<thead>
<tr>
<th>Error Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>DP</td>
<td>Bad DEFINE FILE statement.</td>
</tr>
<tr>
<td>DH</td>
<td>Hollerith field error in DATA statement.</td>
</tr>
<tr>
<td>DL</td>
<td>DATA list and variable list are not same length.</td>
</tr>
<tr>
<td>DN</td>
<td>DO-end missing or incorrectly nested. This message is not printed during pass 3, if it is followed by the statement number of the erroneous statement, rather than the ISN.</td>
</tr>
<tr>
<td>DO</td>
<td>Syntax error in DO or implied DO.</td>
</tr>
<tr>
<td>DP</td>
<td>DO loop parameter not integer or real.</td>
</tr>
<tr>
<td>EX</td>
<td>Syntax error in EXTERNAL statement.</td>
</tr>
<tr>
<td>GT</td>
<td>Syntax error in GO TO statement.</td>
</tr>
<tr>
<td>GV</td>
<td>Assigned or computed GO TO variable must be integer or real.</td>
</tr>
<tr>
<td>HO</td>
<td>Hollerith field error.</td>
</tr>
<tr>
<td>IE</td>
<td>Error reading input file. Control returns to the monitor.</td>
</tr>
<tr>
<td>IF</td>
<td>Logical IF statement cannot be used with DO, DATA, INTEGER, etc.</td>
</tr>
<tr>
<td>LI</td>
<td>Argument of logical IF is not type Logical.</td>
</tr>
<tr>
<td>LT</td>
<td>Input line too long, too many continuations.</td>
</tr>
<tr>
<td>MK</td>
<td>Misspelled keyword.</td>
</tr>
<tr>
<td>ML</td>
<td>Multiply defined line number.</td>
</tr>
<tr>
<td>MM</td>
<td>Mismatched parenthesis.</td>
</tr>
<tr>
<td>MO</td>
<td>Expected operand is missing.</td>
</tr>
<tr>
<td>MT</td>
<td>Mixed variable types (other than integer and real).</td>
</tr>
<tr>
<td>OP</td>
<td>Error writing output file. Control returns to the monitor.</td>
</tr>
<tr>
<td>OP</td>
<td>Illegal operator.</td>
</tr>
<tr>
<td>OT</td>
<td>Type / operator use illegal (for example, A.AND.B where A and/or B not typed Logical).</td>
</tr>
<tr>
<td>PD</td>
<td>Compiler stack overflow; statement too big and/or too many nested loops.</td>
</tr>
<tr>
<td>PH</td>
<td>Bad program header line.</td>
</tr>
<tr>
<td>QL</td>
<td>Nesting error in EQUIVALENCE statement.</td>
</tr>
<tr>
<td>QS</td>
<td>Syntax error in EQUIVALENCE statement.</td>
</tr>
<tr>
<td>RD</td>
<td>Attempt to redefine the dimensions of a variable.</td>
</tr>
<tr>
<td>RT</td>
<td>Attempt to redefine the type of a variable.</td>
</tr>
<tr>
<td>RW</td>
<td>Syntax error in READ/WRITE statement.</td>
</tr>
<tr>
<td>SF</td>
<td>Bad arithmetic statement function.</td>
</tr>
<tr>
<td>SN</td>
<td>Illegal subroutine name in CALL.</td>
</tr>
<tr>
<td>SS</td>
<td>Error in subscript expression, that is, wrong number, syntax.</td>
</tr>
<tr>
<td>ST</td>
<td>Compiler symbol table full, program too big. Causes an immediate return to the monitor.</td>
</tr>
<tr>
<td>SY</td>
<td>System error, that is, PASS20.SV or PASS22.SV missing, or no room on system for output file. Causes an immediate return to the monitor.</td>
</tr>
<tr>
<td>TD</td>
<td>Bad syntax in type declaration statement.</td>
</tr>
<tr>
<td>US</td>
<td>Undefined statement number. This message is not printed during pass 3. It is followed by the statement number of the erroneous statement, rather than the ISN.</td>
</tr>
<tr>
<td>VE</td>
<td>Version error. One of the compiler programs is absent from SYS or is present in the wrong version.</td>
</tr>
</tbody>
</table>
7.1.2 The LOADER

The FORTRAN IV Loader accepts up to 128 relocatable binary modules as input. It links all modules, including any routines from the FORTRAN library that the program may require, to form a loader-image file of the complete program—that is, an image of the binary code with absolute addresses assigned. This loader-image file can be passed directly to the Run-time system, which loads it into memory and executes it.

In addition to the loader-image file, the Loader generates an optional symbol map, which shows the areas the program occupies in memory.

You can call the Loader automatically by adding the /L option to the COMPILE command line.

The format of the Load command is

```
LOAD out:file.LD,out:map.LS<in:file1.RL...,file9.RL
```

where:

- `file.LD` is a loader image file
- `map.LS` is an optional symbol map
- `file1...,9.RL` are relocatable binary modules

OS/78 permits a maximum of nine input files in a command line. If you wish to specify more than nine relocatable modules for input, use the Loader /C (continue) option. This option enables you to add additional file names on the following line.

7.1.2.1 Specifying I/O Devices - The LOAD /G option causes the Loader to chain directly to the FORTRAN Run-Time System. If you use the /G option and terminate the LOAD command with the RETURN key, execution begins immediately. If you use the /G option and terminate the command with the ESCAPE key, the system calls a special program called the Command Decoder (described in Appendix D). The Command Decoder prints an asterisk (*) to indicate that it is ready to accept your special device and file I/O specifications. This feature makes it possible for you to change the devices used in a FORTRAN program, making the program device-independent.

For example, if the POWER.FT program were written with 3 as the output unit designation, that is,

```
WRITE (3,15)
```

the output generated by program execution would be sent to the line printer. However, if your system does not have a line printer available, you could change the output unit designation by typing

```
*LOAD POWER.LD<POWER.RL/G
```
Press the ESCape key (which echoes as a dollar sign) to call up the Command Decoder, which prompts with an asterisk. Then type /3=4 which assigns I/O unit 3 to the console terminal instead of the line printer. Pressing the ESCape key again executes the program and program output is sent to the terminal. The command line will appear as follows:

`.LOAD POWER. LD<POWER. RL/G ESC *3=4 ESC`

The Command Decoder program also allows you to store the output of an executed program in a file that has not been created at load time.

For example,

`.LOAD POWER. LD<POWER. RL/G ESC *RXA1:HOLD. TM<4 ESC`

will output the results of the program into a file called HOLD. TM on RXA1. Typing

`.TYPE RXA1:HOLD. TM`

will display the contents of this file, that is, the results of the program POWER. FT on the terminal screen.

For further information on I/O specifications at run-time, see the FORTRAN compiler.

7.1.2.2 Running Subprograms - The LOAD command is especially useful to link subprograms. Consider the program shown in Figure 7-1, which computes the volume of a regular polyhedron when given the number of faces and the length of one edge. It consists of a main program and a subroutine. The subroutine does the required computation, using a computed GO TO statement to determine whether the polyhedron is a tetrahedron, cube, octahedron, dodecahedron, or icosahedron, then transfers control to the proper arithmetic expression for performing the calculation.

MAIN PROGRAM:

```
C COMPUTE THE VOLUME OF A REGULAR POLYHEDRON GIVEN
C THE NUMBER OF FACES AND LENGTH OF ONE EDGE
COMMON NFACES,EDGE
S WRITE(4,10)
10 FORMAT(1H, 'TYPE IN NO. OF FACES AND ONE LENGTH EDGE')
READ(4,20) NFACES,EDGE
20 FORMAT(I2,F8.5)
CALL SOLVE
GO TO 5
STOP
END
```

Figure 7-1 Main Program and Subprogram for Calculating the Volume of a Regular Polyhedron
SUBPROGRAM SOLVE:

C SUBROUTINE TO SOLVE PROBLEM AND PRINT ANSWER
C CALLED SOLVE.FT
SUBROUTINE SOLVE
COMMON NFACES, EDGE, VOLUME
IF (NFACES.GT.20) GOTO 8
CUBED = EDGE**3
GO TO(6,6,6,1,6,2,6,3,6,6,6,4,6,6,6,6,6,6,6,6,5,6)*NFACES
1 VOLUME = CUBED**0.11785
   GO TO 20
2 VOLUME = CUBED
   GO TO 20
3 VOLUME = CUBED**0.47140
   GO TO 20
4 VOLUME = CUBED**2.66312
   GO TO 20
5 VOLUME = CUBED**2.18170
   GO TO 20
6 WRITE(4,10) NFACES
10 FORMAT(1H0,'NO REGULAR POLYHEDRON HAS',13,1X,'FACES')
   RETURN
20 WRITE(4,30) VOLUME
30 FORMAT(1H0,'VOLUME=',1F10.2)
   RETURN
8 WRITE(4,10)
40 FORMAT (1H0,'DO NOT SPECIFY MORE THAN 20 FACES')
   RETURN
END

Figure 7-1 (Cont.) Main Program and Subprogram for Calculating the Volume of a Regular Polyhedron

If the number of faces of the solid is other than 4, 6, 8, 12, or 20, or if more than 20 faces are specified, the subroutine displays an error message on the terminal. If the correct input parameters are typed in at the terminal keyboard, the calculation is performed, and the answer is displayed on the terminal screen.

When subprograms are used, the main program and the subprograms must be individually compiled. For the example program, both the main program and subprogram are compiled as follows:

.COMPILE MAIN.FT

and

.COMPILE SOLVE.FT

which create relocatable binary files.
The modules must now be loaded to make a single image file that can be executed. This is done as follows:

```
.LOAD POLY.LD<MAIN.RL,SOLVE.RL
```

This command creates a file POLY.LD and returns to the monitor. Execute the program by typing

```
.EXECUTE POLY.LD
```

which results in the program calling for the required input parameters as follows:

```
TYPE IN NO. OF FACES AND ONE LENGTH EDGE
```

Typing a "6" to represent a cube (preceded by a space since the field specification is I2) and "20" as one length edge in accordance with the field specification requirement as follows:

```
b6 20.0
```

results in the answer being displayed on the terminal screen, and a prompt for the next set of input parameters as follows:

```
VOLUME = 8000.00
TYPE IN NO. OF FACES AND ONE LENGTH EDGE
```

Do not specify blanks or zeroes for the NFACES and EDGES variables. Type CTRL/C to return to the monitor.

**7.1.2.3 LOADER Options** - Loader options are described in Table 7-3. The examples following the table illustrate their use.

### Table 7-3
**Loader Run-Time Options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>/C</td>
<td>Continue the current line of input on the next line of input. When specifying input files to the loader, there may be more than nine files in a command line. The /C option permits the additional files to be put on the following line. Terminate each continuation line (except the last) with a RETURN. Terminate the last line with an ESC.</td>
</tr>
<tr>
<td>/G</td>
<td>Treat the current line as the last line of input, and chain to the FORTRAN IV run-time system when finished (see Section 7.1.3).</td>
</tr>
<tr>
<td>/S</td>
<td>Include system symbols in the loader symbol map. System symbols are identified by an initial number sign (#). This option is only valid when a symbol map output file was specifically defined.</td>
</tr>
</tbody>
</table>
Examples:

\texttt{.LOAD POWER.RL}
\textbf{\small{Loads}} \texttt{POWER.RL} \textbf{\small{and}} \textbf{\small{produces}} \texttt{a} \textbf{\small{loader}} \textbf{\small{image}} \texttt{file POWER.LD.}}

\texttt{.LOAD RXA1:POWER.RL/G}
\textbf{\small{Loads}} \texttt{POWER.RL}, \textbf{\small{produces}} \textbf{\small{a} \texttt{loader image file POWER.LD}}, \textbf{\small{chains}} \texttt{to} \textbf{\small{the}} \textbf{\small{run-time system}}, \textbf{\small{and}} \textbf{\small{executes}} \texttt{the program.}

\texttt{.LOAD PROG.LD,RLOB:MAP<MAIN.RL,SUBA.RL}}
\textbf{\small{Loads and links}} \texttt{MAIN.RL} \textbf{\small{and}} \texttt{SUBA.RL} \textbf{\small{to produce}} \texttt{PROG.LD}, \textbf{\small{and}} \textbf{\small{produces}} \texttt{MAP.LS, a loader symbol map output file, on disk RLOB. Using the type command}}
\texttt{.TYPE RLOB:MAP}
\textbf{\small{will display}} \texttt{the MAP file} \textbf{\small{on the screen. Typing .EXE}} \texttt{PROG.LD} \texttt{will execute the program.}}

\texttt{.LOAD PROG.LD<PROG.RL,PROG1.RL<DATE>)(more file specifications)<ESCAPE>}}
\textbf{\small{Loads and links two input files to produce PROG.LD, then calls}} \texttt{a special system program for accepting file I/O specifications (see Section 7.2.3). The second ESCAPE executes the entire command.}}

7.1.2.4 Loader Error Messages - The Loader displays error messages on the terminal during generation of a loader-image file. Except where indicated in Table 7-4, Loader errors are fatal. When it encounters a fatal error condition, the Loader returns control to the monitor.

<table>
<thead>
<tr>
<th>Error Message</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>BAD INPUT FILE</td>
<td>An input file was not a relocatable binary module.</td>
</tr>
<tr>
<td>BAD OUTPUT DEVICE</td>
<td>The loader image file device was not a directory device, or the symbol map file device was a read-only device. The entire line is ignored.</td>
</tr>
<tr>
<td>EX</td>
<td>The symbol is referenced but not defined.</td>
</tr>
<tr>
<td>INVALID OS/78 OPTION</td>
<td>Attempt to use an option that is not available.</td>
</tr>
<tr>
<td>ME</td>
<td>Multiple entry. The symbol is multiply defined.</td>
</tr>
<tr>
<td>MIXED INPUT</td>
<td>The L option was specified on a line that contained some file other than a library file. The library file (if any) is accepted. Any other input file specification is ignored.</td>
</tr>
</tbody>
</table>

(continued on next page)
Table 7-4 (Cont.)
Loader Error Messages

<table>
<thead>
<tr>
<th>Error Messages</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>NO MAIN</td>
<td>No relocatable binary module contained the section #MAIN.</td>
</tr>
<tr>
<td>OVER CORE</td>
<td>The loader image requires more than 16K of memory.</td>
</tr>
<tr>
<td>OVER IMAG</td>
<td>Output file overflow in the loader image file.</td>
</tr>
<tr>
<td>OVER SYMB</td>
<td>Symbol table overflow. More than 253 (decimal) symbols in one FORTRAN job.</td>
</tr>
<tr>
<td>TOO MANY RALF FILES</td>
<td>More than 128 input files were specified.</td>
</tr>
</tbody>
</table>

7.1.3 The Run-Time System (FRTS)

The run-time system reads, loads, and executes a loader image file produced by the loader. It evaluates arithmetic and logical operations. It also configures a software I/O interface between the FORTRAN IV program and the OS/78 operating system, and then monitors program execution to direct I/O processes and identifies certain types of run-time errors.

The run-time system is automatically called to load and execute the loader image file produced by the loader whenever the /G option is specified to the loader. When chained to the loader, the run-time system reacts in one of two ways. If the LOAD command line was terminated by pressing the RETURN key, the program is executed. If the LOAD command line was terminated with an ESCape, a system program called the Command Decoder is called and indicates that it is running by printing an asterisk (*).

In response to the asterisk, you can supply file specifications to the run-time system. This allows a source program to be written that refers to I/O devices as integer constants or variables. Such a program may be compiled, assembled, and loaded into an image file. This image file may be run any number of times, each time specifying different actual peripheral devices. Thus logical unit 8 may refer in one run to the console terminal and in another run to a diskette file.

Of the nine I/O unit numbers available under FORTRAN IV, two are initially assigned to FORTRAN internal device handlers by the system:

<table>
<thead>
<tr>
<th>I/O Unit</th>
<th>Internal Handler</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Line printer</td>
<td>LA78 only.</td>
</tr>
<tr>
<td>4</td>
<td>Console terminal</td>
<td>Double buffered output, single character input.</td>
</tr>
</tbody>
</table>

The FORTRAN internal handlers listed above are not the same as the OS/78 device handlers. The FORTRAN internal handlers are designed for ASCII text only and will not transfer noncharacter data.

7-12
Additional unit numbers may be assigned, in addition to those listed above, to the FORTRAN internal device handlers by typing (in response to the asterisk generated by the Command Decoder):

\[ n = m \]

where:

\( n \)

is a different unit number (1 to 9) that is also to be assigned to that internal handler; and

\( m \)

is the I/O unit number (3 or 4) of the internal handlers.

This specification causes all program references to logical unit \( n \) to perform I/O to device \( m \). For example,

\[ /6 = 3 \]

assigns the FORTRAN internal line printer handler as I/O unit number 6, in addition to unit number 3.

\[ /3 = 4 \]

assigns I/O unit number 3 to the FORTRAN console terminal handler instead of the internal line printer handler.

I/O unit numbers may be assigned to OS/78 device handlers for nondirectory devices by typing (in response to the asterisk generated by the Command Decoder):

\[ \text{dev:}/n \]

where:

\( \text{dev:} \)

is the standard or assigned designation for any supported nondirectory device; and

\( n \)

is an I/O unit number (1 to 9).

Example:

\[ \text{LQP:}/3 \]

specifies the OS/78 LQP line printer handler to be used as device #3 instead of the FORTRAN internal line printer handler.

Existing directory device files may be assigned I/O unit numbers by typing (in response to the asterisk generated by the Command Decoder):

\[ \text{dev:file.ex}/n \]

where:

\( \text{dev:file.ex} \)

is the standard OS/78 designation for an existing directory device file; and

\( n \)

is an I/O unit number (1 to 9).
Example:

RXA1:FORIO.TM/4 Assigns unit number 4 to Diskette file FORIO.TM rather than to the FORTRAN internal console terminal handler, where FORIO.TM is an existing file on Diskette unit 1.

A directory device file that does not presently exist may be assigned a FORTRAN I/O unit number in the same manner by entering it as an output file on the specification line; however, only one such file may be created on any particular device. For example:

FORIO.TM</9 Assigns unit number 9 to file DSK:FORIO.TM, which has not been created at load time.

In any case, only one device or file specification is permitted on each line, and no more than six directory device files may be created by the FORTRAN program. Excess files after the sixth are accepted and written, but they will not be closed. If a file created by the program has the same file name and extension as a pre-existing file, the old file is automatically deleted when the new file is closed.

The Command Decoder "[n]" specification may be used to optimize storage allocation when assigning files that do not yet exist, where n is a decimal number that indicates the maximum expected length of the file, in blocks.

Each time a run-time I/O specification is terminated by pressing the RETURN key, the Command Decoder is recalled to accept another specification. When a specification is terminated by the ESCape key, the program is run.

The following examples illustrate the use of device and file specifications.

```
C WRITE.FT PROGRAM
DIMENSION ILT(200)
INTEGER ILT
C SPECIFY LOGICAL UNIT NUMBER FOR TTY AS 2 INSTEAD OF 4
1 WRITE (2,10)
10 FORMAT (1X,'WRITING AND READING ASCII SEQ. DATA FILE')
   DO 3 I=1,200
  3 ILT(I)=I**2
   WRITE (6,20) (ILT(I),I=1,200)
20 FORMAT (1H,///,20(10I7,/,))
   REWIND 6
C NOW READ DATA FILE DATA.DA FROM MASS STORAGE DEVICE
READ (8,20) ILT
C OUTPUT FILE TO TTY
WRITE (4,20) ILT
END
```

The above program raises 200 sequential numbers to the power of two. The following sequence of specifications are typed using the COMPILe command as follows:

```
*.COMPILE WRITE.FT/G *2=4
*RXA1!DATA.DA</8
```
The /G option in the command line will call the run-time system to execute the program. Pressing the ESCape key after the command calls the Command Decoder, allowing device and file specifications to be declared. In this case, the terminal (4) is assigned to unit number 2 while the load image file is assigned number 8. A file DATA.DA is created on diskette 1 that contains the results of the program. The contents of this file are then displayed on the terminal screen. The command and the specification strings are executed by the second ESCape.

The second example shows an output file RAY.DA being created on the diskette by PROG1.FT, and then being read from the diskette by PROG2.FT.

```fortran
C THIS PROGRAM WRITES A RECORD OF 400 VARIABLES
C INTO A FILE CALLED RAY.DA
DIMENSION RAY(400)
INTEGER RAY
DEFINE FILE 1 (1,400,U,J)
J=1
DO 5 I=1,400
 5 RAY(I)=2*I
WRITE (1/J) (RAY(I),I=1,400)
CALL EXIT
END
```

The above program writes a record into file RAY.DA. The following command and specification strings are typed to accomplish this.

```fortran
.COMPILE PROG1.FT/G *RAY.DA<1
C READ DIRECT ACCESS FILE RAY.DA FROM MASS STORAGE
C DEVICE CREATED BY PREVIOUS PROGRAM
DIMENSION RAY(400)
INTEGER RAY DEFINE FILE 1(1,400,U,J)
J=1
READ (1,J) (RAY(I),I=1,400)
100 FORMAT (1H,,//,40(10I6,/,))
C DUMP CONTENTS OF DATA FILE ONTO TTY
WRITE (4,100) RAY
CALL EXIT
END
```

The above program then reads out the results of PROG1.FT, and displays the contents of file RAY.DA on the terminal screen. This is done by typing the following command line.

```fortran
.COMPILE PROG2.FT/G *RAY.DA/1
```

Although existing files are specified as though they were input files and nonexistent files are specified as though they were output files, any file that has been assigned a unit number may be used for either input or output.

7.1.3.1 Runtime System Options - Run-time system option specifications are described in Table 7-5.
<table>
<thead>
<tr>
<th>Option</th>
<th>Operation</th>
</tr>
</thead>
</table>
| /C     | Carriage control switch. The first character on every output line is processed as a carriage control character by all FORTRAN internal handlers and also by the OS/78 handlers TTY and LPT. The first character on every output line is processed as data, in the same manner as any other character, by all OS/78 handlers except TTY and LPT. Entering a C option specification on the command line that assigns an I/O unit number to a particular handler reverses the processing of carriage control characters for that device. Thus,  

```
TEMP(2C)
```

assigns file DSK:TEMP. as I/O unit 2. The /C option causes the first character of every output line to be processed as a carriage control character. If C were not specified, these characters would be processed as data.  

```
/C/6=3
```

assigns the FORTRAN internal line printer handler as I/O unit 6, as well as unit 3. The first character of every line will be processed as a carriage control character on unit 3, and as a character of data on unit 6.  

| /E     | Ignore the following run-time system errors, any of which indicates that an error was detected earlier in the compilation loading process:  

1. Illegal subroutine call.  
2. Reference to an undefined symbol.  

The console terminal serves as FORTRAN I/O unit 4 for both input and output. Terminal input is automatically echoed on the console screen. In addition, the run-time system monitors the keyboard continually during execution of a FORTRAN program. Typing CTRL/C at any time causes an immediate return to the monitor. Typing CTRL/B branches to the system traceback routine, and then exits to the monitor. This traceback routine generates a printout, similar to the error traceback, including the current subroutine, the line number in the next higher level subroutine from which it was called, and so forth, to the main program. This facilitates locating infinite loops when debugging a program. The following additional special characters are recognized by the console terminal handler and processed as shown:  

- **DELETE** Deletes last character accepted.  
- **CTRL/U** Deletes current line of input.  
- **CTRL/Z** Signals end-of-file on input.  

7-16
Tentative output files (that is, files created by the FORTRAN program) are closed automatically upon successful completion of program execution provided that one of the following conditions occurs:

1. An END FILE statement referencing the file was executed. FRTS assigns a file length equal to the actual length of the file.

2. The last operation performed on the file was a write operation. FRTS proceeds as though an END FILE statement had been executed.

3. A DEFINE FILE statement referencing the file was executed but an END FILE statement was not executed. Upon completion of program execution, FRTS assigns a file length equal to the length specified in the DEFINE FILE statement.

Execution of a REWIND statement does not close a tentative file, nor does it modify the tentative file length.

7.1.3.2 Run-Time System Error Messages - The run-time system generates two classes of error messages. Messages listed in Table 7-6 identify errors that may occur during execution of a FORTRAN program and errors that may be encountered when the run-time system is reading a loader image file into memory in preparation for execution, or accepting I/O unit specifications. Except where indicated, all run-time system errors cause full traceback and an immediate return to the monitor. Nonfatal errors cause partial traceback, sufficient to locate the error, and execution continues.

The run-time system error traceback feature provides automatic printout of statement numbers ISNs corresponding to the sequence of executable statements that terminated in an error condition. At least one statement number is always printed. This number identifies the erroneous statement or, in certain cases, the last correct statement executed prior to the error. When a statement was compiled under the /N option, however, the system cannot generate meaningful statement numbers during traceback. When a statement is reached through any form of GOTO, the line number for error traceback is not reset. Thus, an error in such a line will give the number of the last executed line in the error traceback.

<table>
<thead>
<tr>
<th>Error Message</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>BAD ARG</td>
<td>Illegal argument to library function.</td>
</tr>
<tr>
<td>CAN'T READ IT!</td>
<td>I/O error on reading loader image file.</td>
</tr>
<tr>
<td>D.F. TOO BIG</td>
<td>Random access file requirements exceed available storage.</td>
</tr>
<tr>
<td>DIVIDE BY 0</td>
<td>Attempt to divide by zero. The resulting quotient is set to zero and execution continues.</td>
</tr>
</tbody>
</table>

(continued on next page)
## TABLE 7-6 (Cont.)
### Run-Time System Error Messages

<table>
<thead>
<tr>
<th>Error Messages</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>EOF ERROR</strong></td>
<td>End of file encountered on input.</td>
</tr>
<tr>
<td><strong>FILE ERROR</strong></td>
<td>Any of the following conditions occurred:</td>
</tr>
<tr>
<td></td>
<td>1. A file specified as an existing file was not found.</td>
</tr>
<tr>
<td></td>
<td>2. A file specified as a nonexistent file would not fit on the</td>
</tr>
<tr>
<td></td>
<td>designated device.</td>
</tr>
<tr>
<td></td>
<td>3. More than one nonexistent file was specified on a single device.</td>
</tr>
<tr>
<td></td>
<td>4. The file specification contained an asterisk (*) as name or</td>
</tr>
<tr>
<td></td>
<td>extension.</td>
</tr>
<tr>
<td><strong>FILE OVERFLOW</strong></td>
<td>Attempt to write outside file boundaries.</td>
</tr>
<tr>
<td><strong>FORMAT ERROR</strong></td>
<td>Illegal syntax in FORMAT statement.</td>
</tr>
<tr>
<td><strong>INPUT ERROR</strong></td>
<td>Illegal character received as input.</td>
</tr>
<tr>
<td><strong>I/O ERROR</strong></td>
<td>Error reading or writing a file, tried to read from an output device,</td>
</tr>
<tr>
<td></td>
<td>or tried to write on an input device.</td>
</tr>
<tr>
<td><strong>MORE CORE REQUIRED</strong></td>
<td>The space required for the program, the I/O</td>
</tr>
<tr>
<td></td>
<td>device handlers, the I/O buffers, and the monitor exceeds the available</td>
</tr>
<tr>
<td></td>
<td>memory.</td>
</tr>
<tr>
<td><strong>NO DEFINE FILE</strong></td>
<td>Direct access I/O attempted without a DEFINE FILE statement.</td>
</tr>
<tr>
<td><strong>NO NUMERIC SWITCH</strong></td>
<td>The referenced FORTRAN I/O unit was not specified to the run-time</td>
</tr>
<tr>
<td></td>
<td>system.</td>
</tr>
<tr>
<td><strong>NOT A LOADER IMAGE</strong></td>
<td>The first input file specified to the run-time system was not a loader</td>
</tr>
<tr>
<td></td>
<td>image file.</td>
</tr>
<tr>
<td><strong>OVERFLOW</strong></td>
<td>Result of a computation exceeds upper bound</td>
</tr>
<tr>
<td></td>
<td>for that class of variable. The result is</td>
</tr>
<tr>
<td></td>
<td>set equal to zero and execution continues.</td>
</tr>
<tr>
<td><strong>PARENS TOO DEEP</strong></td>
<td>Parentheses nested too deeply in FORMAT statement.</td>
</tr>
<tr>
<td><strong>SYSTEM DEVICE ERROR</strong></td>
<td>I/O failure on the system device.</td>
</tr>
<tr>
<td><strong>TOO MANY HANDLERS</strong></td>
<td>Too many I/O device handlers are resident in memory, or files have</td>
</tr>
<tr>
<td></td>
<td>been defined on too many devices.</td>
</tr>
<tr>
<td><strong>UNIT ERROR</strong></td>
<td>I/O unit not assigned, or incapable of executing the requested</td>
</tr>
<tr>
<td></td>
<td>operation.</td>
</tr>
<tr>
<td><strong>UNKNOWN INTERRUPT</strong></td>
<td>A hardware interrupt occurred from a device that the run-time system</td>
</tr>
<tr>
<td></td>
<td>is not using.</td>
</tr>
<tr>
<td><strong>USER ERROR</strong></td>
<td>Illegal subroutine call, or call to undefined subroutine. Execution</td>
</tr>
<tr>
<td></td>
<td>continues only if the E option was requested.</td>
</tr>
</tbody>
</table>
7.1.4 The Library

The OS/78 FORTRAN IV system contains a general purpose FORTRAN library named FORLIB.RL. FORLIB.RL contains mathematical functions and miscellaneous subroutines.

Library functions and subroutines are called in the same manner as user written functions and subroutines. Section lists the library components that are available to FORTRAN programs and illustrates calling sequences, where necessary.

7.2 THE FORTRAN IV SOURCE LANGUAGE

A FORTRAN source program consists of statements using the language elements and the syntax described in this chapter. A statement performs one of the following functions:

- Causes operations such as multiplication, division, and branching to be carried out
- Specifies the type and format of data being processed
- Specifies the characteristics of the source program

FORTRAN statements are composed of keywords (that is, words that the FORTRAN compiler recognizes) that you use with elements of the language set. These elements are constants, variables and expressions. There are two basic types of FORTRAN statements: executable and nonexecutable.

Executable statements specify the action of the program; nonexecutable statements describe the characteristics and arrangement of data, editing information, statement functions, and subprograms that you may include in the program. The compilation of executable statements results in the creation of executable code. Nonexecutable statements provide information only to the compiler; they do not create executable code.

The OS/78 FORTRAN IV language generally conforms to the specifications for American National Standard FORTRAN X3.9-1966. The following enhancements are included in OS/78 FORTRAN:

- You may use any arithmetic expression as an array subscript. If the expression is not of integer type, FORTRAN converts it to integer form.
- You may use alphanumeric literals (character strings delimited by apostrophes or quotation marks) in place of Hollerith constants.
- The statement label list in an ASSIGNED GO TO statement is optional.
- The following Input/Output (I/O) statements have been added:
  
  DEFINE FILE Device-oriented I/O
  READ (u'r)
  WRITE (u'r) Unformatted Direct Access I/O
You may use any arithmetic expression as the initial value, increment, or limit-parameter in the DO statement, or as the control parameter in the COMPUTED GO TO statement.

OS/78 FORTRAN permits constants and expressions in the I/O lists of WRITE statements.

All FORTRAN statements are listed in Section 7.13.

In this chapter, the FORTRAN language statements are grouped into eight categories, each of which is described in a separate section. The name, definition, and section references for each statement category are given in Table 7-7.

<table>
<thead>
<tr>
<th>Category</th>
<th>Function</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assignment Statement</td>
<td>Assign values to named variables and array elements.</td>
<td>7.5</td>
</tr>
<tr>
<td>Specification Statement</td>
<td>Declare the properties of variables, arrays, and functions.</td>
<td>7.6</td>
</tr>
<tr>
<td>DATA Statements</td>
<td>Assign initial values to variables and array elements.</td>
<td>7.7</td>
</tr>
<tr>
<td>Control Statements</td>
<td>Determine order of execution of the object program and terminate its execution.</td>
<td>7.8</td>
</tr>
<tr>
<td>Subprogram Statements</td>
<td>Define functions and subroutines.</td>
<td>7.9</td>
</tr>
<tr>
<td>Input/Output Statements</td>
<td>Transfer data between internal storage and specified input/output devices.</td>
<td>7.10</td>
</tr>
<tr>
<td>FORMAT Statements</td>
<td>Specify formats for data on input/output.</td>
<td>7.11</td>
</tr>
</tbody>
</table>

7.2.1 The FORTRAN Character Set

The FORTRAN character set consists of:

- The upper case letters A through Z
- The numerals 0 through 9
- The special characters in Table 7-8
Table 7-8
FORTRAN Special Characters

<table>
<thead>
<tr>
<th>Character</th>
<th>Name</th>
<th>Character</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>Space</td>
<td>( )</td>
<td>Parentheses</td>
</tr>
<tr>
<td></td>
<td>Tab</td>
<td>,</td>
<td>Comma</td>
</tr>
<tr>
<td>=</td>
<td>Equals</td>
<td>.</td>
<td>Decimal Point</td>
</tr>
<tr>
<td>+</td>
<td>Plus</td>
<td>'</td>
<td>Apostrophe</td>
</tr>
<tr>
<td>-</td>
<td>Minus</td>
<td>&quot;</td>
<td>Quote</td>
</tr>
<tr>
<td>*</td>
<td>Asterisk</td>
<td>$</td>
<td>Dollar Sign</td>
</tr>
<tr>
<td>/</td>
<td>Slash</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

You may type other printable characters such as %, _, and @ only as part of Hollerith constants, alphanumeric literals, or comments.

7.2.2 Elements of a FORTRAN Program

A FORTRAN program consists of FORTRAN statements and optional comments. You group the statements into logical units called program units (a program unit being a sequence of statements which you terminate with an optional END statement).

A program unit can be either a main program or a subprogram. One main program and possibly one or more subprograms form the executable program.

7.2.2.1 Statements - Statements are grouped into two general classes: executable and nonexecutable. Executable statements are the action statements of the program; nonexecutable statements describe data arrangement and data characteristics. Nonexecutable statements may also contain editing and data conversion information.

A program consists of a series of statements, written one statement to a line. (A line is a string of up to 72 characters.) If a statement is too long to fit on one line, you may continue it on up to five additional lines (called continuation lines). (For further information, see Section 7.2.3.4, Continuation Indicator Field.)

A statement can refer to another statement. FORTRAN refers to such a statement by an integer number (called a label) ranging from 1 to 99999. Such a statement is most often referenced for the information it may contain or so that program execution can continue at that statement.

7-21
7.2.2.2 Comments - Comments are lines of text which document program action, indicate program sections and processes, and provide greater ease in reading the source program listing by identifying variables.

The FORTRAN compiler ignores comments; the comments exist only so that you can document what the program is doing.

7.2.3 FORTRAN Lines

A FORTRAN line consists of four fields:
  - Statement Label Field
  - Continuation Indicator Field
  - Statement Field
  - Identification Field

You may skip any of these fields when entering statements, but, except for the identification field, the spaces allotted to each field must remain present. In the case of the identification field, you may type a carriage return before reaching it.

Each printing space represents a single character. The following sections describe the entering of the source program and the information contained in each field.

7.2.3.1 Using a Text Editor - When creating a source program with the OS/78 text editor, you type the lines on a "character-per-column" basis. You may also use the TAB character to format lines.

NOTE

The text editor and terminal advance the terminal print carriage to a predefined print position when you type a TAB. This action, however, is not related to FORTRAN's interpretation of the TAB character. The FORTRAN system interprets a TAB as one character, not the number of characters (up to eight) that it will print.

For example, you may format the following lines in either of the ways shown:

\[
\begin{align*}
\text{C- INITIALIZE ARRAYS} & \quad \text{or} \quad \text{C INITIALIZE ARRAYS} \\
10- \text{ W=3} & \quad \text{or} \quad 10 \text{ W=3} \\
- \text{SEL(1)=111200022D0} & \quad \text{or} \quad \text{SEL(1)=111200022D0}
\end{align*}
\]

where:
- represents a TAB, and
- represents a space character.
Use space characters in a FORTRAN statement to improve the legibility of a line. The compiler ignores all spaces in a statement field except those within a Hollerith constant or alphanumeric literal.

Example:

GO TO and GOTO are equivalent.

The compiler also ignores a \texttt{\textbackslash T A B} in a statement field; it regards a \texttt{\textbackslash T A B} to be the same as a space. However, in the compiler generated source listing, FORTRAN prints the character following the \texttt{\textbackslash T A B} at the next tab stop (located at columns 9,17,25,33, etc.).

7.2.3.2 Statement Label Field - A statement label is a number which FORTRAN uses to reference one statement from another statement.

A statement label (sometimes also called a statement number) consists of from one to five decimal digits ranging from 1 through 99999. Place this label in the first five positions of a statement's first line. Any source program statement that is referenced by another statement must have a statement number.

FORTRAN ignores spaces and leading zeros preceding the statement label, e.g., FORTRAN interprets the following as statement label 105:

\begin{verbatim}
  105
  00105
  105
\end{verbatim}

You may assign statement numbers in any order; however, each statement number must be unique in the program or subprogram. In contrast, a main program and a subprogram may contain identical statement numbers. In this case, FORTRAN understands that reference to these numbers means the numbers in the program unit in which the reference is made.

Example:

Assume that the main program and a subprogram both contain statement number 105. A GOTO statement in the main program will refer to statement number 105 in the main program, not to statement 105 in SUB1. A GOTO in SUB1 will transfer control to 105 in SUB1.

An all-zero statement label is illegal.

You cannot label non-executable statements other than FORMAT statements.

When you type a source program with a terminal, an initial \texttt{\textbackslash T A B} skips over the label and continuation field.

7.2.3.3 Comment Indicator - A comment indicator tells FORTRAN that the text on a line is a comment and that, therefore, it should not process that line.

Type the letter C in column one to indicate that the line is a comment. The compiler will print the contents of that line in the source program listing. However, it ignores the line when it compiles the program.
The following are restrictions on comments.

- All comment lines must begin with the letter C in column one.
- You cannot continue comment lines; consequently each comment line must begin with a C.
- Unlike other statements, the text of a comment can begin in the second space of a line.
- Comment lines must not intervene between a statement's initial line and its continuation line (or lines), or between successive continuation lines.

7.2.3.4 Continuation Indicator Field - A continuation indicator tells FORTRAN that the text on that line is part of the same statement as the preceding line.

You must reserve column six of a FORTRAN line for the continuation indicator even if you do not type a continuation indicator.

FORTRAN defines any character except a space in column 6 to be a continuation indicator.

The following are rules for using continuation indicators:

- You may divide a statement into distinct lines at any point.
- You may precede the continuation indicator with space characters only; you may not precede it with a \( \text{TAB} \) as an initial \( \text{TAB} \) skips over the continuation field.
- The characters beginning in column seven of a continuation line are considered to follow the last character of the previous line as if there were no break at that point.
- You may enter no more than 5 continuation lines for one statement.
- You cannot continue comment lines.
- A comment line must not intervene between a statement's initial line and its continuation line (or lines), or between successive continuation lines.
- You cannot assign statement numbers to continuation lines.

7.2.3.5 Statement Field - Type the text of a FORTRAN statement in columns 7 through 72. A \( \text{TAB} \) may precede the statement field rather than spaces. Note that because the compiler ignores \( \text{TAB} \)'s and spaces (except in Hollerith constants and alphanumeric literals), you can space the text of the statement in any way desired for maximum legibility.
7.2.3.6 Identification Field - Type a sequence number or other such identifying information in columns 73-80 of any line in a FORTRAN program. FORTRAN ignores the characters in this field.

NOTE
The FORTRAN compiler ignores text in these positions. Moreover, FORTRAN does not print a warning message if you accidently type text in this field. This is sometimes the source of inexplicable errors.
You might use this feature when typing punched card input. It is seldom used with terminals.

7.2.4 Blank Lines
You may insert lines consisting only of blanks, \texttt{tab}s, or no characters anywhere in your source program except immediately preceding a continuation line. You would use a blank line to improve the readability of a source listing; the FORTRAN compiler ignores them.

7.2.5 Line Format Summary
The fields and the columns in which they may appear are listed in Table 7-9.

<table>
<thead>
<tr>
<th>Field</th>
<th>Column</th>
</tr>
</thead>
<tbody>
<tr>
<td>Statement Label</td>
<td>1 through 5</td>
</tr>
<tr>
<td>Continuation Indicator</td>
<td>6</td>
</tr>
<tr>
<td>Statement</td>
<td>7 through 72</td>
</tr>
<tr>
<td>Identification</td>
<td>73 through 80</td>
</tr>
</tbody>
</table>

This example shows the placement of fields (the numbers represent column numbers):

1 67

\begin{verbatim}
DIMENSION A(12),B(10,10),C(13,13),D(17,00000001
121,5)
10 READ (1,100005) (A,B,C,D) 00000002
C THE DATA IS STORED ON DECTAPE; USE THE FORTRAN RUN 03
C TIME SYSTEM TO ASSIGN LUN 1 TO DTA:1
CALL UPDATE(A,D) 00000005
IF (.NOT. END) GO TO 10 00000006
\end{verbatim}
7.3 FORTRAN STATEMENT COMPONENTS

The elements of FORTRAN statements are:

- Constants
  A constant is a fixed, self-describing value.

- Variables
  A variable is a symbolic name that represents a stored value.

- Arrays
  An array is a group of variables that you may refer to individually or collectively. The individual values are called array elements. Use a symbolic name to refer to the array.

- Expressions
  An expression can be a constant, variable, array element, or function reference. It may also be a combination of those components and certain other elements (called operators). The operators indicate computations which FORTRAN will perform on the values represented by those components. The result of the computation is a single value.

- Function References
  A function reference is the name of a function (often followed by a list of arguments). After FORTRAN performs the computation indicated by the function definition, it substitutes the computed value in place of the function reference.

7.3.1 Symbolic Names

You use symbolic names to identify many entities within a FORTRAN program unit. Symbolic names consist of a combination of from one to six alphanumeric characters. If you use more than six characters in a symbolic name, FORTRAN reads only the first six.

The first letter of a symbolic name must be a letter. The special characters listed in Table 7-8 may not appear in symbolic names.

Examples of valid and invalid symbolic names are:

<table>
<thead>
<tr>
<th>Valid</th>
<th>Invalid</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUMBER</td>
<td>5Q</td>
</tr>
<tr>
<td>K9</td>
<td>B.4</td>
</tr>
</tbody>
</table>

_table_ (Begins with a numeral)
_(Contains a special character)_

Table 4-1 indicates the types of variables which FORTRAN identifies by symbolic names.

Except as specifically mentioned in this manual, you may not use the same symbolic name to identify more than one FORTRAN entity.

Each variable indicated as "Typed" in Table 7-10 has a data type. The means of specifying the data type of a name are presented in Sections 7.3.2, Data Types, and 7.6.1, Type Declaration Statements.

7-26
Within a subprogram, you may use symbolic names as dummy arguments. A dummy argument may represent a variable, array, array element, constant, expression, or subprogram. However, all subprograms must be uniquely named.

<table>
<thead>
<tr>
<th>Entity</th>
<th>Typed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variables</td>
<td>yes</td>
</tr>
<tr>
<td>Arrays</td>
<td>yes</td>
</tr>
<tr>
<td>Arithmetic statement functions</td>
<td>yes</td>
</tr>
<tr>
<td>Processor-defined functions</td>
<td>yes</td>
</tr>
<tr>
<td>FUNCTION subprograms</td>
<td>yes</td>
</tr>
<tr>
<td>SUBROUTINE subprograms</td>
<td>no</td>
</tr>
<tr>
<td>Common blocks</td>
<td>no</td>
</tr>
<tr>
<td>Block data subprograms</td>
<td>no</td>
</tr>
</tbody>
</table>

### 7.3.2 Data Types

The data type of a FORTRAN element may be inherent in its construction, implied by convention, or you may declare it explicitly. The data types available in FORTRAN, and their definitions, are listed in Table 7-11.

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEGER</td>
<td>A whole number.</td>
</tr>
<tr>
<td>REAL</td>
<td>A decimal number; it can be a whole number, a decimal fraction, or a combination of the two.</td>
</tr>
<tr>
<td>LOGICAL</td>
<td>The logical value &quot;true&quot; or &quot;false&quot;.</td>
</tr>
<tr>
<td>OCTAL</td>
<td>An integer number in radix 8.</td>
</tr>
</tbody>
</table>

### 7.3.3 Constants

A constant represents a fixed value; that is, a constant can represent numeric values, logical values, or character strings. You can specify five types of constants in an OS/78 FORTRAN program: integer, real, octal, logical, and Hollerith.
7.3.3.1 Integer Constants - An integer constant is a whole number with no decimal point. It may have a leading sign.

Format:

snn

where:

nn is a string of from 1 to 7 decimal digits, and
s is an optional algebraic sign.

In OS/78 FORTRAN, an integer constant is a whole signed or unsigned number which contains no more than 7 decimal digits. Integer constants must fall within the range \(-2^{**}23\) to \(2^{**}23\)-1 (-8,388,608 to 8,338,607). When you use integer constants as subscripts, FORTRAN uses them at modulo \(2^{**}12\) (4,096 decimal).

FORTRAN ignores leading zeros in integer constants.

Precede a negative integer constant by a minus symbol. A plus symbol is optional before a positive number because FORTRAN assumes an unsigned constant to be positive; e.g., +27 and 27 are identical.

With the exception of a plus or minus sign, an integer constant cannot contain any character other than the numerals 0 through 9. Specifically, embedded commas and decimal points are not allowed.

Examples:

<table>
<thead>
<tr>
<th>Valid Integer Constants</th>
<th>Invalid Integer Constants</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>99999999999 (Too large)</td>
</tr>
<tr>
<td>-127</td>
<td>3.14 (Decimal point and</td>
</tr>
<tr>
<td>+32123</td>
<td>32,767 (comma not allowed)</td>
</tr>
</tbody>
</table>

7.3.3.2 Real Constants - A decimal real constant is a string of decimal digits with a decimal point. An exponential real constant is a decimal real constant followed by an exponent.

A Decimal Real Constant is a decimal number. It may have a leading sign.

Format:

s.nn
snn.nn
snn.

where:

nn is a string of numeric characters.
. is a decimal point.
s is an optional algebraic sign.

A decimal real constant is a string of decimal digits with a decimal point. Note that you do not always have to type a number following the decimal point, but you must always type the decimal point. The decimal point can appear anywhere in the digit string.
FORTRAN IV

FORTRAN does not limit the number of digits in a decimal real constant, but only the leftmost six digits are significant. For example, in the constant 0.000012345678, all of the non-zero digits are significant (note that FORTRAN only stores 0.000012). However, in the constant 000507, the first three zeros are not significant.

You must precede a negative constant with a minus sign. The plus sign is optional preceding a positive real constant.

Except for algebraic signs and a decimal point, a real decimal constant cannot contain any character other than the numerals 0 through 9.

Examples:

<table>
<thead>
<tr>
<th>Valid Real Constants</th>
<th>Invalid Real Constants</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.14159</td>
<td>1,234,567 (Commas not allowed)</td>
</tr>
<tr>
<td>71712.</td>
<td>879877399. (Too large)</td>
</tr>
<tr>
<td>-.00127</td>
<td>100 (Decimal point missing)</td>
</tr>
<tr>
<td>0.0</td>
<td></td>
</tr>
</tbody>
</table>

An exponential real constant is a decimal real constant followed by a decimal exponent.

Format:

mmEsnm

where:

- mm is an integer or real constant,
- nn is a 1- to 3-digit integer constant,
- E indicates that the constant is an exponential real constant, and
- s is an algebraic sign.

An exponential real constant is a decimal number which you type in scientific notation, that is, in powers of ten. The number, nn, represents a power of 10 by which the preceding real or integer constant is to be multiplied (e.g., 1E6 represents the value 1.0 x 10**6). The magnitude of a real constant cannot be smaller than 10**-615 nor greater than 10**615. The number mm is an integer or real constant.

A real constant occupies three words (i.e., six bytes) of storage. FORTRAN interprets this number as having a degree of precision slightly greater than seven decimal digits.

In OS/78 FORTRAN, a real exponential constant need not contain a decimal point.

A minus symbol must appear between the letter E and a negative exponent; a plus symbol is optional for a positive exponent.

Except for algebraic signs, a decimal point, and the letter E, a real exponential constant cannot contain any character other than the numerals 0 through 9. However, you may omit the decimal point if the number does not have a fractional part.

7-29
Examples:

<table>
<thead>
<tr>
<th>Valid Real Constants</th>
<th>Invalid Real Constants</th>
</tr>
</thead>
<tbody>
<tr>
<td>2E-3</td>
<td>-47E645 (Too large)</td>
</tr>
<tr>
<td>+5.0E3</td>
<td>325E-801 (Too small)</td>
</tr>
<tr>
<td>5E3.2</td>
<td>(decimal point misplaced)</td>
</tr>
</tbody>
</table>

7.3.3.3 Logical Constants - A logical constant specifies a logical value, that is, "true" or "false". Therefore, there are only two possible logical constants. They are:

`.TRUE.`

and

`.FALSE.`

NOTE

You may abbreviate `.TRUE.` and `.FALSE.` as `.T.` and `.F.`

You must type the delimiting periods as they are part of each constant.

Only logical operators can operate on logical constants.

7.3.3.4 Octal Constants - An octal constant is a string of octal digits (0-7 only) preceded by the letter O.

Format:

`DATA/Onum/`  

where:

- `num` is an octal number, and
- `O` identifies the number as an Octal constant.

An octal constant is a digit string (0-7 only) which you may use only in `DATA` statements to enter numbers in radix eight. An octal constant may be of any length, but the FORTRAN compiler uses only the 12 low order digits.

You generally use octal constants to set bits for masking purposes.

Examples:

`DATA JOB/O1032/`  
`DATA BASE /07777/`

NOTE

The character following the first `/` in each of these examples is the letter O, not a zero.
FORTRAN IV

7.3.3.5 Hollerith Constants and Alphanumeric Literals - A Hollerith constant is a string of ASCII characters preceded by 1) a character count, and 2) the letter H.

Format:

\[ nH\text{ccc...c} \]

where:

- \( n \) is an unsigned, non-zero integer constant indicating the number of characters in the string (including spaces and tabs),
- \( c \) is any ASCII character, and
- \( H \) identifies this as a Hollerith constant.

A Hollerith constant is a string of alphanumeric and/or special characters preceded by a number which states how many characters are in the constant and the letter H. You may use any ASCII character (including those which are not part of the FORTRAN character set).

Hollerith constants have no data type. They assume the data type of the context in which they appear.

Examples:

\begin{table}
\begin{tabular}{|l|l|}
\hline
\textbf{Valid} & \textbf{Invalid} \\
\textbf{Hollerith Constants} & \textbf{Hollerith Constants} \\
\hline
16HTODAY'S DATE IS: 1H & 3HABCD (wrong number of characters. This will be stored as ABC.) \\
\hline
\end{tabular}
\end{table}

An alphanumeric literal is a string of ASCII characters delimited with apostrophes or quotation marks.

Format:

\['\text{ccc...c}'\]
\['"\text{ccc...c}"'\]

where:

- \( c \) is a printable ASCII character, and you must type both delimiting apostrophes or quotes.

An Alphanumeric literal is an alternate form of Hollerith constant. Like Hollerith constants, you may use any ASCII character (including those which are not part of the FORTRAN character set).

Alphanumeric literals have no data type. They assume the data type of the context in which they appear.

If you need to type an apostrophe within an alphanumeric literal, type it as two consecutive apostrophes.

Examples:

\[''CHANGE PRINTER PAPER TO PREPRINTED FORM NO. 721''\]
\[''TODAY''S DATE IS: ''\]
You may use a quotation mark (") instead of an apostrophe. However, you may not mix quotation marks and apostrophes. For example, the following literal is not allowed:

"THIS IS A MIXED LITERAL"

but you may type

"THIS ISN'T A MIXED LITERAL"

7.3.4 Variables

A variable is a symbolic name that FORTRAN associates with a storage location. (The FORTRAN compiler assigns the storage locations.) The value of the variable is the value currently stored in that location; you can only change that value by assigning a new value to the variable with an assignment statement.

FORTRAN classifies variables by data type, in the same manner as constants. The data type of a variable indicates

- The type of data it represents,
- Its precision, and
- Its storage requirements.

You may specify the data type of a variable either by type declaration statements (see Section 7.6.1), or by FORTRAN default typing rules (Section 7.3.4.2).

FORTRAN associates two or more variables with each other when each variable uses the same storage location; or, partially associates variables when part (but not all) of the storage which one variable uses is the same as part or all of the storage which another variable uses. You create associations and partial associations with:

- COMMON statements,
- EQUIVALENCE statements, and
- Actual and dummy arguments in subprogram references.

A variable is defined if the storage with which it is associated contains a datum of the same type. You can define a variable prior to program execution by typing a DATA statement or during execution by means of assignment or input statements.

Before you assign a value to a variable, it is an undefined variable, and you should not reference it except to assign a value to it. If you reference an undefined variable, an unknown value (garbage) will be obtained.

If you associate variables of differing types with the same storage location, then defining the value of one variable (for example, by assignment) causes the value of the other variable to become not defined.
7.3.4.1 Data Type Specification - Declaration statements (Section 7.6.1) associate given variables with specified data types. For example:

```fortran
INTEGER VAR1
```

This statement indicates that FORTRAN will associate the integer variable VAR1 with a 3-word storage location.

You can only explicitly declare the data type of a variable once in a program unit.

An explicit specification takes precedence over default specification.

7.3.4.2 Default Data Types - FORTRAN assumes all variables having names beginning with I, J, K, L, M, or N represent integer data; variables having names beginning with any other letter are real variables. For example:

<table>
<thead>
<tr>
<th>Real Variables</th>
<th>Integer Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALPHA</td>
<td>KOUNT</td>
</tr>
<tr>
<td>BETA</td>
<td>ITEM</td>
</tr>
<tr>
<td>TOTAL</td>
<td>NTOTAL</td>
</tr>
</tbody>
</table>

7.3.5 Arrays

An array is a group of contiguous storage locations which you reference with a single symbolic name, the array name. You reference the individual storage locations, called array elements, by a subscript appended to the array name.

An array can have from one to seven dimensions.

The following FORTRAN statements establish arrays:

- Type declaration statements (Section 7.6.1),
- DIMENSION statements (Section 7.6.2), and
- COMMON statements (Section 7.6.3).

Each of these statements defines

- The name of the array,
- The number of dimensions in the array, and
- The number of elements in each dimension.
7.3.5.1 **Array Declarations** - Use an array declaration to instruct FORTRAN to reserve storage for an array.

Format:

```
[[typ]] a(d [,d] ...)
```

where:

- `[[typ]]` is a data type declaration,
- `a` is the array name, and
- `d` is a number specifying the number of elements in that part of the array.

An array is a group of variables that have the same symbolic name; you address the elements of the array by means of a subscript.

Declare a variable to be an array by specifying the symbolic name which identifies the array within a program unit and which indicates the properties of that array. The number of dimension declarators `d` indicates the number of dimensions in the array. The minimum number of dimensions is one and the maximum number is seven.

You must declare the size (that is, the number or elements) of an array in order to reserve the needed amount of locations in which to store the array. The value of a dimension declarator specifies the number of elements in that dimension. For example, a dimension declarator value of 50, for example, `TABLE(50)`, indicates that the dimension contains 50 elements. The dimension declarators can be constant or variable.

The rules governing the dimensioning of arrays are as follows (characters enclosed within parentheses represent subscripted characters). In the equation:

```
L(n)=M(l)[1+M(2)+M(3)+M(2)M(3)M(4)...M(n-1)m(n)]
```

let:

- `L` = length of the entire array
- `n` = total number of dimensions in the array
- `M(i)` = maximum subscript for each dimension in the array, where `i` specifies which dimension in the array is being referenced.

In the above equation, `L` must not exceed 4095 in any case.

For example,

```
L(1) = M(l)<4096
L(2) = M(l)[1+M(2)]<4096
L(3) = M(l)[1+M(2)+M(2)M(3)]<4096
```

etc.

In the above equation, `L` must not exceed 2047 when transmitting arrays, individual arrays, elements, or subportions of an array to subprograms.

For example,

```
L(1) = M(l)<2047
L(2) = M(l)[1+M(2)]<2047
L(3) = M(l)[1+M(2)M(3)]<2047
```

etc.
The number of elements in an array is always equal to the product of the number of elements in each dimension. More specifically, the array IAB dimensioned as (3,4) has 12 elements (3 x 4 = 12) and takes 48 words of storage. Although FORTRAN stores arrays as a series of sequential storage locations, you may best visualize and reference arrays as if they were a single or multi-dimensional rectilinear matrices, dimensioned on a row, column, and plane basis. For example, Figure 7-2 represents a 3-row, 3-column, 2-plane array.

![Figure 7-2 Array Representation](image)

Specify the size of an array by an array declaration written as a subscripted array name. In an array declaration, however, each subscript quantity is a dimension of the array and must be either an integer variable or an integer constant.

An array name can appear in only one declaration statement within a program unit.

Use variable dimension declarations to define adjustable arrays (see Section 7.3.5.6).

7.3.5.2 Array Storage (Order of Subscript Progression) - OS/78 FORTRAN always stores arrays in memory as a linear sequence of values. For example, FORTRAN stores a 1-dimensional array with its first element in the first storage location and its last element in the last storage location of the sequence. FORTRAN stores a multi-dimensional array such that the leftmost subscripts vary most rapidly. For example, in the array ARRAY(3,2,2) the progression is:

\[
\begin{aligned}
&\text{ARRAY}(1,1,1) \\
&\text{ARRAY}(2,1,1) \\
&\text{ARRAY}(3,1,1) \\
&\text{ARRAY}(1,2,1) \\
&\text{ARRAY}(2,2,1) \\
&\text{ARRAY}(3,2,1) \\
&\text{ARRAY}(1,1,2) \\
&\text{ARRAY}(2,1,2) \\
&\text{ARRAY}(3,1,2) \\
&\text{ARRAY}(1,2,2) \\
&\text{ARRAY}(2,2,2) \\
&\text{ARRAY}(3,2,2)
\end{aligned}
\]

This is called the "order of subscript progression". For example, consider in Figure 7-3 the array declarators and the arrays that they create.
The arrows labeled "memory positions" show the order in which FORTRAN stores information in memory. This order is critically important when you use an unsubscripted array name in a READ or WRITE statement as this is the order in which FORTRAN fills memory or prints data.

7.3.5.3 Subscripts - A subscript is the means by which you address individual elements in an array.

Format:

\[(s[,s])\ldots\]

where:

\[s\] is an integer subscript expression.

Use a subscript following the array to specify which element in the array FORTRAN will reference.

In any subscripted array reference, you must type one subscript expression for each dimension you define for that array (i.e., one for each dimension declaration). For example, you could use the following entry to refer to the element located in the first row, third column, second level of the array TEMP in Figure 7-2 (which is the element occupying memory position 16).

\[\text{TEMP}(1,3,2)\]

Note, however, that an array reference such as TEMP(1,3) would be illegal because the third subscript is not indicated.

Each subscript expression can be any valid integer expression. If the value of a subscript expression is not an integer, FORTRAN converts it to integer before using it.

A subscript can be a compound expression, that is,

- Subscript quantities may contain arithmetic expressions that involve addition, subtraction, multiplication, division, and exponentiation. For example, \((I+J,K*5,L/2)\) and \((I**3,(J/4+K)*L,3)\) are valid subscripts.

- A subscript may contain function references. For example, \(\text{TABLE}((\text{IABS}(N))*K\text{OUNT},2,3)\) is a valid array element identifier.

- Subscripts may contain nested array element identifiers as subscripts. For example, in the subscript \((I(J(K(L)),M+N,ICOUNT))\), the first subscript quantity given is a nested 3-level subscript.
7.3.5.4 Data Type of an Array - Specify the data type of an array in the same way as the data type of a variable; that is, implicitly by the initial letter of the name, or explicitly by a type declaration statement. (See Section 7.6.1.)

All of the values in an array are of the same data type. FORTRAN converts any value you assign to an array element to the data type of the array.

7.3.5.5 Array References without Subscripts - In the following statements, you may type an array name without a subscript to specify that you wish to use the entire array.

Type Declaration Statements

COMMON statement
DATA statement
EQUIVALENCE statement
FUNCTION statement
SUBROUTINE statement
CALL statement
Input/Output statements

Using unsubscripted array names in any other statement is illegal.

7.3.5.6 Adjustable Arrays - Use an adjustable array in a subprogram so that the subprogram can process arrays of different sizes. Do this by passing the bounds as well as the array name as subprogram arguments or dummy arguments.

An adjustable array declarator differs from a standard array declarator in that the adjustable declarator has variable dimension declarators (which are simply integer variables). In such an array declaration, each dimension declarator must be either an integer constant or an integer dummy argument. This array name must also appear as a dummy argument. (Consequently, you may not use adjustable array declarators in main program units.)

Upon entry to a subprogram containing adjustable array declarators, FORTRAN associates each dummy argument in a dimension declarator with an integer actual argument. FORTRAN uses these values to form the actual array declaration. These integer variables determine the size of the adjustable array for that single execution of the subprogram.

You must not change the values of the dummy adjustable array declarator arguments within subprogram.

The effective size of the dummy array must be equal to or less than the actual size of the associated array.
The function in the following example computes the sum of the elements of a two-dimensional array. Note the use of the integer variables M and N to control the iteration.

```fortran
FUNCTION SUM(A,M,N)
DIMENSION A(M,N)
SUM = 0.
DO 10, I = 1,M
  DO 10, J = 1,N
  10 SUM = SUM + A(I,J)
RETURN
END
```

Following are sample calls on SUM:

```fortran
DIMENSION A1(10,35), A2(3,56)
SUM1 = SUM(A1,10,35)
SUM2 = SUM(A2,3,56)
SUM3 = SUM(A1,10,10)
```

If there are more dimensions in the adjustable array than in the array being passed to the subroutine, you must indicate a value of 1 for that dimension declaration.

7.4 EXPRESSIONS

An expression is a combination of elements which represents a single value. FORTRAN relates an element in an expression to another element in the same expression by operators and parentheses. The expression can be a single basic component, such as a constant or variable, or a combination of basic components with one or more operators. Operators specify computations to be performed (using the values of the basic components) to obtain a single value.

Expressions can be classified as arithmetic, relational, or logical. Arithmetic expressions yield numeric values; relational and logical expressions produce logical values.

7.4.1 Arithmetic Expressions

Form arithmetic expressions with arithmetic elements and arithmetic operators. The evaluation of such an expression yields a single numeric value.

An arithmetic expression element may be any of the following:

- A numeric constant,
- A numeric variable,
- A numeric array element,
- An arithmetic expression within parentheses,
- An arithmetic function reference.

Arithmetic operators specify a computation which FORTRAN will perform using the values of arithmetic elements; they produce a numeric value as a result. The operators and their meanings are listed in Table 7-12.
Table 7-12
Arithmetic Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>**</td>
<td>Exponentiation</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
</tr>
<tr>
<td>+</td>
<td>Addition and Unary Plus</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction and Unary Minus</td>
</tr>
</tbody>
</table>

The operators listed in Table 7-12 are called binary operators, because you would use each in conjunction with two elements. You can also use the + and - symbols as unary operators because, when you write them immediately preceding an arithmetic element, they indicate a positive or negative value.

7.4.1.1 Rules for Writing Arithmetic Expressions - Observe the following rules in structuring compound arithmetic expressions:

- An expression cannot contain two adjacent and unseparated operators. For example, the expression A*B is not permitted.

- You must include all operators; no operation is implied. For example, the expression A(B) does not specify multiplication although this is implied by standard algebraic notation. You must type A*(B) to obtain a multiplication of the elements.

- When you use exponentiation, the base quantity and its exponent may be of different types. For example, the expression ABC**13 involves a real base and an integer exponent. The permitted base/exponent type combinations and the type of the result of each combination are given in Table 7-13.

- You must assign a value to a variable or array element before you use it in an arithmetic expression. If you do not, the elements are undefined.

Table 7-13
Base/Exponent Combinations

<table>
<thead>
<tr>
<th>BASE</th>
<th>EXponent</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Integer</td>
</tr>
<tr>
<td>Integer</td>
<td>Yes</td>
</tr>
<tr>
<td>Real</td>
<td>Yes</td>
</tr>
</tbody>
</table>
In addition, you can only exponentiate a negative element by an integer element; you cannot exponentiate an element having a value of zero by another zero-value element.

In any valid exponentiation, the result is of the same data type as the base element.

7.4.1.2 Evaluation Hierarchy — FORTRAN evaluates arithmetic expressions in an order determined by a precedence it associates with each operator. The precedence of the operators is listed in Table 7-14.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>**</td>
<td>First</td>
</tr>
<tr>
<td>* and /</td>
<td>Second</td>
</tr>
<tr>
<td>+ and -</td>
<td>Third</td>
</tr>
<tr>
<td>=</td>
<td>Fourth</td>
</tr>
</tbody>
</table>

Whenever two or more operators of equal precedence (such as + or -) appear, FORTRAN evaluates them from left to right. However, FORTRAN evaluates exponentiation from right to left. For example, A**B**C is evaluated as A**(B**C) where FORTRAN computes the parenthetical subexpression (B**C) first.

7.4.1.3 Data Type of an Arithmetic Expression — The way in which OS/78 FORTRAN determines the data type of an expression is as follows:

- Integer operations — FORTRAN performs integer operations only on integer elements. (When you use octal constants and logical entities in an arithmetic context, FORTRAN treats them as integers.) In integer arithmetic, any fraction that results from a division is truncated, not rounded. For example, the value of the expression in integer arithmetic

  \[ \frac{1}{3} + \frac{1}{3} + \frac{1}{3} \]

  is zero, not one.

- Real operations — FORTRAN performs real operations on real elements or a combination of real and integer elements. FORTRAN converts integer elements to real by giving each a fractional part equal to zero. It then evaluates the expression using real arithmetic. Note, however, that in the statement \( Y = (I/J)*X \), FORTRAN performs an integer division operation on I and J and then performs a real multiplication on the result and X.
7.4.2 Relational Expressions

A relational expression consists of two arithmetic expressions which you separate by a relational operator. The value of the expression is either true or false, depending on whether or not the stated relationship exists.

A relational operator tests for a relationship between two arithmetic expressions. These operators are listed in Table 7-15.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Relationship</th>
</tr>
</thead>
<tbody>
<tr>
<td>.LT.</td>
<td>Less than</td>
</tr>
<tr>
<td>.LE.</td>
<td>Less than or equal to</td>
</tr>
<tr>
<td>.EQ.</td>
<td>Equal to</td>
</tr>
<tr>
<td>.NE.</td>
<td>Not equal to</td>
</tr>
<tr>
<td>.GT.</td>
<td>Greater than</td>
</tr>
<tr>
<td>.GE.</td>
<td>Greater than or equal to</td>
</tr>
</tbody>
</table>

The delimiting periods preceding and following a relational operator are part of the operator and must be present.

In a relational expression, FORTRAN evaluates the arithmetic expressions first to obtain their values. It then compares those values to determine if the relationship stated by the operator exists. For example, the expression:

```
APPLE+PEACH .GT. PEAR+ORANGE
```

tests the relationship, "The sum of the real variables APPLE and PEACH is greater than the sum of the real variables PEAR and ORANGE." If this relationship does exist, the value of the expression is true; if not, the expression is false.

All relational operators have the same precedence. Thus, if two or more relational expressions appear within an expression, FORTRAN evaluates the relational operators from left to right. Note that arithmetic operators have a higher precedence than relational operators.

Use parentheses to alter the evaluation of arithmetic expressions in a relational expression exactly as in any other arithmetic expression. However, as FORTRAN evaluates arithmetic operators before relational operators, it is unnecessary to enclose an arithmetic expression preceding or following a relational operator in parentheses.
7.4.3 Logical Expressions

A logical expression may be a single logical element, or it may be a combination of logical elements and logical operators. A logical expression yields a single logical value, either true or false.

A logical element can be any of the following:

- A logical constant,
- A logical variable,
- A logical array element,
- A relational expression,
- A logical expression enclosed in parentheses,
- A logical function reference (functions and function references are described in Chapter 8).

The logical operators are listed in Table 7-16.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Example</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>.AND.</td>
<td>A .AND. B</td>
<td>Logical conjunction. The expression is true if, and only if, both A and B are true.</td>
</tr>
<tr>
<td>.OR.</td>
<td>A .OR. B</td>
<td>Logical disjunction (inclusive OR). The expression is true if, and only if, either A or B, or both, is true.</td>
</tr>
<tr>
<td>.XOR.</td>
<td>A .XOR. B</td>
<td>Logical exclusive OR. The expression is true if A is true and B is false, or vice versa. It is false if both elements have the same value.</td>
</tr>
<tr>
<td>.EQV.</td>
<td>A .EQV. B</td>
<td>Logical equivalence. The expression is true if, and only if, both A and B have the same logical value, whether true or false.</td>
</tr>
<tr>
<td>.NOT.</td>
<td>.NOT. A</td>
<td>Logical negation. The expression is true if, and only if, A is false.</td>
</tr>
</tbody>
</table>

NOTE

A and B can be expressions or constants.

You must type the delimiting periods of logical operators.

A logical expression, like an arithmetic expression may consist of basic elements.
For example:

```
.TRU.
X .GE. 3.14159
```
or

```
TVAL .AND. INDEX
BOOL(M) .OR. K .EQ. LIMIT
```

where:

BOOL is either a logical function with one argument or a one-dimensional logical array.

You may enclose logical expressions within parentheses, e.g.,

```
A .AND. (B .OR. C)
```
or

```
(A .AND. B) .OR. C
```

Note that these expressions evaluate differently, e.g., if A is false and C is true, then the first yields a false value while the second yields a true.

7.4.3.1 Logical Operator Hierarchy - A summary of all operators that may appear in a logical expression, and the order in which FORTRAN evaluates them is listed in Table 7-17.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>**</td>
<td>First</td>
</tr>
<tr>
<td>*,/</td>
<td>Second</td>
</tr>
<tr>
<td>+,-</td>
<td>Third</td>
</tr>
<tr>
<td>Relational Operators</td>
<td>Fourth</td>
</tr>
<tr>
<td>.NOT.</td>
<td>Fifth</td>
</tr>
<tr>
<td>.AND.</td>
<td>Sixth</td>
</tr>
<tr>
<td>.OR.</td>
<td>Seventh</td>
</tr>
<tr>
<td>.XOR.,.EQV.</td>
<td>Eighth</td>
</tr>
</tbody>
</table>
7.4.4 Use of Parentheses

In an expression, FORTRAN evaluates all subexpressions you place within parentheses first. When you nest parenthetic subexpressions (that is, one subexpression is contained within another) the most deeply nested subexpression is evaluated first, the next most deeply nested subexpression is evaluated second, and so on, until FORTRAN computes the entire parenthetical expression.

When you type more than one operation within a parenthetical subexpression, FORTRAN performs the required computations according to a hierarchy of operators.

Parentheses do not imply multiplication. For example, \((A+B)(C+D)\) is illegal.

The following illustrates a typical numeric expression using numeric operators and a function reference is the familiar formula for obtaining one of the roots of a quadratic equation.

\[
\frac{-b + \sqrt{b^2 - 4ac}}{2a}
\]

which might be coded

\[
(-B + \text{SQRT}(B^2-4A*C))/(2A)
\]

Note how the parentheses affect the order of evaluation. Also note that one parentheses pair is required by the SQRT function. An example of the effect of parentheses is shown below (the numbers below the operators indicate the order in which FORTRAN performs the operations).

\[
\begin{align*}
4 & + 3 \times 2 - 6 \div 2 = 7 \\
& \quad 2 \quad 1 \quad 4 \quad 3 \\
(4 & + 3) \times 2 - 6 \div 2 = 11 \\
& \quad 1 \quad 2 \quad 4 \quad 3 \\
(4 & + 3 \times 2 - 6) \div 2 = 2 \\
& \quad 2 \quad 1 \quad 3 \quad 4 \\
((4 & + 3) \times 2 - 6) \div 2 = 4 \\
& \quad 1 \quad 2 \quad 3 \quad 4
\end{align*}
\]

Evaluation of expressions within parentheses takes place according to the normal order of precedence.

Nonessential parentheses, such as in the expression

\[4 + (3*2) - (6/2)\]

have no effect on the evaluation of the expression.

The use of parentheses to specify the evaluation order is often important where evaluation orders that are algebraically equivalent might not be computationally equivalent when carried out on a computer.

FORTRAN evaluates operators of equal rank from left to right.
7.5 ASSIGNMENT STATEMENTS

Assignment statements evaluate expressions and assign their values to variables or elements in an array.

There are three types of assignment statements:

- Arithmetic assignment statement,
- Logical assignment statement,
- ASSIGN statement (see Section 7.2.3.1).

7.5.1 Arithmetic Assignment Statement

The arithmetic assignment statement assigns a numerical value to a variable or array element.

Format:

\[ v = e \]

where:

- \( v \) is a variable or array element name.
- \( e \) is an expression.

The arithmetic assignment statement assigns the value of the expression on the right of an equal sign to the variable or array element on the left of the equal sign. If you had previously assigned a value to the variable, an assignment statement replaces it with the value on the right side of the equals sign.

Note that the equal sign does not mean "is equal to", as in mathematics. It means "is replaced by". Thus, the statement:

\[ \text{KOUNT} = \text{KOUNT} + 1 \]

means, "Replace the current value of the integer variable \text{KOUNT} with the sum of that current value and the integer constant 1".

Although the symbolic name to the left of the equal sign can be undefined, you must previously have assigned values to all symbolic references in an expression (i.e., the right side of the equals sign).

An expression must yield a value that conforms to the requirements of the variable or array element to which you assign it (for example, a real expression that produces a value greater than 8,338,608 is illegal if the entity on the left of the equal sign is an INTEGER variable).

If the data type of the variable or array element on the left of the equal sign is the same as that of the expression on the right, FORTRAN assigns the value directly. If the data types are different, FORTRAN converts the value of the expression to the data type of the entity on the left of the equal sign before it is assigned.
Examples:

**Valid Statements**

\[
\text{BETA} = \frac{-1}{(2. \ast X)} + A \ast A/(4. \ast (X \ast X))
\]

\[
\text{PI} = 3.14159
\]

\[
\text{SUM} = \text{SUM} + 1.
\]

**Invalid Statements**

\[
3.14 = A-B \quad \text{(Entity on the left must be a variable or array element.)}
\]

\[
-J = I**4 \quad \text{(Entity on the left must not be signed.)}
\]

\[
\text{ALPHA} = ((X+6)\ast B/(X-Y)) \quad \text{(Invalid; left and right parentheses do not balance.)}
\]

### 7.5.2 Logical Assignment Statements

Use a logical assignment statement to assign a true or false value to a logical variable.

**Format:**

\[
v = e
\]

**where:**

- \( v \) is a variable or array element of type logical, and
- \( e \) is a logical expression.

The logical assignment statement is similar to the arithmetic assignment statement, but it operates on logical data. The logical assignment statement evaluates the expression on the right side of an equal sign and assigns the resulting logical value, either true or false, to the variable or array element on the left.

The variable or array element on the left of the equal sign must be of type LOGICAL; its value can be undefined before the assignment.

You must have previously assigned values, either numeric or logical, to all symbolic references that appear in an expression. The expression must yield a logical value.

**Examples:**

\[
\text{PAGEND} = .FALSE.
\]

\[
\text{PRNTOK} = \text{LINE} .LE. 132 .AND. .NOT. \text{PAGEND}
\]

\[
\text{ABIG} = A \ast .GT. B \ast .AND. A \ast .GT. C \ast .AND. A \ast .GT. D
\]

### 7.6 SPECIFICATION STATEMENTS

This section discusses FORTRAN specification statements. Specification statements are nonexecutable statements which provide information necessary for the proper allocation and initialization of variables and names you use in a program.
7.6.1 Type Declaration Statements

Type declaration statements explicitly define the data type of symbolic names.

Format:

\[ \text{typ \ v} [, \text{v}] \ldots \]

where:

\text{typ} \quad \text{is one of the following data type specifiers:}

\begin{align*}
\text{LOGICAL} \\
\text{INTEGER} \\
\text{REAL}
\end{align*}

\text{v} \quad \text{is a typed variable or array.}

A type declaration statement causes the specified symbolic names to have the specified data type; it overrides the data type implied by the initial letter of a symbolic name.

A type declaration statement can define arrays by including array declarators in the list. In each program unit, an array name can appear only once in an array declarator. Note, however, that

\begin{verbatim}
DIMENSION ISUM(3,4)
INTEGER ISUM
\end{verbatim}

is legal.

Type declaration statements should precede all executable statements and all specification statements. You must precede the first use of any symbolic name with its declaration statement if you do not use the default type declaration.

You can explicitly declare the data type of a symbolic name only once.

You must not label type declaration statements. The FORTRAN entities which you may type are:

\begin{itemize}
  \item Arithmetic Statement Functions
  \item Arrays
  \item functions
  \item Variables
\end{itemize}

Examples:

\begin{verbatim}
INTEGER COUNT, MATRIX(4,4), SUM
REAL MAN, IABS
LOGICAL SWITCH
\end{verbatim}

7.6.2 DIMENSION Statement

The DIMENSION statement defines the number of dimensions in an array and the number of elements in each dimension.

Format:

\[ \text{DIMENSION a(d)} [, a(d) \ldots] \ldots \]
where:

\[
\begin{align*}
\text{a} & \quad \text{is the symbolic name of an array} \\
\text{d} & \quad \text{is the dimension declarator}
\end{align*}
\]

Example:

\[
\text{DIMENSION ARRAY}(6,7,4)
\]

The DIMENSION statement allocates storage locations, one for each element in each dimension, for each array in the DIMENSION statement. You may declare any number of arrays in one dimension statement. Each storage location is six or twelve bytes in length as determined by the data type of the array. The amount of storage FORTRAN assigns to an array is equal to 6 or 12 times the product of all dimension declarators in the array declarator for that array. For example,

\[
\text{DIMENSION ARRAY}(4,4), \text{MATRIX}(5,5,5)
\]

defines ARRAY as having 16 real elements of 6 words each, and MATRIX as having 125 integer elements, also of 6 words each.

You cannot declare more than 7 dimensions to an array. There is also a limit of 4095 elements to any array. Each size specification must be a non-zero positive integer constant.

For further information concerning arrays and the storage of array elements, see Section 2.6.

Array declarators can also appear in type declaration and COMMON statements; however, in each program unit, an array name can appear in only one array declarator.

You must not label DIMENSION statements.

Examples:

\[
\begin{align*}
\text{DIMENSION BUD}(12,24,10) \\
\text{DIMENSION X}(5,5,5), \text{Y}(4,85), \text{Z}(100) \\
\text{DIMENSION MARK}(4,4,4,4)
\end{align*}
\]

7.6.3 EXTERNAL Statement

The EXTERNAL statement permits the use of external procedure names (functions, subroutines, and FORTRAN library functions) as arguments to other subprograms.

Format:

\[
\text{EXTERNAL } v \left[ v, v \right] \ldots
\]

where:

\[
\begin{align*}
v & \quad \text{is the symbolic name of a subprogram or the name of a dummy argument which is associated with a subprogram.}
\end{align*}
\]

Example:

\[
\text{EXTERNAL SIN, COS, ABS}
\]
FORTRAN IV

Any subprogram which you use as an argument to another subprogram must appear in an EXTERNAL statement in the calling subprogram. Thus, the purpose of the EXTERNAL statement is to declare names to be subprogram names. This distinguishes the external name \( v \) from other variable or array names.

The subprogram may be ones that you write or those which are part of the FORTRAN library. The EXTERNAL statement declares each name \( v \) to be the name of a procedure external to the program unit. Such a name can then appear as an actual argument to a subprogram.

NOTE

If you use a complete function reference such as a call to the SQRT external function in a reference such as \( \text{CALL SORT(A,SQRT(B),C)} \), the function reference is a value (the square root of \( B \)) and you do not need to define it as an external statement. You would only have to define it if you were passing the function name, i.e., \( \text{CALL SORT(A,SQRT,C)} \).

FORTRAN reserves the names you declare in an external statement throughout the compilation of the program; you cannot use it in any other declaration statement, with the exception of a type statement.

Example:

<table>
<thead>
<tr>
<th>Main Program</th>
<th>Subprograms</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXTERNAL SIN,COS,TAN</td>
<td>SUBROUTINE TRIG ((x,f,y))</td>
</tr>
<tr>
<td>.</td>
<td>( y = f(x) )</td>
</tr>
<tr>
<td>.</td>
<td>RETURN END</td>
</tr>
<tr>
<td>CALL TRIG ((\text{ANGLE},\text{SIN},\text{SINE}))</td>
<td>.</td>
</tr>
<tr>
<td>.</td>
<td>FUNCTION TAN ((x))</td>
</tr>
<tr>
<td>.</td>
<td>( \text{TAN} = \text{SIN}(x) / \text{COS}(x) )</td>
</tr>
<tr>
<td>CALL TRIG ((\text{ANGLE},\text{COS},\text{COSINE}))</td>
<td>.</td>
</tr>
<tr>
<td>.</td>
<td>RETURN END</td>
</tr>
<tr>
<td>CALL TRIG ((\text{ANGLE},\text{TAN},\text{TANGNT}))</td>
<td>.</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
</tr>
</tbody>
</table>

The CALL statements pass the name of a function to the subroutine TRIG. The function is subsequently invoked by the function reference \( f(x) \) in the second statement of TRIG. Thus, the second statement becomes in effect:

\[
\begin{align*}
  y & = \text{SIN}(x) \\
  y & = \text{COS}(x) \\
  y & = \text{TAN}(x)
\end{align*}
\]

depending upon which CALL statement invoked TRIG. The functions SIN and COS are examples of trigonometric functions supplied in the FORTRAN Library.
7.6.4 COMMON Statement

Use a COMMON statement so that programs and subprograms can share information.

Format:

\[
\text{COMMON } \left[ [\left[ \text{cb} \right] / \right] \text{nlist} / \left[ \left[ \text{cb} \right] / \text{nlist} \right] \ldots
\]

where:

- \text{cb} is a symbolic name or is blank. If the first \text{cb} is blank, you can omit the first pair of slashes, and
- \text{nlist} is a list of variable names, array names, and array declarators separated by commas.

Example:

\[
\text{COMMON } /\text{AREAl/A,B } //\text{C,D}
\]

The COMMON statement enables you to establish storage that two or more programs and/or subprograms may share and to name the variables and arrays that will occupy the common storage. The use of common storage conserves storage and provides a means to implicitly transfer arguments between a calling program and a subprogram. The transfer is implicit because no actual tranferral takes place; instead, the program unit references the common storage area.

\text{FORTRAN} determines the length of a COMMON block by the number of components and the amount of storage each component requires. COMMON blocks may be of any length, subject to the limitations of available memory.

\text{nlist}, which appears after each common name \text{cb}, lists the names of the variables and arrays that will occupy the common area \text{cb}. \text{FORTRAN} places the items for a common within common storage area in the order in which you list them in the COMMON statement or statements.

Elements you place into common storage in one program unit should agree in data type with elements reference in a second. This is because assignment of storage is on a storage unit-for-storage unit basis, not variable-for-variable.

Either label COMMON storage areas or leave them blank (unlabeled). If you label the common area, type a symbolic name within slashes immediately before the list of items that will occupy the \text{cb} area.

For example, the statement

\[
\text{COMMON/AREAl/A,B,C/AREAl2/TAB(13,3,3)}
\]

establishes two labeled common areas (i.e., AREAl and AREAl2).

If you are declaring a common storage area to be blank common, then you may omit the double slashes (//) if and only if it is the first declaration of any common statement. Unlabeled common area is called "blank common". If the blank common declaration is not the first declaration in a COMMON statement, then the double slashes are mandatory.

For example, the statement

\[
\text{COMMON/AREAl/A,B,C/TAB(3,3,3)}
\]

establishes one labeled area (AREAl) and one unlabeled common area.
FORTRAN IV

A given labeled common name may appear more than once in the same COMMON statement and in more than one COMMON statement within the same program or subprogram.

During compilation of a source program, FORTRAN will bring together all items you list for each labeled and blank common area in the order in which the items appear in the source program statements.

For example, the series of source program statements:

```
COMMON/ST1/A,B,C/ST1/TAB(2,2)//C,D,E
  .
COMMON/ST1/TST(3,4)//M,N
  .
COMMON/ST2/X,Y,Z//O,P,Q
```

has the same effect as the single statement

```
COMMON/ST1/A,B,C,TST(3,4)/ST2/TAB(2,2),X,Y,Z//C,D,E,M,N,O,P,Q
```

FORTRAN treats each labeled common area as a separate, specific storage area. You assign the contents of a common area, i.e., variables and arrays, initial values by DATA statements in a BLOCK DATA subprogram. Declarations of a given common area in different subprograms must contain the same number, size, and order of variables and arrays as the reference array.

Common block names must be unique with respect to all subroutine and function names.

The largest definition of a given common area must be loaded first.

Storage allocation for blocks of the same name begins at the same location for all program units FORTRAN executes together. For example, if a program contains:

```
COMMON A,B,C/R/X,Y,Z
```
as its first COMMON statement, and a subprogram has:

```
COMMON /R/U,V,W //D,E,F
```
as its first COMMON statement, the values represented by X and U are stored in the same location. A similar correspondence holds for A and D in blank common.

If one program unit references a part of a common block, then you must use dummy variables to establish the proper correspondence. For example, if you declare a common block to contain:

```
A,B,C,D,E,F,G,H,I,J,K
```
and a subprogram wishes to reference the storage location pointed to by K, then you must declare a common block as follows in the subprogram:

```
COMMON A,B,C,D,E,F,G,H,I,J,K
```
The declaration COMMON K in the subprogram would cause a correspondence between variable A in the main program and variable K in the subprogram. (Note that any other sequence of variables names would also be correct.)
Instead of declaring each variable contained in the COMMON block, you may substitute a dummy array (provided that you are careful to match-up proper storage lengths).

You may also define an array in a COMMON statement. You may not otherwise subscript array names. Also, you cannot assign individual array elements to COMMON.

7.6.5 EQUIVALENCE Statement

Use an EQUIVALENCE statement to associate different variables with the same storage.

Format:

    EQUIVALENCE (nlist) [ , (nlist) ] ...

where:

    nlist is a list of variables and array elements, separated by commas. At least two components must be present in each list.

Example:

    EQUIVALENCE (A,B), (C,D(16),E,F)

The EQUIVALENCE statement declares two or more entities to be associated (either totally or partially) with the same storage location.

NOTE

    EQUIVALENCE differs from COMMON in that EQUIVALENCE associates different variable names with the same storage area in a program unit. COMMON may associate different variable names with the same storage area but it always makes the association between program units.

The EQUIVALENCE statement causes FORTRAN to allocate all of the variables or array elements contained in one parenthesized list beginning at the same storage location.

You can also use the EQUIVALENCE statement to equate variable names. For example, the statement

    EQUIVALENCE (FLTLEN, FLENTH, FLIGHT)

causes FLTLEN, FLENTH, and FLIGHT to have the same value provided they are also of the same data type.
An EQUIVALENCE statement in a subprogram must not contain dummy arguments.

Examples:

```
EQUIVALENCE (A,B), (B,C)   (has the same effect as EQUIVALENCE (A,B,C))
EQUIVALENCE (A(1),X), (A(2),Y), (A(3),Z)
```

7.6.5.1 Making Arrays Equivalent - When you make an element of an array equivalent to an element of another array, the EQUIVALENCE statement also sets equivalences between other elements of the two arrays. Thus, if you make the first elements of two equal-sized arrays equivalent, both arrays share the same storage space. Moreover, if you make the third element of a 5-element array equivalent to the first element of another array, the last three elements of the first array overlap the first three elements of the second array.

The EQUIVALENCE statement must not attempt to assign the same storage location to two or more elements of the same array, nor to assign memory locations in any way that is inconsistent with the normal linear storage of array elements (for example, making the first element of an array equivalent with the first element of another array, then attempting to set an equivalence between the second element of the first array and the sixth element of the other).

In the EQUIVALENCE statement only, it is possible to identify an array element with a single subscript (that is, the linear element number), even though you have defined one as being multi-dimensional.

For example, the statements:

```
DIMENSION TABLE (2,2), TRIPLE (2,2,2)
EQUIVALENCE (TABLE(4), TRIPLE(7))
```

result in the entire array TABLE sharing a portion of the storage space. FORTRAN allocates to array TRIPLE as illustrated in Figure 7-4. In Figure 7-4, the elements with asterisks are those explicitly mentioned in the above EQUIVALENCE statement.

<table>
<thead>
<tr>
<th>Array TRIPLE</th>
<th>Array TABLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Element</td>
<td>Element</td>
</tr>
<tr>
<td>Number</td>
<td>Number</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------</td>
</tr>
<tr>
<td>TRIPLE(1,1,1)</td>
<td>TABLE(1,1)</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>TRIPLE(2,1,1)</td>
<td>TABLE(2,1)</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>TRIPLE(1,2,1)</td>
<td>TABLE(1,2)</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>TRIPLE(2,2,1)</td>
<td>TABLE(2,2)</td>
</tr>
<tr>
<td>4</td>
<td>4*</td>
</tr>
<tr>
<td>TRIPLE(1,1,2)</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>TRIPLE(2,1,2)</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>TRIPLE(1,2,2)</td>
<td></td>
</tr>
<tr>
<td>7*</td>
<td></td>
</tr>
<tr>
<td>TRIPLE(2,2,2)</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

Figure 7-4 Equivalence of Array Storage
Figure 7-4 also illustrates that the following two statements

    EQUIVALENCE (TABLE(1),TRIPLE(4))
    EQUIVALENCE (TRIPLE(1,2,2), TABLE(4))

result in the same alignment of the two arrays.

7.6.5.2 EQUIVALENCE and COMMON Interaction - When you make components equivalent to entities in common, it can cause FORTRAN to extend the common block beyond its original boundaries.

An EQUIVALENCE statement can only extend common beyond the last element of the previously established common block. It must not attempt to increase the size of common in such a way as to place the extended portion before the first element of existing common. (See Figure 7-5.)

![Valid Extension of Common](image)

**Valid Extension of Common**

<table>
<thead>
<tr>
<th>DIMENSION A(4),B(6)</th>
<th>A(1)</th>
<th>A(2)</th>
<th>A(3)</th>
<th>A(4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMMON A</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EQUIVALENCE (A(2),B(1))</td>
<td>B(1)</td>
<td>B(2)</td>
<td>B(3)</td>
<td>B(4) B(5) B(6)</td>
</tr>
<tr>
<td>Exiting Common</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Extended Portion</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

![Illegal Extensions of Common](image)

**Illegal Extensions of Common**

<table>
<thead>
<tr>
<th>DIMENSION A(4),B(6)</th>
<th>A(1)</th>
<th>A(2)</th>
<th>A(3)</th>
<th>A(4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMMON A</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EQUIVALENCE(A(2),B(3))</td>
<td>B(1)</td>
<td>B(2)</td>
<td>B(3) B(4) B(5) B(6)</td>
<td></td>
</tr>
<tr>
<td>Extended Portion</td>
<td></td>
<td></td>
<td>Existing Common</td>
<td>Extended Portion</td>
</tr>
</tbody>
</table>

![Figure 7-5 Legal and Illegal Common Extensions](image)

If you assign two components to the same or different common blocks, you must not make them equivalent to each other.

7.7 DATA STATEMENTS AND BLOCK DATA SUBPROGRAMS

The DATA initialization statement permits the assignment of initial values to variables and array elements prior to program execution.

Format:

    DATA nlist/clist/ [ ] [ , ] nlist/clist/ ] ...

where:

- nlist is a list of one or more variable names, array names, or array element names separated by commas,
- clist is an optional separator, and
- is a list of constants.
Example:

```
DATA A,B,C(3),C(7)/4.0,8.1,16.0,28.0/
```

The DATA statement causes FORTRAN to assign the constant values in each clist to the entities in the preceding nlist. FORTRAN assigns values in a one-to-one manner in the order in which they appear, from left to right. [[IS COMMA OPTIONAL OR MANDATORY]]

When an unsubscripted array name appears in a DATA statement, FORTRAN assigns values to every element of that array. The associated constant list must therefore contain enough values to fill the array. FORTRAN fills array elements in the order of subscript progression. (See Section 2.6.1.)

When you assign Hollerith data to a variable or array element, the number of characters that you can assign depends on the data type of the component. If the number of characters in a Hollerith constant or alphanumeric literal is less than the capacity of the variable or array element, the constant is padded on the right with spaces. If the number of characters in the constant is greater than the maximum number that the variable can hold, it ignores the rightmost excess characters.

When you assign the same value to more than one item in nlist, you may use a repeat specification. Write the repeat specification as N*D where N is an integer that specifies how many times the value of item D is to be used. For example, a DATA specification of /3*20/ specifies that the value 20 is to be assigned to the first three items named in the preceding list. Also, the statement

```
DATA M,N,L /3*20/
```

assigns the value 20 to the variables M, N, and L. The number of constants in a constant list must correspond exactly to the number of entities specified in the preceding name list. The data types of the data elements and their corresponding symbolic names must agree.

FORTRAN IV converts the constant to the type of the variable being initialized.

Example:

```
INTEGER A(10),BELL,K(5,5,5)
DATA A,BELL,STARS/10*0,7, '*****'/K/25*0,25*1,25*2,25*3,25*4,25*5/
```

The DATA statement assigns zero to all ten elements of array A, the value 7 to the variable BELL, and four asterisks to the real variable STARS. The 125 element array, K, is initialized so that each of the five planes (i.e., the third dimension declarator) has a different value.

When you initialize an array, you must initialize the entire array, e.g., the DATA statement in the following

```
DIMENSION K
DATA K /10*1/
```

is illegal.
You could effect the same thing as follows:

```fortran
DIMENSION I(30),K(10)
EQUIVALENCE (I,K)
DATA K/10*1/
```

The values you assign with a DATA statement may also be assigned with a BLOCK DATA subprogram. However, note that initial values for variables in COMMON storage may not be specified in subprograms which may be overlaid at execution time. If a subprogram will be overlaid, then you should only initialize these variables in a BLOCK DATA subprogram. (DIGITAL recommends that you only initialize variables in COMMON storage with BLOCK DATA subprograms.)

Use a BLOCK DATA to initialize variables you place into COMMON storage.

Format:

```
BLOCK DATA
```

Use the BLOCK DATA subprogram to assign initial values to entities in common blocks, at the same time establishing and defining those blocks. It consists of a BLOCK DATA statement followed by a series of specification statements.

The statements FORTRAN allows in a BLOCK DATA subprogram are:

- Type Declaration
- DIMENSION
- COMMON
- EQUIVALENCE
- DATA

The specification statements in the BLOCK DATA subprogram establish and define common blocks, assign variables and arrays to those blocks, and assign initial values to those components.

A BLOCK DATA statement must be the first statement of a BLOCK DATA subprogram. You must not label the BLOCK DATA statement.

A BLOCK DATA subprogram must not contain any executable statements.

If you initialize any entity in a common block in a BLOCK DATA subprogram, you must enter a complete set of specification statements to establish the entire block, even though some of the components in the block do not appear in a DATA statement. You can define initial values for more than one common area with the BLOCK DATA subprogram.

### 7.8 CONTROL STATEMENTS

FORTRAN normally executes statements in the order in which you write them. However, it is frequently desirable to change the normal program flow by transferring control to another section of the program or to a subprogram. Transfer of control from a given point in the program may occur every time that point is reached in the program flow, or may be based on a decision made at that point.
Transfer of control, whether within a program unit or to another program unit, is performed by control statements. These statements also govern iterative processing, suspension of program execution, and program termination. The types of control statements discussed in this chapter are:

ASSIGN
CONTINUE
DO
END
IF
GO TO
PAUSE
STOP

A second kind of statement for transferring control, subprograms, are discussed in Chapter 8.

7.8.1 GOTO Statements

GOTO statements transfer control within a program unit, either to the same statement every time or to one of a set of statements, based on the value of an expression.

The three types of GOTO statements are:

- Unconditional GOTO statement,
- Computed GOTO statement, and
- Assigned GOTO statement.

7.8.1.1 Unconditional GOTO Statement - Transfers control to the same statement every time executed.

Format:

GOTO st

where:

st is the label of an executable statement in the same program unit as the GOTO statement.

Example:

GOTO 50

The unconditional GOTO statement transfers control to the statement identified by the specified label. The statement label must identify an executable statement in the same program unit as the GOTO statement.

Examples:

GOTO 7734
GOTO 99999
GOTO 27.5 (Invalid; the statement label is improperly formed.)
FORTRAN IV

7.8.1.2 Computed GOTO Statement - Transfers control to a statement based on the value of an expression within the statement.

Format:

\[ \text{GOTO (slist) } [ , ] \ e \]

where:

slist is a list of one or more executable statement labels separated by commas,
\[ , \] is an optional separator, and
e is an integer expression the value of which falls within the range 1 to n (where n is the number of statement labels in slist).

Example:

\[ \text{GOTO (10,200,25), NUMBER} \]

Use the computed GOTO to transfer control to one statement out of a list of statements. The computed GOTO thus acts as a multi-directional switch.

The computed GOTO statement evaluates the integer expression e. The GOTO statement then transfers control to the e'th statement label in slist. That is, if the list contains (30,20,30,40), and the value of e is 2, the GOTO statement transfers control to statement 20, and so on.

You may include any number of statements in slist but you must use each number as a label within the program.

The comma following (slist) is optional.

If the value of the expression is less than 1, or greater than the number of labels in the slist, unpredictable results occur.

Examples:

\[ \text{GOTO (12,24,36),INCHES} \]
\[ \text{GOTO (320,330,340,350,360)ISITU(J,K)+1} \]

7.8.1.3 ASSIGN and ASSIGNed GOTO Statement - Use the ASSIGN statement to assign a statement label to a variable name.

Format:

\[ \text{ASSIGN st to } v \]

where:

st is the label of an executable statement in the same program unit as the ASSIGN statement, and
v is an integer variable.

Example:

\[ \text{ASSIGN 50 TO NUMBER} \]
FORTRAN IV

Use the ASSIGN statement to associate a statement label with an integer variable. You can then use the variable as a transfer destination in a subsequent ASSIGNED GOTO statement.

NOTE

The statement number must be in the same program unit.

The statement label st must not be the label of a FORMAT statement.

The ASSIGN statement assigns the statement number to the variable in a manner similar to that of an arithmetic assignment statement, with one exception: the variable becomes defined for use as a statement label reference and becomes undefined as an integer variable.

FORTRAN must execute an ASSIGN statement before the ASSIGNED GOTO statement in which it will use the assigned variable. The ASSIGN statement and the ASSIGNED GOTO statement must occur in the same program unit.

For example, the statement

ASSIGN 100 TO NUMBER

associates the variable NUMBER with the statement label 100.

Arithmetic operations on the variable, such as in the statement

NUMBER = NUMBER + 1

then become invalid, as FORTRAN cannot alter a statement label. (This is because a statement refers to a location in memory and is not a number.) The statement:

NUMBER = 10

disassociates NUMBER from statement 100, assigns it an integer value 10, and returns it to its status as an integer variable. After you make such an assignment, you can no longer use it in an ASSIGNED GOTO statement.

Examples:

ASSIGN 10 TO NSTART

ASSIGN 99999 TO KSTOP

ASSIGN 250 TO ERROR \(\text{(ERROR must have been defined as an integer variable.)}\)

The ASSIGNED GOTO transfers control to a statement which is represented by a variable.

Format:

\[ \text{GOTO v} [ [ , ] (\text{slist}) ] \]

where:

\(v\) is an integer variable,
\(,\) is an optional separator, and
\(\text{slist}\) (when present) is a list of one or more executable statement labels separated by commas.
Example:

GOTO NUMBER,(10,35,15)

The ASSIGNED GOTO statement transfers control to the statement whose label was most recently assigned to the variable \( v \) by an ASSIGN statement. (See Section XXX.)

The variable \( v \) must be of integer type. In addition, you must have previously assigned to it a statement label number with an ASSIGN statement (not an arithmetic assignment statement).

The ASSIGNED GOTO statement and its associated ASSIGN statement must reside in the same program unit. Also, statements to which FORTRAN transfers control must be executable statements in the same program unit.

Examples:

ASSIGN 50 TO IGO
GOTO IGO

GOTO INDEX, (300,450,1000,25)

If the statement label value of \( v \) is not present in the list \( \text{slist} \) (and a list is specified), control transfers to the next executable statement following the ASSIGNED GOTO statement.

NOTE

You must label the statement following an ASSIGNED GOTO; otherwise, FORTRAN can never execute that statement.

7.8.2 IF Statements

An IF statement causes a conditional control transfer or the conditional execution of a statement. There are two types of IF statements:

- Arithmetic IF statements, and
- Logical IF statements.

7.8.2.1 Arithmetic IF Statement - Use the arithmetic IF as a three-way branching statement. The branching depends on whether the value of an expression is less than, equal to, or greater than zero.

Format:

IF (e) st1, st2, st3

where:

- \( e \) is an arithmetic expression, and
- \( \text{st1, st2, st3} \) are the labels of executable statements in the same program unit.
Example:

IF (I-K) 10, 20, 30

Use the arithmetic IF statement for conditional control transfers. This statement can transfer control to one of three statements, based on the value of an arithmetic expression.

You may use logical expressions in arithmetic IF statements. In such a case, FORTRAN first converts the logical expression value to an integer. If you use a complex expression, FORTRAN only uses the real portion.

Normal use of the arithmetic IF requires that all three labels, st1, st2, and st3, must be present. However, they need not refer to three different st. If desired, one or two labels can refer to the same statement.

OS/78 FORTRAN allows you to type less than three numbers. If you type either one or two numbers, then if a condition is not met (e.g., e is greater than zero), then control passes to the next statement.

Example:

IF (ALPHA) 10
STOP

In this statement, control transfers to statement number 10 if ALPHA is negative. If ALPHA is positive or equal to zero, execution stops.

The arithmetic IF statement first evaluates the expression in parentheses and then transfers control to one of the three statement labels that follow expression e. The values upon which FORTRAN makes the selection are listed in Table 7-18.

<table>
<thead>
<tr>
<th>If the Value is:</th>
<th>Control Passes to:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Less than 0</td>
<td>Labels st1</td>
</tr>
<tr>
<td>Equal to 0</td>
<td>Label st2</td>
</tr>
<tr>
<td>Greater than 0</td>
<td>Label st3</td>
</tr>
</tbody>
</table>

Examples:

IF (THETA-CHI) 50,50,100

This statement transfers control to statement 50 if the real variable THETA is less than or equal to the real variable CHI. Control passes to statement 100 only if THETA is greater than CHI.

IF (NUMBER/2*2-NUMBER) 20,40

This statement transfers control to statement 40 if the value of the integer variable NUMBER is even and to statement 20 if it is odd.
FORTRAN IV

7.8.2.2 Logical IF Statement - Use a logical IF statement for conditional execution of statements.

Format:

    IF (e) st

where:

    e    is a logical expression, and
    st   is a complete FORTRAN statement. The statement can be any executable statement except a DO statement or another logical IF statement.

Example:

    IF(X .EQ. Y) Z=4

A logical IF statement causes a conditional statement execution. FORTRAN bases the decision to execute the statement on the value of a logical expression within the statement.

The logical IF statement first evaluates the logical expression. If the value of the expression is true, FORTRAN transfers control to the executable statement within the IF statement. If the value of the expression is false, control transfers to the next executable statement following the logical IF; in this case, FORTRAN does not execute statement st.

Examples:

    IF (J .GT. 4 .OR. J .LT. 1) GOTO 250
    IF (REF(J,K) .NE. HOLD) REF(J,K) = REF(J,K)*A(K,J)
    IF (.NOT. X) CALL SWITCH(S,Y)

7.8.3 DO Statement

Use the DO statement to repeatedly execute a block of statements.

Format:

    DO st i=e1,e2 [ ,e3 ]

where:

    st   is the label of an executable statement which physically follows in the same program unit,
    i    is an unsubscripted real or integer variable,
    e1   (the initial value of i) is an integer or real constant or expression,
    e2   (the terminal value of i) is an integer or real constant or expression and must be greater than e1, and
    e3   (the value by which i will be incremented each time it executes the statements in the range of the DO loops) an integer real constant or expression.
Example:

```
DO 10 I=1,10,2
DO 20 I=J,K,L
```

The DO statement causes FORTRAN to repeatedly execute the statements in its range a specified number of times.

The range of a DO statement is defined as the series of statements that follow the DO statement up to and including its specified terminal statement st, that is, the statements that follow the DO statement, up to and including the terminal statement are in the range of the DO loop.

The variable i is called the control (or index) variable of the DO and el, e2, e3 are the initial, terminal, and increment parameters respectively.

The terminal statement of a DO loop is identified by the label st that appears in the DO statement. This terminal statement must not be a GOTO statement, an arithmetic IF statement, a RETURN statement, PAUSE statement, STOP statement, or another DO statement. A logical IF statement is acceptable as the terminal statement, provided it does not contain any of the above statements.

The DO statement first evaluates the expressions el, e2, e3 to determine values for the initial, terminal, and increment parameters. FORTRAN then assigns value of the initial parameter to the control variable. FORTRAN then repeatedly executes the statements in the range of the DO loop.

The increment parameter must be positive, and the value of the terminal parameter must not be less than that of the initial parameter.

The value of the increment parameter must not be zero.

After each execution of the range of the DO loop, FORTRAN adds the increment value to the value of the index. It then compares the result to the terminal value. If the index value is not greater than the terminal value, FORTRAN reexecutes the range using the new value of the index i.

The number of executions of the DO range, called the iteration count, is given by

```
MAX(1,(e2-el)/e3) + 1
```

FORTRAN always executes the range of a DO statement at least once.

7.8.3.1 DO Iteration Control - You can terminate the execution of a DO by a statement within the range that transfers control outside the loop. When you transfer out of the DO loop's range, the control variable of the DO remains defined with its current value.

When execution of a DO loop terminates, if other DO loops share the same terminal statement, control transfers outward to the next most enclosing DO loop in the DO nesting structure (Section 7.4.2). If no other DO loop share this terminal statement, or if this DO is the outermost DO, control transfers to the first executable statement following the terminal statement.
You may alter the values of i, el, e2, and e3. If you alter the value of i, the loop will not be executed the number of times which you originally specified. If you alter the values of the expressions, you do not affect the looping as FORTRAN "remembers" these values. The control variable i is available for reference as a variable within the range.

The range of a DO loop can contain other DO statements, as long as those "nested" DO loops conform to certain requirements.

You can transfer control out of a DO loop, but you cannot transfer into a loop from elsewhere in the program. Exceptions to this rule are described in the following sections.

Examples:

DO 100 K=1,50,2                              (25 iterations, K=49 during final iteration)
DO 25 IVAR=1,5                               (5 iterations, IVAR=5 during final iteration)
DO NUMBER=5,40,4                              (Invalid; statement label missing)
DO 40 M=2.10                                 (Invalid; decimal point instead of comma)

The last example illustrates a common clerical error. It is a valid arithmetic assignment statement in the FORTRAN language; i.e.,

DO40M = 2.10

7.8.3.2 Nested DO Loops - A DO loop may contain one or more complete DO loops. The range of an inner nested DO must lie completely within the range of the next outer loop. Nested loops may share the same terminal statement. (See Figure 7-6.)

<table>
<thead>
<tr>
<th>Correctly Nested DO Loops</th>
<th>Incorrectly Nested DO Loops</th>
</tr>
</thead>
<tbody>
<tr>
<td>DO 45 K=1,10</td>
<td>DO 15 K=1,10</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>DO 35 L=2,50,2</td>
<td>DO 25 L=1,20</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>35 CONTINUE</td>
<td>15 CONTINUE</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>DO 45 M=1,20</td>
<td>DO 30 M=1,15</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>45 CONTINUE</td>
<td></td>
</tr>
</tbody>
</table>

Figure 7-6 Nesting of DO Loops
In the correctly nested DO loops, note that the diagrammed lines do not cross. They do, however, share the same statement (45). In the incorrectly nested DO loops, the loop defined by DO 25 crosses the ranges of the other two DO loops.

Note that you may nest loops to a depth of (at least) 10 levels.

7.8.3.3 Control Transfers in DO Loops — Within a nested DO loop structure, you can transfer control from an inner loop to an outer loop. A transfer from an outer loop to an inner loop is illegal.

If two or more nested DO loops share the same terminal statement, you can transfer control to that statement only from within the range of the innermost loop, that is, the terminal statement belongs solely to the innermost DO statement. Any other transfer to that statement constitutes a transfer from an outer loop to an inner loop because the shared statement is part of the range of the innermost loop.

The following rules govern the transfer of program control from within the DO statements range or the ranges of nested DO statements.

- FORTRAN permits a transfer out of the range of any DO statement at any time. When such a transfer executes, the controlling DO statement's index variable retains its current value.

- FORTRAN permits a transfer into the range of a DO statement from within the range of any:
  1. DO loop;
  2. nested DO loop; or
  3. extended range loop (in which you leave the loop via a GOTO, execute statements elsewhere, and return to the original loop).

7.8.3.4 Extended Range — A DO loop is said to have an extended range if it contains a control statement that transfers control out of the loop and if, after the execution of one or more statements, another control statement returns control back into the loop. In this way, FORTRAN extends the range of the loop to include all of the executable statements between the destination statement of the first transfer and the statement that returns control to the loop.

Figure 7-7 illustrates valid and invalid control transfers.
### FORTRAN IV

#### Valid Control Transfers

<table>
<thead>
<tr>
<th>Line</th>
<th>Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>CONTINUE</td>
</tr>
<tr>
<td>20</td>
<td>A=B+C</td>
</tr>
<tr>
<td>30</td>
<td>X=A*D</td>
</tr>
<tr>
<td>35</td>
<td>CONTINUE</td>
</tr>
<tr>
<td>50</td>
<td>D=E/F</td>
</tr>
<tr>
<td></td>
<td>GOTO 30</td>
</tr>
</tbody>
</table>

#### Invalid Control Transfers

<table>
<thead>
<tr>
<th>Line</th>
<th>Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>A=B+C</td>
</tr>
<tr>
<td>30</td>
<td>D=E/F</td>
</tr>
<tr>
<td>35</td>
<td>CONTINUE</td>
</tr>
<tr>
<td>40</td>
<td>X=A*D</td>
</tr>
<tr>
<td>45</td>
<td>CONTINUE</td>
</tr>
<tr>
<td>50</td>
<td>CONTINUE</td>
</tr>
<tr>
<td></td>
<td>GOTO 30</td>
</tr>
</tbody>
</table>

**Figure 7-7 Control Transfers and Extended Range**

The following rules govern the use of a DO statement extended range:

- The transfer out statement for an extended range operation must be contained by the most deeply nested DO statement that contains the location to which the return transfer is to be made.

- A transfer into the range of a DO statement is permitted only if the transfer is made from the extended range of that DO statement.

- The extended range of a DO statement must not contain another DO statement.

- The extended range of a DO statement cannot change the index variable or indexing parameters of the DO statement.

- You may execute subprograms within an extended range.

#### 7.8.4 CONTINUE Statement

Insert a CONTINUE statement where you do not wish any statement to be executed.

**Format:**

```fortran
st CONTINUE
```
where:

   st   is a statement label.

A CONTINUE statement is a statement that holds a place in the program without performing any operations.

You may place CONTINUE statements anywhere in the source program without affecting the program sequence of execution. CONTINUE statements are commonly used as the last statement of a DO statement range in order to avoid ending with a GOTO, PAUSE, STOP, RETURN, arithmetic IF, another DO statement, or a logical IF statement containing one of the previous statements. However, they are valid throughout a source program.

Note that you also use a CONTINUE as a transfer point for a GOTO statement within the DO loop that is intended to begin another repetition of the loop.

Example:

In the following sequence, the labeled CONTINUE statement provides a legal termination for the range of the DO loop.

```
    DO 45 ITEM=1,1000
    STOCK=VUNTTRY(ITEM)
    IF (STOCK .EQ. TALLY) GO TO 45
    CALL UPDATE(STOCK,TALLY)
    IF (ITEM .EQ. LAST) GO TO 77
    CONTINUE
```

7.8.5 PAUSE Statement

The PAUSE statement temporarily suspends program execution to permit some action on the part of the user.

Format:

```
PAUSE [[num]]
```

where:

   num   is an optional integer variable or expression containing one to five digits.

The PAUSE statement prints the display (if you have specified one) at your terminal, suspends program execution, and waits for you to type the RETURN key. This causes program execution to resume with the first executable statement following the PAUSE.

7-67
Examples:

PAUSE "13731"
PAUSE 'MOUNT TAPE REEL #3'

7.8.6 STOP Statement

Use the STOP statement to terminate program execution.

Format:

STOP

The STOP statement terminates program execution and returns control to the operating system. If you do not type a STOP statement, a "stop" occurs when FORTRAN transfers control to an END statement in the main program unit.

A CALL EXIT statement is equivalent to STOP and closes any temporary files at the last block written on the file. Control returns to the OS/78 Monitor.

Examples:

STOP

999999 STOP

7.8.7 END Statement

The END statement marks the end of every program unit and it must be the last source line of every program unit.

Format:

END

In a main program, if control reaches the END statement, execution of the program terminates; in a subprogram, a RETURN statement is implicitly executed.

In the main program, END is equivalent to STOP. In a subprogram, it is equivalent to RETURN.

A program cannot reference an END statement.

Control returns to the OS/78 Monitor after FORTRAN executes an END statement.

If you do not type an END statement as the last statement in your program, FORTRAN appends one.
7.9 **SUBPROGRAMS**

Procedures you use repeatedly in a program may be written once and then referenced each time you need the procedure. Procedures that you may reference are either internal (written and contained within the program in which they are referenced) or external (self-contained executable procedures that you may compile separately). The kinds of procedures that you may reference are:

- Arithmetic Statement Functions,
- External Functions,
- Subroutines, and
- Intrinsic functions (FORTRAN-defined functions).

7.9.1 **Subprogram Arguments**

Since you may reference subprograms at more than one point throughout a program, many of the values which the subprogram uses may be changed each time the subprogram is called. Dummy arguments in subprograms represent the actual values which the subprogram will use. The arguments are passed to the subprogram when FORTRAN transfers control to it.

Functions and subroutines use dummy arguments to indicate the type of the actual arguments they represent and whether the actual arguments are variables, array elements, arrays, subroutine names, or the names of external functions. You must use each dummy argument within a subprogram as if it were a variable, array, array element, subroutine, or external function identifier. You enter dummy arguments in an "argument list" which you associate with the identifier assigned to the subprogram; actual arguments are normally given in an argument list which you associate with a call made to the subprogram.

The position, number, and type of each dummy argument in a subprogram must agree with the position, number, and type of each argument in the argument list of the subprogram reference.

Dummy arguments may be:

- Variables,
- Array names,
- Subroutine identifiers, or
- Function identifiers.

When you reference a subprogram, FORTRAN replaces its dummy arguments by the corresponding actual arguments which you supply in the reference. All appearances of a dummy argument within a function or subroutine are related to the given actual arguments. Except for subroutine identifiers and literal constants, a valid association between dummy and actual arguments occurs only if both are of the same type; otherwise, the result of the subprogram will be unpredictable. Argument associations may be carried through more than one level of subprogram reference if a valid association is maintained through each level. FORTRAN terminates the dummy/actual argument associations which it establishes when you reference a subprogram. This occurs when FORTRAN completes the operations defined in the subprogram.

7-69
The following rules govern the use and form of dummy arguments:

- The number and type of the dummy arguments of a procedure must be the same as the number and type of the actual arguments given each time you reference the procedure.

- Dummy argument names may not appear in EQUIVALENCE, DATA, or COMMON statements.

- A variable dummy argument should have a variable, an array element identifier, an expression, or a constant as its corresponding argument.

- An array dummy argument should have either an array name or an array element identifier as its corresponding actual argument. If the actual argument is an array, the length of the dummy array should be less than or equal to that of the actual array. FORTRAN associates each element of a dummy array directly with the corresponding elements of the actual array.

- A dummy argument representing an external function must have an external function as its actual argument.

- A dummy argument representing a subroutine identifier should have a subroutine name as its actual argument.

- You may define (or redefine) a dummy argument in a referenced subprogram only if its corresponding actual argument is a variable. If dummy arguments are array names, then you may redefine the elements of the array.

7.9.2 User-written Subprograms

FORTRAN transfers control to a function by means of a function reference. It transfers control to a subroutine by a CALL statement. A function reference is the name of the function, together with its arguments, appearing in an expression. A function always returns a value to the calling program. Both functions and subroutines may return additional values via assignment to their arguments. A subprogram can reference other subprograms, but it cannot, either directly or indirectly, reference itself (that is, FORTRAN is not recursive).

7.9.2.1 Arithmetic Statement Functions (ASE) - Use an Arithmetic Statement Function to define a one statement, self-contained computational procedure.

Format:

\[
\text{nam} ([\text{a}[,.\text{a}] \ldots]) = \text{e}
\]

where:

- \text{nam} is the name you assign to the ASF,
- \text{a} is a dummy argument, and
- \text{e} is an expression.

Examples:

\[
\text{PROOT}(A,B,C) = (-B+\text{SQRT}(B**2 - 4*A*C))/(2*A)
\]

\[
\text{NROOT}(A,B,C) = (-B-\text{SQRT}(B**2 - 4*A*X))/(2*A)
\]

7-70
An arithmetic statement function is a computing procedure which you define by a single statement, similar in form to an arithmetic assignment statement. The appearance of a reference to the function within the same program unit causes FORTRAN to perform the computation and make the resulting value available to the expression in which the ASF reference appears.

The expression e is an arithmetic expression that defines the computation to be performed by the ASF.

You reference an ASF in the same manner as an external function.

Format:

\[ \text{nam} \left( \left[ a \left[ a, \ldots \right] \right] \right) \]

where:

- \text{nam} is the name of the ASF, and
- a is an actual argument.

\textbf{NOTE}

You must define all ASFs before you type any executable statements.

When a reference to an arithmetic statement function appears in an expression, FORTRAN associates the values of the actual arguments with the dummy arguments in the ASF definition. FORTRAN then evaluates the expression in the defining statement and uses the resulting value to complete the evaluation of the expression containing the function reference.

Specify the data type of an ASF either implicitly by the initial letter of the name or explicitly in a type declaration statement.

Dummy arguments in an ASF definition only indicate the number, order, and data type of the actual arguments. You may use the same names to represent other entities elsewhere in the program unit. Note also that with the exception of data type, FORTRAN does not associate declarative information (such as placement in COMMON or declaration as an array) with the ASF dummy arguments. Note that you cannot use the name of the ASF to represent any other entity within the same program unit.

The expression in an ASF definition may contain function references.

Any reference to an ASF must appear in the same program unit as the definition of that function. You cannot use an ASF name in an \texttt{EXTERNAL} statement.

An ASF reference must appear as, or be part of, an expression; you must not use it as the left side of an assignment statement.

Actual arguments must agree in number, order, and data type with their corresponding dummy arguments. You must assign values to actual argument before the reference to the arithmetic statement function.
FORTRAN IV

Examples:

Definitions

VOLUME(RADIUS) = 4.189*RADIUS**3

SINH(X) = (EXP(X)-EXP(-X))*0.5

AVG(A,B,C,3.) = (A+B+C)/3.  (invalid; constant as dummy argument not permitted)

AVG(A,B,C) = (A+B+C)/3.  (definition)

.

GRADE = AVG(TEST1,TEST2,XLAB)

IF (AVG(P,D,Q).LT.AVG(X,Y,Z))  GOTO 300

FINAL = AVG(TEST3,TEST4,LAB2)  (Invalid; data type of third argument does not agree with dummy argument)

7.9.2.2 FUNCTION Subprogram - A FUNCTION is an external computing procedure that returns a value. You use this value as an expression or as part of an expression.

Format:

[[typ]] FUNCTION nam(a [[,a...]])

where:

typ  is an optional data type specifier,

nam  is a name of the function, and

a    is one of a maximum of six dummy arguments.

A FUNCTION subprogram is a program unit that consists of a FUNCTION statement followed by a series of statements that define a computing procedure. FORTRAN transfers control to a FUNCTION subprogram by a function reference and returns to the calling program unit when it encounters a RETURN statement.

You must always specify at least one argument to a FUNCTION. You may specify other arguments explicitly or place them in COMMON.

A FUNCTION subprogram returns a single value to the calling program unit by assigning that value to the function's name. FORTRAN determines the data type of the returned value by the function's name unless you have explicitly specified the data type.

A function reference that transfers control to a FUNCTION subprogram has the form:

nam ( [[a [[,a]] ...]])

where:

nam  is the symbolic name of the function, and

a    is an actual argument.

7-72
When FORTRAN transfers control to a function subprogram, FORTRAN associates the values you supply by the actual arguments (if any) with the dummy arguments (if any) in the FUNCTION statement. FORTRAN then executes the statements in the subprogram.

NOTE

You may not pass an array to a subprogram if it contains more than 2047 elements. You must implicitly pass larger arrays in COMMON.

You must assign a value to the name of the function before FORTRAN executes a RETURN statement in that function. When FORTRAN returns control to the calling program unit, it makes the value you have assigned to the function's name available to the expression that contains the function reference; it then uses this value to complete the evaluation of the expression.

NOTE

You can store variables that a FUNCTION requires in COMMON rather than passing them explicitly.

You may specify the type of a function name implicitly, explicitly in the FUNCTION statement, or explicitly in a type declaration statement.

The FUNCTION statement must be the first statement of a function subprogram. You may not label a FUNCTION statement.

A FUNCTION subprogram must not contain a SUBROUTINE statement, a BLOCK DATA statement, or a FUNCTION statement (other than the initial statement of the subprogram). A function may, however, call another function or subroutine so long as the call is not directly or indirectly recursive.

7.9.2.3 SUBROUTINE Subprograms - A SUBROUTINE is an external computing procedure that you may repeatedly call from a program or subprogram.

Format:

```
SUBROUTINE nam ( [ ( a [, a ] ... ) ] )
```

where:

- nam is the name of the subroutine, and
- a is a dummy argument.

A SUBROUTINE subprogram is a program unit that consists of a SUBROUTINE statement followed by a series of statements that define a computing procedure. FORTRAN transfers control to a SUBROUTINE subprogram by a CALL statement and returns to the calling program unit by a RETURN statement.
When FORTRAN transfers control to a subroutine, it associates the values you supply with the actual arguments (if any) in the CALL statement with the corresponding dummy arguments (if any) in the SUBROUTINE statement. You may not specify more than six arguments in a subroutine call. FORTRAN then executes the statements in the subroutine.

The SUBROUTINE statement must be the first statement of a subroutine; it must not have a statement label.

A SUBROUTINE subprogram cannot contain a FUNCTION statement, a BLOCK DATA statement, or a SUBROUTINE statement (other than the initial statement of the subprogram).

Example:

```
C        MAIN PROGRAM
COM     COMMON NFACES, EDGE, VOLUME
READ (4,65) NFACES, EDGE
65     FORMAT(I2,F8.5)
     CALL PLYVOL
     WRITE (4,66) VOLUME
66     FORMAT (' VOLUME=','F)
     STOP
     END

SUBROUTINE PLYVOL
COM     COMMON NFACES, EDGE, VOLUME
     CUBED = EDGEX3
     GOTO (6,6,6,6,1,6,2,6,3,6,6,4,6,6,6,6,6,5,6,NFACES
1     VOLUME = CUBED * 0.11785
     RETURN
2     VOLUME = CUBED
     RETURN
3     VOLUME = CUBED * 0.47140
     RETURN
4     VOLUME = CUBED * 7.66312
     RETURN
5     VOLUME = CUBED * 2.18170
     RETURN
6     WRITE (4,100) NFACES
100    FORMAT(' NO REGULAR POLYHEDRON HAS ',I3,'FACES.\')
     RETURN
     END
```

The subroutine in this example computes the volume of a regular polyhedron, given the number of faces and the length of one edge. It uses a computed GOTO statement to determine whether the polyhedron is a tetrahedron, cube, octahedron, dodecahedron, or icosahedron, and to transfer control to the proper procedure for calculating the volume. If the number of faces of the body is other than 4, 6, 8, 12, or 20, the subroutine transmits an error message to logical unit 4 as indicated in the WRITE statement.

7.9.3 CALL Statement

The CALL statement causes the execution of a SUBROUTINE subprogram; it can also specify an argument list for use by the subroutine.

Format:

```
CALL s [( [a] [,a] ... ])
```
where:

- **s** is the name of a SUBROUTINE subprogram, a user-written assembly language routine, or a DEC-supplied system subroutine, or a dummy argument associated with one of the above.
- **a** is an actual argument.

The CALL statement associates the values in the argument list (if the list is present) with the dummy arguments in the subroutine and then transfers control to the first executable statement of the subroutine.

The arguments in the CALL statement must agree in number, order, and data type with the dummy arguments in the subroutine definition. They can be variables, arrays, array elements, constants, expressions, alphanumerical literals, or subprogram names (if those names have been specified in an EXTERNAL statement, as described in Section 5.4). Note that an unsubscripted array name in the argument list refers to the entire array.

Examples:

```
CALL CURVE (BASE,3.14159+X,Y,LIMIT,R(LT+2))
CALL PNTOUT (A,N,'ABCD')
```

### 7.9.4 RETURN Statement

Use the RETURN statement to return control from a subprogram unit to the calling program unit.

**Format:**

```
RETURN
```

When FORTRAN executes a RETURN statement in a FUNCTION subprogram, it returns control to the statement that contains the function reference (see Section XXX). When FORTRAN executes a RETURN statement in a SUBROUTINE subprogram, it returns control to the first executable statement following the CALL statement which initiated execution of the subprogram.

A RETURN statement must not appear in a main program unit.

Example:

```
SUBROUTINE CONVRT (N,ALPH,DATA,PRNT,K)
DIMENSION DATA(N), PRNT(N)
IF (N .LT. 10) GO TO 100
DATA(K+2) = N-(N/10) * N
N = N/10
DATA(K+1) = N
PRNT(K+2) = ALPH(DATA(K+2)+1)
PRNT(K+1) = ALPH(DATA(K+1)+1)
RETURN
```

```
100  PRNT(K+2) = ALPH(N+1)
RETURN
END
```
7.9.5 FORTRAN Library Functions

The FORTRAN library functions are listed and described in Section 7.12. You write function references to FORTRAN library functions in the same form as function references to user-defined functions. For example,

\[ R = 3.14159 \times \text{ABS}(X-1) \]

causes the absolute value of X-1 to be calculated, multiplied by the constant 3.14159, and assigned to the variable R.

The data type of each library function and the data type of the actual arguments is specified in Appendix B. Arguments you pass to these functions may not be array names or subprogram names.

Processor-defined function references are local to the program unit in which they occur and do not affect or preclude the use of the name for any other purpose in other program units.

7.10 INPUT/OUTPUT STATEMENTS

You specify input of data to a program by READ statements and output by WRITE statements. You use some form of these statements in conjunction with format specifications to control translation and editing of the data between internal representation and character (readable) form.

Each READ or WRITE statement contains a reference to the logical unit to or from which data transfer is to take place. You may associate a logical unit to a device or file.

READ and WRITE statements fall into the following three categories:

- **Unformatted Sequential I/O**
  Unformatted sequential READ and WRITE statements transmit binary data without translation.

- **Formatted Sequential I/O**
  Formatted sequential READ and WRITE statements transmit character data using format specifications to control the translation of data to characters on output, and to internal form on input.

- **Unformatted Direct Access I/O**
  Unformatted direct access READ and WRITE statements transmit binary data without translation to and from direct access files.

The auxiliary I/O statements, REWIND and BACKSPACE do not perform data transfer, but perform file positioning. The ENDFILE statement writes a special record that will cause an end-of-file condition when read by a READ statement. The BACKSPACE statement repositions a file to the previous record. The DEFINE FILE statement declares a logical unit to be connected to a direct access file and specifies the characteristics of the file.

7.10.1 Defining I/O Operations

FORTRAN I/O operations require knowledge of logical unit numbers, format specifiers, and record transmission.
7.10.1.1 Input/Output Devices and Logical Unit Numbers - OS/78

FORTRAN uses the I/O services of the operating system and thus supports all peripheral devices that are supported by the operating system. I/O statements refer to I/O devices by means of logical unit numbers which are integer constants or variables with a positive value.

The default logical unit numbers are:

3  Line Printer
4  Terminal

The logical unit number must be in the range 1 through 9. For more information, see Sections 7.1.2.1 and 7.1.3.

7.10.1.2 Format Specifiers - Use format specifiers in formatted I/O statements. A format specifier is the statement label of a FORMAT statement. Section 7.11 discusses FORMAT statements.

7.10.1.3 Input/Output Record Transmission - I/O statements transmit data in terms of records. The amount of information that one record can contain, and the way in which records are separated, depend on the medium involved.

For unformatted I/O, specify the amount of data which FORTRAN will transmit by an I/O statement. FORTRAN determines the amount of information it will transmit by the I/O statement and by specifications in the associated format specification.

If an input statement requires only part of a record, the excess portion of the record is lost. In the case of formatted sequential input or output, you may transmit one or more additional records by a single I/O statement.

7.10.2 Input/Output Lists

An I/O list specifies the data items to be manipulated by the statement containing the list. The I/O list of an input or output statement contains the names of variables, arrays, and array elements whose values FORTRAN will transmit. In addition, the I/O list of an output statement can contain constants and expressions.

Format:

s[,s]...

where:

s  is a simple list or an implied DO list.

The I/O statement assigns input values to, or outputs values from, the list elements in the order in which they appear, from left to right.
7.10.2.1 Simple Lists - A simple I/O list consists of a single variable, array, array element, constant, or expression.

When an unsubscripted array name appears in an I/O list, a READ statement inputs enough data to fill every element of the array; a WRITE statement outputs all of the values contained in the array. Data transmission starts with the initial element of the array and proceeds in the order of subscript progression, with the leftmost subscript varying most rapidly. For example, if the unsubscripted name of a 2-dimensional array defined as:

```fortran
DIMENSION ARRAY(3,3)
```

appears in a READ statement, that statement assigns values from the input record(s) to ARRAY(1,1), ARRAY(2,1), ARRAY(3,1), ARRAY(1,2), and so on, through ARRAY(3,3).

If, in a READ statement, you input the individual subscripts for an array, you must input the subscripts before their use in the array. If, for example, FORTRAN executes the statement:

```fortran
READ (1,1250) J,K,ARRAY(J,K)
1250 FORMAT (I1,X,I1,X,F6.2)
```

and the input record contains the values:

```
1,3,721.73
```

FORTRAN assigns the value 721.73 to ARRAY(1,3). FORTRAN assigns the first input value to J and the second to K, thereby establishing the actual subscript values for ARRAY(J,K). Variables that you use as subscripts in this way must appear to the left of their use in the array subscript.

You may use any valid expression in an output statement I/O list. However, the expression must not cause FORTRAN to attempt further I/O operations. A reference in an output statement I/O list expression to a FUNCTION subprogram that itself performs input/output is illegal.

You must not include an expression in an input statement I/O list except as a subscript expression in an array reference.

7.10.2.2 Implied DO Lists - Use an implied DO list to specify iteration within an I/O list.

Format:

```fortran
(list,i=e1,e2)
```

where:

- list is an I/O list,
- i is a control variable definition,
- e1 is the initial value of i, and
- e2 is the terminal value of i.

You use an implied DO list to specify iteration within an I/O list, to transmit only part of an array, or to transmit array elements in a sequence other than the order of subscript progression. The implied DO list functions as though it were a part of an I/O statement that resides in a DO loop.
When you use nested implied DO lists, the first control variable definition is equivalent to the innermost DO of a set of nested loops, and therefore varies most rapidly. For example, the statement:

```
WRITE (5,150) ((FORM(K,L), L=1,10), K=1,10)
150 FORMAT (F10.2)
```

is similar to:

```
DO 50 K=1,10
DO 50 L=1,10
WRITE (5,150) FORM(K,L)
150 FORMAT (F10.2)
50 CONTINUE
```

Since the inner DO loop is executed ten times for each iteration of the outer loop, the second subscript, L, advances from one through ten for each increment of the first subscript. This is the reverse of the order of subscript progression.

The implied DO uses the control variable of the imaginary DO statement to specify which value or values are to be transmitted during each iteration of the loop.

i, el, and e2 have the same form as that used in the DO statement. The rules for the control, initial, and terminal variables of an implied DO list are the same as those for the DO statement. Note, however, an implied DO loop cannot use an increment parameter. The list may contain references to the control variable as long as the value of the control variable is not altered. There is no extended range for an implied DO list.

Examples:

```
WRITE (3,200) (A,B,C, I=1,3)
WRITE (6,15) L,M,(I,(J,P(I),Q(I,J),J=1,L),I=1,M)
READ (1,75) (((ARRAY(M,N,I), I=2,8), N=2,8), M=2,8)
```

FORTRAN transmits the entire list of the implied DO before the incrementation of the control variable. For example:

```
READ (3,999) (P(I), (Q(I,J), J=1,10), I=1,5)
```

assigns input values to the elements of arrays P and Q in the order:

```
P(1), Q(1,1), Q(1,2), ..., Q(1,10),
P(2), Q(2,1), Q(2,2), ..., Q(2,10),
  .  .  .  .  .
P(5), Q(5,1), Q(5,2), ..., Q(5,10)
```

When processing multi-dimensional arrays, you may use a combination of a fixed subscript and subscript or subscripts that varies according to an implied DO. For example:

```
READ (3,5555) (BOX(1,J), J=1,10)
```

assigns input values to BOX(1,1) through BOX(1,10), then terminates without affecting any other element of the array.
It is also possible to output the value of the control variable directly, as in the statement:

\[
\text{WRITE (6,1111) (I, I=1,20)}
\]

which simply prints the integers 1 through 20.

7.10.3 Input/Output Forms

7.10.3.1 Unformatted Sequential Input/Output - FORTRAN provides two types of I/O—unformatted and formatted. Unformatted input and output is the transfer of data in internal (binary) format without conversion or editing. Use unformatted I/O when data output by a program is to be subsequently input by the same program (or a similar program). Unformatted I/O saves execution time because it eliminates the data conversion process, preserves greater precision in the external data, and usually conserves file storage space.

7.10.3.2 Formatted Sequential Input/Output - Use formatted input and output statements in conjunction with FORMAT statements to translate and edit data on output for ease of interpretation, and, on input, to convert data from external format to internal format.

7.10.3.3 Unformatted Direct Access Input/Output - Use unformatted direct access READ and WRITE statements to perform direct access I/O with a file on a direct access device. Use the DEFINE FILE statement to establish the number of records, and the size of each record, in a file to which FORTRAN will perform direct access I/O. Each direct access READ or WRITE statement contains an integer expression that specifies the number of the record to be accessed. The record number must not be less than one nor greater than the number of records you define for the file.

In OS/78 FORTRAN, the expression that specifies the record number can be of any type. FORTRAN converts it to integer type if necessary.

7.10.4 READ Statements

FORTRAN provides the following READ statements.

7.10.4.1 Unformatted Sequential READ Statement - Use unformatted sequential read statements to assign fields to a record without translating stored information into external form.

Format:

\[
\text{READ (u) [list]}
\]

where:

\[
u \text{ is a logical unit number from 1 to 9, and list is an I/O list.}
\]
The unformatted sequential READ statement inputs one unformatted record from a logical unit and assigns the fields of the record without translation to the I/O list elements in the order in which they appear, from left to right.

An unformatted sequential READ statement transmits exactly one record. If the I/O list does not use all of the values in the record, FORTRAN discards the remainder of the record. If FORTRAN exhausts the contents of the record before the I/O list is satisfied, an error condition results.

You must only use the unformatted sequential READ statement to read records that were created by unformatted sequential WRITE statements.

If you use an unformatted WRITE statement that does not contain an I/O list, FORTRAN skips the next record.

Examples:

```
READ (1) FIELD1, FIELD2
assign values to variables FIELD1 and FIELD2.
```

```
READ (8)
Advance logical unit 8 one record.
```

7.10.4.2 Formatted Sequential READ Statement - Use formatted sequential read statements to transmit information in external format.

Format:

```
READ (u,f) [[ list ]]
```

where:

- \( u \) is a logical unit number from 1 to 9,
- \( f \) is a format statement number, and
- list is an I/O list.

The formatted sequential READ statement transfers data from the indicated logical unit. FORTRAN converts transmitted characters to internal format as specified by the format specification. FORTRAN assigns the resulting values to the elements of the I/O list.

If the FORMAT statement associated with a formatted input statement contains a Hollerith constant or alphanumeric literal, input data will be read and stored directly into the format specification. For example, the statements

```
READ (5,100)
100 FORMAT (5H DATA)
```

cause five characters to be read and stored in the Hollerith format descriptor. If the character string were HELLO, statement 100 would become:

```
100 FORMAT (5HHELLO)
```

If there is no \( H \) field, the record is skipped.
If the number of elements in the I/O list is less than the number of fields in the input record, the excess portion of the record is discarded. If the number of elements in the list exceeds the number of input fields, an error condition results unless the format specifications state that one or more additional records are to be read (see Section 10.8).

If no I/O list is present, data transfer is between the record and the format specification.

Examples:

```
READ (1,300) ARRAY
300          FORMAT (20F8.2)

Read a record from logical unit 1, assign fields to ARRAY.

READ (5,50)
50          FORMAT (25H PAGE HEADING GOES HERE)

Read 25 characters from logical unit 5, place them in the FORMAT statement.
```

The CHKEOF subroutine returns a non-zero value if the logical end of a file is encountered during a formatted READ operation.

CHKEOF accepts one real, integer, or logical argument. After the next formatted READ operation, this argument will be set to a non-zero value if the logical end-of-file was encountered. Otherwise, it will be set to zero.

Only use CHKEOF when reading one record from the logical unit.

The following is an example of the use of CHKEOF.

```
.
.
CALL CHKEOF (EOF)
READ (N,101) DATA
IF (EOF .NE. 0) GO TO 9999
.
.
```

7.10.4.3 Unformatted Direct Access READ Statement - Use an unformatted direct access read statement to transmit a value or values to a direct access device in internal format.

Format:

```
READ (u'r') [[list]]
```

where:

- `u` is a logical unit number from 1 to 9,
- `r` is the record number, and
- `list` is an I/O list.

The unformatted direct access READ statement positions the input file to a specified record and transfers the fields in that record to the elements in the I/O list without translation.
u may be an unsigned integer constant or a positive integer variable. r may also be a variable. If there are more fields in the input record than elements in the I/O list, FORTRAN discards the excess portion of the record. If there is insufficient data in the record to satisfy the requirements of the I/O list, an error condition results.

The unit number in the unformatted direct access READ statement must refer to a unit that you have previously

Examples:

READ (1'10) LIST(1),LIST(8)  Read record 10 of a file on logical unit 1, assign two INTEGER values to specified elements of array LIST.

READ (4'58) (RHO(N),N=1,5)  Read record 58 of a file on logical unit 4, assign five real values to array RHO.

7.10.5 WRITE Statements

7.10.5.1 Unformatted Sequential WRITE Statement - FORTRAN includes the following formatted and unformatted WRITE statements. Use as unformatted sequential write statement to transmit values in their internal representation to a logical unit.

Format:

WRITE (u) [list]

where:

u is a logical unit number from 1 to 9, and
list is an I/O list.

The unformatted sequential WRITE statement transmits the values of the elements in the I/O list to the specified logical unit, without translation, as one unformatted record.

The logical unit specifier is an integer variable or an integer constant from 1 to 9.

If an unformatted WRITE statement contains no I/O list, one null record is output to the specified unit.

A record may hold 85 single precision variables. If the list elements fill more than one record, FORTRAN writes successive records until the list is completed. Thus, if there are 100 variables on the list, FORTRAN uses two records; one record contains 85 variables and the second contains 15 variables. For example

DIMENSION X(200)
WRITE (6) X

will produce three records on logical unit 6, the first containing X(1) to X(85), the second X(86) to X(170), and the third X(171) to X(200). If the amount of data FORTRAN will transmit exceeds the record size, an error condition results. If the WRITE statement does not completely fill the record with data, FORTRAN zero fills the unused portion of the record.
FORTRAN IV

Examples:

```
WRITE (1) (LIST(K), K=1,5)      Output the contents of elements 1
               through 5 of array LIST to logical
               unit 1.

WRITE (4)                        Write a null record on logical unit
```

4. 7.10.5.2 Formatted Sequential WRITE Statement - Use a formatted
sequential write statement to translate a value from its internal
representation to character format and then transmit it to a logical
unit.

Format:

```
WRITE (u,f) [[list]]
```

where:

- `u` is a logical unit number from 1 to 9,
- `f` is a format statement number, and
- `list` is an I/O list.

The formatted sequential WRITE statement transfers data to the
specified logical unit. The I/O list specifies a sequence of values
which FORTRAN converts to characters and positions as specified by a
format specification.

The logical unit specifier may be an integer variable.

If no I/O list is present, data transfer is entirely between the
record and the format specification.

The data FORTRAN transmits by a formatted sequential WRITE statement
normally constitutes one formatted record. The format specification
can, however, specify that additional records are to be written during
the execution of that same WRITE statement.

FORTRAN rounds numeric data output under format control during the
conversion to external format. (If such data is subsequently input
for additional calculations, loss of precision may result. In this
case, unformatted output is preferable to formatted output.)

The records FORTRAN transmits by a formatted WRITE statement must not
exceed the length that the specified device can accept. For example,
a line printer typically cannot print a record that is longer than 132
characters. [If longer, run to next line]

Examples:

```
WRITE (6, 650)  (Output the contents of the
650 FORMAT ('HELLO, THERE')   FORMAT statement to logical
               unit 6.)

WRITE (1,95) AYE, BEE, CEE   (Write one record of three
95 FORMAT (F8.5, F8.5, F8.5) fields to logical unit 1.)

WRITE (1,950) AYE, BEE, CEE   (Write three separate records
950 FORMAT (F8.5) of one field each to logical
               unit 1.)
```
In the last example, format control arrives at the rightmost parenthesis of the FORMAT statement before all elements of the I/O list have been output. Each time this occurs, FORTRAN terminates the current record and initiates a new record. Thus, FORTRAN writes three separate records. (See Section 10.5.)

7.10.5.3 Unformatted Direct Access WRITE Statement - Use an unformatted direct access write statement to transmit a value in its internal representation to a specific record on a direct access device.

Format:

```fortran
WRITE (u'r') [list]
```

where:

- `u` is a logical unit number from 1 to 9,
- `r` is the record number, and
- `list` is an I/O list.

The unformatted direct access WRITE statement transmits the values of the elements in the I/O list to a particular record position on a direct access file. The data is written in internal format without translation.

The logical unit specifier `r` may be an unsigned integer constant or integer variable.

A record may hold a maximum of 85 single precision variables. If the list elements fill more than one record, FORTRAN writes successive records until the list is completed. Thus, if there are 100 variables on the list, FORTRAN uses two records; one record contains 85 variables and the second contains 15 variables. For example

```fortran
DIMENSION X(200)
WRITE (6) X
```

will produce three records on unit 6, the first containing `X(1)` to `X(85)`, the second `X(86)` to `X(170)`, and the third `X(171)` to `X(200)`. If the amount of data FORTRAN will transmit exceeds the record size, an error condition results. If the WRITE statement does not completely fill the record with data, FORTRAN zero fills the unused portion of the record.

Examples:

```fortran
WRITE (2'35) (NUM(K), K=1,10)  (Output ten integer values to record 35 of the file connected to logical unit 2.)
```

```fortran
WRITE (3'J) ARRAY
```

(Output the entire contents of ARRAY to the file connected to logical unit 3 into the record indicated by the value of J.)

7.10.6 Auxiliary Input/Output Statements

You use statements in this category to perform file management functions.
7.10.6.1 **BACKSPACE** Statement - Use the **BACKSPACE** statement to reposition a file to the previous record accessed.

Format:

```
BACKSPACE u
```

where:

`u` is a logical unit number from 1 to 9.

The **BACKSPACE** statement repositions a currently open sequential file backward one record and repositions it to the beginning of that record. On the execution of the next I/O statement for that unit, that record is available for processing.

The unit number must refer to a directory-structured device (e.g., disk). A file must be open on that device.

If the file is positioned at the first record, FORTRAN ignores the **BACKSPACE** statement.

Example:

```
BACKSPACE 4  (Reposition open file on logical unit 4 to beginning of the previous record.)
```

7.10.6.2 **DEFINE FILE** Statement - The **DEFINE FILE** statement establishes the size and structure of a file upon which FORTRAN will perform direct access I/O.

Format:

```
DEFINE FILE u (m,n,U,v) [u(m,n,U,v)] ...
```

where:

`u` is an integer constant or variable that specifies the logical unit number,

`m` is an integer constant or variable that specifies the number of records in the file,

`n` is an integer constant or variable that specifies the length, in words, of each record,

`U` specifies that the file is unformatted (binary) and the letter `U` is the only acceptable entry in this position, and

`v` is an integer variable, called the associated variable of the file.

The **DEFINE FILE** is the means by which you specify the attributes of a direct access device. Once the file characteristics have been established, you should always specify them in the same manner.

At the conclusion of each direct access I/O operation, FORTRAN assigns the record number of the next higher numbered record in the file to `v`.

The **DEFINE FILE** statement specifies that a file containing `m` fixed-length records of `n` words each exists or is to exist, on logical unit `u`. The records in the file are sequentially numbered from 1 through `m`.

You must type the **DEFINE FILE** statement before the first direct access I/O statement that refers to the specified file.
The DEFINE FILE statement also establishes the integer variable \( v \) as the associated variable of the file. At the end of each direct access I/O operation, the FORTRAN I/O system places in \( v \) the record number of the record immediately following the one just read or written. Because the associated variable always points to the next sequential record in the file (unless you redefine it by an assignment or input statement), you can use direct access I/O statements to perform sequential processing of the file. The logical unit number \( u \) cannot be passed as a dummy argument to a DEFINE FILE statement in a subroutine.

If more than one program unit processes the file, or in an overlay environment, the associated variable should be placed in a resident common block.

Example:

```fortran
DEFINE FILE 3 (1000,48,U,NREC)
```

This statement specifies that logical unit 3 is to be connected to a file of 1000 fixed-length records, each record of which is 48 words long. The records are numbered sequentially from 1 through 1000, and are unformatted. After each direct access I/O operation on this file, the integer variable NREC will contain the record number of the record immediately following the one just processed.

7.10.6.3 ENDFILE Statement - The ENDFILE statement writes an end-file record to the specified sequential unit.

Format:

```fortran
ENDFILE u
```

where:

- \( u \) is a logical unit number from 1 to 9.

Use the ENDFILE statement to write an end-of-file mark on a directory-structured device. [Note that you cannot write additional information to that device after the ENDFILE statement.]

The ENDFILE statement must be written to a formatted output file.

No rewind occurs after this statement.

Example:

```fortran
ENDFILE 2   (Output an end-file record to logical unit 2.)
```

7.10.6.4 REWIND Statement - The REWIND statement repositions a currently open sequential file to be repositioned to the beginning of the file.

Format:

```fortran
REWIND u
```

where:

- \( u \) is a logical unit number from 1 to 9.
FORTRAN IV

Use the REWIND statement to position a directory-structured device to its first record.

If the file is already at its first record, FORTRAN ignores the REWIND statement.

The unit number in the REWIND statement must refer to a directory-structured device (e.g., disk). A file must be open on that device.

Example:

REWIND 3 (Reposition logical unit 3 to beginning of currently open file.)

7.11 FORMAT STATEMENTS

FORMAT statements are nonexecutable statements used in conjunction with formatted I/O statements. The FORMAT statement describes the format in which FORTRAN transmits data fields, and the data conversion and editing to be performed to achieve that format.

The FORMAT statement has the form:

st FORMAT (g1f1s1 [f2s2] ... [fnqn])

where:

f is a field descriptor, or a group of field descriptors enclosed in parentheses,
s is a field separator (either a comma or slash),
g is zero or more slash (/) record terminators
st is a mandatory statement number.

including the parentheses is called the format specification. You must enclose the list in parentheses.

A field descriptor in a format specification has the form:

[r]cw[.d]

where:

r represents a repeat count which specifies that FORTRAN is to apply the field descriptor to r successive fields. If you omit the repeat count, FORTRAN assumes it to be 1.
c is a format code,
w is the field width, and
d is the number of characters to the right of the decimal point, and should be less than w.

The terms r, w, and d must all be unsigned integer constants less than or equal to 255.

The field separators are comma and slash. A slash can also be a record terminator. Use a slash to skip records or lines in a record.

The field descriptors used in format specifications are as follows:

1. Integer: Iw
2. Logical: Lw
3. Real: \text{Fw.d, Ew.d, Dw.d, Gw.d, Bw.d}

4. Literal and editing: \text{Aw, nH, nP, nX, Tn, $, ', ..', /}

(In the alphanumeric and editing field descriptors, \( n \) specifies the number of characters or character positions.)

You can precede the F, E, D, or G field descriptors by a scale factor of the form:

\[ \text{n}_p \]

where:

\[ n \]

is an optionally signed integer constant in the range \(-127\) to \(+127\) the scale factor specifies the number of positions the decimal point is to be scaled to the left or right.

During data transmission, FORTRAN scans the format specification from left to right. FORTRAN then performs data conversion by correlating the values in the I/O list with the corresponding field descriptors. In the case of H field descriptors and alphanumeric literals, data transmission takes place entirely between the field descriptor and the external record.

For example, consider the following data for input (where \( b \) equals a blank space).

\[ b10.2bb6732bb3967.61 \]

To read this data, use the following FORMAT statement in conjunction with a READ statement.

\[ 20 \text{ FORMAT (LX,F3.1,2X,I4,2X,F6.2)} \]

where the field descriptor:

1X Indicates a blank space.

F3.1 Indicates a 3-digit real number with 1 decimal place.

2X Indicates 2 blank spaces.

I4 Indicates a 4-digit integer number.

2X Indicates 2 blank spaces.

F6.2 Indicates a 6-digit real number with 2 decimal places.

7.11.1 Field Descriptors

The individual field descriptors that can appear in a format specification are described in detail in the following sections. The field descriptors ignore leading spaces in the external field, but treat embedded and trailing spaces as zeros.
7.11.1.1 I Field Descriptor - The I field descriptor governs the translation of integer data.

Format:

Iw

Input

The I field descriptor causes an input statement to read w characters from an external record. FORTRAN then assigns the character as an integer value to the corresponding integer element of the I/O list. The external data must be an integer; it must not contain a decimal point or exponent field.

The I field descriptor interprets an all-blank field as a zero value.

If the value of the external field exceeds the range of the corresponding integer list element, an error occurs. If the first non-blank character of the external field is a minus symbol, the I field descriptor causes the field to be stored as a negative value; FORTRAN treats a field preceded by a plus symbol, or an unsigned field, as a positive value.

Examples:

<table>
<thead>
<tr>
<th>Format</th>
<th>External Field</th>
<th>Internal Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>I4</td>
<td>2788</td>
<td>2788</td>
</tr>
<tr>
<td>I3</td>
<td>-26</td>
<td>-26</td>
</tr>
<tr>
<td>I9</td>
<td>312</td>
<td>312</td>
</tr>
<tr>
<td>I9</td>
<td>3.12</td>
<td>not permitted; error</td>
</tr>
<tr>
<td></td>
<td>-871</td>
<td>-87</td>
</tr>
</tbody>
</table>

Output

On output, the I field descriptor transmits the value of the corresponding integer I/O list element, right justified, to an external field w characters in length. It also replaces any leading zeros with spaces. If the value does not fill the field, FORTRAN inserts leading spaces. If the value of the list element is negative, the field will have a minus symbol as its leftmost non-blank character. Space must therefore be included in w for a minus symbol if you expect one to be output. FORTRAN suppresses plus symbols and you need not account for them in w. If w is too small to contain the output value, FORTRAN fills the entire external field with asterisks.

Examples:

<table>
<thead>
<tr>
<th>Format</th>
<th>Internal Value</th>
<th>External Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>I3</td>
<td>284</td>
<td>284</td>
</tr>
<tr>
<td>I4</td>
<td>-284</td>
<td>-284</td>
</tr>
<tr>
<td>I5</td>
<td>174</td>
<td>174</td>
</tr>
<tr>
<td>I2</td>
<td>3244</td>
<td>**</td>
</tr>
<tr>
<td>I3</td>
<td>-473</td>
<td>***</td>
</tr>
<tr>
<td>I7</td>
<td>29.812</td>
<td>not permitted; error</td>
</tr>
</tbody>
</table>
7.11.1.2 F Field Descriptor - The F field descriptor specifies the data conversion and editing of real values.

Format:

Fw.d

Input

On input, the F field descriptor causes FORTRAN to read w characters from the external record and to assign the characters as a real value to the corresponding I/O list element. If the first non-blank character of the external field is a minus sign, FORTRAN treats the field as a negative value; FORTRAN assumes a field preceded by a plus sign (or an unsigned field) to be positive. FORTRAN considers an all-blank field to have a value of zero. In all appearances of the F field descriptor, w must be greater than or equal to d+ where the extra character is the decimal point.

If the field contains neither a decimal point nor an exponent, FORTRAN treats it as a real number of w digits, in which the rightmost d digits are to the right of the decimal point. If the field contains an explicit decimal point, the location of that decimal point overrides the location you specify in the field descriptor. If the field contains an exponent, FORTRAN uses the exponent to establish the magnitude of the value before it assigns the value to the list element.

Examples:

<table>
<thead>
<tr>
<th>Format</th>
<th>External Field</th>
<th>Internal Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>F8.5</td>
<td>123456789</td>
<td>123.45678</td>
</tr>
<tr>
<td>F8.5</td>
<td>-1234.567</td>
<td>-1234.56</td>
</tr>
<tr>
<td>F8.5</td>
<td>24.77E+2</td>
<td>2477.0</td>
</tr>
<tr>
<td>F5.2</td>
<td>1234567.89</td>
<td>123.45</td>
</tr>
</tbody>
</table>

Output

On output, the F field descriptor causes FORTRAN to round the value of the corresponding I/O list element to d decimal positions and to transmit an external field w characters in length, right justified. If the converted data consists of fewer than w characters, FORTRAN inserts leading spaces; if the data exceeds w characters, FORTRAN fills the entire field with asterisks.

The field width must be large enough to accommodate 1) a minus sign, if you expect one to be output (FORTRAN suppresses plus signs), 2) at least one digit to the left of the decimal point, 3) the decimal point itself, and 4) d digits to the right of the decimal. For this reason, w should always be greater than or equal to (d+3).

Examples:

<table>
<thead>
<tr>
<th>Format</th>
<th>Internal Value</th>
<th>External Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>F8.5</td>
<td>2.3547188</td>
<td>2.35472</td>
</tr>
<tr>
<td>F9.3</td>
<td>8789.7361</td>
<td>8789.736</td>
</tr>
<tr>
<td>F10.4</td>
<td>-23.24352</td>
<td>-23.2435</td>
</tr>
<tr>
<td>F5.2</td>
<td>325.013</td>
<td>****</td>
</tr>
<tr>
<td>F5.2</td>
<td>-.2</td>
<td>-0.20</td>
</tr>
</tbody>
</table>
7.11.1.3 E Field Descriptor - The E field descriptor specifies the transmission of real values in exponential format.

Format:

Ew.d

Input

The E field descriptor causes an input statement to input w characters from an external record. It interprets and assigns that data in exactly the same way as the F field descriptor.

Examples:

<table>
<thead>
<tr>
<th>Format</th>
<th>External Field</th>
<th>Internal Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>E9.3</td>
<td>734.432E3</td>
<td>734432.0</td>
</tr>
<tr>
<td>E12.4</td>
<td>1022.43E-6</td>
<td>1022.43E-6</td>
</tr>
<tr>
<td>E15.3</td>
<td>52.37596</td>
<td>52.37596</td>
</tr>
</tbody>
</table>

Output

The E field descriptor causes an output statement to transmit the value of the corresponding list element to an external field w characters in width, right justified. If the number of characters in the converted data is less than w, FORTRAN inserts leading spaces; if the number of characters exceeds w, FORTRAN fills the entire field with asterisks. The corresponding I/O list element must be of real type.

FORTRAN transmits data output under control of the E field descriptor in a standard form, consisting of

1. a minus sign if the value is negative (plus signs are suppressed),
2. a zero,
3. a decimal point,
4. d digits to the right of the decimal, and
5. a 3-character exponent of the form:
   \[ e+n\text{n} \]
   or
   \[ e-n\text{n} \]

where:

\[ \text{n} \] is a 2-digit integer constant.

The d digits to the right of the decimal point represent the entire value, scaled to a decimal fraction.

Because w must be large enough to include a minus sign (if any are expected), a zero, a decimal point, and an exponent, in addition to d digits, w should always be equal to or greater than d+7.
Examples:

<table>
<thead>
<tr>
<th>Format</th>
<th>Internal Value</th>
<th>External Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>E9.2</td>
<td>475867.222</td>
<td>0.48E+06</td>
</tr>
<tr>
<td>E12.5</td>
<td>475867.222</td>
<td>0.47587E+06</td>
</tr>
<tr>
<td>E12.3</td>
<td>0.00069</td>
<td>0.690E-03</td>
</tr>
<tr>
<td>E10.3</td>
<td>-0.5555</td>
<td>-0.556E+00</td>
</tr>
<tr>
<td>E5.3</td>
<td>56.12</td>
<td>****</td>
</tr>
</tbody>
</table>

7.11.1.4 G Field Descriptor - The G field descriptor transmits real data in a form that is in effect a combination of the F and E field descriptors.

Format:

Gw.d

Input

On input, the G field descriptor functions identically to the F field descriptor.

Output

On output, the G field descriptor causes FORTRAN to transmit the value of the corresponding I/O list element to an external field w characters in length, right justified. The form in which the value is output is a function of the magnitude of the value, as described in Table 7-19.

Table 7-19
Effect of Data Magnitude on G Format Conversions

<table>
<thead>
<tr>
<th>Data Magnitude</th>
<th>Effective Conversion</th>
</tr>
</thead>
<tbody>
<tr>
<td>m &lt; 0.1</td>
<td>Ew.d</td>
</tr>
<tr>
<td>0.1 &lt; m &lt; 1.0</td>
<td>F(w-4).d, 4X</td>
</tr>
<tr>
<td>1.0 &lt; m &lt; 10.0</td>
<td>F(w-4).(d-1), 4X</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>10d-2 &lt; m &lt; 10d-1</td>
<td>F(w-4).1, 4X</td>
</tr>
<tr>
<td>10d-1 &lt; m &lt; 10d</td>
<td>F(w-4).0, 4X</td>
</tr>
<tr>
<td>m &gt; 10d</td>
<td>Ew.d</td>
</tr>
</tbody>
</table>

The 4X field descriptor is inserted by the G field descriptor for values within its range, and means that four spaces are to follow the numeric data representation.
The field width, \( w \), must include

1. space for a minus sign, if any are expected (plus signs are suppressed),
2. at least one digit to the left of the decimal point,
3. the decimal point itself,
4. \( d \) digits to the right of the decimal, and
5. (for values that are outside the effective range of the \( G \) field descriptor) a 4-character exponent.

Therefore, \( w \) should always be equal to or greater than \( d+7 \).

Examples:

<table>
<thead>
<tr>
<th>Format</th>
<th>Internal Value</th>
<th>External Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>G13.6</td>
<td>0.01234567</td>
<td>0.123457E-01</td>
</tr>
<tr>
<td>G13.6</td>
<td>-0.12345678</td>
<td>-0.123457</td>
</tr>
<tr>
<td>G13.6</td>
<td>1.23456789</td>
<td>1.23457</td>
</tr>
<tr>
<td>G13.6</td>
<td>12.34567890</td>
<td>12.3457</td>
</tr>
<tr>
<td>G13.6</td>
<td>123.45678901</td>
<td>123.457</td>
</tr>
<tr>
<td>G13.6</td>
<td>-1234.56789012</td>
<td>-1234.57</td>
</tr>
<tr>
<td>G13.6</td>
<td>12345.67890123</td>
<td>12345.7</td>
</tr>
<tr>
<td>G13.6</td>
<td>123456.78901234</td>
<td>123457.</td>
</tr>
<tr>
<td>G13.6</td>
<td>-1234567.89012345</td>
<td>-0.123457E+07</td>
</tr>
</tbody>
</table>

For comparison, consider the following example of the same values output under the control of an equivalent \( F \) field descriptor.

<table>
<thead>
<tr>
<th>Format</th>
<th>Internal Value</th>
<th>External Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>F13.6</td>
<td>0.01234567</td>
<td>0.012346</td>
</tr>
<tr>
<td>F13.6</td>
<td>-0.12345678</td>
<td>-0.123457</td>
</tr>
<tr>
<td>F13.6</td>
<td>1.23456789</td>
<td>1.234568</td>
</tr>
<tr>
<td>F13.6</td>
<td>12.34567890</td>
<td>12.345679</td>
</tr>
<tr>
<td>F13.6</td>
<td>123.45678901</td>
<td>123.456789</td>
</tr>
<tr>
<td>F13.6</td>
<td>-1234.56789012</td>
<td>-1234.567890</td>
</tr>
<tr>
<td>F13.6</td>
<td>12345.67890123</td>
<td>12345.678901</td>
</tr>
<tr>
<td>F13.6</td>
<td>123456.78901234</td>
<td>123456.789012</td>
</tr>
<tr>
<td>F13.6</td>
<td>-1234567.89012345</td>
<td>**************</td>
</tr>
</tbody>
</table>

**NOTE**

Only the first 6 digits in external representation are accurate.

7.11.1.5 \( L \) Field Descriptor - The \( L \) field descriptor specifies the transmission of logical data.

Format:

\[ Lw \]
FORTRAN IV

Input

The L field descriptor causes an input statement to read w characters from an external record. If the first non-blank character of that field is the letter T or the string .T, FORTRAN assigns the value .TRUE. to the corresponding I/O list element. (The corresponding I/O list element must be of logical type.) If the first non-blank character of the field is the letter F or the string .F, or if the entire field is blank, FORTRAN assigns the value .FALSE.. Any other value in the external field causes an error condition.

Output

The L field descriptor causes an output statement to transmit either the letter T, if the value of the corresponding list element is .TRUE. or the letter F, if the value is .FALSE., to an external field w characters wide. The letter T or F is in the rightmost position of the field, preceded by w-1 spaces.

Examples:

<table>
<thead>
<tr>
<th>Format</th>
<th>Internal Value</th>
<th>External Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>L5</td>
<td>.TRUE.</td>
<td>T</td>
</tr>
<tr>
<td>L1</td>
<td>.FALSE.</td>
<td>F</td>
</tr>
</tbody>
</table>

7.11.1.8 A Field Descriptor - The A field descriptor specifies the transmission of alphanumeric data.

Format:

Aw

Input

On input, the A field descriptor causes w characters to be read from the external record and stored in ASCII format in the corresponding I/O list element. (The corresponding I/O list element may be of any data type.) The maximum number of characters that FORTRAN can store in a variable or array element depends on the data type of that element, as listed in Table 7-20.

<table>
<thead>
<tr>
<th>I/O List Element</th>
<th>Maximum Number of Characters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logical</td>
<td>6</td>
</tr>
<tr>
<td>Integer</td>
<td>6</td>
</tr>
<tr>
<td>Real</td>
<td>6</td>
</tr>
</tbody>
</table>

If w is greater than the maximum number of characters that FORTRAN can store in the corresponding I/O list element, only the rightmost six characters are assigned to that entity; the leftmost excess characters are lost. If w is less than the number of characters that FORTRAN can store, it assigns w characters to the list element, left justified, and adds trailing spaces to fill the variable or array element.
FORTRAN IV

Examples:

<table>
<thead>
<tr>
<th>Format</th>
<th>External Field</th>
<th>Internal Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>A6</td>
<td>PAGE #</td>
<td>PAGE # (Integer)</td>
</tr>
<tr>
<td>A6</td>
<td>PAGE #</td>
<td>GE # (Real)</td>
</tr>
</tbody>
</table>

Output

On output, the A field descriptor causes FORTRAN to transmit the contents of the corresponding I/O list element to an external field w characters wide. If the list element contains fewer than w characters, the data appears in the field right-justified with leading spaces. If the list element contains more than w characters, FORTRAN transmits only the leftmost w characters.

Examples:

<table>
<thead>
<tr>
<th>Format</th>
<th>Internal Value</th>
<th>External Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>A5</td>
<td>OHMS</td>
<td>OHMS</td>
</tr>
<tr>
<td>A5</td>
<td>VOLTS</td>
<td>VOLTS</td>
</tr>
<tr>
<td>A5</td>
<td>AMPERES</td>
<td>AMPER</td>
</tr>
</tbody>
</table>

7.11.1.7 H Field Descriptor

Format:

nHccc...c

where:

n specifies the number of characters that are to be transmitted, and

c is an ASCII character.

Input

When the H field descriptor appears in a format specification, data transmission takes place between the external record and the field descriptor itself.

The H field descriptor causes an input statement to read n characters from the external record and to place them in the field descriptor, with the first character appearing immediately after the letter H. FORTRAN replaces any characters that had been in the field descriptor prior to input by the input characters.

Output

The H field descriptor causes an output statement to transmit the n characters in the field descriptor following the letter H to the external record. An example of the use of H field descriptors for input and output follows:

```
WRITE (4,100)
100 FORMAT (4H ENTER PROGRAM TITLE, UP TO 20 CHARACTERS)
READ (4,200)
200 FORMAT (20H TITLE GOES HERE )
```
The WRITE statement transmits the characters from the H field descriptor in statement 100 to the user's terminal. The READ statement accepts the response from the keyboard, placing the input data in the H field descriptor in statement 200. The new characters replace the string TITLE GOES HERE; if the user enters fewer than 20 characters, FORTRAN fills the remainder of the H field descriptor with spaces to the right.

7.11.1.8 Alphanumeric Literals − You may use an alphanumeric literal in place of an H field descriptor. For output, both types of format specifiers function identically. However, you cannot use an alphanumeric literal on input.

You write an apostrophe character within an alphanumeric literal as two apostrophes. For example:

50 FORMAT ('TODAY'S DATE IS: ',I2, '/ ',I2, '/ ',I2)

FORTRAN treats a pair of apostrophes used in this manner to be a single character.

7.11.1.9 X Field Descriptor − The X field descriptor causes spaces to be skipped in a record.

Format:

nX

Input

The X field descriptor causes an input statement to skip over the next n characters in the input record.

Output

The X field descriptor causes an output statement to transmit n spaces to the external record. For example:

WRITE (5,90) NPAGE
90 FORMAT (13H1PAGE NUMBER ,I2,16X,23HGRAPHIC ANALYSIS, CONT.)

The WRITE statement prints a record similar to:

PAGE NUMBER nn GRAPHIC ANALYSIS, CONT.

where "nn" is the current value of the variable NPAGE. FORTRAN does not print the numeral 1 in the first H field descriptor, instead using it to advance the printer paper to the top of a new page. Printer carriage control is explained in Section 10.5.

7.11.1.10 T Field Descriptor − The T field descriptor is a tabulation specifier.

Format:

Tn
where:

\[ n \]
indicates the character position of the external record. The value of \( n \) must be greater than or equal to one, but not greater than the number of characters allowed in the external record.

Input

On input, the \( T \) field descriptor causes FORTRAN to position the external record to its \( n \)th character position. For example, if a READ statement inputs a record containing:

```
ABC XYZ
```

under control of the FORMAT statement:

```
10  FORMAT (T7,A3,T1,A3)
```

the READ statement would input the characters XYZ first, then the characters ABC.

Output

On output to devices other than the line printer or terminal, the \( T \) field descriptor states that subsequent data transfer is to begin at the \( n \)th character position of the external record. For output to a printing device, data transfer begins at position \( n-1 \). This is because FORTRAN reserves the first position of a printed record for a carriage control character (see Section 10.5) which is never printed.

Example the statements:

```
WRITE(4,25)
25  FORMAT (T51,'COLUMN 2',T21,'COLUMN 1')
```

would cause the following line to be printed:

```
Position 20  Position 50
COLUM 1  COLUM 2
```

7.11.1.11 \$ Descriptor - The character \$ (dollar sign) appearing in a format specification modifies the carriage control specified by the first character of the record. The \$ descriptor is intended primarily for interactive I/O and causes the terminal print position to be left at the end of the text written (rather than returned to the left margin) so that a typed response will appear on the same line following the output.

Example:

```
A=5
WRITE (4,100) A
READ (4,200) B
100  FORMAT (' SAMPLE NO.', I2, ' IS: ',$)
200  FORMAT (A6)
WRITE (4,200) B
END
```

This program outputs

SAMPLE NO. 5 IS: RED RED
7.11.2 Scale Factor

You can alter the location of the decimal point in real values during input or output through the use of a scale factor.

Format:

\[ \text{n}P \]

where:

\[ n \]

is a signed or unsigned integer constant in the range -127 to +127 specifying the number of positions the decimal point is to be moved to the right or left.

You may place a scale factor anywhere in a format specification, but it must precede the field descriptors with which it is to be associated. It has the forms:

\[ \text{nPFw.d} \quad \text{nPEw.d} \quad \text{nPGw.d} \]

Data input under control of one of the above field descriptors is multiplied by \(10^{n}\) before FORTRAN assigns it to the corresponding I/O list element. For example, a 2P scale factor multiplies an input value by .01, moving the decimal point two places to the left; a -2P scale factor multiplies an input value by 100, moving the decimal point two places to the right. If the external field contains an explicit exponent, however, the scale factor has no effect.

Examples:

<table>
<thead>
<tr>
<th>Format</th>
<th>External Field</th>
<th>Internal Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>3PE10.5</td>
<td>37.614</td>
<td>.037614</td>
</tr>
<tr>
<td>3PE10.5</td>
<td>37.614E2</td>
<td>3761.4</td>
</tr>
<tr>
<td>-3PE10.5</td>
<td>37.614</td>
<td>37614.</td>
</tr>
</tbody>
</table>

The effect of the scale factor on output depends on the type of field descriptor with which it is associated. For the F field descriptor, FORTRAN multiplies the value of the I/O list element by \(10^{n}\) before it transmits it to the external record. Thus, a positive scale factor moves the decimal point to the right; a negative scale factor moves the decimal point to the left.

FORTRAN adjusts values output under control of an E or D field descriptor with a scale factor by multiplying the basic real constant portion of each value by \(10^{n}\) and subtracting \(n\) from the exponent. Thus a positive scale factor moves the decimal point to the right and decreases the exponent; a negative scale factor moves the decimal point to the left and increases the exponent.

FORTRAN suspends the effect of the scale factor while the magnitude of the data to be output is within the effective range of the G field descriptor, since \(G\) supplies its own scaling function. The \(G\) field descriptor functions as an \(E\) field descriptor when the magnitude of the data value is outside its range; the effect of the scale factor is therefore the same as described for that field descriptor.

Note that on input, and on output under control of an \(F\) field descriptor, a scale factor actually alters the magnitude of the data; on output, a scale factor attached to an \(E\) or \(G\) field descriptor merely alters the form in which the data is transmitted. Note also that on input a positive scale factor moves the decimal point to the left and a negative scale factor moves the decimal point to the right, while on output the effect is just the reverse.
If you do not attach a scale factor to a field descriptor, FORTRAN assumes a scale factor of zero. Once you specify a scale factor, however, it applies to all subsequent real field descriptors in the same format specification, unless another scale factor appears. You may only restate a scale factor of zero by an explicit 0P specification.

Some examples of scale factor effect on output are:

<table>
<thead>
<tr>
<th>Format</th>
<th>Internal Value</th>
<th>External Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1PE12.3</td>
<td>-270.139</td>
<td>-2.701E+02</td>
</tr>
<tr>
<td>1PE12.2</td>
<td>-270.139</td>
<td>-2.70E+02</td>
</tr>
<tr>
<td>-1PE12.2</td>
<td>-270.139</td>
<td>-0.03E+04</td>
</tr>
</tbody>
</table>

7.11.3 Grouping and Group Repeat Specifications

You can apply any field descriptor (except H, T, P, or X) to a number of successive data fields by preceding that field descriptor with an unsigned integer constant, called a repeat count, that specifies the number of repetitions. For example, the statements:

```
20 FORMAT (E12.4,E12.4,E12.4,15,I5,I5,I5)
```

and

```
20 FORMAT (3E12.4,4I5)
```

have the same effect.

Similarly, you may repeatedly apply a group of field descriptors to data fields by enclosing those field descriptors in parentheses, with an unsigned integer constant, called a group repeat count, preceding the left parenthesis. For example:

```
50 FORMAT (2I8,3(F8.3,E15.7))
```

is equivalent to:

```
50 FORMAT (I8,I8,F8.3,E15.7,F8.3,E15.7,F8.3,E15.7)
```

You can enclose an H or X field descriptor, which could not otherwise be repeated, in parentheses. FORTRAN then treats it as a group repeat specification, thus allowing it to be repeated a desired number of times.

If you omit a group repeat count, FORTRAN assumes it to be 1.

7.11.4 Carriage Control

FORTRAN never transmits the first character of a record to a printing device; instead, FORTRAN interprets this first character as a carriage control character. The FORTRAN I/O system recognizes certain characters for this purpose; the effects of these characters are shown in Table 7-21.
FORTRAN IV

Table 7-21
Carriage Control Characters

<table>
<thead>
<tr>
<th>Character</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>space</td>
<td>Advance one line</td>
</tr>
<tr>
<td>0 zero</td>
<td>Advance two lines</td>
</tr>
<tr>
<td>1 one</td>
<td>Advances to top of next page</td>
</tr>
<tr>
<td>+ plus</td>
<td>Do not advance (allows overprinting)</td>
</tr>
</tbody>
</table>

FORTRAN treats any character other than those described in Table 10-2 as though it is a space, and deletes it from the print line.

7.11.5 Format Specification Separators

You generally separate field descriptors in a format specification from one another by commas. You may also use the slash (/) record terminator to separate field descriptors. A slash causes FORTRAN to terminate the input or output of the current record and to initiate a new record.

You may omit the comma when using a slash. Also, you need not type a comma after a Hollerith constant.

Example:

```plaintext
WRITE (5,40) K,L,M,N,O,P
40 FORMAT (3A6/I6,2F8.4)
```

is equivalent to:

```plaintext
WRITE (5,40) K,L,M
40 FORMAT (3A6)
WRITE (5,50) N,O,P
50 FORMAT (I6,2F8.4)
```

It is possible to bypass input records or to output blank records by the use of multiple slashes. If n consecutive slashes appear between two field descriptors, they cause FORTRAN to skip n-1 records on input or n-1 blank records to be output. (The first slash terminates the current record; the second slash terminates the first skipped or blank record, and so on.) If n slashes appear at the beginning or end of a format specification, however, they result in n skipped or blank records, because the initial and terminal parentheses of the format specification are themselves a record initiator and record terminator, respectively. An example of the use of multiple record terminators is:

```plaintext
WRITE (5,99)
99 FORMAT ('1'T51'HEADING LINE'/'T51'SUBHEADING LINE'//)
```
FORTRAN IV

The above statements output the following:

Column 50, top of page

HEADING LINE
(blank line)
SUBHEADING LINE
(blank line)
(blank line)

7.11.6 Short Field Termination

A field descriptor such as fw.d specifies that an input statement is to read w characters from the external record. If the data field in question contains fewer than w characters, the input statement would read some characters from the following field unless the short field were padded with leading zeros or spaces. To avoid the necessity of doing so, you may terminate an input field containing fewer than w characters by a comma. The comma overrides the field descriptor's field width specification. This practice, called short field termination, is particularly useful when entering data from a terminal keyboard. You may also use it in conjunction with I, F, E, D, G, and L field descriptors.

Examples:

    READ (6,100) I,J,A,B
    100 FORMAT (2I6,2F10.2)

If the external record input by the above statements contains:

1,-2,1.0,35

Then the following assignments take place:

I = 1
J = -2
A = 1.0
B = 0.35

Note that the physical end of the record also serves as a field terminator. Note also that the d part of a w.d specification is not affected as illustrated by the assignment to B.

You may only terminate fields of fewer than w characters by a comma. If you follow a field of w characters or greater by a comma, FORTRAN will consider the comma to be part of the following field.

Two successive commas, or a comma following a field of exactly w characters, constitutes a null (zero-length) field. Depending on the field descriptor in question, the resulting value assigned is 0, 0.0, 0D0, or .FALSE.

You cannot use a comma to terminate a field that is to be read under control of an A, H, or alphanumeric literal field descriptor. If FORTRAN encounters the physical end of the record before it has read w characters, however, short field termination is accomplished and FORTRAN assigns the characters that were input successfully. It also appends trailing spaces to fill the corresponding I/O list element or the field descriptor.

7-102
7.11.7 Format Control Interaction with Input/Output Lists

FORTRAN initiates format control with the beginning of execution of a formatted I/O statement. The action of format control depends on information provided jointly by the next element of the I/O list (if one exists) and the next field descriptor of the FORMAT statement. FORTRAN interprets both the I/O list and the format specification from left to right.

If the I/O statement contains an I/O list, at least one field descriptor of a type other than H, X, T, or P must exist in the format specification. An execution error occurs if this condition is not met.

When FORTRAN executes a formatted input statement, it reads one record from the specified unit and initiates format control; thereafter, additional records can be read as indicated by the format specification. Format control demands that a new record be input whenever a slash is encountered in the format specification, or when the last outer right parenthesis of the format specification is reached and additional I/O list elements remain.

Each field descriptor of types I, F, E, G, L, and A corresponds to one element in the I/O list. No list element corresponds to an H, X, P, T, or alphanumeric literal field descriptor. In the case of H and alphanumeric literal field descriptors, data transfer takes place directly between the external record and the format specification.

When format control encounters an I, F, E, G, L, or A field descriptor, it determines if a corresponding element exists in the I/O list. If so, format control transmits data, appropriately converted to or from external format, between the record and the list element, then proceeds to the next field descriptor (unless the current one is to be repeated). If there is no corresponding list element, format control terminates.

When FORTRAN reaches the last outer right parenthesis of the format specification, it determines whether or not there are more I/O list elements to be processed. If not, format control terminates. If additional list elements remain, however, FORTRAN terminates the current record, initiates a new one, and format control reverts to the right-most top-level group repeat specification (the one whose left parenthesis matches the next-to-last right parenthesis of the format specification). If no group repeat specification exists in the format specification, format control returns to the initial left parenthesis of the format specification. Format control then continues from that point.

7.11.8 Summary of Rules for Format Statements

The following is a summary of the rules pertaining to the construction and use of the format statement and its components, and to the construction of the external fields and records with which a format specification communicates.

General

1. You must always label a FORMAT statement.

2. In a field descriptor such as rIw or nX, the terms r, w, and n must be unsigned integer constants greater than zero. You may omit the repeat count and field width specification.
3. In a field descriptor such as Fw.d, the term d must be an
unsigned integer constant. It must be present in F, E, D,
and G field descriptors even if it is zero. The decimal
point must also be present. The field width specification w
must be greater than d. The w and d must either both be
present or both omitted.

4. In a field descriptor such as nHcc...c, exactly n characters
must be present following the H format code. Any ASCII
character may appear in this field descriptor (an
alphanumeric literal field descriptor follows the same rule).

5. In a scale factor of the form nP, n must be a signed or
unsigned integer constant in the range -127 to 127 inclusive.
Use of the scale factor applies to F, E, D, and G field
descriptors only. Once you specify a scale factor, it
applies to all subsequent real or double precision field
descriptors in that format specification until another scale
factor appears. FORTRAN requires an explicit OP
specification to reinstate a scale factor of zero.

6. FORTRAN does not permit a repeat count in H, X, T or
alphanumeric literal descriptors unless you enclose those
field descriptors in parentheses and treats them as a group
repeat specification.

7. If an I/O list is present in the associated I/O statement,
the format specification must contain at least one field
descriptor of a type other than H, X, P, T or alphanumeric
literal.

Input

1. You must precede an external input field with a negative
value by a minus symbol; you may optionally precede a
positive value by a plus sign.

2. An external field whose input conversion is governed by an I
field descriptor must have the form of an integer constant.
It cannot contain a decimal point or an exponent.

3. FORTRAN handles an external field whose input conversion by
an F, E, or G field descriptor must have the form of an
integer constant or a real or double precision constant. It
can contain a decimal point and/or an E or D exponent field.

4. If an external field contains a decimal point, the actual
size of the fractional part of the field, as indicated by
that decimal point, overrides the d specification of the
corresponding real field descriptor.

5. If an external field contains an exponent, it causes the
scale factor (if any) of the corresponding field descriptor
to be inoperative for the conversion of that field.

6. The field width specification must be large enough to
accommodate, in addition to the numeric character string of
the external field, any other characters that can be present
(algebraic sign, decimal point, and/or exponent).

7. A comma is the only character that is acceptable for use as
an external field separator. You use it to terminate input
of fields that are shorter than the number of characters
expected, or to designate null (zero-length) fields.
FORTRAN IV

Output

1. A format specification must not demand the output of more characters than can be contained in the external record (for example, a line printer record cannot contain more than 133 characters including the carriage control character).

2. The field width specification w must be large enough to accommodate all characters that FORTRAN may generate by the output conversion, including an algebraic sign, decimal point, and exponent. (The field width specification in an E field descriptor, for example, should be large enough to contain d+7 characters.)

3. FORTRAN uses the first character of a record output to a line printer or terminal for carriage control; FORTRAN never prints it. The first character of such a record should be a space, 0,1,$, or +. FORTRAN treats any other character as a space and deletes it from the record.

7.12 LIBRARY FUNCTIONS AND SUBROUTINES

Library functions and subroutines are called in the same manner as user written functions and subroutines. This section lists the library components that are available to FORTRAN programs and illustrates calling sequences, where necessary. Arguments must be of the correct number and type, but need not have the same name as those shown in the illustrative examples. Certain library routines are used by the FORTRAN system programs and are not available to a user's FORTRAN program. These routines may be identified by a number sign (#) in the entry point or section name, and are not listed in the following section.

7.12.1 ABS (Single-Precision Absolute Value)

ABS calculates the absolute value of a real variable by leaving the variable unchanged if it is positive (or zero) and negating the variable if it is negative.

7.12.2 ACOS (Single-Precision Arc-Cosine Function)

ACOS calculates and returns the primary arc-cosine (in radians) of a real argument less than or equal to 1.0 according to the relation:

If \( x > 0.0 \), \( \text{ACOS}(x) = \text{ATAN} \frac{\sqrt{1-x^2}}{x} \)

If \( x < 0.0 \), \( \text{ACOS}(x) = -\text{ATAN} \frac{\sqrt{1-x^2}}{x} \)

If \( x = 0.0 \), \( \text{ACOS}(x) = 0/2.0 \)
7.12.3 AINT (Single-Precision Floating-Point to Integer)

AINT is a floating-point truncation function. Given a real argument, it truncates the fractional part of the argument and returns the integral part as an integer. This is accomplished by taking the absolute value of the argument, aligning and normalizing this result, then restoring the original sign. AINT, IFIX, and INT perform identical functions.

7.12.4 ALOG (Single-Precision Natural Logarithm)

ALOG calculates and returns the natural (Naperian) logarithm of a real argument greater than zero. Any negative or zero argument returns an error message and a value of 0.0. The algorithm used is an 8-term Taylor series approximation.

7.12.5 ALOG10 (Single-Precision Common Logarithm)

ALOG10 calculates and returns the common (base 10) logarithm of a real argument greater than zero. Any negative or zero argument returns an error message and a value of 0.0. The calculation is accomplished by calling ALOG to compute the natural logarithm and executing a change of base.

7.12.6 AMAX0 (Single-Precision Maximum Value)

AMAX0 accepts an arbitrary number of integer arguments and returns a real value equal to the largest of the arguments.

7.12.7 AMAX1 (Single-Precision Maximum Value)

AMAX1 accepts an arbitrary number of real arguments and returns a real value equal to the largest of the arguments.

7.12.8 AMIN0 (Single-Precision Minimum Value)

AMIN0 accepts an arbitrary number of integer arguments and returns a real value equal to the smallest of the arguments.

7.12.9 AMIN1 (Single-Precision Minimum Value)

AMIN1 accepts an arbitrary number of real arguments and returns a real value equal to the smallest of the arguments.

7.12.10 AMOD (Single-Precision A Modulo B)

AMOD accepts two real arguments and returns a real value equal to the remainder when the first argument is divided by the second argument. If the second argument is not sufficiently large to prevent overflow, an error message and a value of 0.0 are returned.
7.12.11 ASIN (Single-Precision Arc-Sine)

ASIN calculates and returns the arc-sine (in radians) of a real argument in the range [-1, 1] according to the relation:

\[ \text{ASIN}(X) = \text{ATAN}(X/\sqrt{1-X^2}) \]

If the argument falls outside the range [-1, 1], an error message results.

7.12.12 ATAN (Single-Precision Arc-Tangent)

ATAN calculates and returns the primary arc-tangent (in radians) of a real argument. The argument is first reduced according to the relations:

1. If \( x < 2^{-14} \), \( \text{atan}(x) = x \)
2. If \( x > 2^{-14} \), \( \text{atan}(x) = 1/x \)
3. If \( x > 1.0 \), \( \text{atan}(x) = \pi/2 - \text{atan}(1/x) \)
4. If \( x < 0 \), \( \text{atan}(x) = -\text{atan}(-x) \)

and the arc-tangent is then computed by a power series approximation.

7.12.13 ATAN2 (Single-Precision Arc-Tangent of Two Arguments)

ATAN2 accepts two real arguments, assumed to be an abscissa and an ordinate respectively, and calculates the arc-tangent of the quotient of the first argument divided by the second argument. This is accomplished by calling ATAN to find the principal arc-tangent of the quotient and then adjusting the result, depending upon the quadrant in which a point defined by the arguments falls, according to the relations:

- Argument in first quadrant: \( \text{atan2}(y, x) = \text{atan}(y/x) \)
- Argument in second quadrant: \( \text{atan2}(y, x) = \pi/2 - \text{atan}(y/x) \)
- Argument in third quadrant: \( \text{atan2}(y, x) = -\text{atan}(y/x) \)
- Argument in fourth quadrant: \( \text{atan2}(y, x) = -\pi/2 - \text{atan}(y/x) \)

7.12.14 CGET (Character Get Subroutine)

The calling sequence:

\[ \text{CALL CGET (STRING, N, CHAR)} \]

causes the Nth character to be unpacked from STRING and stored in CHAR as a variable in the range 0, 63, where STRING is a character string in A6 format.
7.12.15 CHKEOF (Check for End-of-File Subroutine)

CHKEOF accepts one real, integer or logical argument. After the next formatted read operation, this argument will be set to non-zero if the logical end-of-file was encountered, or to 0 if the logical end-of-file was not encountered. The following is an example of the use of CHKEOF:

```
.
.
CALL CHKEOF(EOF)
READ (N,101)DATA
IF (EOF.NE.0) GO TO 999
.
.
```

7.12.16 CLOCK (Initialize Clock Subroutine)

The purpose of the CLOCK subroutine is to initialize the KK8-B real-time clock. The calling sequence is:

```
CALL CLOCK (FUNCTN,RATE)
```

Where functn can have a value of 0 or 8, specifying multiple or single A/D channel input, respectively. The value of rate is preset to 100 Hz. Any value of rate other than 0 or 8 causes an error message; any value of rate other than 100 is ignored.

7.12.17 COS (Single-Precision Cosine Function)

COS calculates and returns the cosine of a real argument (in radians) by applying the identity:

\[ \cos(x) = \sin(x + \pi/2) \]

7.12.18 COSH (Single-Precision Hyperbolic Cosine Function)

COSH calculates and returns the hyperbolic cosine of a real argument according to the relations:

- If \(|x| < 88.029\)
  \[ \cosh(x) = 1/2 \left( \exp(x) + \frac{1.0}{\exp(x)} \right) \]
- If \(|x| > 88.028\) and \(|x| - \log_2(2) < 88.028\)
  \[ \cosh(x) = \exp(|x| - \log_2(2)) \]
- If \(|x| - \log_2(2) > 88.028\)
  \[ \cosh(x) = 37773777777_8 \]

The third relation produces an error message.
7.12.19 CPUT (Character Put Subroutine)

The calling sequence:

    CALL CPUT(STRING,N,CHAR)

causes CPUT to insert CHAR as the Nth character in STRING, where
STRING is a character string stored in A6 format, and CHAR is a number
in the range [0, 63] which is interpreted as a character. The
following program illustrates the use of CGET and CPUT.

    DATA STR/'HEY!'/
    WRITE(4,100) STR
100 FORMAT(' HEY! IN ASCII ',A6)
    WRITE(4,101)
101 FORMAT(' HEY! IN DECIMAL')
    DO 10 I=1,4
        CALL CGET(STR,I,ICHAR)
        WRITE(4,102) ICHAR
10  CONTINUE
102 FORMAT(I6)
    DO 20 I=1,6
        J=I+1
        CALL CPUT(STR,I,J)
20  CONTINUE
    WRITE(4,103) STR
103 FORMAT(' NEW STRING ',A6)
    CALL EXIT
    END

.R F4
*CHRC/G$
    HEY! IN ASCII  HEY!
    HEY! IN DECIMAL
    0
    5
    25
    33
    NEW STRING  BDFHJL

7.12.20 DATE (OS/78 Date Subroutine)

DATE accepts three integer arguments, accesses the current OS/78
system date, and returns an integer from 1 to 12 corresponding to the
current month as the first argument, an integer from 1 to 31
corresponding to the current day as the second argument, and an
integer from 1970 to 1977 corresponding to the current year as the
third argument.

7.12.21 DIM (Single-Precision Positive Real Difference)

DIM calculates and returns the positive difference of two real
arguments. That is, if the first argument is larger than the second
argument, DIM returns the difference between the arguments; if the
first argument is less than or equal to the second argument, DIM
returns 0.0.
7.12.22 EXP (Single-Precision Exponential Function)

EXP calculates and returns the exponential function of a real argument. The algorithm uses a numerical method after Kogbetliantz (IBM Journal of Research and Development, April, 1957, pp 110-5). EXP3 (Base Raised to an Exponent) Exp3 accepts two real or integer arguments, that is, a base and an exponent and performs the calculation

\[ a = b^e \]

If the first argument is outside the range \([0, 12]\), the value returned in the second argument is unpredictable. If EXTLVL is called on a PDP-8, the second argument will always be set to zero.

7.12.23 FLOAT (Integer-to-Floating-Point Conversion)

FLOAT accepts an integer argument and returns a real variable equal to the argument.

7.12.24 IABS (Integer Absolute Value Function)

IABS calculates and returns the absolute value of an integer variable by leaving the variable unchanged if it is positive (or zero), and negating the variable if it is negative.

7.12.25 IDIM (Integer Positive Difference Function)

IDIM calculates and returns the positive difference of two integer arguments. That is, if the first argument is larger than the second argument, IDIM returns the difference between the arguments; if the first argument is less than or equal to the second argument, IDIM returns a value of 0.

7.12.26 IFIX (Single-Precision Floating-Point-to-Integer Function)

IFIX is a floating-point truncation function. Given a real argument, it truncates the fractional part of the argument and returns the integral part as an integer. IFIX, AINT and INT perform the same function.

7.12.27 INT (Single-Precision Floating-Point-to-Integer)

INT is a floating-point truncation function that performs the same function as AINT and IFIX.

7.12.28 ISIGN (Integer Transfer of Sign Function)

ISIGN accepts two integer arguments, calculates the absolute value of the first argument, and returns this value if the second argument is positive (or zero), or the negative of this value if the second argument is negative.
7.12.29 MAX0 (Single-Precision Maximum Value)

MAX0 accepts an arbitrary number of integer arguments and returns an integer result equal to the largest of the arguments.

7.12.30 MAX1 (Single-Precision Maximum Value)

MAX1 accepts an arbitrary number of real arguments and returns an integer result equal to the largest of the arguments.

7.12.31 MIN0 (Single-Precision Minimum Value Function)

MIN0 accepts an arbitrary number of integer arguments and returns an integer value equal to the smallest of the arguments.

7.12.32 MIN1 (Single-Precision Minimum Value Function)

MIN1 accepts an arbitrary number of real arguments and returns an integer value equal to the smallest of the arguments.

7.12.33 MOD (Integer A Modulo B Function)

MOD accepts two integer arguments and returns an integer value equal to the remainder when the first argument is divided by the second argument. If the second argument is not sufficiently large to prevent overflow, an error message and a value of 0 are returned.

7.12.34 SIGN (Single-Precision Transfer of Sign)

SIGN accepts two real arguments, calculates the absolute value of the first argument, and returns this value if the second argument is positive (or zero), or the negative of this value if the second argument is negative.

7.12.35 SIN (Single-Precision Sine Function)

SIN calculates and returns the sine of a real argument (in radians). The argument is reduced to the first quadrant, and the sine is then computed from a Taylor series expansion.
7.12.36 **SINH** (Single-Precision Hyperbolic Sign)

SINH calculates and returns the hyperbolic sine of a real argument according to the relations:

- If $0.10 < |x| < 87.929$, $\text{SINH}(x) = \frac{1}{2} \left[ \text{EXP}(x) - \frac{1}{\text{EXP}(x)} \right]
- If |x| < 0.10$, $\text{SINH}(x) = x + x^3/6 + x^5/120$
- If $|x| > 88.028$, $\text{SINH}(x) = \left[ \text{EXP}(|x| - \log_2 2) \right] \cdot [\text{signum}(x)]$

7.12.37 **SRT** (Single-Precision Square Root Function)

SQRT calculates and returns the (positive) square root of a positive real argument. Any negative argument results in an error message.

7.12.38 **TAN** (Single-Precision Tangent Function)

TAN calculates and returns the tangent of a real argument (in radians). This is accomplished by computing the quotient of the sine of the argument divided by the cosine of the argument; thus, if the cosine of the argument is zero, an error message is returned.

7.12.39 **TANH** (Single-Precision Hyperbolic Tangent)

TANH calculates and returns the hyperbolic tangent of a real argument by computing the quotient of the hyperbolic sine of the argument divided by the hyperbolic cosine of the argument.

7.12.40 **TIME** (Read Time of Day)

TIME may be called as a subroutine with one real or integer argument, or as a function with a dummy argument. It returns the elapsed time since the clock was started. This result will be in seconds.

7.13 **FORTRAN LANGUAGE SUMMARY**

<table>
<thead>
<tr>
<th>Statement</th>
<th>Form</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>$a=b$</td>
<td>The value of expression $b$ is assigned to the variable $a$.</td>
</tr>
<tr>
<td>Arithmetic</td>
<td>$t \text{ nam}(al...)=x$</td>
<td>The value of expression $x$ is assigned to $f(al...)$ after parameter substitution.</td>
</tr>
<tr>
<td>Statement Function</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Definition</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Statement</td>
<td>Form</td>
<td>Effect</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------------------------</td>
<td>------------------------------------------------------------------------</td>
</tr>
<tr>
<td>ASSIGN</td>
<td>ASSIGN n TO v</td>
<td>Statement number n is assigned as the value of integer variable v for use in an assigned GOTO statement.</td>
</tr>
<tr>
<td>BACKSPACE</td>
<td>BACKSPACE u</td>
<td>Peripheral device u is backspaced one record.</td>
</tr>
<tr>
<td>BLOCK DATA</td>
<td>BLOCK DATA</td>
<td>Identifies a block data subprogram.</td>
</tr>
<tr>
<td>CALL</td>
<td>CALL prog CALL prog(al...)</td>
<td>Invokes subroutine named prog, supply arguments when required.</td>
</tr>
<tr>
<td>COMMON</td>
<td>COMMON/block1/a,b,.../...</td>
<td>Variables (a,b,...) are assigned to a common block.</td>
</tr>
<tr>
<td>CONTINUE</td>
<td>CONTINUE</td>
<td>No processing, target for transfers.</td>
</tr>
<tr>
<td>DATA</td>
<td>DATA varlist/var/...</td>
<td>Assigns initial or constant values to variables.</td>
</tr>
<tr>
<td>DEFINE FILE</td>
<td>DEFINE FILE a(b,c,U,v)</td>
<td>Describes a mass storage file for direct access I/O.</td>
</tr>
<tr>
<td>DIMENSION</td>
<td>DIMENSION array (v1...,v7)</td>
<td>Storage allocated according to dimensions specified for the array.</td>
</tr>
<tr>
<td>DO</td>
<td>DO st l-e1,e2,e3</td>
<td>Statements following the DO up to statement st are iterated for values of integer variable i, starting at i=e1, incrementing by e3, and terminating when i&gt;e2.</td>
</tr>
<tr>
<td>END</td>
<td>END</td>
<td>Cease program compilation; equivalent to STOP in main program or RETURN in subprogram.</td>
</tr>
<tr>
<td>END FILE</td>
<td>END FILE u</td>
<td>Writes END-OF-FILE character in file u.</td>
</tr>
<tr>
<td>EQUIVALENCE</td>
<td>EQUIVALENCE (v1,v2,...,)</td>
<td>Identifies same storage location for variables within parentheses.</td>
</tr>
</tbody>
</table>
### FORTRAN IV

<table>
<thead>
<tr>
<th>Statement</th>
<th>Form</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXTERNAL</td>
<td>EXTERNAL subprogram</td>
<td>Declares a subprogram for use by other subprograms.</td>
</tr>
<tr>
<td>FORMAT</td>
<td>(spec1,spec2,.../...)</td>
<td>Specifies conversions between internal and external representations of data.</td>
</tr>
<tr>
<td>FUNCTION</td>
<td>name(a1,...)</td>
<td>Indicates an external function definition.</td>
</tr>
<tr>
<td>GO TO</td>
<td>(1) GO TO n</td>
<td>Transfers control to:</td>
</tr>
<tr>
<td></td>
<td>(2) GO TO (n1,...nk),e</td>
<td>(1) statement n</td>
</tr>
<tr>
<td></td>
<td>(3) GO TO v</td>
<td>(2) to statement n1 if e=1, to statement nk if e=k.</td>
</tr>
<tr>
<td></td>
<td>GO TO v,(n1,...nk)</td>
<td>(3) transfers control to state-number assigned to v optionally checking that v is assigned one of the labels n1,...nk.</td>
</tr>
<tr>
<td>IF</td>
<td>IF(arith expr)n1,n2,n3</td>
<td>Transfers control to n1 if expr&lt;0, n2 if = 0, or n3 if &gt; 0.</td>
</tr>
<tr>
<td>IF</td>
<td>IF(logical expr)st</td>
<td>Executes statement if expression has a value .TRUE., otherwise executes the next statement.</td>
</tr>
<tr>
<td>Logical</td>
<td>v=e</td>
<td>Value of expression E is assigned to variable V.</td>
</tr>
<tr>
<td>Assignment</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PAUSE</td>
<td>PAUSE [num]</td>
<td>Program execution interrupted and number printed, if given.</td>
</tr>
<tr>
<td>READ</td>
<td>READ(u,f) list</td>
<td>Reads a record from a peripheral device according to specifications given in the argument of the statement.</td>
</tr>
<tr>
<td></td>
<td>READ(u,f)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>READ(u) list</td>
<td></td>
</tr>
<tr>
<td></td>
<td>READ(a'r) list</td>
<td></td>
</tr>
<tr>
<td>RETURN</td>
<td>RETURN</td>
<td>Returns control from a subprogram to the calling program.</td>
</tr>
<tr>
<td>REWIND</td>
<td>REWIND u</td>
<td>Repositions designated unit to the beginning of the file.</td>
</tr>
<tr>
<td>STOP</td>
<td>STOP</td>
<td>Terminate program execution.</td>
</tr>
</tbody>
</table>
### FORTRAN IV

<table>
<thead>
<tr>
<th>Statement</th>
<th>Form</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUBROUTINE</td>
<td>SUBROUTINE nam[(al...)]</td>
<td>Declares name to be a subroutine subprogram and al,..., if supplied are dummy arguments.</td>
</tr>
<tr>
<td>WRITE</td>
<td>WRITE(u,f) list</td>
<td>Writes a record to a peripheral device according to specifications given in the arguments of the statement.</td>
</tr>
<tr>
<td></td>
<td>WRITE(u,f)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>WRITE(u) list</td>
<td></td>
</tr>
<tr>
<td></td>
<td>WRITE(a'r) list</td>
<td></td>
</tr>
</tbody>
</table>

Operators in each type are shown in order of descending precedence.

<table>
<thead>
<tr>
<th>Type</th>
<th>Operator</th>
<th>Operates Upon</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>** exponentiation</td>
<td>arithmetic or logical constants, variables, and expressions</td>
</tr>
<tr>
<td></td>
<td>* multiplication</td>
<td></td>
</tr>
<tr>
<td></td>
<td>/ division</td>
<td></td>
</tr>
<tr>
<td></td>
<td>+ addition</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- subtraction</td>
<td></td>
</tr>
<tr>
<td>Relational</td>
<td>.GT. greater than</td>
<td>arithmetic or logical constants variables, and expressions (all relational operators have equal priority)</td>
</tr>
<tr>
<td></td>
<td>.GE. greater than or equal to</td>
<td></td>
</tr>
<tr>
<td></td>
<td>.LT. less than</td>
<td></td>
</tr>
<tr>
<td></td>
<td>.LE. less than or equal to</td>
<td></td>
</tr>
<tr>
<td></td>
<td>.EQ. equal to</td>
<td></td>
</tr>
<tr>
<td></td>
<td>.NE. not equal to</td>
<td></td>
</tr>
</tbody>
</table>

**Logical**

- .NOT. .NOT.A is true if and only if A is false
- .AND. A.AND.B is true if and only if A and B are true
- .OR. A.OR.B is true if and only if either A or B is true.
- .EQV. A.EQV.B is true if and only if A and B are both true or A and B are both false.
- .XOR. A.XOR.B is true if and only if A is true and B is false or B is true and A is false

(precedence same as .XOR.)

(precedence same as .EQV.)
CHAPTER 8

BATCH

OS/78 BATCH processing is ideally suited to lengthy or after-hour jobs consisting of fixed sequences of OS/78 system commands not requiring operator intervention. BATCH allows you to create a file that contains the OS/78 commands. Output can be line printer listings and files. These files contain program output or comprehensive summaries (log) of all action taken by the program, OS/78 BATCH, and the operator.

BATCH provides optional spooling of output files. Spooling allows printed output to be diverted to a fast file-structured device, such as a diskette, for later transferral to the slower device. This feature serves to increase perceived throughput on the system. A line printer, although optional, makes the use of BATCH more effective.

With a few exceptions, BATCH executes standard OS/78 commands.

8.1 BATCH PROCESSING UNDER OS/78

OS/78 BATCH maintains an input file and an output log. The BATCH input file consists of a series of BATCH commands. The input file must reside on the system device (disk pack or diskette). Its default extension is .BI. Each command in the BATCH input file generally occupies one line.

The BATCH output file is a line printer listing (log) on which BATCH prints job headers, certain messages that result from conditions within the input file, an image of each line in the input file, and certain types of user output. BATCH supports only the LA78 line printer for the output of the BATCH log. Listings, however, can be output on the LQP78 line printer. If a line printer is not present in the system, the output file is displayed on the terminal.

BATCH accepts program and data files from any input device in the system. User output files may be directed to any output device in the system.

BATCH also permits optional spooling of output files. When spooling is requested, every output request to a non-file-structured device output file is assigned a file name from a list of names maintained by BATCH and directed to a file-structured spool device instead of the user-specified device. Spooling of output files increases BATCH throughput, permitting slow output operations to be postponed until a more favorable time. For example, a batch processing run that generates many output listings may be initialized to reroute all listings to a specified diskette. The listings on the diskette may then be dumped onto the appropriate hard copy device after the run, when more time is available. The spool device may be any OS/78 file-structured device.
BATCH

BATCH is called via the SUBMIT command. The format for a BATCH command string is:

    SUBMIT spool-dev:<dev:input/options

where:

spool-dev: is the device on which to spool output. If not specified, no spooling is performed. Note that spooling applies only to non-file-structured output.

dev:input is the input device and file. The default extension for BATCH input files is .BI. The default input device is DSK.

/options are listed in Table 8-1.

Table 8-1
BATCH Run-Time Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>/E</td>
<td>Treat OS/78 Monitor errors as non-fatal errors. If /E is not specified, monitor errors cause the current BATCH job to be terminated.</td>
</tr>
<tr>
<td>/H</td>
<td>Do not output a BATCH log.</td>
</tr>
<tr>
<td>/Q</td>
<td>Output an abbreviated BATCH log, consisting of $JOB and $MSG lines.</td>
</tr>
<tr>
<td>/T</td>
<td>Output the BATCH log to the terminal. This option need be specified only if a line printer is available. If a line printer is not available, the BATCH log is automatically output to the terminal.</td>
</tr>
<tr>
<td>/U</td>
<td>BATCH will not pause for operator response to $MSG lines. Any attempt to use TTY: as an input device to an unattended BATCH stream will cause the current job to be aborted.</td>
</tr>
</tbody>
</table>

8.2 BATCH COMMANDS

A BATCH command is a character or string of characters that begins with the first character of a line in the BATCH input file. Each BATCH command must be followed by a RETURN. The files may contain form feed characters, but form feed characters are ignored by BATCH on input.

BATCH recognizes four monitor level commands. These commands allow routine housekeeping operations in a multi-job, batch processing environment and provide communication between the BATCH user and the operator. Table 8-2 lists the BATCH commands, which may be considered as an extension of the OS/78 Monitor command set. Note that the first character of the $JOB, $MSG and $END commands is a dollar sign (SHIFT/4).
### Table 8-2
BATCH Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$JOB</td>
<td>Initialize for a new job and display a job header on the output file. The remainder of the $JOB record is included in the job header but ignored by BATCH. It should be used for job identification, to provide correlation between terminal output, line printer output and spool device output.</td>
</tr>
<tr>
<td>$MSG text</td>
<td>Sound the terminal buzzer and print the message specified by the argument text at the terminal. If the /U option was not specified, implying that an operator is present, BATCH will pause until any key is struck at the keyboard. If the /U option was specified, processing continues uninterrupted.</td>
</tr>
<tr>
<td>$END</td>
<td>Terminate batch processing and exit to the monitor. A $END command should be the last line of every BATCH input file.</td>
</tr>
<tr>
<td>/</td>
<td>Copy the line onto the output log, then ignore it. BATCH assumes that every line beginning with a slash is a comment.</td>
</tr>
</tbody>
</table>

Any record that begins with a dollar sign character but is not one of the BATCH commands listed above is copied onto the output file and ignored by BATCH.

A BATCH processing job consists of a $JOB command line and all of the commands that follow it up to the next $JOB or $END command. Normally, all the commands submitted by one user are processed as a single job, and all output from these commands appears under one job header.

After BATCH encounters a $JOB command, it scans the input file until the next OS/78 command is read. Any lines that follow the $JOB command and precede the first command are written onto the log and ignored by BATCH.

The first character of every command is a period (.). An exception to this is the use of an asterisk (*) whose function is to accept a command string that indicates input/output files. It is further described in the example program given in Section 8.3. The rest of the line contains a command, which should appear in standard OS/78 format. However, commands that would be terminated with an ESCape under interactive OS/78 should be terminated with a dollar sign under BATCH. Most OS/78 commands are legal input to BATCH except the MEMORY and SQUISH SYS: commands. ODT will go to the terminal for input instead of the BATCH file. It is not usually meaningful to invoke ODT, EDIT, or other interactive programs under BATCH.

BATCH executes an OS/78 command by stripping off the initial period (.) character and passing the remainder of the line to the monitor. BATCH then passes control to the monitor, which executes the command as though it had been typed at the keyboard. Monitor commands that return control to the monitor level should be followed by a BATCH command or another monitor command. The SUBMIT command can be used to chain from one BATCH stream to another.
BATCH

The general rules and conventions associated with BATCH processing are as follows:

1. The dollar sign ($) is always in the first character position of the BATCH command lines $JOB, $MSG and $END.

2. Each job must have a $JOB and $END command.

3. OS/78 commands can be spelled out entirely or in accordance with the accepted abbreviations. Also, a period (.) must precede every command.

4. Wildcards can be specified only for the OS/78 commands COPY, DELETE, DIRECT, LIST, RENAME and TYPE.

5. Comments may only be included as separate comment lines.

6. Only 80 characters per control statement are allowed.

7. Continuation file specification lines (where necessary) are specified by an asterisk at the beginning of the line:

   +LOAD prog,suba,subb*
   *subc,subd,sube*

8.3 THE BATCH INPUT FILE

The following file is an example of a BATCH input file.

   $JOB
   +DATE 13-MAY-79
   /LIST ANY HELP FILES AND STARTING BLOCKS
   ,DIR *.hl/b
   $MSG INSERT DISKETTE INTO DRIVE 1 - TYPE ANY KEY TO CONTINUE
   /LIST DIRECTORY OF DISKETTE 2 - SPOOL TO FILE BICHA1
   ,DIR TTY:<rxa1:
   /COPY FILE FROM DISKETTE 2 DISKETTE 1
   ,COPY RXAO:<RXA1:POWER,FT
   /COMPILE FORTRAN PROGRAM POWER,FT
   ,COMPILE POWER,FT
   /LOAD FORTRAN PROGRAM AND CHAIN TO RUN TIME SYSTEM
   ,LOAD POWER,RL/6*
   /STORE RESULTS OF FORTRAN PROGRAM IN FILE 'HOLD'
   +HOLD,TM</4*
   /EXECUTE FORTRAN PROGRAM AND DISPLAY RESULTS ON SCREEN
   ,EXE POWER,LD
   /END OF JOB NO.1 - START NEXT JOB
   $JOB
   /ASSEMBLE FILE SAMPLE AND PRODUCE CREF LISTING
   ,PAL SAMPLE/C-LS
   /EXECUTE PROGRAM SAMPLE
   ,EXE SAMPLE,BN
   /END OF EXAMPLE AND BATCH INPUT FILE
   $END

The file was created using the Editor and is named BATSAM.BI. For the example used, both the PAL8 and FORTRAN IV system programs were on diskette RXAO.
BATCH is started using the SUBMIT command and typing

.SUBMIT RXAO:<BATSAM

when spooling is desired, or

.SUBMIT BATSAM

when spooling is not desired. The default extension for the BATCH input file is .BI.

BATCH begins processing by printing a job header and executing the DATE command. BATCH next executes the DIRECT command, which displays any Help file names and their starting blocks. Wildcards are used in this command.

BATCH continues to scan the input file for the next line. BATCH processes the $MSG command, sounds the terminal buzzer, and copies the $MSG record onto the terminal. Assuming that an operator is present, processing is suspended until any key is typed at the terminal.

The operator performs the task specified by the $MSG command (that of placing diskette 2 into Drive 1) and types any key, allowing BATCH to continue reading the input file.

BATCH then executes the DIRECT command, which lists the directory of RXA1 on the line printer. Since spooling is active because RXA0 was specified as the spooling device in the SUBMIT command line and a non-file-structured device is specified as output in the DIRECT command, BATCH intercepts this output and stores it in a temporary file on the spool device. The output is stored in a file named BTCHAL. BATCH then outputs the message

#SPOOL TO FILE BTCHAL

on both the console terminal and the line printer, if available. If another file is routed to the spool device, it will be assigned the file name BTCHA2, and any successive files will be named in the following sequence:

BTCHA3
...
BTCHA9
BTCHB0
...
BTCHZ9

A total of 260 spool files are permitted. If output to a spool file is generated by a program that appends a default extension to output file names, the spool file will be assigned a standard default extension. All of the spool files may then be transferred to the terminal or line printer by using the TYPE or LIST command with the input file specification dev:BTCH??.*.

You may type CTRL/C at any time during a batch processing run. Typing CTRL/C at the program level causes an effective jump to location 07600, which recalls the BATCH program. BATCH program then recognizes the CTRL/C and terminates the BATCH run. Sometimes two CTRL/C's are required to be typed in succession to return to the monitor.
Continuing with our example program, the next command copies the FORTRAN program POWER.FT from Drive 1 to Drive 0. The program then is compiled creating a module POWER.RO that is used by the LOAD command to generate a loader image file. The LOAD command is given the /G option, which chains to the run-time system for program execution.

Note that the ESCape function, specified via a dollar sign ($), calls a system program called the Command Decoder. This program (which normally displays an asterisk when it is running) allows the run-time system to accept file input/output specifications. Similarly, an asterisk is used for specifying input/output files. In this case, a file HOLD.TM is designated to store the output of the executed FORTRAN program. The file specification line is then followed by a dollar sign (again serving as an ESCape key function) to execute the LOAD command. After the execution of this command, the loader image file POWER.LD has been created, and its executable results stored in the HOLD.TM file. The EXECUTE command executes the program POWER.FT, displaying the results on the terminal.

BATCH then encounters the second $JOB command. The first job is terminated and a new header is printed. The second job calls CREF to assemble a source program named SAMPLE and produce a CREF listing. The results of this command is to produce a binary file called SAMPLE.BN and a CREF listing stored in the file SAMPLE.LS. Typing the TYPE or LIST commands will print this file on the terminal or line printer, respectively. The program is then executed. The BATCH job is terminated upon encountering the $END command and control returns to the monitor.

The BATCH $END command always must appear as the last record in the input file to terminate batch processing and cause BATCH to recall the monitor and re-establish interactive processing under OS/78.

8.4 BATCH ERROR MESSAGES

BATCH generates two types of error messages. They are BATCH error messages and system error messages. BATCH level error messages appear in the form:

#BATCH ERR

System error messages are generated by OS/78 system programs. When these occur, BATCH will append a "#" character to the beginning of the message, so that it appears in the form:

#SYSTEM ERROR

Any occurrence of an error normally causes BATCH to terminate the current job and scan the input file for the next $JOB command. If the /E option was specified, BATCH treats errors as non-fatal and continues the BATCH run.

Table 8-3 lists the BATCH error messages, their meanings, and the probable cause for the error.
<table>
<thead>
<tr>
<th>Message</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>#BAD LINE JOB ABORTED</td>
<td>The BATCH program detected a line in the input file that did not have one of the characters period (.), slash, dollar sign or asterisk as the first character of the record. The record is ignored, and BATCH scans the input file for the next $JOB command.</td>
</tr>
<tr>
<td>#BATCH SQUISHING SYS::</td>
<td>BATCH is running and attempting to execute a SQUISH on the system device. This may cause BATCH.SV or the BATCH input file to be moved.</td>
</tr>
<tr>
<td>BATCH.SV NOT FOUND</td>
<td>A copy of BATCH.SV must exist on the system device. Control returns to the monitor.</td>
</tr>
<tr>
<td>DEV NOT IMPLEMENTED</td>
<td>BATCH cannot accept input from the specified input device because its handler is not permanently resident. Only input from SYS: is permitted. Control returns to the Command Decoder (See Appendix D). In most cases, type CTRL/C and then retype the command, using the correct parameters.</td>
</tr>
<tr>
<td>#ILLEGAL INPUT</td>
<td>A file specification designated TTY as an input device when the /U option indicated that an operator is not available. The current job is terminated, and BATCH scans the input file for the next $JOB command.</td>
</tr>
<tr>
<td>ILLEGAL SPOOL DEVICE</td>
<td>The device specified as a spooling output device must be file-structured. Control returns to the Command Decoder (See Appendix D). In most cases, type CTRL/C and then retype the command, using the correct parameters.</td>
</tr>
<tr>
<td>#INPUT FAILURE</td>
<td>Either a hardware problem prevented BATCH from reading the next line of the input file, or BATCH read the last line of the input file without encountering a $END command line. If a hardware problem exists, correct the problem and type any character at the terminal to resume processing.</td>
</tr>
</tbody>
</table>

(continued on next page)
### Table 8-3 (Cont.)
**BATCH Error Messages**

<table>
<thead>
<tr>
<th>Message</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>#MANUAL HELP NEEDED</td>
<td>BATCH is attempting to operate an I/O device, such as TTY, that will require operator intervention. If the /U option was specified to indicate that an operator is not present, this message is suppressed, the current job is terminated, and BATCH scans the input file for the next $JOB command record. If an operator is present, the $MSG command provides notification of the action that should be taken.</td>
</tr>
<tr>
<td>#MONITOR OVERLAYED</td>
<td>The BATCH program was called to accept and transmit a file specification, but found that a user program had overlaid part or all of BATCH. BATCH then executes the next command.</td>
</tr>
<tr>
<td>#SPOOL TO FILE BITCH1</td>
<td>Where the &quot;A&quot; may be any character of the alphabet and the &quot;1&quot; may be any decimal digit. This message indicates that BATCH has intercepted a non-file-structured output file and routed it to the spool device. This is, in general, an error condition. Spool device file names are assigned sequentially, beginning with file BITCH1. Standard default extensions may be assigned by some system programs.</td>
</tr>
<tr>
<td>#SYS ERROR</td>
<td>A hardware problem prevented BATCH from performing an I/O operation. Program execution halts, and the system must be restarted by pressing the START pushbutton.</td>
</tr>
</tbody>
</table>

### 8.5 RESTRICTIONS UNDER OS/78 BATCH

OS/78 BATCH is unprotected from user errors. The BATCH program resides in locations 5000 to 7577 in the highest memory field available. BATCH also uses the following locations in field 0 and the highest memory field available.

- **LOCATION**
- **USED AS:**
  - 07777 Batch processing flag.
  - ff7774-ff7777 Internal pointers (ff = field number).

Both the monitor and the Command Decoder (see Appendix D) check the batch processing flag (bit 1 of 07777) whenever they are entered from the program level. Any user program that modifies location 07777 may cause batch processing to be terminated prematurely before the next line of the BATCH input file is read unless bit 1 is left alone.
When the monitor is entered from the program level (JMP to 07600 or 07605), it checks the batch processing flag and reads a new copy of the BATCH program into memory if batch processing is in progress. The Command Decoder, however, does NOT perform this operation. Thus, the Command Decoder must not be called unless the BATCH program is already in memory.

Therefore, large user programs may be loaded over the BATCH program as long as they do not modify the last four locations in field 3. However, once a user memory load has overwritten the BATCH program, execution must remain at the program level until the monitor has been re-entered and a new copy of the BATCH program is read into memory. The Command Decoder must not be called after a user program has been loaded over the BATCH program.

In general, this restriction applies only to loader programs and only when the loader calls the Command Decoder more than once while building a large memory load. Multiple calls to the Command Decoder may be avoided when loading large programs during batch processing if the memory load is first built in the monitor environment and then saved with the SAVE command for subsequent execution under BATCH.

However, the memory image of any program that overlays the BATCH program cannot be saved with the SAVE command. After the load operation but before the save is executed, BATCH will be read back into memory, destroying part of the user program. Thus, the monitor SAVE operation will cause part of the BATCH program to be saved instead of that part of the user program which originally overlaid the BATCH program.

A BATCH job must never move or delete either BATCH.SV or the BATCH input file. Also, SYS: should never be SQUISHED while BATCH is running since this may cause these files to be moved to another region on the system device.
CHAPTER 9
OCTAL DEBUGGING TECHNIQUE (ODT)

ODT is an interactive system program that makes debugging PAL8 programs easy by providing selective execution and memory examination, modification, and searching facilities.

9.1 ODT FEATURES

ODT features include examination and modification of memory locations, and the use of instruction breakpoints to return control to ODT. ODT makes no use of the program interrupt facility and, with few restrictions, is invisible to a user program.

The breakpoint is one of ODT's most useful features. To accomplish this, ODT acts as a monitor to the user program and permits you to insert ODT-controlled halts (breakpoints) in your program. You decide how far the program is to run and instruct ODT to insert a breakpoint in the program which, when encountered, transfers control to transfer back to ODT. ODT immediately preserves in its internal storage locations the contents of the AC and L at the breakpoint. It then displays the location at which the breakpoint occurred, as well as the contents of the AC at that point. ODT will allow examination and modification of any location of the program (or those ODT locations containing the AC and L). You can move the breakpoint or delete it, and request that ODT continue running the program. This causes ODT to restore the AC and L, execute the "trapped" instruction and continue in the program until the breakpoint is again encountered, or the program is terminated.

9.2 CALLING AND USING ODT

When ODT is being used, a complete assembly listing of the program should be available for the program that is being debugged.

Before calling ODT, place the program that is to be debugged in memory as the "current program". For example, if it is a memory image file, type

```
.GET SYS SAMPLE
.ODT
```

If it is a binary file, type

```
.LOAD SAMPLE
.ODT
```
Little of the memory that is being used is disturbed by the running of ODT, because the sections of the program which ODT may occupy when in memory are preserved on the system device and swapped back into memory as necessary. ODT uses the Job Status Word of the particular program to determine whether or not swapping should occur. If the program does not use locations 0-1777 in field 0, less swapping occurs during use of the breakpoint feature.

If any amount of a program is typed directly into memory (in octal), the Core Control Block of the program may not reflect the true extent of the program. If octal additions are made below location 2000 in field 0, ODT may give erroneous results. This condition is corrected by changing the Job Status Word, which is stored in location 7746 of field 0, and which can be examined and changed using ODT as explained later in this chapter. Location 7745 of field 0 is the 12-bit starting address of the program in memory and location 7744 contains the field designation in the form 62n3, where n is the field designation of the starting address.

When using the breakpoint feature of ODT, keep certain operating characteristics in mind:

1. If a breakpoint is inserted at a location which contains an auto-indexed instruction, the auto-index register is incremented immediately after the breakpoint. Thus, when control returns to the user in ODT, the register will have been increased by one. The breakpoint instruction is executed properly, but the index register, if examined, will be one greater than it should.

2. ODT keeps track of the terminal display I/O flag and restores the terminal flag when it continues from a breakpoint.

3. The breakpoint feature uses and does not restore locations 4, 5, and 6 in the memory field in which the breakpoint is set.

4. The breakpoint feature of ODT uses the table of user-defined device names as scratch storage, destroying any device names that may have been created and creating garbage entries. Therefore, it is advisable not to use user-defined device names in programs being debugged with ODT breakpoints. After a session with ODT in which breakpoints are used, give a DEALIGN command to clear out the user-device name table.

5. Breakpoints must not be set in the monitor, in the device handlers, or between a CIF and the following JMP or JMS instruction.

6. ODT should not be used to debug programs which use interrupts.

If an operation is attempted in non-existent memory, ODT ignores the command and types "?". Thus, attempting to examine locations in field 4 and above, ODT responds with ?.

Typing CTRL/C returns control to the monitor. You can then save your program on any file-structured device with the SAVE command. If you want to do this, do not issue any other commands before the SAVE command.
9.3 ODT Commands

9.3.1 Special Characters

These characters will be illustrated using the SAMPLE.PA program created in Chapter 4.

9.3.1.1 Slash (/) - Open This Location - The location examination character (/) causes the location addressed by the octal number preceding the slash to be opened for modification and its contents are displayed in octal. The open location can then be modified by typing the desired octal number and closing the location by using the RETURN key. Any octal number from 1 to 6 digits in length is legal input for addresses and from 1 to 4 digits for data. If the number is not octal or if the DELETE key is typed, a question mark (?) is displayed and the number is ignored. If more than the maximum number of digits are entered, ODT accepts only the last 6 entered (for addresses) or the last 4 (for data). Typing / with no preceding argument reopens the most recently opened location. For example,

```
200 /7300 RET
000201/6046 RET
000201/6046 6048?
000201/6046 2345 RET
/2345
```

The above example modifies locations in field 0. If locations are examined in other fields, specify the location as ffnnnn/ where ff is the field (0-37) and nnnn is the location in octal.

9.3.1.2 RETURN - Close Location - If you have typed a valid octal number after the content of a location is displayed by ODT, pressing the RETURN key causes that number to replace the original contents of the opened location and the location to be closed. If you type nothing, the location is closed but the contents of the location are not changed. For example,

```
201/6046 RET
000201/6046 2345 RET
/2345 6046
```

location 201 is unchanged.
location 201 is changed to contain 2345.
place 6046 back in location 201.

Typing another command will also close an opened register.

9.3.1.3 LINE FEED - Close Location, Open Next Location - The LINE FEED key has the same effect as the RETURN key, but, in addition, the next sequential location is opened and its contents displayed. For example,

```
200 /7300 LF
000201/2346 6046 LF
000202/1216
```

location 200 is closed unchanged and 201 is opened. Type change.
201 is closed (containing 6046) and 202 is opened.
9.3.1.4 Semicolon (;) - Change Location, Close It and Open Next - The semicolon inserts the octal value (if any) that follows it into the currently opened location, closes that location and opens the next sequential location for modification. For example,

202/1216 1234:5670
000202/1234
000203/5670

A series of octal values can be deposited sequentially using the semicolon character. Typing multiple semicolons skips a memory location for each semicolon typed and will accept an octal value at the location skipped to. For example,

202/1234 1216;+0000
000202/1216
000203/5670
000204/6041
000205/0000

9.3.1.5 nnnn+ and nnnn- - Open Location Relative to Another - The nnnn+ and nnnn- commands open the current location plus or minus an offset value for modification and prints the content of that location. If n is omitted, it is assumed to be 1. For example,

200/7300 3+
000203/5670 3217

or

207/6046 2-
000205/0000 5204

9.3.1.6 Circumflex (^) - Close Location, Take Contents as Memory Reference Instruction and Open Effective Address - The circumflex will close an open location just as will the RETURN key. Further, it will interpret the contents of the location as a memory reference instruction, open the location referenced and display its contents. For example,

202/1216 ^

1216 symbolically is "TAD, this page, relative location 16," so ODT opens location 216.

000216 /0220

The indirect bit is used in determining the effective address.

9.3.1.7 Underline (_) - Close Location, Open Indirectly - The back arrow will close the currently open location and then interpret its contents as the address of the location (in the same field) whose contents it is to print and open for modification. For example,

202/1216 _
000216 /0220
000220 /0215

9-4
9.3.2 Illegal Characters

Any character that is neither a valid control character nor an octal digit causes the current line to be ignored and a question mark printed. For example,

```
2:?
2U?
206/1617 67K?
```

ODT opens no location.

ODT ignores a partial number and closes location 206.

/1617

9.3.3 Control Commands

9.3.3.1 ffnnnnG - Transfer Control to Program at Location nnnn of Field ff - Clear the AC and L then go to the location ffnnnn. The breakpoint, if any, will be inserted. Typing G alone will cause a start at location 000000.

9.3.3.2 ffnnnnB - Set Breakpoint at Location nnnn of Field ff - Instructs ODT to establish a breakpoint at the location ffnnnn. A breakpoint may be changed to another location whenever ODT is in control, by simply typing ffnnnnB where ffnnnn is the new location. Only one breakpoint may be in effect at a time; therefore, requesting a new breakpoint removes any previously existing one. A breakpoint may be established at location 000000.

The breakpoint (B) command does not make the actual exchange of ODT instruction for user instruction, it only sets up the mechanism for doing so. The actual exchange occurs when a G or a C command is executed.

When, during execution, the program encounters the location containing the breakpoint, control passes immediately to ODT (via locations 0004-0006 of the instruction field). The contents of the accumulator and contents of the link at the point of the interruption are saved in special locations accessible to ODT. The user instruction that the breakpoint was replacing is restored, before the address of the trap and the content of the AC are displayed. The restored instruction has not been executed at this time, and will not be executed until the "continue from breakpoint" command is given. Any location used, including those containing the stored accumulator and Link, can now be modified. The breakpoint can also be moved or removed at this time.

9.3.3.3 B - Remove Breakpoint - Typing B alone removes any previously established breakpoint and restores the actual contents of the breakpoint location. B should be typed before saving the current program if a breakpoint was set. Typing B where no breakpoint is set has no effect.

9-5
NOTE

If a breakpoint set by ODT is not encountered while ODT is running the object (user's) program, the instruction which causes the break to occur will not be removed from the user's program.

9.3.3.4 A - Open (AC) Location - When the breakpoint is encountered the contents of the accumulator and the contents of the link are saved for later restoration. Typing A after having encountered a breakpoint opens for modification the location in which the AC was saved and prints its contents. This location may now be modified in the normal manner (see Slash) and the modification will be restored to the accumulator when the C or G command is given (contains either 0000 or 0001).

9.3.3.5 L - Open C(L) Location - Typing L opens the link storage location for modification and prints its contents. The link location may now be modified as usual (see Slash) and that modification will be restored to the Link when the C or G command is given.

9.3.3.6 C - Continue From a Breakpoint - Typing C, after having encountered a breakpoint, causes ODT to restore the contents of the accumulator, link, and breakpoint location, and transfer control to the breakpoint location. The user program then runs until the breakpoint is again encountered or the program halts or exits to the monitor.

9.3.3.7 nnnnC - Continue nnnn+1 Times from Breakpoint - A breakpoint may be established at some location within a loop of a program. Since loops often run to many iterations, some means must be available to prevent a break from occurring each time the break location is encountered. This is the function of nnnnC (where nnnn is an octal number). The "continue" operation is done nnnn+1 times. If nnnn is omitted, 0 is assumed. Thus, 2C will allow a loop in which the breakpoint is again encountered to be executed three times.

Given the following program ADD1, which increases the value of the accumulator by increments of 1, the use of the Breakpoint and Continue commands may be illustrated.

```assembly
*200
000200  0200  *200  CLA CLL
000201  7300  A,  TAD ONE
000202  1206  B,  ISZ CNT
000203  2207  JMP B
000204  5202  JMP A
000205  5201  HLT
000206  7402  ONE,  1
000207  0000  CNT,  0
```

9-6
Assemble the program and call ODT by typing the following:

```
,FA1 ADD1/L

.ODT
000201B
000200G
000201 0:0000
C
000201 0:0001
C
000201 0:0002
4C
000201 0:0006
```

ODT has been loaded and started. A breakpoint is inserted at location 0201 and execution stops here showing the accumulator initially set to 0000. The use of the Continue command (C) executes the program until the breakpoint is again encountered (after one complete loop) and shows the accumulator to contain a value of 0001. Again execution continues, incrementing the AC to 0002. At this point, the command 4C is used, allowing execution of the loop to continue five more times before stopping at the breakpoint. The contents of the accumulator have now been incremented to 0006.

9.3.3.8 D - Open Data Field - Typing D opens for modification the internal ODT location containing the data field which was in effect at the last breakpoint. Contents of D always appears as a value between 0 and 37.

9.3.3.9 F - Open Current Field - Typing F opens for modification the internal ODT location containing the field used by ODT in the W (search) command, in the and ^ (indirect addressing) commands, or in the last breakpoint (depending upon which was used most recently). A value between 0 and 37.

9.3.3.10 M - Open Search Mask - Typing M causes ODT to open for modification the internal ODT location containing the current value of the search mask and print its contents. Initially the mask is set to 7777. It may be changed by opening the mask location and typing the desired value after the value printed by ODT, then closing the location.

9.3.3.11 M (L) - Open Lower Search Limit - The word immediately following the mask storage location contains the location at which the search is to begin. Pressing the LINE FEED key to close the mask location causes the lower search limit to be opened for modification and its contents displayed. Initially the lower search limit is set to 0000. It may be changed by typing the desired lower limit after that printed by ODT, then closing the location.
9.3.3.12  **M (LF) (LF)**  - Open Upper Search Limit - The next sequential word after the lower search limit contains the location with which the search is to terminate. Pressing the LINE FEED key to close the lower search limit causes the upper search limit to be opened for modification and its contents printed. Initially, the upper search limit is 7577. It may be changed by typing the desired upper search limit after the one printed by ODT, then closing the location with the RETURN key.

9.3.3.13  **nnnnW**  - Word Search - The command nnnnW (where nnnn is an octal number) will cause ODT to conduct a search of a defined section of memory, using the mask and the lower and upper limits that the user has specified, as indicated above. The word searching operation determines if a given quantity is present in any of the locations of that defined section of memory. A search never alters the contents of any location.

The search is conducted as follows. ODT masks each location within the limits specified and compares the result to the quantity for which it is searching. If the two quantities are identical, the address and the actual unmasked contents of the matching location are displayed, and the search continues until the upper limit is reached. The search includes the upper limit.

For example, locations 3000 through 3777 are to be searched for all ISZ instructions, regardless of what location they refer to (that is, search for all locations beginning with an octal 2, which is the operation code for an ISZ instruction).

```
M/7777 7000 (LF)  Change the mask to 7000; open lower search limit.
000041/5273 3000 (LF)  Change the lower limit to 3000; open upper limit.
000042/1335 3777 (RET)  Change the upper limit to 3777; close location.
2000W 2000W  Initiate the search for ISZ instructions.
0000005 /2331  There are four ISZ instructions in this section of memory. Note that these might also
0000006 /2324  be data values that happen to start with an octal 2.
0000011 /2222
0000033 /2575
```

9.4  **ERRORS**

The only legal inputs are command characters and octal digits. Any other character will cause the character or line to be ignored and a question mark to be printed by ODT. When G is used, it must be preceded by an address to which control will be transferred. If G is not preceded by an address, a question mark will not be displayed, but control will be transferred to location 0.

9.5  **PROGRAMMING NOTES**

ODT will not turn on the program interrupt. It does, however, turn off the interrupt when a breakpoint is encountered, to prevent spurious interrupts.
OCTAL DEBUGGING TECHNIQUE (ODT)

Breakpoints are fully invisible to "open location" commands; however, breakpoints must not be placed in locations which the user program will modify in the course of execution or the breakpoint may be destroyed. Caution should be used in placing a breakpoint between a call to USR function code 10 (USRIN) and the following call to USR function code 11 (USROUT).

If a trap set by ODT is not encountered by your program, the breakpoint instruction will not be removed. If the SAVE command is to be used, the B command must be used to remove the breakpoint before typing CTRL/C. ODT should not be run under BATCH.

9.6 ODT COMMAND SUMMARY

Table 9-1 presents a brief summary of the ODT commands.

<table>
<thead>
<tr>
<th>Command</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>fffnnn/</td>
<td>Opens location designated by the octal number fffnnn, where ff represents the memory field (0-37). ODT displays the contents of the location, a space, and waits for the user to enter a new value for that location or close the location. If you omit ff, field 0 is assumed.</td>
</tr>
<tr>
<td>/</td>
<td>Reopens the latest opened location.</td>
</tr>
<tr>
<td>nnnn; or nnnn</td>
<td>Deposits nnnn in the currently opened location, close that location and open the next location in sequence for modification. The semicolon (;) lets you deposit a series of octal values in sequential locations. To skip locations in the sequence, type a semicolon for each location that you want to skip.</td>
</tr>
<tr>
<td>RETURN key</td>
<td>Closes the currently open location, if any.</td>
</tr>
<tr>
<td>LINE FEED key</td>
<td>Closes the currently open location, opens the next sequential location for modification, and displays the contents of that location.</td>
</tr>
<tr>
<td>nnnn+</td>
<td>Opens the current location plus n and displays the contents of that location.</td>
</tr>
<tr>
<td>nnnn-</td>
<td>Opens the current location minus n and displays its contents.</td>
</tr>
</tbody>
</table>

(continued on next page)
## OCTAL DEBUGGING TECHNIQUE (ODT)

Table 9-1 (Cont.)
ODT Command Summary

<table>
<thead>
<tr>
<th>Command</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>^ (Circumflex)</td>
<td>Closes the current location, reads its contents as a memory-reference instruction and opens the location it points to, displaying its contents.</td>
</tr>
<tr>
<td></td>
<td>• ODT makes no distinction between instruction op-codes when you use this command. It treats all op-codes as memory-reference instructions.</td>
</tr>
<tr>
<td></td>
<td>• Take care when you use this command with indirectly referenced auto-index registers. If you use the command in this way, the contents of the auto-index register is incremented by one. Check to see that the register contains the proper value before proceeding.</td>
</tr>
<tr>
<td>_ (Underline)</td>
<td>Closes the current location, takes the contents of the current location as a 12-bit address and opens that address for modification, displaying its contents.</td>
</tr>
<tr>
<td>ffnnnnG</td>
<td>Transfers control of the program to location ffnnnn, where ff represents the memory field.</td>
</tr>
<tr>
<td>ffnnnnB</td>
<td>Establishes a breakpoint at location ffnnnn, where ff represents the memory field. Only one breakpoint is allowed at any given time.</td>
</tr>
<tr>
<td>B</td>
<td>Removes the breakpoint, if any.</td>
</tr>
<tr>
<td>A</td>
<td>Opens for modification the location in which the contents of the accumulator were stored when the breakpoint was encountered.</td>
</tr>
<tr>
<td>L</td>
<td>Opens for modification the location in which the contents of the link were stored when the breakpoint was encountered.</td>
</tr>
<tr>
<td>C</td>
<td>Continues from a breakpoint.</td>
</tr>
<tr>
<td>nnnnC</td>
<td>Continues from a breakpoint nnnn+1 times before interrupting the user's program at the breakpoint location.</td>
</tr>
<tr>
<td>M</td>
<td>Opens the search mask, initially set to 7777, which can be changed by typing a new value.</td>
</tr>
</tbody>
</table>

(continued on next page)
Table 9-1 (Cont.)
ODT Command Summary

<table>
<thead>
<tr>
<th>Command</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>M ( LF )</td>
<td>Opens the lower search limit. Type in the location (four octal digits) where the search will begin.</td>
</tr>
<tr>
<td>M ( LF ) ( LF )</td>
<td>Opens the upper search limit. Type in the location (four octal digits) where the search will terminate.</td>
</tr>
<tr>
<td>nnnnW</td>
<td>Searches the portion of memory as defined by the upper and lower limits for the octal value nnnn. Search can only be done on a single memory field at a time. See the F command.</td>
</tr>
<tr>
<td>D</td>
<td>Opens for modification the word containing the data field (0-37) that was in effect at the last breakpoint. To change the field, enter a field number in the range 0-37.</td>
</tr>
<tr>
<td>F</td>
<td>Opens for modification the word containing the field (0-37) used by ODT in the W (search) command, in the (^) and (^) (indirect addressing) commands, or in the last breakpoint (depending upon which was used most recently). To change the field, enter a field number in the range 0-37.</td>
</tr>
<tr>
<td>CTRL/O</td>
<td>Interrupts a long search output and waits for a new ODT command.</td>
</tr>
<tr>
<td>DELETE key</td>
<td>Cancels previous number typed, up to the last non-numeric character entered. ODT responds with a question mark, after which you enter the correct location value.</td>
</tr>
</tbody>
</table>
APPENDIX A

ASCII CHARACTER SET

The following table shows the standard ASCII character set used by OS/78 software. Control codes marked with an asterisk (*) have no significance to OS/78.

<table>
<thead>
<tr>
<th>7-Bit Decimal</th>
<th>8-Bit Octal</th>
<th>6-Bit Octal</th>
<th>Character/Control Code</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>200</td>
<td></td>
<td>NUL</td>
<td>Fill character - ignored on input</td>
</tr>
<tr>
<td>001</td>
<td>201</td>
<td></td>
<td>SOH *</td>
<td></td>
</tr>
<tr>
<td>002</td>
<td>202</td>
<td></td>
<td>STX *</td>
<td></td>
</tr>
<tr>
<td>003</td>
<td>203</td>
<td></td>
<td>ETX</td>
<td>CTRL/C - Return control to monitor</td>
</tr>
<tr>
<td>004</td>
<td>204</td>
<td></td>
<td>EOT *</td>
<td></td>
</tr>
<tr>
<td>005</td>
<td>205</td>
<td></td>
<td>ENQ *</td>
<td></td>
</tr>
<tr>
<td>006</td>
<td>206</td>
<td></td>
<td>ACK *</td>
<td></td>
</tr>
<tr>
<td>007</td>
<td>207</td>
<td></td>
<td>BEL</td>
<td>CTRL/G - Sound audible signal</td>
</tr>
<tr>
<td>008</td>
<td>210</td>
<td></td>
<td>BS *</td>
<td></td>
</tr>
<tr>
<td>009</td>
<td>211</td>
<td></td>
<td>HT</td>
<td>CTRL/I - Horizontal tab</td>
</tr>
<tr>
<td>010</td>
<td>212</td>
<td></td>
<td>LF</td>
<td>CTRL/J - LINE FEED (new line)</td>
</tr>
<tr>
<td>011</td>
<td>213</td>
<td></td>
<td>VT</td>
<td>CTRL/K - TAB (vertical)</td>
</tr>
<tr>
<td>012</td>
<td>214</td>
<td></td>
<td>FF</td>
<td>CTRL/L - Form feed</td>
</tr>
<tr>
<td>013</td>
<td>215</td>
<td></td>
<td>CR</td>
<td>Cursor or carriage RETURN (CTRL/M)</td>
</tr>
<tr>
<td>014</td>
<td>216</td>
<td></td>
<td>SO *</td>
<td>CTRL/O - Abort current terminal output</td>
</tr>
<tr>
<td>015</td>
<td>217</td>
<td></td>
<td>SI</td>
<td></td>
</tr>
<tr>
<td>016</td>
<td>220</td>
<td></td>
<td>DLE *</td>
<td></td>
</tr>
<tr>
<td>017</td>
<td>221</td>
<td></td>
<td>DC1</td>
<td>CTRL/Q - Disable terminal output</td>
</tr>
<tr>
<td>018</td>
<td>222</td>
<td></td>
<td>DC2 *</td>
<td></td>
</tr>
<tr>
<td>019</td>
<td>223</td>
<td></td>
<td>DC3</td>
<td>CTRL/S - Enable terminal output</td>
</tr>
<tr>
<td>020</td>
<td>224</td>
<td></td>
<td>DC4 *</td>
<td></td>
</tr>
<tr>
<td>021</td>
<td>225</td>
<td></td>
<td>NAK</td>
<td>CTRL/U - Delete current input line</td>
</tr>
<tr>
<td>022</td>
<td>226</td>
<td></td>
<td>SYN *</td>
<td></td>
</tr>
<tr>
<td>023</td>
<td>227</td>
<td></td>
<td>ETB *</td>
<td></td>
</tr>
<tr>
<td>024</td>
<td>230</td>
<td></td>
<td>CAN *</td>
<td></td>
</tr>
<tr>
<td>025</td>
<td>231</td>
<td></td>
<td>EM *</td>
<td></td>
</tr>
<tr>
<td>026</td>
<td>232</td>
<td></td>
<td>SUB</td>
<td>CTRL/Z - End of file</td>
</tr>
<tr>
<td>027</td>
<td>233</td>
<td></td>
<td>ESC</td>
<td>ALTMODE</td>
</tr>
<tr>
<td>028</td>
<td>234</td>
<td></td>
<td>FS *</td>
<td></td>
</tr>
<tr>
<td>029</td>
<td>235</td>
<td></td>
<td>GS *</td>
<td></td>
</tr>
<tr>
<td>030</td>
<td>236</td>
<td></td>
<td>RS *</td>
<td></td>
</tr>
<tr>
<td>031</td>
<td>237</td>
<td></td>
<td>US *</td>
<td></td>
</tr>
<tr>
<td>032</td>
<td>240</td>
<td>40</td>
<td>SP</td>
<td>Space</td>
</tr>
</tbody>
</table>
### ASCII Character Set

<table>
<thead>
<tr>
<th>7-Bit</th>
<th>8-Bit</th>
<th>6-Bit</th>
<th>Character/Control Code</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decimal</td>
<td>Octal</td>
<td>Octal</td>
<td></td>
<td></td>
</tr>
<tr>
<td>033</td>
<td>241</td>
<td>41</td>
<td>!</td>
<td>Exclamation point</td>
</tr>
<tr>
<td>034</td>
<td>242</td>
<td>42</td>
<td>&quot;</td>
<td>Quotation mark</td>
</tr>
<tr>
<td>035</td>
<td>243</td>
<td>43</td>
<td>#</td>
<td>Dollar sign</td>
</tr>
<tr>
<td>036</td>
<td>244</td>
<td>44</td>
<td>$</td>
<td>Percent sign</td>
</tr>
<tr>
<td>037</td>
<td>245</td>
<td>45</td>
<td>%</td>
<td>Ampersand</td>
</tr>
<tr>
<td>038</td>
<td>246</td>
<td>46</td>
<td>'</td>
<td>Apostrophe</td>
</tr>
<tr>
<td>039</td>
<td>247</td>
<td>47</td>
<td>(</td>
<td>Left parenthesis</td>
</tr>
<tr>
<td>040</td>
<td>250</td>
<td>50</td>
<td>)</td>
<td>Right parenthesis</td>
</tr>
<tr>
<td>041</td>
<td>251</td>
<td>51</td>
<td>*</td>
<td>Asterisk</td>
</tr>
<tr>
<td>042</td>
<td>252</td>
<td>52</td>
<td>+</td>
<td>Plus sign</td>
</tr>
<tr>
<td>043</td>
<td>253</td>
<td>53</td>
<td>,</td>
<td>Comma</td>
</tr>
<tr>
<td>044</td>
<td>254</td>
<td>54</td>
<td>-</td>
<td>Hyphen or minus sign</td>
</tr>
<tr>
<td>045</td>
<td>255</td>
<td>55</td>
<td>.</td>
<td>Period</td>
</tr>
<tr>
<td>046</td>
<td>256</td>
<td>56</td>
<td>/</td>
<td>Slash (right)</td>
</tr>
<tr>
<td>047</td>
<td>257</td>
<td>57</td>
<td></td>
<td></td>
</tr>
<tr>
<td>048</td>
<td>260</td>
<td>60</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>049</td>
<td>261</td>
<td>61</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>050</td>
<td>262</td>
<td>62</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>051</td>
<td>263</td>
<td>63</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>052</td>
<td>264</td>
<td>64</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>053</td>
<td>265</td>
<td>65</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>054</td>
<td>266</td>
<td>66</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>055</td>
<td>267</td>
<td>67</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>056</td>
<td>270</td>
<td>70</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>057</td>
<td>271</td>
<td>71</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>058</td>
<td>272</td>
<td>72</td>
<td>:</td>
<td>Colon</td>
</tr>
<tr>
<td>059</td>
<td>273</td>
<td>73</td>
<td>;</td>
<td>Semicolon</td>
</tr>
<tr>
<td>060</td>
<td>274</td>
<td>74</td>
<td>&lt;</td>
<td>Left angle bracket or less than sign</td>
</tr>
<tr>
<td>061</td>
<td>275</td>
<td>75</td>
<td>=</td>
<td>Equals sign</td>
</tr>
<tr>
<td>062</td>
<td>276</td>
<td>76</td>
<td>&gt;</td>
<td>Right angle bracket or greater than sign</td>
</tr>
<tr>
<td>063</td>
<td>277</td>
<td>77</td>
<td>?</td>
<td>Question mark</td>
</tr>
<tr>
<td>064</td>
<td>300</td>
<td>00</td>
<td>@</td>
<td>At sign (null in 6-bit)</td>
</tr>
<tr>
<td>065</td>
<td>301</td>
<td>01</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>066</td>
<td>302</td>
<td>02</td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>067</td>
<td>303</td>
<td>03</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>068</td>
<td>304</td>
<td>04</td>
<td>D</td>
<td></td>
</tr>
<tr>
<td>069</td>
<td>305</td>
<td>05</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>070</td>
<td>306</td>
<td>06</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>071</td>
<td>307</td>
<td>07</td>
<td>G</td>
<td></td>
</tr>
<tr>
<td>072</td>
<td>310</td>
<td>10</td>
<td>H</td>
<td></td>
</tr>
<tr>
<td>073</td>
<td>311</td>
<td>11</td>
<td>I</td>
<td></td>
</tr>
<tr>
<td>074</td>
<td>312</td>
<td>12</td>
<td>J</td>
<td></td>
</tr>
<tr>
<td>075</td>
<td>313</td>
<td>13</td>
<td>K</td>
<td></td>
</tr>
<tr>
<td>076</td>
<td>314</td>
<td>14</td>
<td>L</td>
<td></td>
</tr>
<tr>
<td>077</td>
<td>315</td>
<td>15</td>
<td>M</td>
<td></td>
</tr>
<tr>
<td>078</td>
<td>316</td>
<td>16</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>079</td>
<td>317</td>
<td>17</td>
<td>O</td>
<td></td>
</tr>
<tr>
<td>080</td>
<td>320</td>
<td>20</td>
<td>P</td>
<td></td>
</tr>
<tr>
<td>081</td>
<td>321</td>
<td>21</td>
<td>Q</td>
<td></td>
</tr>
<tr>
<td>082</td>
<td>322</td>
<td>22</td>
<td>R</td>
<td></td>
</tr>
<tr>
<td>083</td>
<td>323</td>
<td>23</td>
<td>S</td>
<td></td>
</tr>
<tr>
<td>084</td>
<td>324</td>
<td>24</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>085</td>
<td>325</td>
<td>25</td>
<td>U</td>
<td></td>
</tr>
<tr>
<td>086</td>
<td>326</td>
<td>26</td>
<td>V</td>
<td></td>
</tr>
<tr>
<td>087</td>
<td>327</td>
<td>27</td>
<td>W</td>
<td></td>
</tr>
<tr>
<td>088</td>
<td>330</td>
<td>30</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>089</td>
<td>331</td>
<td>31</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>090</td>
<td>332</td>
<td>32</td>
<td>Z</td>
<td>Left square bracket</td>
</tr>
<tr>
<td>7-Bit Decimal</td>
<td>8-Bit Octal</td>
<td>6-Bit Octal</td>
<td>Character/Control Code</td>
<td>Remarks</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------</td>
<td>-------------</td>
<td>------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>092</td>
<td>334</td>
<td>34</td>
<td>\</td>
<td>Backslash</td>
</tr>
<tr>
<td>093</td>
<td>335</td>
<td>35</td>
<td>[</td>
<td>Right square bracket</td>
</tr>
<tr>
<td>094</td>
<td>336</td>
<td>36</td>
<td>^</td>
<td>Circumflex</td>
</tr>
<tr>
<td>095</td>
<td>337</td>
<td>37</td>
<td></td>
<td></td>
</tr>
<tr>
<td>096</td>
<td>340</td>
<td>40</td>
<td>-</td>
<td>Grave accent</td>
</tr>
<tr>
<td>097</td>
<td>341</td>
<td>41</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>098</td>
<td>342</td>
<td>42</td>
<td>b</td>
<td></td>
</tr>
<tr>
<td>099</td>
<td>343</td>
<td>43</td>
<td>c</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>344</td>
<td>44</td>
<td>d</td>
<td></td>
</tr>
<tr>
<td>101</td>
<td>345</td>
<td>45</td>
<td>e</td>
<td></td>
</tr>
<tr>
<td>102</td>
<td>346</td>
<td>46</td>
<td>f</td>
<td></td>
</tr>
<tr>
<td>103</td>
<td>347</td>
<td>47</td>
<td>g</td>
<td></td>
</tr>
<tr>
<td>104</td>
<td>350</td>
<td>50</td>
<td>h</td>
<td></td>
</tr>
<tr>
<td>105</td>
<td>351</td>
<td>51</td>
<td>i</td>
<td></td>
</tr>
<tr>
<td>106</td>
<td>352</td>
<td>52</td>
<td>j</td>
<td></td>
</tr>
<tr>
<td>107</td>
<td>353</td>
<td>53</td>
<td>k</td>
<td></td>
</tr>
<tr>
<td>108</td>
<td>354</td>
<td>54</td>
<td>l</td>
<td></td>
</tr>
<tr>
<td>109</td>
<td>355</td>
<td>55</td>
<td>m</td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>356</td>
<td>56</td>
<td>n</td>
<td></td>
</tr>
<tr>
<td>111</td>
<td>357</td>
<td>57</td>
<td>o</td>
<td></td>
</tr>
<tr>
<td>112</td>
<td>360</td>
<td>60</td>
<td>p</td>
<td></td>
</tr>
<tr>
<td>113</td>
<td>361</td>
<td>61</td>
<td>q</td>
<td></td>
</tr>
<tr>
<td>114</td>
<td>362</td>
<td>62</td>
<td>r</td>
<td></td>
</tr>
<tr>
<td>115</td>
<td>363</td>
<td>63</td>
<td>s</td>
<td></td>
</tr>
<tr>
<td>116</td>
<td>364</td>
<td>64</td>
<td>t</td>
<td></td>
</tr>
<tr>
<td>117</td>
<td>365</td>
<td>65</td>
<td>u</td>
<td></td>
</tr>
<tr>
<td>118</td>
<td>366</td>
<td>66</td>
<td>v</td>
<td></td>
</tr>
<tr>
<td>119</td>
<td>367</td>
<td>67</td>
<td>w</td>
<td></td>
</tr>
<tr>
<td>120</td>
<td>370</td>
<td>70</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>121</td>
<td>371</td>
<td>71</td>
<td>y</td>
<td></td>
</tr>
<tr>
<td>122</td>
<td>372</td>
<td>72</td>
<td>z</td>
<td></td>
</tr>
<tr>
<td>123</td>
<td>373</td>
<td>73</td>
<td>{</td>
<td></td>
</tr>
<tr>
<td>124</td>
<td>374</td>
<td>74</td>
<td>}</td>
<td></td>
</tr>
<tr>
<td>125</td>
<td>375</td>
<td>75</td>
<td>}</td>
<td></td>
</tr>
<tr>
<td>126</td>
<td>376</td>
<td>76</td>
<td>~</td>
<td>Tilde</td>
</tr>
<tr>
<td>127</td>
<td>377</td>
<td>77</td>
<td>DEL</td>
<td>RUBOUT</td>
</tr>
</tbody>
</table>
APPENDIX B

USEFUL MATHEMATICAL SUBROUTINES

Since the DECstation hardware does not contain built-in multiplication or division instructions, you must incorporate these functions explicitly in your assembly language programs. The subroutines listed in this appendix illustrate efficient ways to do these calculations. These routines perform single-precision, unsigned calculations. This means that all numbers are positive values represented by 12-bit words in the range from 0 to 7777 (octal), inclusive. Another important point is that single-precision, unsigned numbers may be regarded as integers or as fractions, depending on whether the binary point (analogous to the decimal point in the decimal number system) is assumed to be located to the right of the right-most bit (in the case of integers) or to the left of the left-most bit (in the case of fractions).

B.1 UNSIGNED INTEGER MULTIPLICATION SUBROUTINE

/CALCULATES A*B, WHERE A AND B ARE UNSIGNED INTEGERS. IF A*B>7777
/A JMP IS MADE TO LOCATION 'ERROR'.

/CALLING SEQUENCE:
/  JMS ML
/  A
/  B
/CONTROL RESUMES HERE WITH (I.E., AT THE THIRD LOCATION AFTER THE
/JMS INSTRUCTION) WITH ACCUMULATOR(AC)=A*B, A AND B ARE PRESERVED,
/LINK=0 ON RETURN.

/NOTE: USE REPETITIVE ADDITION INSTEAD OF THIS SUBROUTINE
/NOTE: IF EITHER ARGUMENT WILL ALWAYS BE LESS THAN ABOUT 50 (OCTAL).

/NOTE: STARRED INSTRUCTIONS MAY BE REMOVED IF THEIR EFFECTS
/ARE NOT NEEDED.

/NOTE: REMOVING ALL OVERFLOW TEST INSTRUCTIONS GIVES A SUBROUTINE
/WHICH TRUNCATES THE RESULT, RETURNING A*B MOD 10000.

ML,
  0
CLA /* IGNORE AC
TAD I ML
ISZ ML
DCA ML2
TAD KM14
DCA COUNT
ML1,
  CLL RAL /*ACCUMULATE LSB(PRODUCT)
DCA PROD
SZL /* OVERFLOW TEST
JMP ERROR /*
USEFUL MATHEMATICAL SUBROUTINES

```
tad ml2 /get next bit of multiplier
ral
ica ml2
szl /if set, add multiplicand into
    /the partial product
tad i ml /multiplicand
cll /* overflow test
tad prod /partial product
isz count /loop for 12 bits
jmp ml1
isz ml
snl /* overflow test
jmp i ml /*return
cla /*
jmp error /*

/variables
ml2, 0 /multiplier
prod, 0 /partial product
count, 0
km14, -14
```

B.2 UNSIGNED FRACTIONAL MULTIPLICATION SUBROUTINE

```
/calculates a*b/10000, where a and b are unsigned integers.
/calling sequence:
    jms ml
    a
    b
/control resumes here with ac=a*b/10000.
a and b are preserved.

/note: there are no error conditions.
ml, 0
cla /* may be removed if ac=0 on entry
tad i ml /get multiplier
isz ml
dca ml2
tad km14
ml1, dca prod
tad ml2 /get next bit of multiplier
cll rar
dca ml2
szl /if set, add multiplicand into
tad i ml
cll
tad prod /the partial product
rar /keep msb (prod) only
isz count /loop for 12 bits
jmp ml1
isz ml
jmp i ml /*return

/variables
ml2, 0 /multiplier
prod, 0 /partial product
count, 0
km14, -14
```
B.3 UNSIGNED INTEGER DIVISION SUBROUTINE

/ CALCULATES A/B, WHERE A AND B ARE UNSIGNED INTEGERS SUCH THAT 
/ B IS NOT 0, IF B=0 THEN A JMP IS MADE TO LOC "ERROR". 
/ QUOTIENT IS RETURNED IN AC, REMAINDER IS AVAILABLE IN 
/ LOCATION "REMAIN".

/CALLING SEQUENCE:
/   JMS DV 
/   A 
/   B 

/CONTROL RESUMES HERE (I.E., AT THE THIRD LOCATION AFTER THE 
/JMS INSTRUCTION) WITH AC=QUOTIENT(A/B), REMAIN=REMAINDER(A/B). 
/A AND B ARE PRESERVED. LINK=0 ON RETURN.

/NOTE: STARRED INSTRUCTIONS MAY BE REMOVED IF THEIR EFFECTS 
/ARE NOT NEEDED.

DV, 0
CLA    /* IGNORE AC
TAD I DV /*GET ARGS
ISZ DV
DCA DA /*DIVIDEND
TAD I DV
ISZ DV
SNA    /* D = 0 TEST
JMP ERROR /*
CLL CIA /*SUBTRACTION WILL BE DONE BY ADDING
DCA DB /*DIVISOR
DCA REMAIN /*CLEAR MSB (DIVIDEND)
CLL CLA CMA RAL /*SET INITIAL VALUE OF QUOTIENT TO 7776. AS 
/QUOTIENT IS SHIFTED LEFT, THE 0 BIT WILL SHI 
/LEFT, SERVING AS A FLAG TO STOP THE DIVISION 
/LOOP AFTER ALL 12 BITS ARE COMPUTED.
/DIV MAIN LOOP: COMPUTE NEXT BIT OF PARTIAL 
DV1,
DCA QUOT /*QUOTIENT INITIALIZE OR STORE PARTIAL
TAD DA /*QUOTIENT SHIFT REMAINDER LEFT TO NEXT BIT
CLL RAL
DCA DA
TAD REMAIN /*ONE BIT INTO MSB(DIVIDEND)
RAL
DCA REMAIN /*TRIAL SUBTRACTION
TAD REMAIN
TAD DB /*SUBTRACT DIVISOR FROM DIVIDEND
SZL /*STORE BACK REMAINDER ONLY IF .GE. 0
DCA REMAIN
CLA
TAD QUOT
RAL /*ROTATE LINK INTO PARTIAL QUOTIENT
SZL /*DO LOOP ONCE FOR EACH BIT
JMP DV1
JMP I DV /*RETURN

/VARIABLES

DA, 0 /*LSB(DIVIDEND)
REMAIN, 0 /*MSB(DIVIDEND), REMAINDER
DB, 0 /*-DIVISOR
QUOT, 0 /*PARTIAL QUOTIENT
B.4 UNIGNED FRACTIONAL DIVISION SUBROUTINE

/CALCULATES 10000*A/B, WHERE A AND B ARE UNIGNED INTEGERS SUCH THAT
/B .NE. 0 AND A<B. IF B=0 OR A .GE. B THEN A JMP IS MADE TO LOCATION *ER

/CALLING SEQUENCE:
/ JMS DV
/ A
/ B

/CONTROL RESUMES HERE (I.E., AT THE THIRD LOCATION AFTER THE
/JMS INSTRUCTION) WITH AC=QUOTIENT (10000*A/B).
/A AND B ARE PRESERVED. LINK=0 ON RETURN.

/NOTE: STARRED INSTRUCTIONS MAY BE REMOVED IF THEIR EFFECTS
/ARE NOT NEEDED.

DV,
O
CLA /* IGNORE AC
TAD I DV /* GET ARG5
ISZ DV
DCA DA /*DIVIDEND
TAD I DV
ISZ DV
SNA /* B = 0 TEST
JMP ERROR /*
CIL CIA /*SUBTRACTION WILL BE DONE BY ADDING
DCA DB /*DIVISOR
TAD DA /* OVERFLOW TEST
TAD DB /*
SZL CLA /*
JMP ERROR /*
CIL CLA CMA RAL /*SET INITIAL VALUE OF QUOTIENT TO 7776. AS QU
/IS SHIFTED LEFT, THE 0 BIT WILL SHIFT LEFT,
/SERVING AS A FLAG TO STOP THE DIVISION LOOP
/AFTER ALL 12 BITS ARE COMPUTED.
/MAIN LOOP: COMPUTE NEXT BIT OF PARTIAL
/QUOTIENT INITIALIZE OR STORE PARTIAL
/DIVIDEND remainder left to NEXT BIT
DV1,
DCA QUOT /*TRIAL SUBTRACTION
CIL DA TAD DA /*SUBTRACT DIVISOR FROM DIVIDEND
DCA RAL /*STORE BACK REMAINDER ONLY IF .GE. 0
DCA DA
TAD QUOT /*LINK=1 IFF SUBTRACT SUCCEEDED
CIL RAL /*ROTATE LINK INTO PARTIAL QUOTIENT
NZL /*DO LOOP ONCE FOR EACH BIT
JMP DV1
JMP I DV /*RETURN

/VARIABLES
DA, 0 /*DIVIDEND (REMAINDER)
DB, 0 /*DIVISOR
QUOT, 0 /*PARTIAL QUOTIENT
APPENDIX C
USER SERVICE ROUTINE

The User Service Routine, or USR, is a collection of subroutines that perform the operations of opening and closing files, loading device handlers, program chaining, and calling the Command Decoder. The USR provides these functions not only for the system itself but for any programs running under the OS/78 system. A summary of USR functions is given in Table C-1.

USR is used only by PAL8 programs. Users who code in BASIC or FORTRAN IV can skip this appendix and Appendix F, "Using Device Handlers".

The USR resides on the system device and, when called, is swapped into memory. Typically, a series of calls is involved in any I/O operation and it is inefficient to repeat the swap every time. Therefore, the calling program can lock USR in memory in the first 2000 (octal) words of field 1, through the use of USRIN function. To perform any operations on files, it is necessary to have the relevant device handlers available.

This is accomplished with the FETCH function which loads a device handler into memory. Since the handlers reside in the calling program, space must be reserved for them. Special characteristics of OS/78 device handlers and their use are described in Appendix F.

An attempt to call the USR with a code greater than 13 (octal) will cause a Monitor Error 4 message to be printed and the program to be aborted.

C.1 CALLING THE USR

Performing any USR function is done by issuing a JMS instruction followed by the proper arguments. Calls to the USR take a standardized calling sequence. This standard call should be studied before progressing to the operation of the various USR functions.

C.1.1 Standard USR Call

In the remainder of this chapter, the following calling sequence is referenced:

\[
\begin{align*}
\text{TAD VAL} & \quad \text{The contents of the AC is applicable in some cases only.} \\
\text{CDF N} & \quad \text{Where N is the value of the current program field multiplied by 10 (octal).}
\end{align*}
\]
USER SERVICE ROUTINE

**CIF 10**
The instruction field must be set to 1.

**JMS I (USR)**
Where USR is either 7700 or 0200, (see Section C.1.2).

**FUNCTION**
This word contains an integer from 1 to 13 (octal) indicating which USR operation is to be performed as described in the USR Functions Summary Table (Table C-1).

**ARG(1), ARG(2), ARG(3)**
The number and meaning of these argument words varies with the particular USR function to be performed.

**error return**
When applicable, this is the return address for any errors.

**normal return**
The operation was successful. The AC is cleared and the data field is set to current field.

Table C-1
Summary of USR Functions

<table>
<thead>
<tr>
<th>Function Code</th>
<th>Name</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>FETCH</td>
<td>Loads a device handler into memory. Return the entry address of the handler.</td>
</tr>
<tr>
<td>2</td>
<td>LOOKUP</td>
<td>Searches the file directory on any device to locate a specified permanent file.</td>
</tr>
<tr>
<td>3</td>
<td>ENTER</td>
<td>Creates and opens for output a tentative file on a specified device.</td>
</tr>
</tbody>
</table>
| 4             | CLOSE   | Closes the currently open tentative file on the specified device, making it a permanent file. Also, any previous permanent file with the same file name and extension is deleted.
| 5             | DECODE  | Calls the Command Decoder. The function of the Command Decoder is described in Appendix D.     |
| 6             | CHAIN   | Loads a specified memory image file from the system device and starts it.                      |
| 7             | ERROR   | Prints an error message of the form USER ERROR n AT LOCATION xxxxx.                            |
| 10            | USRIN   | Loads the USR into memory. Subsequent calls to the USR are by an effective JMS to location 10200. |
| 11            | USROUT  | Dismisses the USR from memory and restores the previous contents of locations 10000 to 11777. |

(continued on next page)
USER SERVICE ROUTINE

Table C-1 (Cont.)
Summary of USR Functions

<table>
<thead>
<tr>
<th>Function Code</th>
<th>Name</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>INQUIRE</td>
<td>Ascertains whether a given device exists and, if so, whether its handler is in memory.</td>
</tr>
<tr>
<td>13</td>
<td>RESET</td>
<td>Resets system tables to their initial cleared state.</td>
</tr>
<tr>
<td>14 - 17</td>
<td></td>
<td>Not currently used; these request numbers are reserved for future use.</td>
</tr>
</tbody>
</table>

This calling sequence can change from function to function. For example, some functions take no value in the AC and others have fewer or greater numbers of arguments. However, this format is generally followed.

The value of the data field preceding the JMS to the USR is exceedingly important. The data field MUST be set to the current field, and the instruction field MUST be set to 1. Note that a CDF is not explicitly required if the data field is already correct. When a doubt exists as to the data field setting, an explicit CDF should be executed.

Three other restrictions apply to all USR calls, as follows:

1. The USR can never be called from any address between 10000 and 11777. Attempting to do so results in the:

   MONITOR ERROR 4 AT xxxxx (ILLEGAL USR CALL)

   message and termination of program execution. The value of xxxxx is the address of the calling sequence (in all such MONITOR ERROR messages).

2. Several USR calls take address pointers as arguments. These pointers always refer to data in the same memory field as the call.

3. When calling the USR from field 1, these address pointers must never refer to data that lies in the area 10000 to 11777.

C.1.2 Direct and Indirect Sequence

A user program can call the USR in two ways. First, by performing a JMS to location 17700. In this case, locations 10000 to 11777 are saved on a special reserved area on the system device, and the USR is then loaded into 10000 to 11777. When the USR operation is completed, locations 10000 to 11777 are restored to their previous values.
NOTE

By setting bit 11 of the Job Status Word to a 1, you can avoid this saving and restoring of memory for programs that do not use locations 0000 to 1777 in field 1.

Alternatively, a program can keep the USR permanently resident in memory at locations 10000 to 11777 by using the USRIN function (see Section C.2.8). Once the USR has been brought into memory, a USR call is made by performing a JMS to location 10200. This is the most efficient way of calling the USR. When USR operations have been completed, the program restores locations 10000 to 11777 to their original state by executing the USROUT function, if necessary (see Section C.2.9).

C.2 USR FUNCTIONS

C.2.1 FETCH Device Handler (Function Code = 1)

Device handlers must be loaded into memory to make them available to the USR and user program for I/O operations on that device. Before performing a LOOKUP, ENTER, or CLOSE function on any device, the handler for that device must be loaded by FETCH.

The FETCH function takes two forms:

1. Load a device handler corresponding to a given device name.

   First, the following is an example of loading a handler by name from memory field 0:

   | CLA        | /AC MUST BE CLEAR |
   | CDF 0      | /DF = CURRENT FIELD |
   | CIF 10     | /IF = 1 |
   | JMS I (USR 1) | /FUNCTION CODE = 1 |
   | DEVICE RXA1 | /GENERATES TWO WORDS: ARG(1) |
   | 6001        | /AND ARG(2) |
   | JMP ERR     | /ARG(3) |
   | .           | /ERROR RETURN |
   | .           | /NORMAL RETURN |

   ARG(1) and ARG(2) contain the device name in standard format. If the normal return is taken, ARG(2) is changed to the device number corresponding to the device loaded. ARG(3) contains the following information:

   Bits 0 to 4 contain the page number into which the handler is to be loaded (handlers are always loaded and used in field 0).
Bit 11 is 0 if the user program can only accept a 1-page handler for this FETCH operation.

Bit 11 is 1 if there is room for a 2-page handler.

Notice that in the example above, the handler for RXA1 is to be loaded into locations 6000 to 6377. After a normal return, ARG(3) is changed to contain the entry point of the handler.

2. Load a device handler corresponding to a given device number.

A different set of arguments is used to fetch a device handler by number. The following is an example of this form:

```
TAD VAL
CDF 0
CIF 10
JMS I (USR)
1
6001
JMP ERR
.

/AC IS NOT ZERO
/DF = CURRENT FIELD
/IF = 1
/FUNCTION CODE = 1
/ARG(1)
/ERROR RETURN
/NORMAL RETURN
```

On entry to the USR, the AC contains the device number in bits 8 to 11 (bits 0 to 7 are ignored). The format for ARG(1) is the same as that for ARG(3) in the previous example. Following a normal return ARG(1) is replaced with the entry point of the handler.

The conditions that can cause an error return to occur in both cases are as follows:

1. There is no device corresponding to the given device name or device number, or

2. An attempt was made to load a two-page handler into one page. If this is an attempt to load the handler by name, the contents of ARG(2) have been changed already to the internal device number.

In addition, one of the following Monitor errors can be printed, followed by a return to the Monitor:

<table>
<thead>
<tr>
<th>Error Message</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>MONITOR ERROR 4 AT xxxxx (ILLEGAL USR CALL)</td>
<td>Results if bits 8 to 11 of the AC are zero (and bits 0 to 7 are non-zero).</td>
</tr>
<tr>
<td>MONITOR ERROR 5 AT xxxxx (I/O ERROR ON SYS)</td>
<td>Results if a read error occurs while loading the device handler.</td>
</tr>
</tbody>
</table>

The FETCH function checks to see if the handler is in memory, and if it is not, then the handler and all co-resident handlers are loaded. While the FETCH operation is essentially a simple one, the following points should be noted:

1. Device handlers are always loaded into memory field 0.

2. The entry point that is returned may not be on the page desired. This would happen if the handler were already resident.
USER SERVICE ROUTINE

3. Never attempt to load a handler into page 37 or into page 0. Never load a two page handler into page 36 since this will corrupt the OS/78 resident monitor.

For more information on using device handlers, see Appendix F.

NOTE

Two or more device handlers are "co-resident" when they are both included in the same memory pages, for example, RXA0 and RXA1.

C.2.2 LOOKUP Permanent File (Function Code = 2)

This request locates a permanent file entry on a given device, if one exists. An example of a typical LOOKUP would be:

TAD VAL / LOAD DEVICE NUMBER
CDF 0 / DF = CURRENT FIELD
CIF 10 / IF = 1
JMS I (USR)
2 / FUNCTION CODE = 2
NAME / ARG(1), POINTS TO FILE NAME
0 / ARG(2)
JMP ERR / ERROR RETURN
... / NORMAL RETURN
NAME, FILENAME PROG,PA

This request looks up a permanent file with the name PROG,PA. The device number on which the lookup is to be performed must be in AC bits 8 to 11 when the call to USR is made. ARG(1) must contain a pointer to the file name. Note that the file name block must be in the same memory field as the call, and that it cannot be in locations 10000 to 11777. The device handler must have been previously loaded into memory. If the normal return is taken, ARG(1) is changed to the starting block of the file and ARG(2) is changed to the file length in blocks as a negative number. If the device specified is a readable, non-file structured device (for example, the terminal), then ARG(1) and ARG(2) are both set to zero.

If the error return is taken, ARG(1) and ARG(2) are unchanged. The following conditions cause an error return:

1. The device specified is a write-only device.
2. The file specified was not found.
In addition, specifying illegal arguments can cause one of the following monitor errors, followed by a return to the Keyboard Monitor:

<table>
<thead>
<tr>
<th>Error Message</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>MONITOR ERROR 2 AT xxxxx (DIRECTORY I/O ERROR)</td>
<td>Results if an I/O error occurred while reading the device directory.</td>
</tr>
<tr>
<td>MONITOR ERROR 3 AT xxxxx (DEVICE HANDLER NOT IN CORE)</td>
<td>Results if the device handler for the specified device is not in memory.</td>
</tr>
<tr>
<td>MONITOR ERROR 4 AT xxxxx (ILLEGAL USR CALL)</td>
<td>Results if bits 8 to 11 of the AC are zero.</td>
</tr>
</tbody>
</table>

The LOOKUP function is the standard method of opening a permanent file for input.

C.2.3 ENTER Output (Tentative) (File Function Code = 3)

The ENTER function is used to create a tentative file entry to be used for output. An example of a typical ENTER function is as follows:

```
TAB VAL
CDF 0 /DF = CURRENT FIELD
CIF 10 /IF = 1
JMS I (USR)
3 /FUNCTION CODE = 3
NAME /ARG(1) POINTS TO FILE NAME
0 /ARG(2)
JMP ERROR /ERROR RETURN
0 /NORMAL RETURN
```

```
NAME, FILENAME TEST. FA
```

Bits 8 and 11 of the AC contain the device number of the selected device; the device handler for this device must be loaded into memory before performing an ENTER function. If bits 0 to 7 of the AC are non-zero, this value is considered to be a declaration of the maximum length of the file. The ENTER function searches the file directory for the smallest empty file that contains at least the declared number of blocks. If bits 0 to 7 of the AC are zero, the ENTER function locates the largest available empty file.

On the normal return, the contents of ARG(1) are replaced with the starting block of the file. The 2's complement of the actual length of the created tentative file in blocks (which can be equal to or greater than the requested length) replaces ARG(2).

**NOTE**

If the selected device is not file structured but permits output operations (for example, a line printer), the ENTER operation always succeeds. In this case, ARG(1) and ARG(2) are both zeroed on return.
USER SERVICE ROUTINE

If the error return is taken, ARG(1) and ARG(2) are unchanged. The following conditions cause an error return:

1. The device specified by bits 8 to 11 of the AC is a read only device.

2. No empty file exists which satisfies the request length requirement.

3. Another tentative file is already active on this device (only one output file can be active at any given time).

4. The first word of the file name was 0 (an illegal file name).

In addition, one of the following monitor errors can occur, followed by a return to the Keyboard Monitor:

<table>
<thead>
<tr>
<th>Error Message</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>MONITOR ERROR 2 AT xxxxx (DIRECTORY I/O ERROR)</td>
<td>Results if an I/O error occurred while reading or writing the device directory.</td>
</tr>
<tr>
<td>MONITOR ERROR 3 AT xxxxx (DEVICE HANDLER NOT IN MEMORY)</td>
<td>Results if the device handler for the specified device is not in memory.</td>
</tr>
<tr>
<td>MONITOR ERROR 4 AT xxxxx (ILLEGAL USR CALL)</td>
<td>Results if AC bits 8 to 11 are zero.</td>
</tr>
<tr>
<td>MONITOR ERROR 5 AT xxxxx (I/O ERROR ON SYS)</td>
<td>Read error on the system device while bringing in the overlay code for the ENTER function.</td>
</tr>
<tr>
<td>MONITOR ERROR 6 AT xxxxx (DIRECTORY OVERFLOW)</td>
<td>Results if a directory overflow occurred (no room for tentative file entry in directory).</td>
</tr>
</tbody>
</table>

C.2.4 The CLOSE Function (Function Code = 4)

The CLOSE function has a dual purpose: first, it is used to close the current active tentative file, making it a permanent file. Second, when a tentative file becomes permanent, it is necessary to remove (delete) any permanent file having the same name; this operation is also performed by the CLOSE function. An example of CLOSE usage follows:

```
TAD VAL /GET DEVICE NUMBER
CDF 0 /DF = CURRENT FIELD
CIF 10 /IF = 1
JMS I (USR) 4 /FUNCTION CODE = 4
NAME /ARG(1)
15 /ARG(2)
JMP ERR /ERROR RETURN
.
.
NAME, FILENAME TEST.PA
```

C-8
The device number is contained in AC bits 8 to 11 when calling the USR. ARG(1) is a pointer to the name of the file to be closed and/or deleted and ARG(2) contains the number of blocks to be used for the new permanent file.

The normal sequence of operations on an output file is:

1. FETCH the device handler for the output device.

2. ENTER the tentative file on the output device, getting the starting block and the maximum number of blocks available for the file.

3. Perform the actual output using calls to the device handler, keeping track of how many blocks are written, and checking to insure that the file does not exceed the available space.

4. CLOSE the tentative file, making it permanent. The CLOSE operation would always use the same file name as the one used for ENTER performed in step 2. The closing file length would have been computed in step 3 and used in the CLOSE call.

After a normal return from CLOSE, the active tentative file is permanent and any permanent file having the specified file name already stored on the device is deleted. If the specified device is a non-file structured device that permits output (the lineprinter, for example) the CLOSE function will always succeed.

NOTE

The user must be careful to specify the same file names to the ENTER and the CLOSE functions. Failure to do so can cause several permanent files with identical names to appear in the directory. If CLOSE is intended only to be used to delete some existing file, then the number of blocks (ARG(2)) should be zero.

The following conditions cause the error return to be taken:

1. The device specified by bits 8 to 11 of the AC is a read-only device.

2. There is neither an active tentative file to be made into a permanent file nor a permanent file with the specified name to be deleted.

In addition, one of the following Monitor errors can occur:

<table>
<thead>
<tr>
<th>Error Message</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>MONITOR ERROR 1 AT xxxxx (CLOSE ERROR)</td>
<td>Results if the length specified by ARG(2) exceeded the allotted space.</td>
</tr>
<tr>
<td>MONITOR ERROR 2 AT xxxxx (DIRECTORY I/O ERROR)</td>
<td>Results if an I/O error occurred while reading or writing the device directory.</td>
</tr>
</tbody>
</table>
**USER SERVICE ROUTINE**

<table>
<thead>
<tr>
<th>Error Message</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>MONITOR ERROR 3 AT xxxxx</td>
<td>Results if the device handler for the specified device is not in memory.</td>
</tr>
<tr>
<td>(DEVICE HANDLER NOT IN MEMORY)</td>
<td></td>
</tr>
<tr>
<td>MONITOR ERROR 4 AT xxxxx</td>
<td>Results if AC bits 8 to 11 are zero.</td>
</tr>
<tr>
<td>(ILLEGAL USR CALL)</td>
<td></td>
</tr>
</tbody>
</table>

**C.2.5 Call Command Decoder (DECODE) (Function Code = 5)**

The DECODE function causes the USR to load and execute the Command Decoder. The Command Decoder accepts (from the terminal) a list of input and output devices and files, along with various options. The Command Decoder performs a LOOKUP on all input files, sets up necessary tables in the top page of field 1, and returns to the user program.

A typical call to the Command Decoder looks as follows:

CDF 0 /DF = CURRENT FIELD
CIF 10 /IF = 1
JMS I (USR
5 /FUNCTION CODE = 5
2001 /ARG(1), ASSUMED INPUT EXTENSION (.PA)
0 /ARG(2), ZERO TO PRESERVE ALL
	/TENTATIVE FILES
	/NORMAL RETURN

ARG(1) is the assumed input extension; in this example it is ".PA". On return from the Command Decoder, information is stored in tables located in the last page of memory field 1. The DECODE function also resets all system tables as in the RESET function (see RESET function, Section C.2.11); if ARG(2) is 0, all currently active tentative files remain open; if ARG(2) is non-zero, all tentative files are deleted, and the normal return is to ARG(2) instead of ARG(2)+1.

The DECODE function has no error return (Command Decoder error messages are given in Appendix D). However, the following Monitor error can occur:

<table>
<thead>
<tr>
<th>Error Message</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>MONITOR ERROR 5 AT xxxxx</td>
<td>I/O error occurred while reading or writing on the system device.</td>
</tr>
<tr>
<td>(I/O ERROR ON SYS)</td>
<td></td>
</tr>
</tbody>
</table>

**C.2.6 CHAIN Function (function Code = 6)**

The CHAIN function permits a program to run another program with the restriction that the program chained to must be a memory image (.SV) file located on the system device. A typical implementation of the CHAIN function follows:

CDF 0 /DF = CURRENT FIELD
CIF 10 /IF = 1
JMS I (USR
6 /FUNCTION CODE = 6
BLOCK /ARG(1), STARTING BLOCK NUMBER
USER SERVICE ROUTINE

There is no normal or error return from CHAIN. However, the following monitor error can occur:

<table>
<thead>
<tr>
<th>Error Message</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>MONITOR ERROR 5 AT</td>
<td>I/O error occurred while reading or writing on the system</td>
</tr>
<tr>
<td>xxxx    (I/O ERROR</td>
<td>device.</td>
</tr>
<tr>
<td>ON SYS)</td>
<td></td>
</tr>
<tr>
<td>CHAIN ERR</td>
<td>If an attempt is made to CHAIN to a file which is not a</td>
</tr>
<tr>
<td></td>
<td>memory image (.SV) file. Control returns to the Monitor.</td>
</tr>
</tbody>
</table>

The CHAIN function loads a memory image file located on the system device beginning at the block number specified as ARG(1) (which is normally determined by performing a LOOKUP on the desired file name). Once loaded, the program is started at an address that is one greater than the starting address specified by the program's Core Control Block.

CHAIN automatically performs a USROUT function (see Section C.2.9), if necessary, to dismiss the USR from memory, and a RESET to clear all system tables (see Section C.2.11), but CHAIN does not delete tentative files. Normally, it is best to set bit 11 of the Job Status Word before chaining to an OS/78 system program.

The areas of memory altered by the CHAIN function are determined by the contents of the Core Control Block of the memory image file loaded by CHAIN. The Core Control Block for the file is set up by ABSLDR or LOADER programs. It can be modified by performing a SAVE command with specific arguments. Every page of memory in which at least one location was saved is loaded. If the page is one of the "odd numbered" pages (pages 1, 3, etc.; locations 0200 to 0377, 0600 to 0777, etc.), the previous page is always loaded. In addition, CHAIN always alters the contents of locations 07200 to 07577.

**NOTE**

CHAIN destroys a necessary part of the ODT resident breakpoint routine. Thus an ODT breakpoint should never be attempted across a CHAIN.

With the above exceptions, programs can pass data back and forth in memory while chaining. For example, FORTRAN programs normally leave the COMMON area in memory field 1 unchanged. This COMMON area can then be accessed by the program activated by the CHAIN.

C.2.7 Signal User ERROR (Function Code = 7)

The USR can be called to print a user error message for a program. The following is a possible ERROR call:

```
CDF 0 /DF = CURRENT FIELD
CIF 10 /IF = 1
JMS I (USR
7 /FUNCTION CODE = 7
2 /ARG(1), ERROR NUMBER
```
The **ERROR** function causes a message of the form:

**USER ERROR n AT xxxxx**

to be printed. Here **n** is the error number given as ARG(1); **n** must be between 0 and 11 (octal), and **xxxx** is the address of ARG(1). If ARG(1) in the sample call above was at location 500 in field 0, the message:

**USER ERROR 2 AT 00500**

would be printed. Following the message, the USR returns control to the Keyboard Monitor, preserving the user program intact.

The error number is arbitrary. Two numbers have currently assigned meanings:

<table>
<thead>
<tr>
<th>Error Message</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>USER ERROR 0 AT xxxx</td>
<td>During a RUN, GET, or R command, this error message indicates that an error occurred while loading the memory image.</td>
</tr>
<tr>
<td>USER ERROR 1 AT xxxx</td>
<td>During a FORTRAN program execution, this error indicates that a call was made to a subroutine that was not loaded.</td>
</tr>
</tbody>
</table>

**C.2.8 Lock USR in Memory (USRIN) (Function Code = 10)**

When a program must make a number of calls to the USR, it is advantageous to have the program avoid reloading the USR each time a USR call is made. The USR can be brought into memory and kept there by the USRIN function. The calling sequence for the USRIN function is as follows:

```
CDF 0      /DF = CURRENT FIELD
CIF 10     /IF = 1
.JMS I (7700)      \FUNCTION CODE = 10
10          /NORMAL RETURN
.
.
```

The USRIN function saves the contents of locations 10000 to 11777 in a reserved area on SYS: (provided the calling program loads into this area as indicated by the current Job Status Word), then loads the USR, and finally returns control to the user program.

Once the USR is locked in memory, you can call it at location 10200.

**NOTE**

If bit 11 of the current Job Status Word is a 1, the USRIN function will not save the contents of locations 10000 through 11777.
C.2.9 Dismiss USR From Memory (USROUT) (Function Code = 11)

When a program has loaded the USR into memory with the USRIN function and no longer wants or needs the USR in memory, the USROUT function restores the original contents of locations 10000 to 11777. The calling sequence for the USROUT function is as follows:

CDF 0       /DF = CURRENT FIELD
CIF 10      /IF = 1
JMS I (200) /DO NOT JMS TO 17700!!
11          /FUNCTION CODE = 11
.            /NORMAL RETURN
.
.

Subsequent calls to the USR must be made by performing a JMS to location 7700 in field 1.

NOTE

If bit 11 of the current Job Status Word is a 1, the contents of memory are not changed by the USROUT function. In this case USROUT is a redundant operation since memory was not preserved by the USRIN function.

C.2.10 Ascertain Device Information (INQUIRE) (Function Code = 12)

On some occasions a user may wish to determine what internal device number corresponds to a given device name or whether the device handler for a specified device is in memory, without actually performing a FETCH operation. INQUIRE performs these operations for the user. The function call for INQUIRE closely resembles the FETCH handler call.

INQUIRE, like FETCH, has two forms:

1. Obtain the device number corresponding to a given device name and determine if the handler for that device is in memory.

An example of the INQUIRE call is shown below:

CLA       /AC MUST BE CLEAR
CDF 0     /DF = CURRENT FIELD
CIF 10    /IF = 1
JMS I (USR) 12 /FUNCTION CODE = 12
DEVICE RLOB /GENERATES TWO WORDS:
             /ARG(1) AND ARG(2)
             0       /ARG(3)
JMP ERR  /ERROR RETURN
.       /NORMAL RETURN
.
.

ARG(1) and ARG(2) contain the device name in standard format. When the normal return is taken, ARG(2) is changed to the device number corresponding to the given name, and ARG(3) is changed to either the entry point of the device handler if it is already in memory, or zero if the corresponding device handler has not yet been loaded.

C-13
2. Determine if the handler corresponding to a given device number is in memory.

A slightly different set of arguments is used to inquire about a device by its device number:

| TAD VAL  | /AC IS NON-ZERO |
| CDF 0    | /DF = CURRENT FIELD |
| CIF 10   | /IF = 1 |
| JMS I (USR) | /FUNCTION CODE = 12 |
| 12       | /ARG(1) |
| 0        | /ERROR RETURN |
| JMR ERR  | /NORMAL RETURN |

On entry to INQUIRE, AC bits 8 to 11 contain the device number.

NOTE

If AC bits 0 to 7 are non-zero, and bits 8 to 11 are zero (an illegal device number), the

MONITOR ERROR 4 AT xxxxx

message is displayed and program execution is terminated.

On normal return ARG(1) is set to the entry point of the device handler if it is already in memory, or zero if the corresponding device handler has not yet been loaded. The error return in both cases is taken only if there is no device corresponding to the device name or number specified.

C.2.11 RESET System Tables (Function Code = 13)

Resetting the system tables effectively removes from memory all device handlers except the system handler. An example of the RESET function is shown below:

| CDF 0    | /DF = CURRENT FIELD |
| CIF 10   | /IF = 1 |
| JMS I (USR) | /FUNCTION CODE = 13 |
| 13       | /O PRESERVES TENTATIVE FILES |
| 0        | /NORMAL RETURN |

RESET removes all device handlers, other than that for the system device, from memory. This operation should be done anytime a user program modifies any page in which a device handler was loaded.
USER SERVICE ROUTINE

RESET also deletes all currently active tentative files (files that have been entered but not closed). This results in zeroing bits 9 through 11 of every entry in the Device Control Word Table (see Section B.3.5 of OS/8 Software Support Manual). If RESET is to be used to delete all active tentative files, then ARG(1) must be non-zero and the normal return is ARG(1) rather than to ARG(1)+1. For example, the following call would serve this purpose.

CDF 0 /DF:CURRENT FIELD
CIF 10 /IF = 1
JMS I 1 (USR
13 /FUNCTION CODE = 13
CLA CMA /NON-ZERO INSTRUCTION

The normal return would execute the CLA CMA, and all active tentative files on all devices would be deleted. The Keyboard Monitor currently does not reset the Monitor tables. If user programs which do not call the Command Decoder are used, it is wise to do a RESET operation before loading device handlers. The RESET will ensure that the proper handler will be loaded into memory.
OS/78 provides a powerful subroutine called the Command Decoder for use by all system and user programs. The Command Decoder is normally called when a program starts running. When called, the Command Decoder displays an asterisk (*) and then accepts a command line from the console terminal that includes a list of I/O devices, file names, and various option specifications. The Command Decoder validates the command line for accuracy, performs a LOOKUP on all input files, and sets up various tables for the calling program (refer to Appendix C, Section C.2.2, for more information on LOOKUP).

D.1 COMMAND DECODER CONVENTIONS

The command line has the following general form in response to the Command Decoder asterisk:

*output files <input files/(options)

There can be 0 to 3 output files and 0 to 9 input files specified.

<table>
<thead>
<tr>
<th>Output File Examples</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXPLE.EX</td>
<td>Output to a file named EXPLE.EX on device DSK (the default file storage device).</td>
</tr>
<tr>
<td>LPT:</td>
<td>Output to the LPT. This format generally specifies a non-file-structured device.</td>
</tr>
<tr>
<td>RXA2:EXPLE.EX</td>
<td>Output to a file named EXPLE.EX on device RXA2.</td>
</tr>
<tr>
<td>RL0A:EXPLE.EX[99]</td>
<td>Output to a file named EXPLE.EX on device RL0A. A maximum output file size of 99 blocks is specified.</td>
</tr>
<tr>
<td>(null)</td>
<td>No output specified.</td>
</tr>
</tbody>
</table>

An input file specification has one of the following forms:

<table>
<thead>
<tr>
<th>Input File Format</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>RXA2:INPUT</td>
<td>Input from a file named INPUT.df on device RXA2. Where df is the assumed input file extension specified to the Command Decoder.</td>
</tr>
<tr>
<td>RXA2:INPUT.EX</td>
<td>Input from a file named INPUT.EX on device RXA2. In this case .EX overrides the assumed input file extension.</td>
</tr>
</tbody>
</table>
THE COMMAND DECODER

Input File Format | Meaning
--- | ---
INPUT.EX | Input from a file named INPUT.EX. If there is no previously specified input device, input is from device DSK, the default file storage device; otherwise, the input device is the same as the last specified input device.
TTY: | Input from terminal; no file name is needed for non-file structured devices.
(null) | Repeats input from the previous device specified (must not be first in input list, and must refer to a non-file structured device).

NOTE
Whenever a file extension is left off an input file specification, the Command Decoder first performs a LOOKUP for the given name appending a specified assumed extension. If the LOOKUP fails, a second LOOKUP is made for the file appending a null (zero) extension.

The Command Decoder verifies that the specified device names, file names, and extensions consist only of the characters A through Z and 0 through 9. If not, a syntax error is generated and the command line is considered to be invalid.

Two kinds of options can be specified: alphanumeric and numeric. Alphanumeric option switches are denoted by a single alphanumeric character preceded by a slash (/) or a string of characters enclosed in parentheses. Numeric options can be specified in octal numbers from 1 to 37777777 preceded by an equal sign (=). These options are passed to the user program and are interpreted differently by each program.

Finally, the Command Decoder permits the command line to be terminated by either the RETURN or ESCape key. This information is also passed to the user program.

D.2 COMMAND DECODER ERROR MESSAGES

If an error in the command line is detected by the Command Decoder, one of the following error messages is displayed. After the error message, the Command Decoder starts a new line, prints an *, and waits for another command line. The erroneous command is ignored.

<table>
<thead>
<tr>
<th>Error Message</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ILLEGAL SYNTAX</td>
<td>The command line is formatted incorrectly.</td>
</tr>
<tr>
<td>TOO MANY FILES</td>
<td>More than three output files or nine input files were specified. (Or in special mode, more than one output file or more than five input files.)</td>
</tr>
</tbody>
</table>
THE COMMAND DECODER

<table>
<thead>
<tr>
<th>Error Message</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>device DOES NOT EXIST</td>
<td>The specified device name does not correspond to any permanent device name or any user assigned device name.</td>
</tr>
<tr>
<td>name NOT FOUND</td>
<td>The specified input file name was not found on the selected device.</td>
</tr>
</tbody>
</table>

D.3 CALLING THE COMMAND DECODER

The Command Decoder is initiated by the DECODE function of the USR. DECODE saves the contents of locations 0 to 1777 of field 0 on the system scratch blocks, and brings the Command Decoder into that area of memory and starts it. When the command line has been entered and properly interpreted, the Command Decoder exits to the USR, which restores the original contents of 0 to 1777 and returns to the calling program.

NOTE

By setting bit 10 of the Job Status Word to a 1; you can avoid this saving and restoring of memory for programs that do not occupy locations 0 to 1777.

The DECODE call can reside in the area between 00000 to 01777 and still function correctly. A typical call would appear as follows:

CDF 0 /SET DATA FIELD TO CURRENT FIELD
CIF 10 /INSTRUCTION FIELD MUST BE 1
JMS I (USR /USR=7700 IF USR IS NOT IN MEMORY
/ OR USR=0200 IFUSRIN WAS PERFORMED
S /DECODE,FUN CTION = 5
2001 /ARG(1), ASSUMED INPUT EXTENSION
O /ARG(2), ZERO TO PRESERVE
. /ALL TENTATIVE FILES
. /NORMAL RETURN

ARG(1) is the assumed input extension in SIXBIT notation. If an input file name is given with no specified extension, the Command Decoder first performs a LOOKUP for a file having the given name with the assumed extension. If the LOOKUP fails, the Command Decoder performs a second LOOKUP for a file having the given name and a null (zero) extension. In this example, the assumed input extension is ".PA".

DECODE performs an automatic RESET operation to remove from memory all device handlers except those equivalent to the system device. As in the RESET function, if ARG(2) is zero, all currently active tentative files are preserved. If ARG(2) is non-zero, all tentative files are deleted, and DECODE returns to ARG(2) instead of ARG(2)+1.

As the Command Decoder normally handles all of its own errors, there is no error return from the DECODE operation.

D-3
THE COMMAND DECoder

D.4 COMMAND DECoder TABLES

The Command Decoder sets up various tables in the top page of field 1 that describe the command line typed to the user program.

D.4.1 Output Files

The output file table that begins at location 17600 has room for three entries. Each entry is five words long and has the following format:

<table>
<thead>
<tr>
<th>WORD 1</th>
<th>USER SPECIFIED FILE LENGTH</th>
<th>4-BIT DEVICE NUMBER</th>
</tr>
</thead>
<tbody>
<tr>
<td>WORD 2</td>
<td>FILE NAME CHARACTER 1</td>
<td>FILE NAME CHARACTER 2</td>
</tr>
<tr>
<td>WORD 3</td>
<td>FILE NAME CHARACTER 3</td>
<td>FILE NAME CHARACTER 4</td>
</tr>
<tr>
<td>WORD 4</td>
<td>FILE NAME CHARACTER 5</td>
<td>FILE NAME CHARACTER 6</td>
</tr>
<tr>
<td>WORD 5</td>
<td>FILE EXTENSION CHARACTER 1</td>
<td>FILE EXTENSION CHARACTER 2</td>
</tr>
</tbody>
</table>

Bits 0 to 7 of word 1 in each entry contain the file length, if the file length was specified with the square bracket construction in the command line. Otherwise, those bits are zero.

The entry for the first output file is in locations 17600 to 17604, the second is in locations 17605 to 17611, and the third is in locations 17612 to 17616. If word 1 of any entry is zero, the corresponding output file was not specified. A zero in word 2 means that no file name was specified.

Also, if word 5 of any entry is zero no file extension was specified for the corresponding file. It is left to the user program to take the proper action in these cases.

These entries are in a format that is acceptable to the ENTER function.

D.4.2 Input Files

The input file table that begins at location 17617 has room for nine entries. Each entry is two words long and has the following format:

<table>
<thead>
<tr>
<th>WORD 1</th>
<th>MINUS FILE LENGTH</th>
<th>4-BIT DEVICE NUMBER</th>
</tr>
</thead>
<tbody>
<tr>
<td>WORD 2</td>
<td>STARTING BLOCK OF FILE</td>
<td></td>
</tr>
</tbody>
</table>
THE COMMAND DECODER

Bits 0 to 7 of word 1 contain the file length as a negative number. Thus, 377 (octal) in these bits is a length of one block, 376 (octal) is a length of two blocks, etc. If bits 0 to 7 are zero, the specified file has a length greater than or equal to 256 blocks or a non-file structured device was specified.

NOTE

Restricting the actual specified size to 255 blocks can cause some problems if the program has no way of detecting end-of-file conditions. If this is a problem, your program probably should use the special mode of the Command Decoder described in Section D.5 and perform its own LOOKUP on the input files to obtain the exact file length.

The two-word input file list entries beginning at odd numbered locations from 17617 to 17637 inclusive. If location 17617 is zero, no input files were indicated in the command line. If less than nine input files were specified, the unused entries in the input file list are zeroed (location 17641 is always set to zero to provide a terminator even when no files are specified).

D.4.3 Command Decoder Option Table

Five words are reserved beginning at location 17642 to store the various options specified in the command line. The format of these five words is as follows:

```
<table>
<thead>
<tr>
<th>17642</th>
<th>HIGH ORDER 11 BITS OF = N OPTIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>17643</td>
<td>A B C D E F G H I J K L</td>
</tr>
<tr>
<td>17644</td>
<td>M N O P Q R S T U V W X</td>
</tr>
<tr>
<td>17645</td>
<td>Y Z 0 1 2 3 4 5 6 7 8 9</td>
</tr>
<tr>
<td>17646</td>
<td>LOW ORDER 12 BITS OF = N OPTIONS</td>
</tr>
</tbody>
</table>
```

Each of the bits in locations 17643 and 17645 corresponds to one of the possible alphanumeric option switches. The corresponding bit is 1 if the switch was specified, 0 if not specified.
NOTE

If no option is specified, the Command Decoder zeroes location 17646 and bits 1 to 11 of 17642. Thus, typing = 0 is meaningless since the user program cannot tell that any option was specified.

Bit 0 of location 17642 is 0 if the command line was terminated by a carriage return, 1 if it was terminated by an ESCape.

D.4.4 Example

To clarify some of the preceding, consider the interpretation of the following command line:

*BIN[10]<TTY:,RXA2:PARA.PA,MAIN.PA/L=14200

If this command line is typed, the Command Decoder would return to the calling program with the following values in the system tables:
17600 0242  DSK: IS DEVICE NUMBER 2
       0211
       1600  FILE NAME IS BIN
       0000
17604 0000  NULL EXTENSION
17605
17616  REMAINING ENTRIES IN OUTPUT TABLES ARE ZERO
17617  0016  FIRST TERMINAL INPUT
       0000
17620  0016  SECOND TERMINAL INPUT
17621  0000
17622  7667  RXA2: PARA PA IS 5 BLOCKS LONG, BEGINNING AT 100 (8)
17623  0100
17624  0007  RXA2: MAIN PA IS 256 (10) OR MORE BLOCKS LONG, BEGINNING AT BLOCK 105 (8)
17625  0105
17626  REMAINING ENTRIES IN INPUT TABLES ARE ZERO
17627
17641  4001  LINE WAS TERMINATED BY ESC (1 OF 14200)
17642  0001
17643  0000  /L WAS ONLY OPTION SWITCH SPECIFIED
17644  0000
17645  0000
17646  4200  LSD'S OF 14200 WAS SPECIFIED
NOTE

The entries for terminal (where no input file name is specified) have a starting block number and file size of zero. This is always true of the input table for a non-file structured device, or a file structured device on which no file name is given.

D.5 SPECIAL MODE OF THE COMMAND DECODER

Occasionally the user program does not want the Command Decoder to perform the LOOKUP on input files, leaving this option to the user program itself. Programs such as format conversion routines which access non-standard file structures could use this special format. If the input files were not OS/78 format, a command decoder LOOKUP operation would fail. The capability to handle this case is provided in the OS/78 Command Decoder. This capability is generally referred to as the "special mode" of the Command Decoder.

D.5.1 Calling the Command Decoder Special Mode

The special mode call to the Command Decoder is identical to the standard DECODE call except that the assumed input file extension, specified by ARG(1), is equal to 5200. The value 5200 corresponds to an assumed extension of ".*", which is illegal. Therefore, the special mode of the Command Decoder in no way conflicts with the normal mode.

D.5.2 Operation of the Command Decoder in Special Mode

In special mode the Command Decoder is loaded and inputs a command line as usual. The appearance of the command line is altered by the special mode in these respects:

1. Only one output file can be specified.
2. No more than five input files can be specified, rather than the nine acceptable in normal mode.
3. The characters asterisk (*) and question mark (?) are legal in file names and extensions, both in input files and on output files. It is strongly suggested that these characters be tested by the user program and treated either as special options or as illegal file names. The user program must not ENTER an output file with an asterisk or a question mark in its name since the OS/78 system programs will have difficulty manipulating or deleting files named this way.
THE COMMAND DECODER

The output and option table set up by the Command Decoder is not altered in special mode. Entries in the input table are changed to the following format:

<table>
<thead>
<tr>
<th>WORD 1</th>
<th>4-BIT DEVICE NUMBER</th>
<th>WORD 2</th>
<th>FILE NAME CHARACTER 1</th>
<th>FILE NAME CHARACTER 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>WORD 3</td>
<td>FILE NAME CHARACTER 3</td>
<td>FILE NAME CHARACTER 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WORD 4</td>
<td>FILE NAME CHARACTER 5</td>
<td>FILE NAME CHARACTER 6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WORD 5</td>
<td>FILE EXTENSION CHARACTER 1</td>
<td>FILE EXTENSION CHARACTER 2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

BITS 0-7 ARE ALWAYS 0

OUTPUT FILE NAME 6 CHARACTER

FILE EXTENSION 2 CHARACTERS

The table entry for the first input file is in locations 17605 to 17611; the second in locations 17612 to 17616; the third in location 17617 to 17623; the fourth in locations 17624 to 17630; and the fifth in locations 17631 to 17635. A zero in word 1 terminates the list of input files. If word 2 of an entry is zero, no input file name was specified.
APPENDIX E
OS/78 DEFAULT FILE NAME EXTENSIONS

This appendix lists the default file name extensions used in OS/78.

<table>
<thead>
<tr>
<th>Extension</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>.BA</td>
<td>BASIC source file (default extension for a BASIC input file).</td>
</tr>
<tr>
<td>.BI</td>
<td>Batch input file (input for BATCH).</td>
</tr>
<tr>
<td>.BN</td>
<td>Absolute binary file (default extension for ABSLDR and BITMAP input files; also used as default extension for PAL8 binary output file).</td>
</tr>
<tr>
<td>.CM</td>
<td>Command file.</td>
</tr>
<tr>
<td>.DI</td>
<td>Directory listing file.</td>
</tr>
<tr>
<td>.FT</td>
<td>FORTRAN language source file (default extension for FORTRAN input files).</td>
</tr>
<tr>
<td>.HL</td>
<td>Text file accessed by the HELP command.</td>
</tr>
<tr>
<td>.HN</td>
<td>Device handler save image file used by the SET HANDLER command.</td>
</tr>
<tr>
<td>.LD</td>
<td>FORTRAN load module file (default assumed by run-time system, FORTRAN IV loader).</td>
</tr>
<tr>
<td>.LS</td>
<td>Assembly listing Output file (default extension for PAL8).</td>
</tr>
<tr>
<td>.MP</td>
<td>File containing a loading map (used by the Linking Loader, MAP command).</td>
</tr>
<tr>
<td>.PA</td>
<td>PAL8 source file.</td>
</tr>
<tr>
<td>.RA</td>
<td>RALF assembly language file (FORTRAN IV).</td>
</tr>
<tr>
<td>.RL</td>
<td>Relocatable binary file (default extension for a Linking Loader input file).</td>
</tr>
<tr>
<td>.SV</td>
<td>Memory image file (SAVE file); default for the R, RUN, SAVE, and GET commands.</td>
</tr>
<tr>
<td>.TM</td>
<td>Temporary file generated by system.</td>
</tr>
</tbody>
</table>
APPENDIX F

USING DEVICE HANDLERS

A device handler is a system subroutine that is used by all parts of the OS/78 system and by all standard system programs to perform I/O transfers. All device handlers are called in the same way, and they all perform the same basic operation of reading or writing a specified number of 128-word records beginning at a selected memory address. This appendix contains information relevant to assembly language (PAL8) programming only.

These subroutines effectively mask the unique characteristics of different I/O devices from the calling program; thus, programs that use device handlers properly are effectively "device independent". Changing devices involves merely changing the device handlers used for I/O.

Device handlers have another important feature. They are able to transfer a number of records as a single operation. On a file-structured device, a single operation could transfer an entire track or more. This capability significantly increases the speed of operation of programs that have large buffer areas.

All device handlers occupy two memory pages, and can run in any page of field 0, except in Pages 0, 36 and 37.

NOTE

A "record" is 128 words of data; thus, an OS/78 block consists of two 128-word records.

F.1 CALLING DEVICE HANDLERS

Device handlers are loaded into a user selected area in memory field 0 by the FETCH function of the USR (see Appendix C). This is possible because the SYS: handler is always resident. FETCH stores in ARG(1) the entry point of the handler loaded. The handler is called by executing a JMS to the specified entry point address. Handler calls have the following format:

CDF N  /WHERE N IS THE VALUE OF THE CURRENT PROGRAM FIELD TIMES 10 (OCTAL)
CIF 0  /DEVICE HANDLER ALWAYS IN FIELD 0
JMS I ENTRY  /ENTRY CONTAINS THE HANDLER ENTRY POINT
ARG(1)  /FUNCTION CONTROL WORD
ARG(2)  /BUFFER ADDRESS
ARG(3)  /STARTING BLOCK NUMBER
JMP ERR  /ERROR RETURN
USING DEVICE HANDLERS

/NORMAL RETURN (I/O TRANSFER COMPLETE)

ENTRY, 0

/ENTRY CONTAINS ARG(3) FROM THE FETCH OPERATION

NOTE

The entry point for SYS is always 07607.

As with calls to the USR, it is important to set the data field is set to the current program field before the device handler is called. On exit from the device handler, the data field will remain set to the current program field. The accumulator need not be zero when calling a handler; it will be zero when the normal return is taken.

ARG(1) is the function control word, and contains the following information:

<table>
<thead>
<tr>
<th>Bits</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit 0</td>
<td>0 for an input operation (read), 1 for an output operation (write).</td>
</tr>
<tr>
<td>Bits 1 to 5</td>
<td>The number of 128 word records to be transferred. If bits 1-5 are zero and the device is non-file structured (i.e., TTY, LPT, etc.) the operation is device dependent. If the device is file structured, a read/write of 40 (octal) pages is performed.</td>
</tr>
<tr>
<td>Bits 6 to 8</td>
<td>The memory field in which the transfer is to be performed.</td>
</tr>
<tr>
<td>Bits 9 to 11</td>
<td>unused (device dependent) bits should be left zero.</td>
</tr>
</tbody>
</table>

ARG(2) is the starting location of the transfer buffer.

ARG(3) is the number of the block at which the transfer is to begin. The user program initially determines this value by performing a LOOKUP or ENTER operation. After each transfer the user program should itself add to the current block number (this argument) the actual number of blocks transferred, equal to one-half the number of 128-word records specified, rounded up if the number of records was odd.

Error returns are either: fatal or non-fatal. When an error return occurs and the contents of the AC are negative, the error is fatal. A fatal error can be caused by a bad data checksum (CRC) or an attempt to write on a read-only device (or vice versa). The meaning can vary from device to device, but in all cases it is serious enough to indicate that the data transferred, if any, is invalid.

When an error return occurs and the contents of the AC are greater than or equal to zero, a non-fatal error has occurred. This error always indicates detection of the end-of-file character (CTRL/Z) on non-file-structured input devices. For example, when a CTRL/Z is typed during input from device TTY, the TTY handler inserts a CTRL/Z code in the buffer and takes the error exit with the AC equal to zero. While non-file-structured input devices can detect the end-of-file condition, file-structured devices cannot; furthermore, no device handler takes a non-fatal error return when doing output.
The following restrictions apply to the use of device handlers:

1. File-structured Handlers - If bits 1-5 of the function control word (ARG(1)) are zero, the handler transfers an entire memory page (40 (octal) blocks). Be careful when using this type of transfer not to overlay the handler itself on the system that resides in the various parts of Fields 0, 1, and 2.

2. The user program must never specify an input into locations 07600 to 07777, 17600 to 17777, or 27600-27777, or the page(s) in which the device handler itself resides. In general, 7600-7777 in every memory field are reserved for use by system software. Those areas should be used with caution.

3. Note that the amount of data transferred is given as a number of 128-word records, exactly one half of an OS/78 block. Attempting to output an odd number of records will change the contents of the last 128 words of the last block written.

4. The specified buffer address does not have to begin at the start of a page. The specified buffer cannot overlap fields; rather the address will "wrap around" memory. For example, a write of 2 pages starting at location 07600 would cause locations 07600-07777 and 00000-00177 of field 0 to be written.

5. If bits 1-5 of the function control word (ARG(1)) are zero, a non-file-structured device-dependent operation occurs. Users should not expect a 40-page (full field) transfer of data. The CLOSE operation of the USR calls the handler with bits 1-5 and 9-11 of the function control word 0. This condition means "perform any special close operation desired". Non-file structured handlers, which need no special handling on the conclusion of data transfers, should treat this case as a NOP. An example of usage of such special codes would be where the line printer would perform a line feed.

F.2 OS/78 DEVICE HANDLERS

This section describes briefly the operation of standard OS/78 device handlers, including normal operation, any special initialization operations for function control word=0, terminating conditions, and response to control characters typed at the keyboard.

F.2.1 Line Printer (LPT, LQP)

1. Normal Operation

These handlers unpack characters from the buffer and print them on the line printer. The character horizontal tab (ASCII 211) causes sufficient spaces to be inserted to position the next character at a "tab stop" (every eighth column, by definition). The character vertical tab (ASCII 213) causes a skip to the next paper position for vertical tabulation if the line printer hardware provides that feature. The character form feed (ASCII 214) causes a skip to the top of the next page. Finally, the handler maintains a record of the current print column and starts a new line after the full width has been printed. This handler functions properly only on ASCII data.
2. Initialization for Function Control Word = 0

Before printing begins, the line printer handler issues a form feed to space to the top of the next page.

3. Terminating Condition

On detection of a CTRL/Z character in the buffer, the line printer handler issues a form feed and immediately takes the normal return. Attempting to input from the line printer forces a fatal error to be returned. A fatal error is also returned if the line printer error flag is set. There are no fatal errors associated with the line printer handler if the device is off line.

F.2.2 File-Structured Devices (SYS, DSK, RXAx, RLxx, VXA0)

1. Normal Operation

These handlers transfer data directly between the device and the buffer.

2. Initialization for Function Control Word = 0

None.

3. Terminating Conditions

A fatal error is returned whenever the transfer sets one of the error flags in the device status register to be set. For example, a fatal error would result if a CRC error occurred while reading data from a diskette. The device handlers generally try to perform the operation three times before giving up and returning a fatal error. All errors for file-structured devices are fatal.

4. Terminal Interaction

Typing CTRL/C forces a return to the Monitor except when using the system device handler (SYS).

NOTE

The system and non-system device handlers for the RX01 and RX02 when called with a negative block number (4000-7777) take the error return with the complement of the device size in the AC (single density = 7022, double density = 6044). This allows you to determine whether the device is on RX01 or on RX02.
F.2.3 Terminal Handlers (TTY, SLUX, VLUX)

Listed are the features of the terminal handlers:

1. These handlers read a line at a time. Whenever the user types CR, they enter a CR/LF into the buffer, echo a CR/LF, pad the remainder of the buffer with nulls, and return to the calling program. The characters are inserted into the buffer one character per word. Thus every third character is a null so far as OS/78 is concerned.

2. The DELETE deletes the previous character from the buffer. The handler will echo a backslash (\) for a hard-copy device or delete the character from the screen for a video terminal. This can be altered by the SET term SCOPE command.

3. CTRL/U echoes as ^U and erases the current line, allowing the user to retype it. (It also starts a new line.) The buffer pointer is reset to the beginning of the buffer.

4. CTRL/Z echoes as ^Z (followed by CR, LF) and is used to signal end-of-file (end of input). The ^Z enters the buffer and the remainder of the buffer is padded with nulls. The error return is taken with a positive AC (non-fatal error).

5. Nulls are ignored.

6. CTRL/C echoes as ^C and returns control to the Monitor via location 07600.

On output: (either normal output or when echoing input)

1. CTRL/C on keyboard echoes as ^C and returns control to the Monitor.

2. CTRL/O on keyboard stops further echoing. All visible output ceases (although the buffer continues to be filled) until either the handler is reloaded into memory or the user types any character on the keyboard. Not operative during input.

3. CTRL/S causes the handler to suspend output to the terminal. No characters are lost, and output resumes when a CTRL/Q is typed. CTRL/S and CTRL/O do not echo. These characters are operative only upon output. On input, they are treated like other input characters. This is useful on high-speed video terminals.

F.2.4 Multiple Input Files

There is a peculiarity associated with reusing Device Handler areas in OS/78. This is illustrated by the following example:

Assume a program has reserved locations 1000-1377 for its input handler and locations 7400-7577 for its output handler. If the program gives a USR FETCH command to load the RXA1 handler as an input device handler, both RX handlers will load into 1000-1377, since they are both co-resident. If another FETCH is issued to load the RXA0 handler as an output device handler, that handler will not be loaded, because it shares space with the RXA1 handler currently in memory. This is fine; however, if the user now switches input devices and FETCHes the terminal handler, as an input device handler, it will destroy the RXA0 handler and the next attempt to output using the RXA0 handler will produce errors. You can get around this problem in two ways.
1. Always assign first the handler which you expect to stay in memory the longest. Although most programs can process more than one input file per program step (for example, an assembly pass is one program step), they can process only one output file; therefore, they assign the output handler before any of the input handlers. In the above example, the problem would be eliminated if the RXA1 handler were assigned first.

2. Always give a USR RESET call before each FETCH. Obviously, this call should not delete any open output files. This means that the USR will always load the new handler, even if another copy is in memory. The user must FETCH the output handler again before issuing the USR CLOSE call; otherwise, the USR will determine that the output handler is not in memory and give a MONITOR ERROR 3 message.
APPENDIX G

OS/78 ERROR MESSAGE SUMMARY

Error messages generated by OS/78 programs are listed in alphabetic order and identified by the system program by which they are generated. Also, system halts that occur as a result of errors are listed. This appendix is only a summary. Refer to the appropriate chapters for more detailed information about specific error conditions.

G.1 SYSTEM HALTS

Errors that occur as a result of a major I/O failure on the system device can cause a system halt. These are as follows.

<table>
<thead>
<tr>
<th>Value of PC</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>00601</td>
<td>A read error occurred while attempting to load ODT.</td>
</tr>
<tr>
<td>07461</td>
<td>An error occurred while reading a program into memory during a CHAIN.</td>
</tr>
<tr>
<td>07605</td>
<td>An error occurred while attempting to write the Monitor area onto the system scratch blocks.</td>
</tr>
<tr>
<td>07702</td>
<td>A user program has performed a JMS to 7700 in field 0. This is a result of trying to call the USR without first performing a CIF 10.</td>
</tr>
<tr>
<td>07764</td>
<td>A read error occurred while loading a program.</td>
</tr>
<tr>
<td>07772</td>
<td>A read error occurred on the system scratch area while loading a program.</td>
</tr>
<tr>
<td>10066</td>
<td>An input error occurred while attempting to restore the USR.</td>
</tr>
<tr>
<td>10256</td>
<td>A read error occurred while attempting to load the Monitor.</td>
</tr>
<tr>
<td>17676</td>
<td>An error occurred while attempting to read the Monitor from the system device.</td>
</tr>
<tr>
<td>17721</td>
<td>An error occurred while saving the USR area.</td>
</tr>
<tr>
<td>17727</td>
<td>An error occurred while attempting to read the USR from the system device.</td>
</tr>
<tr>
<td>17736</td>
<td>An error occurred while reading the scratch blocks to restore the USR area.</td>
</tr>
</tbody>
</table>
After bootstrapping and retrying the operation that caused the failure, if the error persists, it is the result of a hardware malfunction or a parity error in the system area.

### G.2 ERROR MESSAGES

<table>
<thead>
<tr>
<th>Message</th>
<th>Program</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>?0</td>
<td>SRCCOM</td>
<td>Insufficient memory -- this means that the differences between the files are too large to allow for effective comparison. Use of the /X option may alleviate this problem.</td>
</tr>
<tr>
<td>0</td>
<td>EDIT</td>
<td>Editor failed in reading a device. Error occurred in device handler; most likely a hardware malfunction.</td>
</tr>
<tr>
<td>?1</td>
<td>SRCCOM</td>
<td>Input error on file #1 or less than 2 input files specified.</td>
</tr>
<tr>
<td>1</td>
<td>EDIT</td>
<td>Editor failed in writing onto a device. Generally a hardware malfunction.</td>
</tr>
<tr>
<td>?2</td>
<td>SRCCOM</td>
<td>Input error on file #2.</td>
</tr>
<tr>
<td>2</td>
<td>EDIT</td>
<td>File close error occurred. The output file could not be closed; the file has been deleted.</td>
</tr>
<tr>
<td>2045 REPS</td>
<td>CREF</td>
<td>More than 2044 (decimal) references to one symbol were made.</td>
</tr>
<tr>
<td>?3</td>
<td>SRCCOM</td>
<td>Output file too large for output device. It has been deleted.</td>
</tr>
<tr>
<td>3</td>
<td>EDIT</td>
<td>File open error occurred. This error occurs if the output device is a read-only device or if no output file name is specified for a file-oriented output device.</td>
</tr>
<tr>
<td>?4</td>
<td>SRCCOM</td>
<td>Output I/O error.</td>
</tr>
<tr>
<td>4</td>
<td>EDIT</td>
<td>Device handler error occurred. The Editor could not load the device handler for the specified device. This error should never occur.</td>
</tr>
<tr>
<td>?5</td>
<td>SRCCOM</td>
<td>Could not create output file.</td>
</tr>
<tr>
<td>AA</td>
<td>P4</td>
<td>More than six subroutine arguments are arrays.</td>
</tr>
<tr>
<td>ALREADY EXISTS (filename)</td>
<td>FOTP</td>
<td>An attempt was made to rename an output file with the name of an existing output file.</td>
</tr>
<tr>
<td>Message</td>
<td>Program</td>
<td>Explanation</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>---------</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>AMBIGUOUS SWITCH</td>
<td>CCL</td>
<td>Not enough characters of the command full word option were specified to make it unique. (See Appendix H.)</td>
</tr>
<tr>
<td>ARE YOU SURE?</td>
<td>PIP</td>
<td>Occurs when using the SQUISH command. A response of Y will compress the files.</td>
</tr>
<tr>
<td>AS</td>
<td>F4</td>
<td>Bad ASSIGN statement.</td>
</tr>
<tr>
<td>BAD ARG</td>
<td>PRTS</td>
<td>Illegal argument to library function.</td>
</tr>
<tr>
<td>BAD ARGS</td>
<td>Monitor</td>
<td>The arguments to the SAVE command are not consistent and violate restrictions.</td>
</tr>
<tr>
<td>BAD COMMAND LINE</td>
<td>FORMAT</td>
<td>The command line contains a syntax error. Retype the line correctly.</td>
</tr>
<tr>
<td>BAD DATE</td>
<td>Monitor</td>
<td>The date has not been entered correctly, or incorrect arguments were used, or the date was out of range.</td>
</tr>
<tr>
<td>BAD DEVICE</td>
<td>CCL</td>
<td>The device specified in a system command is not of the correct form.</td>
</tr>
<tr>
<td>BAD DIRECTORY ON DEVICE #n</td>
<td>PIP</td>
<td>PIP is trying to read the directory, but it is not a legal OS/78 directory.</td>
</tr>
<tr>
<td>BAD DISK</td>
<td>FORMAT</td>
<td>Disk pack cannot be reformatted. There are more than 63 bad blocks on the system area (blocks 0–70 octal) contains bad blocks.</td>
</tr>
<tr>
<td>BAD EXTENSION</td>
<td>CCL</td>
<td>Either an extension was specified without a file name or two extensions were specified.</td>
</tr>
<tr>
<td>BAD INPUT DIRECTORY</td>
<td>DIRECT</td>
<td>The directory on the specified input device is not a valid OS/78 directory.</td>
</tr>
<tr>
<td></td>
<td>POTP</td>
<td></td>
</tr>
<tr>
<td>BAD INPUT FILE</td>
<td>LOAD</td>
<td>An input file was not a RALF module.</td>
</tr>
<tr>
<td>BAD INPUT, FILE #n</td>
<td>ABSLDR</td>
<td>Attempt was made to load a non-binary file as file number n of the input file list; or a non-memory image with /I option.</td>
</tr>
<tr>
<td>BAD INPUT, FILE #n</td>
<td>BITMAP</td>
<td>A physical end of file was reached before a logical end of file, or extraneous characters were found in binary file n.</td>
</tr>
<tr>
<td>Message</td>
<td>Program</td>
<td>Explanation</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>---------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>#BAD LINE. JOB ABORTED</td>
<td>BATCH</td>
<td>BATCH detected a record in the input file that did not have one of the characters period, slash, dollar sign, or asterisk as the first character of the record. The record is ignored, and BATCH scans the input file for the next $JOB record.</td>
</tr>
<tr>
<td>BAD NUMBER</td>
<td>CCL</td>
<td>The =n option was used, where n was not in the correct octal format.</td>
</tr>
<tr>
<td>BAD OUTPUT DEVICE</td>
<td>FOTP</td>
<td>This message usually appears when a non-file structured device is specified as the output device for a file-oriented operation.</td>
</tr>
<tr>
<td>BAD OUTPUT DEVICE</td>
<td>LOAD</td>
<td>The loader image file device was not a directory device, or the symbol map file device was a read-only device. The entire command is ignored.</td>
</tr>
<tr>
<td>BAD OUTPUT DIRECTORY</td>
<td>FOTP</td>
<td>The directory on the specified output device is not a valid OS/78 device directory.</td>
</tr>
<tr>
<td>BAD RECOLLECTION</td>
<td>CCL</td>
<td>An attempt was made to use a previously remembered argument after the system date had been changed.</td>
</tr>
<tr>
<td>BAD SYSTEM HEAD</td>
<td>PIP</td>
<td>Attempt to copy an obsolete version of the system head.</td>
</tr>
<tr>
<td>BAD SWITCH OPTION</td>
<td>CCL</td>
<td>The character used with a slash (/) to indicate an option is not a legal option.</td>
</tr>
<tr>
<td>BATCH.SV NOT FOUND ON SYS:</td>
<td>BATCH</td>
<td>A copy of BATCH.SV must exist on the system device. Control returns to the OS/78 Monitor.</td>
</tr>
<tr>
<td>%BATCH SQUISHING SYS:!</td>
<td>BATCH</td>
<td>Batch is running and attempting to squish the system.</td>
</tr>
<tr>
<td>BD</td>
<td>F4</td>
<td>Bad dimensions (too big, or syntax) in DIMENSION, COMMON or type declarations.</td>
</tr>
<tr>
<td>BE</td>
<td>RALF</td>
<td>Illegal equate. The symbol had been defined previously.</td>
</tr>
<tr>
<td>BE</td>
<td>PAL8</td>
<td>PAL8 internal table has overflowed. Fatal error; assembly cannot continue.</td>
</tr>
<tr>
<td>BI</td>
<td>RALF</td>
<td>Illegal index register specification.</td>
</tr>
<tr>
<td>BO</td>
<td>FRTS</td>
<td>No more file buffer available.</td>
</tr>
</tbody>
</table>
## OS/78 Error Message Summary

<table>
<thead>
<tr>
<th>Message</th>
<th>Program</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>BS</td>
<td>F4</td>
<td>Non-declarative statement used BLOCK DATA program.</td>
</tr>
<tr>
<td>BX</td>
<td>RALF</td>
<td>Bad expression. Something in the expression is incorrect, or the expression is not valid in this context.</td>
</tr>
<tr>
<td>CANNOT CHANGE MEMORY CAPACITY WHILE RUNNING BATCH</td>
<td>CCL</td>
<td>A MEMORY command was issued while BATCH was running.</td>
</tr>
<tr>
<td>?CAN'T-DEVICE DOESN'T EXIST</td>
<td>SET</td>
<td>A nonexistent device was referenced.</td>
</tr>
<tr>
<td>?CAN'T-DEVICE IS RESIDENT</td>
<td>SET</td>
<td>No modifications are allowed to the system handler.</td>
</tr>
<tr>
<td>?CAN'T-HANDLER ALREADY RESIDENT</td>
<td>SET</td>
<td>Attempt to replace a handler with one already in the system head.</td>
</tr>
<tr>
<td>?CAN'T-OBsolete HANDLER</td>
<td>SET</td>
<td>Obsolete version of the handler.</td>
</tr>
<tr>
<td>CANT-TOO MANY LOGICAL DEVICES</td>
<td>SET</td>
<td>The system does not allow you to have more than 12 logical devices (excluding SYS, DSK, and TTY). The handler that you are attempting to insert will cause the number of logical devices to exceed 12.</td>
</tr>
<tr>
<td>?CAN'T-UNKNOWN VERSION OF THIS HANDLER</td>
<td>SET</td>
<td>This handler's version number is not recognized; possibly a newer version.</td>
</tr>
<tr>
<td>CAN'T OPEN OUTPUT FILE</td>
<td>PIP</td>
<td>Message occurs due to one of the following:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1. Output file is on a read-only device.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2. No name has been specified for the output file.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3. Output file has zero free blocks.</td>
</tr>
<tr>
<td>CAN'T READ IT</td>
<td>FRTS</td>
<td>I/O error on reading loader image file.</td>
</tr>
<tr>
<td>!CAN'T REMEMBER</td>
<td>CCL</td>
<td>The argument specified in a CCL command line is too long to be remembered or an I/O error occurred.</td>
</tr>
<tr>
<td>CAUTION - THIS COMMAND DESTROYS CONTENTS OF SYSTEM DISK CHANGE DISKS BEFORE PROCEEDING OR CTRL/C.</td>
<td>RLFMT</td>
<td>Drive 0 was selected for formatting. Be sure to replace your system disk pack with the one that you want to format.</td>
</tr>
<tr>
<td>CF</td>
<td>PAL8</td>
<td>CREFS.V not on SYS:</td>
</tr>
<tr>
<td>Message</td>
<td>Program</td>
<td>Explanation</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>---------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>CH</td>
<td>PAL8</td>
<td>Chain to CREF error -- CREF.SV was not found on SYS:.</td>
</tr>
<tr>
<td>CHAIN ERROR</td>
<td>USR Monitor</td>
<td>Chain error</td>
</tr>
<tr>
<td>CL</td>
<td>F4</td>
<td>Bad COMPLEX literal.</td>
</tr>
<tr>
<td>CLOSE ERROR</td>
<td>USR Monitor</td>
<td>Close error</td>
</tr>
<tr>
<td>CLOSE FAILED</td>
<td>CREF</td>
<td>CLOSE on output file failed.</td>
</tr>
<tr>
<td>CO</td>
<td>F4</td>
<td>Syntax error in COMMON statement.</td>
</tr>
<tr>
<td>COMMAND LINE OVERFLOW</td>
<td>CCL</td>
<td>The command line specified with @ construction is more than 512 characters in length.</td>
</tr>
<tr>
<td>COMPARE ERROR</td>
<td>RXCOPY</td>
<td>A compare error has been detected at the track and sector specified.</td>
</tr>
<tr>
<td>CONTRADICTORY SWITCHES</td>
<td>CCL</td>
<td>Either two CCL processor switches were specified in the same command line or the file extension and the processor switch do not agree.</td>
</tr>
<tr>
<td>CORE IMAGE ERR</td>
<td>Monitor</td>
<td>Cannot run system program.</td>
</tr>
<tr>
<td>DA</td>
<td>F4</td>
<td>Bad syntax in DATA statement.</td>
</tr>
<tr>
<td>DE</td>
<td>F4</td>
<td>This type of statement illegal as end of DO loop.</td>
</tr>
<tr>
<td>DE</td>
<td>PAL8</td>
<td>Device error. An I/O failure was detected when trying to read or write a device. Fatal error-assembly cannot continue.</td>
</tr>
<tr>
<td>DELETES PERFORMED ONLY ON INPUT DEVICE GROUP 1 CAN'T HANDLE MULTIPLE DEVICE DELETES</td>
<td>FOTP</td>
<td>More than one input device was specified in the DELETE command when no output specification (device or filename) was included.</td>
</tr>
<tr>
<td>DEV LPT BAD</td>
<td>CREF</td>
<td>The default output device, LPT, cannot be used as it is not available on this system.</td>
</tr>
<tr>
<td>DEV NOT IMPLEMENTED</td>
<td>BATCH</td>
<td>BATCH cannot accept input from the specified input device because its handler is not SYS:. Control returns to the Command Decoder.</td>
</tr>
<tr>
<td>DEVICE DOES NOT HAVE A DIRECTORY</td>
<td>DIRECT</td>
<td>The input device is a non-directory device; for example, TTY. DIRECT can only read directories from file-structured devices.</td>
</tr>
<tr>
<td>Message</td>
<td>Program</td>
<td>Explanation</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>---------</td>
<td>-------------------------------------------------------------</td>
</tr>
<tr>
<td>DEVICE FULL</td>
<td>HELP</td>
<td>Output device storage capacity exhausted.</td>
</tr>
<tr>
<td>DEVICE HANDLER NOT IN CORE</td>
<td>USR Monitor</td>
<td>Handler for device specified not in memory.</td>
</tr>
<tr>
<td>DEVICE IS NOT RX</td>
<td>RXCOPY</td>
<td>One or more of the devices specified to RXCOPY were not RX diskettes.</td>
</tr>
<tr>
<td>DF</td>
<td>PAL8</td>
<td>Device full. Fatal error -- assembly cannot continue.</td>
</tr>
<tr>
<td>DP</td>
<td>F4</td>
<td>Bad DEFINE FILE statement.</td>
</tr>
<tr>
<td>D.F. TOO BIG</td>
<td>FRTS</td>
<td>Product of number of records times number of blocks per record exceeds number of blocks in file.</td>
</tr>
<tr>
<td>DH</td>
<td>F4</td>
<td>Hollerith field error in DATA statement.</td>
</tr>
<tr>
<td>DIRECTORY I/O ERROR</td>
<td>USR Monitor</td>
<td>An I/O error occurred while attempting to read or write a directory block.</td>
</tr>
<tr>
<td>DIRECTORY OVERFLOW</td>
<td>USR Monitor</td>
<td>Directory overflow occurred.</td>
</tr>
<tr>
<td>DIVIDE BY</td>
<td>FRTS</td>
<td>Attempt to divide by zero. The resulting quotient is set to zero and execution continues.</td>
</tr>
<tr>
<td>DL</td>
<td>F4</td>
<td>Data list and variable list are not same length.</td>
</tr>
<tr>
<td>DN</td>
<td>F4</td>
<td>DO-end missing or incorrectly nested. It is followed by the statement number of the erroneous statement rather than the ISN.</td>
</tr>
<tr>
<td>DO</td>
<td>F4</td>
<td>Syntax error in DO or implied DO.</td>
</tr>
<tr>
<td>name DOES NOT EXIST</td>
<td>CCL</td>
<td>The device with the name given is not present on the OS/78 system.</td>
</tr>
<tr>
<td>DP</td>
<td>F4</td>
<td>DO loop parameter not integer or real.</td>
</tr>
<tr>
<td>EG</td>
<td>RALF</td>
<td>The preceding line contains extra code which could not be used by the assembler.</td>
</tr>
<tr>
<td>ENTER FAILED</td>
<td>CREF</td>
<td>Entering an output file was unsuccessful -- possibly output was specified to a read-only device.</td>
</tr>
<tr>
<td>EOF ERROR</td>
<td>FRTS</td>
<td>End of file encountered on input.</td>
</tr>
<tr>
<td>EQUALS OPTION BAD</td>
<td>DIRECT</td>
<td>The =n option is not in the correct range.</td>
</tr>
<tr>
<td>Message</td>
<td>Program</td>
<td>Explanation</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>---------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>ERROR CLOSING FILE</td>
<td>DIRECT</td>
<td>Output device was read-only.</td>
</tr>
<tr>
<td>ERROR IN COMMAND</td>
<td>CCL</td>
<td>A command not entered directly from the console terminal is not a legal CCL command. This error occurs when the argument of a UA, UB, or UC command was not a legal command.</td>
</tr>
<tr>
<td>ERROR ON INPUT DEVICE SKIPPING (filename)</td>
<td>FOTP</td>
<td>The file specified is not transferred (due to a hardware I/O error), but any previous or subsequent files are transferred and indicated in the new directory. Any file of the same name may have been deleted from the output device.</td>
</tr>
<tr>
<td>ERROR ON OUTPUT DEVICE</td>
<td>BITMAP</td>
<td>A hardware I/O error occurred while writing on output device.</td>
</tr>
<tr>
<td>ERROR ON OUTPUT DEVICE SKIPPING (filename)</td>
<td>FOTP</td>
<td>The file specified is not transferred, but any previous or subsequent files are transferred and indicated in the new directory.</td>
</tr>
<tr>
<td>ERROR READING INPUT DIRECTORY</td>
<td>DIRECT</td>
<td>A hardware I/O error occurred while reading the directory.</td>
</tr>
<tr>
<td>ERROR WRITING FILE</td>
<td>DIRECT</td>
<td>A hardware I/O error occurred while writing the output file.</td>
</tr>
<tr>
<td>ERROR WRITING OUTPUT DIRECTORY</td>
<td>FOTP</td>
<td>A hardware I/O error has occurred. The output device has probably been corrupted (directory no longer describes current files).</td>
</tr>
<tr>
<td>ES</td>
<td>RALF</td>
<td>External symbol error.</td>
</tr>
<tr>
<td>EX</td>
<td>F4</td>
<td>Syntax error in EXTERNAL statement.</td>
</tr>
<tr>
<td>FETCH ERROR</td>
<td>HELP</td>
<td>Cannot initialize device handler.</td>
</tr>
<tr>
<td>FILE ERROR</td>
<td>FRTS</td>
<td>Any of the following:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>a. A file specified as an existing file was not found.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>b. A file specified as a non-existing file would not fit on the designated device.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>c. More than 1 nonexistent file was specified on a single device.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>d. File specification contained &quot;*&quot; as name or extension.</td>
</tr>
<tr>
<td>FILE OVERFLOW</td>
<td>FRTS</td>
<td>Attempt to write outside file boundaries.</td>
</tr>
<tr>
<td>Message</td>
<td>Program</td>
<td>Explanation</td>
</tr>
<tr>
<td>---------</td>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>FL</td>
<td>RALF</td>
<td>An error has occurred in an integer-to-real conversion routine.</td>
</tr>
<tr>
<td>FORMAT ERROR</td>
<td>FRTS</td>
<td>Illegal syntax in FORMAT statement.</td>
</tr>
<tr>
<td>FP</td>
<td>RALF</td>
<td>A syntax error was encountered in a real constant.</td>
</tr>
<tr>
<td>FULL *</td>
<td>EDIT</td>
<td>The specified output device has become full. The file is closed; the user must specify a new output file.</td>
</tr>
<tr>
<td>GT</td>
<td>F4</td>
<td>Syntax error in GO TO statement.</td>
</tr>
<tr>
<td>GV</td>
<td>F4</td>
<td>Assigned or computed GO TO variable must be integer or real.</td>
</tr>
<tr>
<td>HO</td>
<td>F4</td>
<td>Hollerith field error.</td>
</tr>
<tr>
<td>IC</td>
<td>RALF</td>
<td>The symbol or expression in a conditional is improperly used, or left angle bracket is missing. The conditional pseudo-op is ignored.</td>
</tr>
<tr>
<td>IC</td>
<td>PAL8</td>
<td>Illegal character. The character is ignored and the assembly continued.</td>
</tr>
<tr>
<td>ID</td>
<td>PAL8</td>
<td>Illegal redefinition of a symbol.</td>
</tr>
<tr>
<td>IE</td>
<td>F4</td>
<td>A hardware I/O error on reading input file. Control returns to the Monitor.</td>
</tr>
<tr>
<td>IE</td>
<td>PAL8</td>
<td>Illegal equals -- an attempt was made to equate a variable to an expression containing an undefined term. The variable remains undefined.</td>
</tr>
<tr>
<td>IE</td>
<td>RALF</td>
<td>An entry point has not been defined, or is absolute, or also is defined as a common section, or external.</td>
</tr>
<tr>
<td>IF</td>
<td>F4</td>
<td>Logical IF statement cannot be used with DO, DATA, INTEGER, etc.</td>
</tr>
<tr>
<td>II</td>
<td>PAL8</td>
<td>Illegal indirect -- an off-page reference was made; a link could not be generated because the indirect bit was already set.</td>
</tr>
<tr>
<td>ILLEGAL *</td>
<td>DIRECT</td>
<td>An asterisk (*) was included in the output file specification or an illegal * was included in the input file name.</td>
</tr>
</tbody>
</table>

G-9
<table>
<thead>
<tr>
<th>Message</th>
<th>Program</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ILLEGAL * OR ?</td>
<td>CCL</td>
<td>An * or ? was used in a CCL command that does not accept the wild card construction.</td>
</tr>
<tr>
<td>ILLEGAL ?</td>
<td>DIRECT</td>
<td>A question mark (?) was included in the output file specification.</td>
</tr>
<tr>
<td>ILLEGAL ARG</td>
<td>Monitor</td>
<td>The SAVE command was not expressed correctly; illegal syntax used.</td>
</tr>
<tr>
<td>#ILLEGAL INPUT</td>
<td>BATCH</td>
<td>A file specification designated TTY or LPT as an input device when the initial dialogue indicated that an operator is not available. The current job is aborted, and BATCH scans the input file for the next $JOB command record.</td>
</tr>
<tr>
<td>ILLEGAL ORIGIN</td>
<td>Loader</td>
<td>A RALF routine tried to store data outside the bounds of its overlay.</td>
</tr>
<tr>
<td>ILLEGAL SPOOL DEVICE</td>
<td>BATCH</td>
<td>The device specified as a spooling output device must be file-structured. Control returns to the Command Decoder.</td>
</tr>
<tr>
<td>ILLEGAL SYNTAX</td>
<td>CCL</td>
<td>The command line was formatted incorrectly.</td>
</tr>
<tr>
<td>?ILLEGAL WIDTH</td>
<td>SET</td>
<td>A width that was 0 or too large was specified; for the TTY, a width of 128 or one not a multiple of 8 was specified.</td>
</tr>
<tr>
<td>INCOMPATIBLE MONITOR - OS/78 V3.0 EXPECTED</td>
<td>SET</td>
<td>The system head contains an old version of the OS/78 Monitor. You cannot change handlers in a system head that has an old version of the Monitor.</td>
</tr>
<tr>
<td>INPUT DEVICE READ ERROR</td>
<td>RXCOPY</td>
<td>Bad sector(s), operation continues.</td>
</tr>
<tr>
<td>INPUT ERROR</td>
<td>CREF</td>
<td>A hardware I/O error occurred while reading the file.</td>
</tr>
<tr>
<td></td>
<td>FRTS</td>
<td>Illegal character received as input.</td>
</tr>
<tr>
<td>INPUT ERROR READING INDIRECT FILE</td>
<td>CCL</td>
<td>CCL cannot read the file specified with the @ construction due to hardware I/O error.</td>
</tr>
<tr>
<td>#INPUT FAILURE</td>
<td>BATCH</td>
<td>Either a hardware problem prevented BATCH from reading the next line of the input file, or BATCH read the last record of the input file without encountering a $END command record.</td>
</tr>
<tr>
<td>Message</td>
<td>Program</td>
<td>Explanation</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>---------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>INSUFFICIENT MEMORY FOR BATCH RUN</td>
<td>BATCH</td>
<td>OS/78 BATCH requires 12K of memory to run. Control returns to the OS/78 Monitor. Type MEMORY 3 and try again.</td>
</tr>
<tr>
<td>I/O</td>
<td>RALF</td>
<td>I/O error (fatal error).</td>
</tr>
<tr>
<td>I/O ERROR</td>
<td>FRTS</td>
<td>Error reading or writing a file, tried to read from an output device, or tried to write on an output device.</td>
</tr>
<tr>
<td>I/O ERROR, FILE #n</td>
<td>ABSLDR</td>
<td>An I/O error has occurred in input file number n.</td>
</tr>
<tr>
<td></td>
<td>BITMAP</td>
<td></td>
</tr>
<tr>
<td>IO ERROR IN (filename) -- CONTINUING</td>
<td>PIP</td>
<td>An error has occurred during a SQUISH transfer.</td>
</tr>
<tr>
<td>I/O ERROR ON SYS:</td>
<td>CCL</td>
<td>An error occurred while doing I/O to the system device. The system must be rebootstraped.</td>
</tr>
<tr>
<td>?I/O ERROR ON SYS:</td>
<td>SET</td>
<td>An I/O error occurred while trying to read or rewrite the handler.</td>
</tr>
<tr>
<td>I/O ERROR TRYING TO RECALL</td>
<td>CCL</td>
<td>An I/O error occurred while CCL was trying to remember an argument.</td>
</tr>
<tr>
<td>I/O READ ERROR</td>
<td>SET</td>
<td>A hardware error occurred while reading from the system device.</td>
</tr>
<tr>
<td>I/O WRITE ERROR</td>
<td>SET</td>
<td>A hardware error occurred while writing to the system device.</td>
</tr>
<tr>
<td>IP</td>
<td>PAL8</td>
<td>Illegal pseudo-op -- a pseudo-op was used in the wrong context or with incorrect syntax.</td>
</tr>
<tr>
<td>IX</td>
<td>RALF</td>
<td>An index register was specified for an instruction which cannot accept one.</td>
</tr>
<tr>
<td>IZ</td>
<td>PAL8</td>
<td>Illegal page zero reference -- The pseudo-op was found in an instruction which did not refer to page zero. The Z is ignored.</td>
</tr>
<tr>
<td>LD</td>
<td>PAL8</td>
<td>The /L or /G options have been specified and ABSLDR.SV is not present on the system.</td>
</tr>
<tr>
<td>LG</td>
<td>PAL8</td>
<td>Link Generated -- only printed if the /E switch was specified to PAL8.</td>
</tr>
<tr>
<td>LI</td>
<td>F4</td>
<td>Argument of logical IF is not of type Logical.</td>
</tr>
<tr>
<td>LOADER I/O ERROR</td>
<td>LOAD</td>
<td>Fatal error message indicating that an error was detected by OS/78 while trying to perform a USR function.</td>
</tr>
<tr>
<td>Message</td>
<td>Program</td>
<td>Explanation</td>
</tr>
<tr>
<td>-------------------------</td>
<td>---------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>LT</td>
<td>F4</td>
<td>Input line too long, too many continuations.</td>
</tr>
<tr>
<td>LT</td>
<td>RALP</td>
<td>The line is longer than 128 characters. The first 127 characters are assembled and listed.</td>
</tr>
<tr>
<td>#MANUAL HELP</td>
<td>BATCH</td>
<td>BATCH is attempting to operate an I/O device, such as a terminal, that will require operator intervention.</td>
</tr>
<tr>
<td>NEEDED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MD</td>
<td>RALP</td>
<td>The tag on the line has been previously encountered at another location or has been used in a context requiring an absolute expression.</td>
</tr>
<tr>
<td>MIXED INPUT</td>
<td>LOAD</td>
<td>The L option was specified on a line that contained some file other than a library file. The library file (if any) is accepted. Any other input file specification is ignored.</td>
</tr>
<tr>
<td>MK</td>
<td>F4</td>
<td>Misspelled keyword.</td>
</tr>
<tr>
<td>ML</td>
<td>F4</td>
<td>Multiply-defined line number.</td>
</tr>
<tr>
<td>MM</td>
<td>F4</td>
<td>Mismatched parenthesis.</td>
</tr>
<tr>
<td>MO</td>
<td>F4</td>
<td>Operand expected but not found.</td>
</tr>
<tr>
<td>MONITOR ERROR 5 AT xxxx (I/O ERROR ON SYS=)</td>
<td>Monitor</td>
<td>A hardware I/O error occurred while doing I/O to the system device.</td>
</tr>
<tr>
<td>MONITOR ERROR 6 AT xxxx (DIRECTORY OVERFLOW)</td>
<td>Monitor</td>
<td>A directory overflow has occurred (no room for tentative file entry in directory).</td>
</tr>
<tr>
<td>#MONITOR OVERLAYED</td>
<td>BATCH</td>
<td>The Command Decoder attempted to call the BATCH monitor to accept and transmit a file specification, but found that a user program had overlayed part of all of the BATCH monitor. Control returns to the monitor level, and BATCH executes the next Monitor command.</td>
</tr>
<tr>
<td>MORE CORE REQUIRED</td>
<td>PRTS</td>
<td>The space required for the program, the I/O device handlers (I/O buffers) and the resident Monitor exceeds the available memory.</td>
</tr>
<tr>
<td>NT</td>
<td>F4</td>
<td>Operand of mixed type or operator does not match operands.</td>
</tr>
<tr>
<td>Message</td>
<td>Program</td>
<td>Explanation</td>
</tr>
<tr>
<td>----------------------</td>
<td>-----------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>MULT SECT</td>
<td>Loader</td>
<td>Any combination of entry point, COMMON section (with the exception of multiple COMMONs) or program section of the same name causes this error.</td>
</tr>
<tr>
<td>NE</td>
<td>RALF</td>
<td>Number error. A number out of range was specified or an 8 or 9 occurred while in octal radix (internal error during FORTRAN assembly pass).</td>
</tr>
<tr>
<td>NO!!</td>
<td>Monitor</td>
<td>The user attempted to start (with .ST) a program that is no longer in memory.</td>
</tr>
<tr>
<td>NO CCL!!</td>
<td>Monitor</td>
<td>CCL.SV is not present on the system device or an I/O error occurred on reading it.</td>
</tr>
<tr>
<td>NO DEFINE FILE</td>
<td>FRTS</td>
<td>Direct access I/O attempted without a DEFINE FILE statement.</td>
</tr>
<tr>
<td>NO FILES OF THE FORM</td>
<td>FOTP</td>
<td>No files of the form (xxxx) specified were found.</td>
</tr>
<tr>
<td>xxxx</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NO HELP</td>
<td>HELP</td>
<td>No help information is present on the command given.</td>
</tr>
<tr>
<td>NO HELP FILE</td>
<td>HELP</td>
<td>No help file is on the system device.</td>
</tr>
<tr>
<td>NO/I</td>
<td>BITMAP</td>
<td>Cannot produce a bitmap of an image file.</td>
</tr>
<tr>
<td>NO/I!</td>
<td>ABSLDR</td>
<td>Use of /I is prohibited after the first file.</td>
</tr>
<tr>
<td>NO INPUT</td>
<td>ABSLDR</td>
<td>No input or binary file was found on the designated device.</td>
</tr>
<tr>
<td>NO INPUT DEVICE</td>
<td>BITMAP</td>
<td>The command line lacks the required parameters.</td>
</tr>
<tr>
<td>NO OUTPUT DEVICE</td>
<td>RXCOPY</td>
<td>The command line lacks the required parameters.</td>
</tr>
<tr>
<td>NO MAIN</td>
<td>LOAD</td>
<td>No RALF module contained section #MAIN.</td>
</tr>
<tr>
<td>NO NUMERIC SWITCH</td>
<td>FRTS</td>
<td>The referenced FORTRAN I/O unit was not specified to the run-time system.</td>
</tr>
<tr>
<td>NO ROOM FOR OUTPUT</td>
<td>DIRECT</td>
<td>Either room on device or room in directory is lacking.</td>
</tr>
<tr>
<td>FILE NAME</td>
<td>PIP</td>
<td>Occurs during the SQUISH command. The output device cannot contain all of the files on the input device.</td>
</tr>
<tr>
<td>NO ROOM IN (file</td>
<td>PIP</td>
<td></td>
</tr>
<tr>
<td>name) -CONTINUING</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Message</td>
<td>Program</td>
<td>Explanation</td>
</tr>
<tr>
<td>-------------------------------------</td>
<td>---------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>NO ROOM, SKIPPING (filename)</td>
<td>FOTP</td>
<td>No space is available on the output device to perform the transfer. Predeletion may already have occurred.</td>
</tr>
<tr>
<td>NOT A LOADER IMAGE</td>
<td>FRTS</td>
<td>The first input file specified to the run-time system was not a loader image file.</td>
</tr>
<tr>
<td>name NOT AVAILABLE</td>
<td>Monitor</td>
<td>The device with the name given is not listed in any system table, or it is not available for use at the moment, or the user tried to obtain input from an output-only device.</td>
</tr>
<tr>
<td>NOT ENOUGH MEMORY</td>
<td>CCL</td>
<td>The number specified in a MEMORY command is greater than is available in the system.</td>
</tr>
<tr>
<td>name NOT FOUND</td>
<td>CCL, Monitor</td>
<td>The file name designated in the command was not found in the device directory.</td>
</tr>
<tr>
<td>?NUMBER TOO BIG</td>
<td>SET</td>
<td>The number specified was out of range.</td>
</tr>
<tr>
<td>OF</td>
<td>F4</td>
<td>I/O error while writing output file. Control returns to the Monitor.</td>
</tr>
<tr>
<td>OLD HANDLER NOT FOUND IN MONITOR</td>
<td>SET</td>
<td>You attempted to replace a handler that is not in the system head.</td>
</tr>
<tr>
<td>OP</td>
<td>F4</td>
<td>Illegal operator.</td>
</tr>
<tr>
<td>OPTION UNKNOWN XXXXX</td>
<td>CCL</td>
<td>Fatal error message indicating that an error was detected by OS/78 while trying to perform a USR function.</td>
</tr>
<tr>
<td>OS78 ENTER ERROR</td>
<td>LOAD</td>
<td>Fatal error message indicating that an error was detected by OS/78 while trying to perform a USR function.</td>
</tr>
<tr>
<td>OT</td>
<td>F4</td>
<td>Type/operator use illegal (for example, A.AND.B where A and/or B not of type Logical).</td>
</tr>
<tr>
<td>OUT DEV FULL</td>
<td>CREF</td>
<td>The output device is full (directory devices only).</td>
</tr>
<tr>
<td>OUTPUT DEVICE READ ERROR</td>
<td>RXCOPY</td>
<td>Bad sector(s), operation continues.</td>
</tr>
<tr>
<td>OUTPUT DEVICE WRITE ERROR</td>
<td>RXCOPY</td>
<td>Bad sector(s), operation continues.</td>
</tr>
<tr>
<td>OVER CORE</td>
<td>LOAD</td>
<td>The loader image requires more memory than is available.</td>
</tr>
<tr>
<td>OVER IMAG</td>
<td>LOAD</td>
<td>Output file overflow in the loader image file.</td>
</tr>
<tr>
<td>Message</td>
<td>Program</td>
<td>Explanation</td>
</tr>
<tr>
<td>----------------------</td>
<td>---------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>OVER SYMB</td>
<td>LOAD</td>
<td>Symbol table overflow. More than 253 (decimal) symbols in one FORTRAN job.</td>
</tr>
<tr>
<td>OVERFLOW</td>
<td>FRTS</td>
<td>Result of a computation exceeds upper bound for that class of variable. The result is set equal to zero and execution continues.</td>
</tr>
<tr>
<td>PARENS TOO DEEP</td>
<td>FRTS</td>
<td>Parentheses nested too deeply in FORMAT statement.</td>
</tr>
<tr>
<td>PD</td>
<td>F4</td>
<td>Compiler stack overflow; statement too big and/or too many nested loops.</td>
</tr>
<tr>
<td>PE</td>
<td>PAL8</td>
<td>Current non-zero page exceeded.</td>
</tr>
<tr>
<td>PH</td>
<td>F4</td>
<td>Bad program header line.</td>
</tr>
<tr>
<td>PH</td>
<td>PAL8</td>
<td>A conditional assembly bracket is still in effect at the end of the input stream -- this is caused by nonmatching &lt; and &gt; characters in the source.</td>
</tr>
<tr>
<td>QL</td>
<td>F4</td>
<td>Nesting error in EQUIVALENCE statement.</td>
</tr>
<tr>
<td>QS</td>
<td>F4</td>
<td>Syntax error in EQUIVALENCE statement.</td>
</tr>
<tr>
<td>RD</td>
<td>F4</td>
<td>Attempt to redefine the dimensions of an array.</td>
</tr>
<tr>
<td>RD</td>
<td>PAL8</td>
<td>A permanent symbol has been redefined using =. The new and old definitions do not match. The redefinition is allowed.</td>
</tr>
<tr>
<td>RE</td>
<td>RALF</td>
<td>Relocatability error. A relocatable expression has been used in a context requiring an absolute expression.</td>
</tr>
<tr>
<td>READ ERR</td>
<td>HELP</td>
<td>Cannot read input device.</td>
</tr>
<tr>
<td>READY, STRIKE</td>
<td>RLFMT</td>
<td>The /P option was selected. Mount the disk pack to be formatted and type a carriage return to begin formatting.</td>
</tr>
<tr>
<td>CARRIAGE RETURN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TO CONTINUE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RT</td>
<td>F4</td>
<td>Attempt to redefine the type of a variable.</td>
</tr>
<tr>
<td>RW</td>
<td>F4</td>
<td>Syntax error on READ/WRITE statement.</td>
</tr>
<tr>
<td>SAVE ERROR</td>
<td>Monitor</td>
<td>An I/O error has occurred while saving the program. The program remains intact in memory.</td>
</tr>
<tr>
<td>Message</td>
<td>Program</td>
<td>Explanation</td>
</tr>
<tr>
<td>-------------------------</td>
<td>---------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>SE</td>
<td>PAL8</td>
<td>Symbol table exceeded -- too many symbols have been defined for the amount of memory available for the symbol table. Fatal error -- assembly cannot continue.</td>
</tr>
<tr>
<td>SF</td>
<td>F4</td>
<td>Bad arithmetic statement function.</td>
</tr>
<tr>
<td>SN</td>
<td>F4</td>
<td>Illegal subroutine name in CALL.</td>
</tr>
<tr>
<td>SORRY -- NO INTERRUPTIONS</td>
<td>PIP</td>
<td>&quot;C (CTRL/C) was typed during a SQUIP; the transfer continues.</td>
</tr>
<tr>
<td>#SPOOL TO FILE BTHCA1</td>
<td>BATCH</td>
<td>Where the &quot;A&quot; may be any character of the alphabet and the &quot;1&quot; may be any decimal digit. This message indicates that BATCH has intercepted a non-file structured output file and routed it to the spool device. This is not, generally, an error condition.</td>
</tr>
<tr>
<td>SS</td>
<td>F4</td>
<td>Error in subscript expression; i.e., wrong number, syntax.</td>
</tr>
<tr>
<td>ST</td>
<td>F4</td>
<td>Compiler symbol table full, program too big. Causes an immediate return to the Keyboard Monitor.</td>
</tr>
<tr>
<td>ST</td>
<td>RALF</td>
<td>User symbol table overflow (fatal error).</td>
</tr>
<tr>
<td>SWITCH NOT ALLOWED HERE</td>
<td>CCL</td>
<td>Either a CCL option was specified on the left side of the &lt; or was used when not allowed.</td>
</tr>
<tr>
<td>SY</td>
<td>F4</td>
<td>System error; i.e., PASS20.SV or PASS2.SV missing, or no room for output file. Causes an immediate return to the Keyboard Monitor.</td>
</tr>
<tr>
<td>SYM OVERFLOW</td>
<td>CREF</td>
<td>More than 896 (decimal) symbols and literals were encountered. Try again using /M option.</td>
</tr>
<tr>
<td>?SYNTAX ERROR</td>
<td>SET</td>
<td>Incorrect format used in SET command or NO specified when not allowed.</td>
</tr>
<tr>
<td>#SYS ERROR</td>
<td>BATCH</td>
<td>A hardware I/O error occurred during BATCH operation.</td>
</tr>
<tr>
<td>SYSTEM DEVICE ERROR</td>
<td>FRTS</td>
<td>I/O failure on the system device.</td>
</tr>
<tr>
<td>SYSTEM ERR</td>
<td>Monitor</td>
<td>An error occurred while doing I/O to the system device. The system should be rebootstrapped.</td>
</tr>
<tr>
<td>Message</td>
<td>Program</td>
<td>Explanation</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>---------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>SYSTEM ERROR</td>
<td>LOAD</td>
<td>Fatal error message indicating that an error was detected by OS/78 while trying to perform a USR function.</td>
</tr>
<tr>
<td>SYSTEM ERROR -- CLOSING FILE</td>
<td>FOTP</td>
<td>Self-explanatory.</td>
</tr>
<tr>
<td>TD</td>
<td>F4</td>
<td>Bad syntax in type declaration statement.</td>
</tr>
<tr>
<td>TOO FEW ARGS</td>
<td>Monitor</td>
<td>An important argument has been omitted from a command.</td>
</tr>
<tr>
<td>TOO MANY FILES</td>
<td>CCL</td>
<td>Too many files were included in a CCL command.</td>
</tr>
<tr>
<td>TOO MANY HANDLERS</td>
<td>FRTS</td>
<td>Too many I/O device handlers are resident in memory, or files have been defined on too many devices.</td>
</tr>
<tr>
<td>TOO MANY RALF FILES</td>
<td>LOAD</td>
<td>More than 128 input files were specified.</td>
</tr>
<tr>
<td>?UNKNOWN ATTRIBUTE FOR DEVICE dev</td>
<td>SET</td>
<td>An illegal attribute was specified for the given device.</td>
</tr>
<tr>
<td>UNIT ERROR</td>
<td>FRTS</td>
<td>I/O unit not assigned, or incapable of executing the requested operation.</td>
</tr>
<tr>
<td>UO</td>
<td>PAL8</td>
<td>Undefined origin -- an undefined symbol has occurred in an origin statement.</td>
</tr>
<tr>
<td>US</td>
<td>RALF</td>
<td>Undefined symbol in an expression.</td>
</tr>
<tr>
<td>USER ERROR</td>
<td>FRTS</td>
<td>Illegal subroutine call, or call to undefined subroutine. Execution continues only if the E option was specified.</td>
</tr>
<tr>
<td>USER ERROR 0 AT xxxx</td>
<td>Monitor</td>
<td>An input error was detected while loading the program. xxxx refers to the Monitor location where the error was generated.</td>
</tr>
<tr>
<td>VE</td>
<td>F4</td>
<td>Version error. One of the compiler programs is absent from SYS.</td>
</tr>
<tr>
<td>WRITE ERR</td>
<td>HELP</td>
<td>Cannot output to device.</td>
</tr>
<tr>
<td>WRONG FLOPPY TYPE</td>
<td>RXCOPY</td>
<td>Attempt to duplicate from an RX02 drive with a double-density diskette to an RX01 single-density diskette drive.</td>
</tr>
<tr>
<td>Message</td>
<td>Program</td>
<td>Explanation</td>
</tr>
<tr>
<td>------------------</td>
<td>---------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>xxxxxx.HN NOT FOUND ON SYS</td>
<td>SET</td>
<td>The handler that you want to insert into the system head does not reside on the system device (SYS), or does not have a .HN extension.</td>
</tr>
<tr>
<td>ZE</td>
<td>PAL8</td>
<td>Page 0 exceeded -- same as PE except with reference to page 0.</td>
</tr>
<tr>
<td>ZERO SYS?</td>
<td>PIP</td>
<td>If any attempt is made to zero the system device directory, this message occurs. Responding with Y causes the directory to be zeroed; any other character prevents destruction of the system directory.</td>
</tr>
</tbody>
</table>
APPENDIX H

OS/78 MULTIFUNCTION OPERATION

DECstation 78 series systems can perform multifunction operations; that is, these systems can run two tasks simultaneously. One task is the normal OS/78 job stream, that runs in the first 12K of memory. The second task (the symbiont) runs in the upper 4K of memory (field 3). The symbiont is an interrupt-driven task that you write in PAL8 assembly language. OS/78 and the symbiont share a common floppy disk and line printer.

BASIC, FORTRAN and PAL8 programs can be compiled, but only BASIC and PAL8 programs can run while a symbiont is active. You cannot run FORTRAN programs because the FORTRAN run time system performs interrupt-driven I/O, which would interfere with symbiont operation.

A demonstration symbiont, called spoolr comes with OS/78. You can use the SPOOLR to queue and print listings on the line printer while you do other OS/78 work at the terminal.

H.1 SYMBIONT COMMANDS

OS/78 commands that control symbiont operation are as follows:

REQUEST symbiont-name

This command automatically sets memory for OS/78 to 12K (MEM 2) and starts up the named symbiont. The default extension for a symbiont save image is .SM; if .SM is not found, the auxiliary default extension is .SV.

CANCEL

This command cancels the currently running symbiont and returns OS/78 to operation in 16K memory.

CAUTION

Do not use the MEM command while a symbiont is running.
H.2 SPOOLER COMMANDS

Type the following command to start SPOOLR.

.REQ SPOOLR

This command starts the spooler.

You can use the following commands once the SPOOLR program is running.

.QUEUE filename list, .../options,...

Enter the specified lists of files for printing on the line printer. Options for the QUEUE command are:

/L Lists contents of queue and prints version number of QUEUE and SPOOLR
/C:n Specifies the number (n) of copies to be printed (in octal)
/N A block letter header will not be added to the files queued
/2 Prints two pages of block letter header rather than one.

H.3 CUSP CODE CONVENTION

You should observe the following conventions when writing OS/78 programs if you want to run them while a symbiont is running.

1. If the program loads into page 0 of field 0, then it must contain the following code:

*1
CIF 30
JMP .-1

Also, locations 0, 1, and 2 must not be used as scratch or as data.

2. The program must not use page 0 of field 0 as a buffer or data area. However, you may swap the OS/8 command decoder in this area.

3. The program must not turn interrupts on or off.

4. The program must not modify the software memory size.

5. The program must not use field 3.
H.4 WRITING A SYMBIONT

To write your own symbiont, you must know the PALB assembly language and the PDP-8 architecture. You must observe the following conventions when you write a symbiont.

1. The symbiont runs only in field 3.

2. The symbiont must turn on interrupts and must disable keyboard interrupts (KIE with AC=1).

3. The CANCEL command branches to the symbiont at location 30003.

4. The symbiont will receive interrupts at location 30001. After saving the AC, Link, etc., the symbiont should pick up location 0 in field 0 and save it in location 0 of field 3.
APPENDIX I

FULL WORD COMMAND SWITCHES

OS/78 allows you to enter complete words in place of most of the standard single-character command options. For example, this command

```
.DIR/BRIEF
```

is the same as

```
.DIR/F
```

This appendix lists the switch options and the full word equivalents that you can use in OS/78 commands. Keep in mind the following rules and features.

- The standard single-character slash options remain legal and are the only single-character switches you should attempt to use in a command.

- Only the first six characters of a full word are significant.

- Every character in a full word switch must be legal. For example, /QUIEX is an illegal switch and will produce the message UNKNOWN SWITCH.

- You may abbreviate any switch option to the number of characters (at least two) necessary to determine the switch uniquely for the command in question. For example, in the DIRECT command, /EXT is a valid abbreviation for the /EXTENDED switch. /EX, however, is not valid because the switch /EXCEPT also begins with these two characters.

- If you fail to type enough characters to identify a switch uniquely, you will receive an AMBIGUOUS SWITCH message.

- You may still enclose single-character switches in parentheses. (ABC) in a command line means the same as A/B/C. However, you may not enter full word switches in this manner.

- :n is the same as the standard =n and may optionally follow any full word switch. Thus /IMAGE:24 is the same as /I=24.
<table>
<thead>
<tr>
<th>Command</th>
<th>Single-Character Switch</th>
<th>Full Word Switch</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMPARE</td>
<td>/B</td>
<td>/BLANKS</td>
</tr>
<tr>
<td></td>
<td>/C</td>
<td>/NOCOMMENTS</td>
</tr>
<tr>
<td></td>
<td>/S</td>
<td>/NO SPACES</td>
</tr>
<tr>
<td></td>
<td>/T</td>
<td>/TABS</td>
</tr>
<tr>
<td></td>
<td>/X</td>
<td>/NOPRINTCOMMENTS</td>
</tr>
<tr>
<td>COMPILE</td>
<td>/G</td>
<td>/GO</td>
</tr>
<tr>
<td>(see PAL and BASIC)</td>
<td>/N</td>
<td>/NOISN</td>
</tr>
<tr>
<td></td>
<td>/Q</td>
<td>/OPTIMIZE</td>
</tr>
<tr>
<td>COPY</td>
<td>/C</td>
<td>/CURRENT</td>
</tr>
<tr>
<td></td>
<td>/D</td>
<td>/NOCOPY</td>
</tr>
<tr>
<td></td>
<td>/F</td>
<td>/FAILSAFE</td>
</tr>
<tr>
<td></td>
<td>/N</td>
<td>/NOPERDELETE</td>
</tr>
<tr>
<td></td>
<td>/O</td>
<td>/OTHER</td>
</tr>
<tr>
<td></td>
<td>/Q</td>
<td>/QUERY</td>
</tr>
<tr>
<td></td>
<td>/R</td>
<td>/RENAME</td>
</tr>
<tr>
<td></td>
<td>/T</td>
<td>/TODAY</td>
</tr>
<tr>
<td></td>
<td>/U</td>
<td>/INDEPENDENTLY</td>
</tr>
<tr>
<td></td>
<td>/V</td>
<td>/EXCEPT</td>
</tr>
<tr>
<td>CREATE</td>
<td>/B</td>
<td>/SPACES</td>
</tr>
<tr>
<td>CREF</td>
<td>/E</td>
<td>KEEP</td>
</tr>
<tr>
<td></td>
<td>/M</td>
<td>MAMMOTH</td>
</tr>
<tr>
<td></td>
<td>/P</td>
<td>NOLIST</td>
</tr>
<tr>
<td></td>
<td>/U</td>
<td>NOSYNTAB</td>
</tr>
<tr>
<td></td>
<td>/X</td>
<td>NOLITERALS</td>
</tr>
<tr>
<td>DELETE</td>
<td>/C</td>
<td>/CURRENT</td>
</tr>
<tr>
<td></td>
<td>/O</td>
<td>/OTHER</td>
</tr>
<tr>
<td></td>
<td>/Q</td>
<td>/QUERY</td>
</tr>
<tr>
<td></td>
<td>/T</td>
<td>/TODAY</td>
</tr>
<tr>
<td></td>
<td>/U</td>
<td>/INDEPENDENTLY</td>
</tr>
<tr>
<td></td>
<td>/V</td>
<td>/EXCEPT</td>
</tr>
<tr>
<td>DIRECT</td>
<td>/B</td>
<td>/BLOCKS</td>
</tr>
<tr>
<td></td>
<td>/C</td>
<td>/CURRENT</td>
</tr>
<tr>
<td></td>
<td>/E</td>
<td>/EXTENDED</td>
</tr>
<tr>
<td></td>
<td>/F</td>
<td>/FAST</td>
</tr>
<tr>
<td></td>
<td>/M</td>
<td>/EMPTY</td>
</tr>
<tr>
<td></td>
<td>/O</td>
<td>/OTHER</td>
</tr>
<tr>
<td></td>
<td>/R</td>
<td>/REMAINDER</td>
</tr>
<tr>
<td></td>
<td>/U</td>
<td>/INDEPENDENTLY</td>
</tr>
<tr>
<td></td>
<td>/V</td>
<td>/EXCEPT</td>
</tr>
<tr>
<td></td>
<td>=n</td>
<td>/COLUMNS:n</td>
</tr>
<tr>
<td>DUPLICATE</td>
<td>/M</td>
<td>/NOCOPY</td>
</tr>
<tr>
<td></td>
<td>/N</td>
<td>/NOMATCH</td>
</tr>
<tr>
<td></td>
<td>/P</td>
<td>/PAUSE</td>
</tr>
<tr>
<td></td>
<td>/R</td>
<td>/READYONLY</td>
</tr>
<tr>
<td>EDIT</td>
<td>/B</td>
<td>/SPACES</td>
</tr>
<tr>
<td></td>
<td>/D</td>
<td>/DELETE</td>
</tr>
</tbody>
</table>

I-2
<table>
<thead>
<tr>
<th>Command</th>
<th>Single-Character Switch</th>
<th>Full Word Switch</th>
</tr>
</thead>
<tbody>
<tr>
<td>FORMAT RL01</td>
<td>/O</td>
<td>/ZERO</td>
</tr>
<tr>
<td></td>
<td>/1</td>
<td>/ONE</td>
</tr>
<tr>
<td></td>
<td>/P</td>
<td>/PAUSE</td>
</tr>
<tr>
<td>LIST</td>
<td>/C</td>
<td>/CURRENT</td>
</tr>
<tr>
<td></td>
<td>/O</td>
<td>/OTHER</td>
</tr>
<tr>
<td></td>
<td>/Q</td>
<td>/QUERY</td>
</tr>
<tr>
<td></td>
<td>/U</td>
<td>/INDEPENDENTLY</td>
</tr>
<tr>
<td></td>
<td>/V</td>
<td>/EXCEPT</td>
</tr>
<tr>
<td>LOAD</td>
<td>/G</td>
<td>/GO</td>
</tr>
<tr>
<td></td>
<td>/I</td>
<td>/IMAGE</td>
</tr>
<tr>
<td></td>
<td>/S</td>
<td>/MULTIPLE</td>
</tr>
<tr>
<td>MAP</td>
<td>/S</td>
<td>/MULTIPLE</td>
</tr>
<tr>
<td></td>
<td>/T</td>
<td>/INVERT</td>
</tr>
<tr>
<td>PAL</td>
<td>/B</td>
<td>/SHIFT</td>
</tr>
<tr>
<td></td>
<td>/C</td>
<td>/CREF</td>
</tr>
<tr>
<td></td>
<td>/E</td>
<td>/NOLINKS</td>
</tr>
<tr>
<td></td>
<td>/F</td>
<td>/NOFILL</td>
</tr>
<tr>
<td></td>
<td>/G</td>
<td>/GO</td>
</tr>
<tr>
<td></td>
<td>/H</td>
<td>/NONPAGINATED</td>
</tr>
<tr>
<td></td>
<td>/J</td>
<td>/NOCONDITIONALs</td>
</tr>
<tr>
<td></td>
<td>/L</td>
<td>/LOAD</td>
</tr>
<tr>
<td></td>
<td>/N</td>
<td>/NOLIST</td>
</tr>
<tr>
<td></td>
<td>/O</td>
<td>/NOORIG</td>
</tr>
<tr>
<td></td>
<td>/S</td>
<td>/NOSYMTAB</td>
</tr>
<tr>
<td></td>
<td>/W</td>
<td>/NOREMEMBERLITERALS</td>
</tr>
<tr>
<td>QUEUE</td>
<td>/C=n</td>
<td>/COPIES:n</td>
</tr>
<tr>
<td></td>
<td>/L</td>
<td>/LIST</td>
</tr>
<tr>
<td></td>
<td>/N</td>
<td>/NOH</td>
</tr>
<tr>
<td>RENAME</td>
<td>/C</td>
<td>/CURRENT</td>
</tr>
<tr>
<td></td>
<td>/O</td>
<td>/OTHER</td>
</tr>
<tr>
<td></td>
<td>/Q</td>
<td>/QUERY</td>
</tr>
<tr>
<td></td>
<td>/T</td>
<td>/TODAY</td>
</tr>
<tr>
<td></td>
<td>/U</td>
<td>/INDEPENDENTLY</td>
</tr>
<tr>
<td></td>
<td>/V</td>
<td>/EXCEPT</td>
</tr>
<tr>
<td></td>
<td>/W</td>
<td>/VERSION</td>
</tr>
<tr>
<td>SQUIrish</td>
<td>/O</td>
<td>/OK</td>
</tr>
<tr>
<td>SUBMIT</td>
<td>/E</td>
<td>/NONFATAL</td>
</tr>
<tr>
<td></td>
<td>/H</td>
<td>/HUSH</td>
</tr>
<tr>
<td></td>
<td>/Q</td>
<td>/QUIET</td>
</tr>
<tr>
<td></td>
<td>/T</td>
<td>/TERMINAL</td>
</tr>
<tr>
<td></td>
<td>/U</td>
<td>/UNATTENDED</td>
</tr>
<tr>
<td></td>
<td>/V</td>
<td>/VERSION</td>
</tr>
<tr>
<td>TYPE</td>
<td>/C</td>
<td>/CURRENT</td>
</tr>
<tr>
<td></td>
<td>/O</td>
<td>/OTHER</td>
</tr>
<tr>
<td></td>
<td>/Q</td>
<td>/QUERY</td>
</tr>
<tr>
<td></td>
<td>/T</td>
<td>/TODAY</td>
</tr>
<tr>
<td></td>
<td>/U</td>
<td>/INDEPENDENTLY</td>
</tr>
<tr>
<td></td>
<td>/V</td>
<td>/EXCEPT</td>
</tr>
</tbody>
</table>
### APPENDIX J

**DECstation Hardware Configuration Summary**

<table>
<thead>
<tr>
<th>MODEL</th>
<th>CPU</th>
<th>Console Terminal</th>
<th>Disk System</th>
<th>Line Printer</th>
<th>Serial I/O Ports</th>
</tr>
</thead>
<tbody>
<tr>
<td>78/40</td>
<td>VT78 (16K)</td>
<td>VT78</td>
<td>1 RX01</td>
<td>LA78 or LQP78</td>
<td>2</td>
</tr>
<tr>
<td>78/50</td>
<td>VT78 (16K)</td>
<td>VT78</td>
<td>1 RX02</td>
<td>LA78 or LQP78</td>
<td>2</td>
</tr>
<tr>
<td>78/60</td>
<td>VT78 (16K)</td>
<td>VT78</td>
<td>2 RX01</td>
<td>LA78 or LQP78</td>
<td>2</td>
</tr>
<tr>
<td>78/70</td>
<td>VT78 (16K)</td>
<td>VT78</td>
<td>2 RX02</td>
<td>LA78 or LQP78</td>
<td>2</td>
</tr>
<tr>
<td>88/50</td>
<td>8A205 (32K)</td>
<td>VT100 or LA36</td>
<td>1 RX02</td>
<td>LA180 or LQP78</td>
<td>2 (optional)</td>
</tr>
<tr>
<td>88/70</td>
<td>8A205 (32K)</td>
<td>VT100 or LA36</td>
<td>2 RX02</td>
<td>LA180</td>
<td>2 (optional)</td>
</tr>
<tr>
<td>88/80</td>
<td>8A205 (32K)</td>
<td>VT100 or LA36</td>
<td>2 RX02</td>
<td>LA180</td>
<td>2 (optional)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1 RL01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>88/90</td>
<td>8A205 (32K)</td>
<td>VT100 or LA36</td>
<td>2 RL01</td>
<td>LA180</td>
<td>2 (optional)</td>
</tr>
<tr>
<td>88/92</td>
<td>8A425 (32K)</td>
<td>VT100 or LA36</td>
<td>2 RL01</td>
<td>LA180</td>
<td>2 (optional)</td>
</tr>
<tr>
<td>88/97</td>
<td>8A425 (64K)</td>
<td>VT100 or LA36</td>
<td>2 RL01</td>
<td>LA180</td>
<td>2 (optional)</td>
</tr>
</tbody>
</table>

* Can be connected to LA34/38, LA36, LA120, VT52, VT100 or equivalent.

J-1
GLOSSARY

ABSOLUTE ADDRESS
A number that is permanently assigned as the address of a memory storage location.

ACCESS TIME
The interval between the instant at which a data transfer is requested and the instant at which the data actually starts transferring.

ACCUMULATOR
The register in which the hardware arithmetic operations are performed (abbreviated AC).

ADDRESS
1. A name or a number which identifies a location in memory, either within a field (12-bits wide) or within all available memory (15-bits wide with left-most bit 0).
2. The part of an instruction that specifies the location of the operand of that instruction.

ALGORITHM
A prescribed sequence of well-defined rules or processes for the solution of a problem.

ALPHANUMERIC
Pertaining to the character set that contains only letters and numbers.

ARGUMENT
A variable or constant which is given in the call of a subroutine as information to it; the independent variables of a function; the known reference factor necessary to find an item in a table or array (that is, the index).

ARRAY
An ordered set of data values. An n-dimensional array is a table having n dimensions.

ASCII
American Standard Code for Information Interchange. Established by American Standards Association to standardize a binary code for printing and control characters.

ASSEMBLE
To translate from a source program to a binary program by substituting binary operation codes for mnemonic operation codes and absolute or relocatable addresses for symbolic addresses.

Glossary-1
GLOSSARY

ASSEMBLER
A program which translates assembly language instructions into machine language and assigns memory locations for variables and constants.

ASSEMBLY LANGUAGE
A symbolic language that translates directly into machine language instructions. Usually there is a one to-one relation between assembly language instructions and machine language instructions.

AUTO-INDEXING
A property of the autoindex registers (locations 0010 through 0017. When one of these locations is addressed indirectly, the contents of that location are incremented by one, rewritten into that same location and then used as the effective address of the current instruction.

AUXILIARY STORAGE
Storage that supplements memory such as a diskettes and disk packs.

BASIC
A high-level easy to learn programming language for arithmetic and string computations Developed by Dartmouth College.

BATCH PROCESSING
A method of using OS/78 with no operator interaction.

BINARY
Pertaining to the number system with a base (or radix) of two. In this system numbers are represented by strings of 1's and 0's.

BINARY CODE
A code that makes use of two distinct values: 0 and 1.

BIT
Contraction of "Binary digit", a bit is the smallest unit of information in the binary system of notation.

BIT MAP
A method of keeping track of used and unused entities by assigning one bit in a table to each entity.

BLOCK
A set of consecutive machine words, characters or digits handled as a unit, particularly with reference to input and output; an OS/78 block is 400 octal contiguous words (two memory pages).

BOOTSTRAP
A program of instructions that are executed when the START pushbutton is pressed or you use the BOOT command. The purpose of a bootstrap is to load and start the OS/78 Monitor.

BREAKPOINT
A location in a program at which that program's execution may be suspended, so that partial results can be examined via ODT.

BUFFER
An area that is usually used for temporary storage. Buffers are often used to hold data being passed between processes or devices which operate at different speeds or different times.

Glossary-2
GLOSSARY

BUG
A mistake in the design or implementation of a program resulting in erroneous results.

BYTE
A group of binary digits usually operated upon as a unit, especially one of six bits.

CALL
To transfer control to a specified routine; to invoke a system command.

CALLING SEQUENCE
A specified arrangement of instructions and data necessary to pass parameters and control to a given subroutine.

CARTRIDGE DISK PACK
A removable data storage medium consisting of a thin disk coated with a magnetic recording material and enclosed in a two-piece protective cover.

CCL (CONCISE COMMAND LANGUAGE)
A system program that simplifies the entry of OS/78 commands by calling the selected system program and decoding its command string.

CHAINING
A program technique which involves dividing a program into sections with each section terminated by a call to the next section.

CHARACTER
A single letter, number or symbol (printing or non-printing) used to represent information.

CLEAR
To erase or reset the contents of a memory location or hardware register.

CLOCK
A hardware device that generates periodic program interrupts when enabled.

COMPILE
To translate a source program written in a high-level language, such as BASIC or FORTRAN, into a binary-coded program. In addition to translating the source language, appropriate subroutines may be selected from a subroutine library and output in binary code along with the main program.

COMPILER
A program which compiles entire high-level language source programs into binary coded programs.

CONCATENATION
The adjacent joining of two strings of characters to produce a longer string.

CODE
To write instructions for a computer using symbols meaningful to the computer or to an assembler, compiler, or other language processor.
GLOSSARY

COMMAND
A directive from the user to the system entered from the keyboard or a BATCH input file.

COMMAND DECORER
A part of OS/78 that interprets file and option specification strings typed by the user.

CONDITIONAL ASSEMBLY
The processes of translating specific sections of an assembly language program into machine code only if certain conditions have been met during the assembly process.

CONFIGURATION
A particular selection of computer, peripherals and interfacing equipment that are to function together.

CONSTANT
Numeric data used but not changed by a program.

CONTROL
Memory address at which the next instruction will be executed. When "control is passed" to a location, the instruction contained in that location will be the next one to be executed.

CONVERSATIONAL PROGRAM
A program which interacts dynamically with on-line users, that is, an interactive program.

COUNTER
A variable or memory location used to control the number of iterations of a program loop.

CPU
Central Processing Unit, the portion of a computer that executes most instructions.

CRASH
Fail Totally. When a system crashes, it will not function at all and must be restarted.

CRC
Cycle Redundancy Check. An error detection technique used for detecting incorrectly read bits in file-structured devices.

CREATE
To open, write, and close a file for the first time.

CROSS REFERENCE PROGRAM (CREF)
A program that generates a sequence-numbered listing file and a table that contains line numbers of all referenced user-defined symbols and literals and their usage.

CYCLE TIME
A basic unit of time in a computer, usually equal to the memory read time plus memory write time. Computer instructions usually execute in multiples of the cycle time.

DATA
A general term used to denote any or all computer-represented facts, numbers, letters and symbols.

DEBUG
To detect, locate, and correct mistakes in a program.
GLOSSARY

DEFAULT
A parameter that is assumed by the system when none is explicitly specified.

DELIMITER
A character that separates and organizes elements of data, particularly the symbols of a programming language.

DEVICE
A peripheral hardware I/O unit and/or a removable data volume mounted on it.

DEVICE CODES
Numbers assigned to each device in the system, used in the computer instructions for those devices.

DEVICE DRIVERS
See Device Handlers.

DEVICE HANDLERS
Routines that perform I/O for specific devices in a standard format. These routines also handle error recovery and provide device independence.

DEVICE INDEPENDENCE
The ability of a computer system to divert the input or output of an executing program from one device to another, either automatically if the specified device is out of order, or by a keyboard command to the Monitor.

DIAGNOSTIC
Pertaining to the detection and isolation of hardware malfunctions.

DIGITAL
Representation of information by discrete units.

DIRECT ACCESS
Same as Random Access.

DIRECT ADDRESS
A number that specifies the location of an instruction operand.

DIRECTORY
A reserved storage area on a mass storage device that describes the layout of the data on that device in terms of file names, length, location, and creation date.

DISKETTE
A removable data volume, consisting of a thin disk coated with a magnetic data-storage material. Also called a floppy disk.

DISK PACK
See cartridge disk pack.

DUMMY ARGUMENTS
Symbolic names used only as placeholders for other actual arguments that will be uniformly substituted for them at a later time. Used within programming language function definitions to represent the independent (supplied) variables.

ECHO
The displaying by the terminal of characters typed on the keyboard.
EDITOR
A program which interacts with the programmer or typist to enter new programs into the computer and edit them as well as modify existing programs.

EFFECTIVE ADDRESS
The address actually used to fetch an instruction operand, that is, the specified address modified by indexing or indirect addressing rules.

ENTRY POINT
A point in a subroutine to which control is transferred when the subroutine is called.

ERROR MESSAGE
A message from computer system to programmer that reports a hardware malfunction or an incorrect format or operation detected by software.

EXECUTE
To cause the computer to carry out an instruction; to run a program on the computer.

FIELD
1. A division of memory containing 4096 decimal (numbered 0-7777 octal) storage cells (locations).
   2. An element of a format specification, particularly of a record.

FILE
A contiguous block of characters or computer words, particularly where stored on a mass storage device and entered in the device's directory.

FILE NAME
A name of one to six alphanumerics chosen by the user to identify a file.

FILE NAME EXTENSION
Two alphanumerics chosen by the programmer or provided by OS/78 to describe the format of information in the file.

FILE-STRUCTURED DEVICE
A device on which files may be stored; also contains a file directory.

FIXED POINT
A format in which one or more computer words (two in the case of OS/8 FORTRAN integers), represent a number with a fixed binary point.

FLAG
A variable or register used to record the status of a program or device. In the latter case it is sometimes called a device flag.

FLOATING POINT
A format in which one or more computer words (three in the case of OS/8 FORTRAN Real and OS/8 BASIC numbers) represent a number with a variable binary point, particularly when there is an exponent part and a mantissa part.

FLOPPY DISK
See diskette.
GLOSSARY

FORMAT
1. A description of a set of valid sequences of language or data elements; syntax.

2. A FORTRAN statement which specifies the arrangement of characters to be used to represent a piece of data.

FORTRAN
A high-level language developed for the scientific community, it stands for FORMula TRANslation.

GARBAGE
Undefined or random data or instructions.

HARD COPY
Computer output in the form of printing on paper and generally in readable form such as listings or other documents.

HEAD
A hardware component that reads, records, or erases data on a file-structured device.

HIGH-LEVEL LANGUAGE
A language in which single statements typically result in more than one machine language instruction, for example, BASIC, FORTRAN.

INDIRECT ADDRESS
An address in a computer instruction which indicates a location in memory where the address of the referenced operand is to be found.

INHIBIT
To prevent, suppress or disallow.

INITIALIZATION CODE
Code which sets counters, switches, and addresses to zero or other starting values at the beginning of or at prescribed points in a computer routine.

INPUT BUFFER
A section of memory used for storage of input data.

INPUT
The process of transferring data to memory from a mass storage device or from other peripheral devices into the AC.

INSTRUCTION
One unit of machine language, usually corresponding to one line of assembly language, which tells the computer what elementary operations to do next.

I/O
Input-output. Refers to transfers of data between memory and peripheral devices.

ITERATION
Repetition of a group (loop) of instructions.

INTERACTIVE
Referring to a mode of using a computer system in which the computer and the user communicate via a computer terminal.
GLOSSARY

INTERRUPT
A hardware facility that executes an effective JMS to location 0 of field 0 upon any of a particular set of external (peripheral) conditions. The processor state is saved so that the interrupted program can be continued following any desired interrupt-level processing.

INTERRUPT DRIVEN
Pertaining to software that uses the interrupt facility of the computer to handle I/O and respond to user requests.

JOB
A unit of code which solves a problem; a program or sequence of programs.

JUMP
A departure from the consecutive sequence of executing instructions. Control is passed to some location in memory which is specified by the jump instruction.

K
Two to the tenth power (1024 in decimal notation). For example, a 4K memory has 4096 words.

KEYBOARD
On a typing device, the array of buttons which causes character codes for letters, numbers, and symbols to be generated when pressed.

LABEL
One or more characters used to identify a source language statement on line or label.

LATENCY
On rotating storage devices, the delay between the instant the device is notified that a transfer is coming and the instant the device is ready to perform the transfer.

LEAST SIGNIFICANT DIGIT
The rightmost digit.

LIBRARY
A collection of standard routines which can be incorporated into other programs.

LINE
A string of characters terminated with a line feed, vertical tab, or form feed character (and usually also a carriage return). The terminator belongs to the line that it terminates.

LINE NUMBER
1. In source languages such as BASIC and FORTRAN, a number which begins a line for purposes of identification. A numeric label.

2. In editors and listings, the number of the line beginning at the first line of the program and counting each line.

3. An automatically-generated indirect address for an off-page reference (PAL8).
GLOSSARY

LINK
1. A one-bit register that is complemented when overflow occurs in the accumulator.
2. An address pointer to the next element of a list or next block of a file.
3. A PAL8 assembler-created indirect address.

LINKAGE
Code that connects two separately coded routines and passes values and/or control between them, particularly when the routines occupy separate fields.

LIST
To output a listing on the terminal or line printer. The listing of a source program is a sequential copy of statements or instructions in the program. Also, a table of data in a program.

LITERAL
A constant that defines itself by requesting the assembler to reserve storage for it, even though no storage location is explicitly specified in the source program.

LOAD
To move data into a hardware register or into memory.

LOADER
A program which takes information in binary format and copies it into memory. An absolute loader loads binary information that has been prepared for the absolute addresses of memory. A relocatable or linking loader loads binary information specified in relative addresses by assigning an absolute address to every relative address.

LOCATION
A numbered or named cell in memory where a word of data or an instruction or address may be stored.

LOOP
A sequence of instructions that is executed repeatedly until a terminating condition occurs. Also, used as a verb meaning to execute this sequence of instructions while waiting for the ending condition.

MACHINE LANGUAGE
The language, particular to each kind of computer, that those computers understand. It is a binary code which contains an operation code to tell the computer what to do and an address to tell the computer what data to perform the operation on.

MAP
A diagram representing memory locations and outlining which locations are used by which programs.

MASK
A combination of bits that is used to clear or accept selected portions of any word, character or register while retaining or ignoring other parts. Also, to clear these selected locations with a mask.

MASK STORAGE
Devices such as a diskette and disk pack that stores large amounts of data readily accessible to the central processing unit.
GLOSSARY

MATRIX
A rectangular array of elements. A two-dimensional table can be considered a matrix.

MEMORY
The main storage in a computer from which instructions must be fetched and executed. Consists of a sequence of consecutively-numbered cells storing one word each.

MEMORY IMAGE FILE
An executable binary code program file created by the system command SAVE from the current contents of memory.

MEMORY MAP
A diagram or table showing specific memory requirements of one or more programs.

MEMORY REFERENCE INSTRUCTION
A computer instruction that accesses the computer memory during its execution, as opposed to a register instruction which only accesses registers in the CPU and I/O instructions which are commands to peripheral devices.

MNEMONIC
A symbolic representation of an operation code or address (for example, X for unknown variable, JMP for jump instruction).

MODE
A state or method of system or program operation.

MODULE
A routine that handles a particular function.

MONITOR
The collection of routines which schedules resources, I/O, system programs, and user programs, and obeys keyboard commands.

MONITOR SYSTEM
Editors, assemblers, compilers, loaders, interpreters, data management programs and other utility programs all automated for the user by a monitor.

MOST SIGNIFICANT DIGIT
The leftmost digit.

NESTING
The inclusion of one program language construction inside another.

NON-DIRECTORY DEVICE
A device such as a terminal or a line printer that cannot store or retrieve files.

NO-OP
Contraction of No Operation, an instruction that specifically causes the computer to delay for one instruction time, and then to get the next instruction.

OBJECT CODE
The result after assembling or compiling source code.

OCTAL
The number system with a base, or radix, of eight.
GLOSSARY

ODT
Octal Debugging Technique, an interactive program for finding and correcting bugs in programs in which the user communicates in octal notation.

OFF-LINE
Pertaining to equipment, devices or events which are not under direct control of the computer.

ONE'S COMPLEMENT
A number formed by setting each bit in an input number to the other bit: 1's become 0's and 0's become 1's.

OP-CODE
The part of a machine language instruction that identifies the operation that the CPU will be required to perform.

OPERAND
The data to be used when an instruction is executed.

OPERATING SYSTEM
See MONITOR SYSTEM

OPERATOR
That symbol or code which indicates an action or operation to be performed (e.g., + or TAD).

ORIGIN
The absolute address of the beginning of a section of code.

OUTPUT
The process of transferring data from memory to a mass storage device or to a listing device such as a line printer.

OVERFLOW
A condition that occurs when a mathematical operation yields a result whose magnitude is larger than the program or system is capable of handling.

PACK
To conserve storage requirements by combining data.

PAGE
A 128 (decimal) word section of memory, beginning at an address which is a multiple of 200 (octal). There are 40 (octal) pages in a field, numbered 0-37 (octal).

PAL
The name of the OS/78 system's assembly language.

PARITY BIT
A bit that indicates whether the total number of binary one digits in a piece of data is even or odd.

PARITY CHECK
A check that tests whether the sum of all the bits in an entity is odd or even.

PASS
One complete reading of a set of input data. An assembler usually requires two passes over a source program in order to translate it into binary code.
PATCH
To modify a routine in an expeditious way, usually by modifying the memory image rather than reassembling it.

PERIPHERAL
In a data processing system, any device distinct from the CPU, which provides the computer system with outside communication.

POINTER
A location containing the address of another word in memory.

PRINTOUT
A loose term referring to almost anything printed by a computer peripheral device; any computer-generated hard copy.

PROGRAM
A unit of instruction and routines necessary to solve a problem.

PROGRAM COUNTER
A register in the CPU that holds the address of the next instruction to be executed.

PROGRAMMABLE
Can be controlled by instructions in a program.

PROMPTING CHARACTER
A character that is displayed on the console terminal and cues the user to perform some action.

PSEUDO-OP
Contraction of "Pseudo Operation", an assembly language directive which does not directly translate into machine code but gives directions to the assembler on how to assemble the code that follows.

RADIX
The base of a number system, the number of digit symbols required by a number system. The decimal number system is radix 10.

RANDOM ACCESS
Pertaining to a storage device where data or blocks of data can be read without regard to their physical order (for example, diskette).

READ
To transfer information from a peripheral device into memory or a register in the CPU.

RECORD
A collection of related items of data treated as a unit, such as a line of source code, or a person's name, address, and telephone number.

REDUNDANCY
In any data, that portion of the total characters or bits that can be eliminated without any loss of information.

REGISTER
A device usually made of semiconductor components that is capable of storing a specified amount of data, frequently one word.

RELATIVE ADDRESS
The number that specifies the difference between the actual address and the base address.
GLOSSARY

RELOCATE
To move a routine from one portion of storage to another and to adjust the necessary address references so that the routine can be executed in the new location.

RESTART
To resume execution of a program.

RETURN
To pass control back to a calling program at a point following the call when a subroutine has completed its execution.

1. The set of instructions at the end of a subroutine that permits control to return to the proper point in the main program.

2. The point in the main program to which control is returned.

3. The name of a key on the terminal.

RING-BUFFER
A storage area for data accessed on a first-in, first-out basis whose area is reused circularly.

ROUTINE
A set of instructions arranged in proper sequence to cause the computer to perform a desired task.

RUN
1. A single continuous execution of a program.

2. To perform that execution.

RUN TIME
The time during which a program is executed.

SCRATCH PAD MEMORY
Any memory or registers used for temporary storage of partial results.

SECTOR
The smallest unit of physical storage on a file-structured device.

SEGMENT
To divide information into segments or to store portions of information program on a mass storage device to be brought into memory as needed.

SERIAL ACCESS
Pertaining to the sequential or consecutive transmission of data to or from memory or a peripheral device.

SERIAL TRANSMISSION
A method of information transfer in which the bits composing the character are sent sequentially on a single path.

SERVICE ROUTINE
A program used for general support of the user; I/O routines, diagnostics, and other utility routines.

SHIFT
A movement of bits to the left or right, usually performed in the accumulator.
GLOSSARY

SIMULATE
To represent the function of a device, system or program with another device, system or program.

SOFTWARE
The executable instructions used in a computer system.

SOURCE LANGUAGE
Any programming language used by the programmer to write a program before it is translated into machine code.

SOURCE PROGRAM
The computer program written in the source language.

SPOOLING
The technique by which output to slow devices is placed into temporary files on mass storage devices to await transmission. This allows more efficient use of the system since programs using low speed devices can run to completion quickly and make room for others.

STATEMENT
An expression or instruction in a source language.

STORAGE
A general term for any device capable of retaining information.

STORE
To enter data into a storage device, especially memory.

STRING
A sequence of characters.

SUBROUTINE
A section of code, usually performing one task, that may be called from various points of a main program.

SUBSCRIPT
A value used to specify a particular item in an array.

SYMBIONT
A program (or task) that can run simultaneously and cooperatively with the OS/78 system. This is a feature of DEC station 78 series systems (see Appendix H).

SYMBOLIC ADDRESS
Alphanumeric characters used to represent a storage location in the context of a particular program. It must be translated to an absolute address by the assembler.

SYMBOL TABLE
A memory storage area or a listing that contains all defined symbols and the binary value associated with each one. Mnemonic operators, labels, and user defined symbols are all placed in the symbol table. (Mnemonic operators stay in the table permanently.)

SYNCHRONOUS
Pertaining to circuits or events where all changes occur simultaneously or in definite timed intervals.

SYSTEM
A combination of software and hardware which performs specific processing operations.
GLOSSARY

SYSTEM DEVICE
A peripheral mass storage device on which the system software resides. Its handler (SYS) is resident in the last page of field 0.

SYSTEM HEAD
The system area reserved on a file-structured device for the OS/78 Keyboard Monitor. It resides on the system device and contains, in addition to the Monitor, programs such as ODT and Command Decoder.

SYSTEM SOFTWARE
DEC-supplied programs which come in the basic software packages. These include editors, assemblers, compilers, loaders, etc.

TABLE
A collection of data sorted for ease of reference, generally a two-dimensional array.

TEMPORARY STORAGE
Storage locations or registers reserved for intermediate results.

TERMINAL
A peripheral device in a system through which data can enter or leave the computer, especially in a display and keyboard.

TEXT
A message or program expressed in characters.

TRACK
The set of all sectors of a file-structured device that are accessible at each head position.

TRUNCATION
The reduction of precision by ignoring one or more of the least significant digits without rounding off.

TWO'S COMPLEMENT
A number used to represent the negative of a given value in many computers. This number is formed from the given binary value by changing all 1's to 0's and all 0's to 1's, then adding 1.

UTILITIES
Programs to perform general useful functions.

VARIABLE
A piece of data whose value changes during the execution, assembly, or compilation of a program.

WORD
A 12-bit unit of data which may be stored in one addressable location in memory or on a peripheral device.
INDEX

Ø FORTRAN carriage control character, 7-101
1 FORTRAN carriage control character, 7-101
/2 QUEUE command option, H-2
/8 LOAD command option, 3-43
/9 LOAD command option, 3-43
/=n DIRECT command option, 3-25

A,
Editor append command, 4-6
field descriptor, 7-95
ODT command, 9-6
ABS BASIC function, 6-48
ABSLDH.SV, 3-11, 3-34, 3-44
Absolute binary files, 3-42
Absolute value function, 6-48
Addition,
BASIC operator (+), 6-17
FORTRAN operator (+), 7-39
PAL8 operator (+), 5-18
Addressing indirect and page zero, 5-29
AINT FORTRAN function, 7-106
 ALOG FORTRAN function, 7-106
 ALOG10 FORTRAN function, 7-106
Alphanumeric literals, 7-31
AMAX0 FORTRAN function, 7-106
AMAX1 FORTRAN function, 7-106, 6-106
AMIN0 FORTRAN function, 7-106
AMIN1 FORTRAN function, 7-106
AMOD FORTRAN function, 7-106
Amersand (&),
BASIC operator, 6-18, 6-33
PAL8 operator, 5-18
AND,
BASIC function, 6-58
PAL8 AND symbol (&), 5-18
.AND. FORTRAN operator, 7-42
Angle brackets (<>),
BASIC relations, 6-21
in PAL8 comments, 5-23
left (<),
Editor command, 4-12
monitor, 2-13
PAL8 conditionals, 5-23
right (>) Editor command, 4-12
Arctangent function, 6-45
Arguments system retention of, 2-17
Arithmetic expressions,
BASIC, 6-17
FORTRAN, 7-38
Arithmetic statements,
FORTRAN, 7-45
BASIC strings, 6-19
Arrays,
BASIC, 6-24
FORTRAN, 7-33, 7-47
ARROW SET command option, 3-62
ASC function, 6-52
ASCII,
6-bit, 5-37
character set, A-1
conversion,
BASIC function, 5-52
PAL8, 5-22
entering PAL8 text strings, 5-37
file format, 2-26
ASF function, 7-70
ASIN FORTRAN function, 7-107
Assembling a PAL8 program, 5-5
CREF command, 3-19
PAL command, 3-53
Assembly language programs,
Command Decoder use, D-1
math subroutines, B-1
Assembler termination, 5-39
ASSIGN,
command, 2-11, 3-2
statement, 7-58
ASSIGNed GOTO, 7-58
Assigning logical device names, 3-2
Asterisk (*),
BASIC, 6-33
double (**),
BASIC operator, 6-17
FORTRAN operator, 7-39
FORTRAN operator, 7-39
prompting symbol, 2-21
PAL8 location counter, 5-14
PAL8 special character, 5-21
wild character, 2-18
At (@) symbol indirect commands, 2-20
ATAN FORTRAN function, 7-107
ATAN2 FORTRAN function, 7-107
ATN function, 6-45
Autoindex registers, 5-28

B,
Editor buffer command, 4-7
ODT command, 9-5
/B,
BASIC option, 6-79
COMPARE option, 3-6
COMPILE command option, 6-79
INDEX (Cont.)

/B (cont.)
DIRECT command option, 3-25
Editor option, 4-21
PAL8 option, 5-9
.BA file name extension, 2-12
BACKSPACE statement, 7-86
BASIC, (6-1),
arithmetic expressions, 6-17
auto line numbering, 6-8
calling an old program, 6-4
changing program text, 6-7
changing text, 6-7
command, 3-3
command summary, 6-80
commands, 6-3
commercial arithmetic, 6-19
compilation, 3-34
constants, 6-13
control statements, 6-36
creating a new program, 6-4
deleting text, 6-7
deleting program lines, 6-12
direct record I/O, 6-69
elements of the language, 6-13
erasing a program, 6-11
error messages, 6-86
execution, 3-34, 3-10
expression operators, 6-17
expressions, 6-17
file control, 6-62
file statements, 6-62
format control, 6-30
function summary, 6-83
functions, 6-43
interaction with BATCH, 6-80
interrupting program execution, 6-12
issuing monitor commands, 6-55
listing a program, 6-5
listing control, 6-12
logical functions, 6-58
merging text, 6-9
numeric,
constants, 6-13
field list use, 6-36
functions, 6-44
variables, 6-15
operating procedures, 6-3
overlay files, 6-80
overview, 6-1
printing format strings, 6-35
program termination, 6-41
program segmentation, 6-74
reading files, 6-65
relational expressions, 6-18
renaming a program, 6-10
resequencing a program, 6-11
running a program, 6-5
saving programs, 6-79

BASIC (cont.)
special print functions, 6-34
statements, 6-23
formatting, 6-22
summary, 6-81
storing a program, 6-10
strings,
concatenation, 6-18
constants, 6-14
functions, 6-49
numbers, 6-20
relations, 6-22
variables, 6-15
subroutines, 6-42
subscripted variables, 6-16
user-defined functions, 6-78
using record I/O files, 6-72
variables, 6-14
writing files, 6-65
writing programs, 6-1
BASIC.SV, 3-3
BAT device handler, 3-67
Batch,
command processing, 8-3
commands, 8-2
error messages, 8-6
input file structure, 8-4
interaction with BASIC, 6-80
introduction, 8-1
SUBMIT command, 3-72
operating restrictions, 8-8
options, 8-2
BATCH.SV, 3-74
BCCOMP.SV, 3-11,, 3-34
.MULTI file name extension, 2-12
BITMAP.SV, 3-48
Blank lines in FORTRAN programs, 7-25
BLOAD.SV, 3-34
BLOCK DATA statement, 7-54
.MULTI file name extension, 2-12
Boolean operators, PAL8, 5-18
BOOT,
command, 3-4
button, 2-6
BOOT.SV, 3-4
Bootstrapping OS/78, 2-6
Brackets ([]), 2-16
PAL8 special characters, 5-22
Breakpoints under ODT, 9-2
BATCHx BATCH files, 3-74
Buffer size of Editor, 4-4
BYE command, 6-11

C,
Editor change command, 4-9
FORTRAN comment indicator, 7-23
ODT command, 9-7

Index-2
INDEX (Cont.)

/C,
COMPARE option, 3-6
COPY command option, 3-14
DELETE command option, 3-24
DIRECT command option, 3-25
FORTRAN Loader option, 7-10
FRTS option, 7-16
LIST command option, 3-41
LOAD command option, 3-44
PAL8 option, 5-9
RENAME command option, 3-56
TYPE command option, 3-76
C:n/ QUEUE command option, H-2
CALL statement, 7-74
CANCEL command, 3-5, H-1
CAPS function, 6-54
Caret - see circumflex
CCL.SV, 3-9, 3-3, 3-4
CDF instruction, 5-30, 5-43
CGET subroutine, 7-107
CHAIN,
statement, 6-74
USR function C-10
Chaining PAL8 programs, C-10
Character conversion function, 6-52
Character set, A-1
BASIC, 6-2, 6-2
FORTRAN, 7-20
PAL8, 5-10
Characters,
output by codes (BASIC), 6-60
PAL8 formatting, 5-12
wild, 2-18
CHKEOF FORTRAN subroutine, 7-108
CHR$ function, 6-53
CIF instruction, 5-30, 5-43
Circumflex (') ODT command, 9-4
CLOCK FORTRAN subroutine, 7-108
CLOSE function (USR), C-8
CLOSE# statement, 6-64, 6-71
.CM file name extension, 2-12
COL,
function, 6-60
SET command option, 3-62
Comma (,),
BASIC, 6-34
FORTRAN use, 7-101
PAL8 special character, 5-21
Command Decoder,
calling from USR, C-1
conventions, D-1
calling, D-3
error messages, D-2
example of use, D-6
option table, D-5
special mode call, D-8
tables, D-4
Commands, OS/78, 2-11, 3-1
argument retention 2-17
Commands, OS/78 (cont.)
ASSIGN, 3-2
BASIC, 3-3
BOOT, 3-4
CANCEL, 3-5
COMPARE, 3-6
COMPILE, 3-10, 7-4
COPY, 3-12
CREATE, 3-18
CREF, 3-19
DATE, 3-21
DEASSIGN, 3-22
DELETE, 3-23
DIRECT, 3-25
DUPLICATE, 3-28
EDIT, 3-32
EXECUTE, 3-34
format, 2-13
FORMAT, 3-35
full word switches, H-1
GET, 3-37
HELP, 2-22, 3-38
incorrect, 2-14
LIST, 3-49
LOAD, 3-42
MAP, 3-46
MEMORY, 3-50
ODT, 3-52, 9-3
PAL, 3-53
QUEUE, 3-54
R, 3-55
RENAME, 3-56
REQUEST, 3-57
SAVE, 3-59
SET, 3-62
SQUIRCH, 3-76
START, 3-71
SUBMIT, 3-72, 8-2
summary, 2-26
TERMINATE, 3-75
TYPE, 3-76
UA/UB/UC, 3-78
ZERO, 3-79
Comments,
BASIC, 6-26
FORTRAN, 7-23
PAL8, 5-12
COMMON statement, 7-50
interaction with EQUIVALENCE, 7-54
COMPARE command, 3-6
error messages, 3-9
option summary, 3-6
COMPILE command, 3-10
option summary, 6-79, 7-5
Computed GOTO, 7-58
Concatenation, 6-18
Conditional,
assembly operators, 5-36
delimiters, 5-23
INDEX (Cont.)

Conditional (cont.)
  transfer,
    BASIC, 6-37
    FORTRAN, 7-60

Constants,
  BASIC, 6-13
  FORTRAN, 7-27
    exponential, 7-29
    integer, 7-28
    Hollerith, 7-31
    logical, 7-30
    octal, 7-30

CONTINUE statement, 7-66

Control character,
  representation, 2-7
  uparrow (^) echo control, 3-63

Control statements,
  BASIC, 6-36
  FORTRAN, 7-56

COPY command, 3-12
  option summary, 3-16
  error messages, 3-17

Core Control Block, 3-55, 3-59, 3-71, 9-2

Correcting errors, 2-15

COS function, 6-45

COS FORTRAN subroutine, 7-108

COSH FORTRAN subroutine, 7-108

CPUT FORTRAN subroutine, 7-109

CREATE command, 3-18

Creating a PAL8 program, 5-4

CREF command, 3-19
  error messages, 3-20
  option summary, 3-19

CREFLS.TM file, 3-19

CREF.SV, 3-19, 3-19

Cross-reference listing, 3-19, 5-7

CTRL/C, 2-7, 2-15
  BASIC use, 6-12
  BATCH use, 8-5
  COPY use, 3-15
  EDIT use, 4-3

CTRL/G, 5-24

CTRL/L, 4-3

CTRL/O, 2-7

BASIC, 6-6, 6-12
  control character, 4-3

ODT use, 9-11

CTRL/Q, 2-7

BASIC use, 6-12

CTRL/S, 2-7

BASIC use, 6-12

CTRL/U, 2-7

BASIC use, 6-12

FORTRAN use, 7-16

CTRL/Z,
  FORTRAN use, 7-16
  with device handlers, F-2, F-4

CUR$ function, 6-59

Current location counter, 5-21
  changing with PAGE, 5-33
  PAL8 symbol, 5-14
  PAL8, 5-21

Current page literals, 5-22

Cursor function, 6-59

D,

  Editor delete command, 4-9
  ODT command, 9-7

/D,

  DUPLICATE command option, 3-30
  Editor option, 4-21
  Dash options (general-purpose), 2-16

DATS function, 6-57

Data field register, 5-30

DATA statement, 6-28, 7-54

DATE,
  command, 3-21
  FORTRAN subroutine, 7-109

Date,
  file transfer by, 3-14
  function, 6-57

DEASSIGN command, 3-22

Debugging,

  function (BASIC), 6-57
  in octal, 3-52, 9-1

DECIMAL pseudo-operator, 5-36

Decimal to octal conversion, 6-55

Decimal point (.), 7-28

DECODE function (USR), C-10

DECstation System Hardware, 1-1

  configuration summary, J-1

DEF statement, 6-56

Default,

  file name extensions, E-1
  file specifications, 2-21

DEFINE FILE statement, 7-86

DEFINE$ statement, 6-78

DELETE,

  BASIC command, 6-7
  key, 2-7, 6-12, 7-16
  OS/78 command, 3-23
  error messages, 3-24
  option summary, 3-24

Demonstration programs, 2-10

Device Control Word Table, C-15

Device handlers, 3-67
  calling, F-1
  description and use, F-1

FETCH function (USR), C-4

  file-structured devices, 2-23, F-4

  function control word, F-2
  line printer, F-3

  loading from PAL8 programs, C-4
INDEX (Cont.)

Device handlers (cont.)
  multiple input file
  restrictions, F-5
  obtaining characteristics via
  USR, C-13
  removal from memory, C-14
  restrictions of use, F-3
  terminals, F-5
Device names,
  permanent, 2-10
  reassigning, 2-10
DEVICE pseudo-operator, 5-37
Devices - see terminals also
  file-structured, 2-23, F-4
  FORTRAN I/O, 7-7
  storage capacity, 2-23
  using RX01/RX02 and RL01K, 2-24
  using VXA0, 2-25
.DI file name extension, 2-12
DIM FORTRAN function, 7-109
Diagnostic messages,
  FORTRAN, 7-5
  PAL8, 5-39
DIMENSION statement,
  BASIC, 6-24
  FORTRAN, 7-47
Direct,
  assignment statements, 5-16
  record I/O, 6-69
DIRECT command, 3-25
  option summary, 3-25
  error messages, 3-27
DIRECT.SV, 3-27
Directory (file), 2-25
Disk pack - see RL01K,
Diskette - see RX01/RX02 also
  bootstrap procedure, 2-5
  making backup copies, 2-8
Dismissing USR, C-13
Division,
  BASIC operator (/), 6-17
  FORTRAN operator (/), 7-39
  PAL8 operator (\), 5-18
DO statement, 7-62, 7-64
Document,
  conventions, Preface
  reference, Preface
  dollar sign ($),
  BASIC, 6-34
  Editor symbol, 4-10
  ESCape key echo, 3-64
  FORTRAN field descriptor, 7-98
  PAL8 special symbol, 5-24
Double,
  asterisk (**), 6-17
  quote ("), 5-22
Double-density diskette
  formatting, 3-30
  DSK device handler, F-4

Dummy arguments in subprograms, 7-69
DUPLICATE command, 3-28
  error messages, 3-31
  option summary, 3-30

E,
  Editor command, 4-7
  FORTRAN field descriptor, 7-92
  ODT command, 9-7
/E,
  CREF command option, 3-19
  DIRECT command option, 3-25
  FRTS option, 7-16
  PAL8 option, 5-9
  SUBMIT command option, 3-72, 8-2
  ECHO option (SET command), 3-62
EDIT command, 3-32
  BASIC, 6-7
EDIT.SV, 3-18, 3-33
Editor - (see CREATE and EDIT
  commands also), 4-1
  alteration commands, 4-9
  append (A) command, 4-6
  buffer (B) command, 4-7
  buffer size, 4-4
  calling procedure, 4-1
  change (C) command, 4-9
  changing existing file, 4-2
  command mode, 4-2, 4-5
  command summary, 4-23
  command usage, 4-13 thru 4-18
  control commands, 4-3
  creating a new file, 4-1
  delete (D) command, 4-9
  Editor, exit (E) command, 4-7
  error messages, 4-21, 4-22
  example session, 4-18
  get (G) command, 4-6
  insert (I) command, 3-6
  interbuffer search (J) command, 4-10
  kill (K) buffer command, 4-8
  list (L) command, 4-6
  list buffer (V) command, 4-9
  listing commands, 4-6
  next (N) command, 4-8
  operating modes, 4-2
  option summary, 4-21
  output commands, 4-7
  output current buffer (E)
    command, 4-7
  output (P) command, 4-7
  page (P) command, 4-4
  quit (Q) command, 4-8
  read (R) command, 4-5, 4-6
  search character (S) command, 4-10

Index-5
INDEX (Cont.)

Editor (cont.)
  search next (F) command, 4-10
  special characters, 4-11
  text mode, 4-2
  text searches, 4-13
  yank (Y) command, 4-10
EJECT pseudo-operator, 5-34
SEND command (BATCH), 8-3
END statement, 6-41, 7-68
END# statement, 6-68
End-of-file,
  BASIC, 6-68
  FORTRAN ENDFILE statement, 7-87
PAL8, 5-37
ENTER function (USR), C-7
Exponentiation, 6-17
Equals sign (=), 2-16
  BASIC, 6-21, 6-23
  Editor command, 4-12
  FORTRAN use, 7-45
  PAL8 direct assignment, 5-16
  PAL8 special character, 5-21
EQUIV statement, 7-52
  interaction with COMMON, 7-54
.EQV. FORTRAN operator, 7-42
Error messages,
  BASIC, 6-86
  BATCH, 8-6
  Command Decoder, D-2
  COMPARE command, 3-9
  COPY command, 3-17
  CREF command, 3-20
  DELETE command, 3-24
  DIRECT command, 3-27
  DUPLICATE command, 3-31
  Editor, 4-21,4-22
  FORMAT command, 3-36
  FORTRAN Loader, 7-11
  FRTS, 7-17
  HELP command, 3-39
  LIST command, 3-41
  MAP command, 3-49
  ODT, 9-8
  OS/78 summary, G-1
  PAL8, 5-39
  SET command, 3-68
  system halts, G-1
  TYPE command, 3-77
ERROR function (USR), C-11
Errors, correcting keyboard, 2-15, 6-12
ESC SET command option, 3-62
ESCAPE key, 2-7
  BATCH operation, 3-73
  dollar sign ($) echo control, 3-64
  Editor intrabuffer search command, 4-10
  FORTRAN use, 7-14
  terminator, 2-13

ESCAPE sequences in BASIC, 6-76
Exclamation point (!),
  BASIC, 6-26
  PAL8 operator, 5-18
EXECUTE command, 3-34
Executing PAL8 programs, 5-7
EXP function, 6-46
EXP FORTRAN function, 7-110
Exponential,
  function, 6-46
  operator (**), 7-39
  real constant, 7-29
Expressions,
  BASIC, 6-17
  FORTRAN, 7-38
  PAL8, 5-18
EXPUNGE pseudo-operator, 5-35
Extensions, file name 2-11, E-1
EXTERNAL statement, 7-48

F,
  Editor search next command, 4-10
  field descriptor, 7-91
/F,
  COPY command option, 3-14
  DIRECT command option, 3-25
  PAL8 option, 5-9
F4.SV, 3-11, 3-34
/ff LOAD command option, 3-42
/ffnnnn, LOAD command option, 3-42
.FALSE. logical constant, 7-30
FETCH function (USR), C-4
FIELD pseudo-operator, 5-29
Fields,
  BASIC numeric, 6-33
  memory data and instruction, 5-30
FILE statement, 6-63
FILE# statement, 6-70
FILENAME pseudo-operator, 5-37
Files, 2-25
Files,
  absolute binary, 3-42
  ASCII format, 2-26
  BASIC, 6-62
  batch input, 3-72
  changing ASCII, 4-2
  closing, 6-71, C-8
  creating ASCII, 4-1
  deleting tentative, C-15
  directory, 2-25
  listing, 3-25
  eliminating empty, 3-70
  fixed length, 6-70
  input table, D-4
  memory-image,
    creation, 3-59
    loading and executing, 3-55, 3-71
Files (cont.)
names, 2-10
  extensions, 2-12, E-1
opening for record I/O, 6-70
output spooling under BATCH, 3-73
output table, D-4
permanent, C-6
postdeletion, 3-13
predeletion, 3-13
protection during transfer, 3-14
resetting BASIC, 6-67
restrictions on multiple input, F-5
specifications,
  using defaults, 2-21
  wildcard input, 2-19
  wildcard output, 2-19
  tentative, C-7
testing for end, 6-68
testing for open, 6-67
transfer by date, 3-14
transfer command COPY, 3-12
types, 2-25
USR search, C-6
variable length string, 6-69
FILEV$ statement, 6-69
FIXS function, 6-55
FLOAT FORTRAN function, 7-110
FIN function, 6-56
FOR statement, 6-39
FORLIB.RL, 3-44, 7-19
FORM FEED, 5-12, 5-24
FORMAT command, 3-35
error messages, 3-36
  option summary, 3-35
Format,
  ASCII files, 2-26
  BASIC statements, 6-22
  characters, PALB, 5-12
  control, BASIC, 6-30
  statements, FORTH, 7-88
OS/78 commands, 2-13
Formatted I/O (FORTRAN), 7-77
Formatting,
  double density diskettes, 3-30
  RL01K disk packs, 3-35
  single-density diskettes, 3-30
FORTRAN,
  arithmetic,
    assignment statements, 7-45
    conditionals, 7-60
  arrays, 7-33, 7-53
  assignment statements, 7-45
  auxiliary I/O statements, 7-85
  BLOCK DATA subprograms, 7-54
  FORTRAN (cont.)
    carriage control, 7-100
    compilation, 3-34, 3-10
    constants, 7-27
    continuation lines, 7-24
    control statements, 7-56
    DATA statements, 7-54
    data types, 7-27, 7-27, 7-33
    direct access I/O, 7-80
    error message summary, 7-5
    execution, 3-34
    exponential constants, 7-29
    expressions, 7-38
    operators, 7-39
    field descriptors, 7-89
    FORMAT statements, 7-88
    format,
      control and I/O lists, 7-103
      specification separators, 7-101
    FRTS, 7-15
    error messages, 7-17
    grouping and group repeat specifications, 7-100
    Hollerith constants, 7-31
    I/O,
      devices, 7-7
      formatting, 7-80
      lists, 7-77
      statements, 7-76
      identification field, 7-25
    integer constants, 7-28
    introduction, 7-1
    label field, 7-23
    language, 7-19
      summary, 7-112
    library, 7-19
      functions, 7-76, 7-105
      subroutines, 7-105
      line format, 7-25
      literals, 7-97
      Loader, 7-7
      error messages, 7-11
      options, 7-10
    logical,
      assignment, 7-46
      conditionals, 7-62
      constants, 7-30
      expressions, 7-42
      operators, 7-42
      unit numbers, 7-77
    octal constants, 7-30
    operating procedures, 7-4
    program loading, 3-42
    real constants, 7-28
    relational expressions, 7-41
    relocatable binary files, 3-44
    runtime system (FRTS), 7-1, 7-12
    I/O assignment, 7-12
    scale factors, 7-99
INDEX (Cont.)

FORTRAN (cont.)
  sequential I/O, 7-80
  special characters, 7-21
  specification statements, 7-46
  statements, 7-21
  storage declaration, 7-50
  subprograms, 7-69, 7-72
  declaration, 7-48
  user-written, 7-70
  subroutines, 7-73
  subscripts, 7-36
  symbols, 7-26
  variables, 7-31
  FOTP.SV, 3-15, 3-23, 3-40, 3-56
  FRTS, 7-12
  FRTS.SV, 3-34
  -FT dash option, 2-17
  .FT file name extension, 2-12
  FUNCTION statement, 7-72
  Functions,
    BASIC, 6-43
    FORTRAN, 7-105
    user-defined, 6-78

  I,
    Editor insert command, 4-6
    field descriptor, 7-90
    PAL8 indirect addressing
    pseudo-operator, 5-29
    /I LOAD command option, 3-43
    I/O statements,
      FORTRAN, 7-76
    I/O,
      BASIC, 6-26
      direct record (BASIC), 6-69
      DIM FORTRAN function, 7-110
      IF GOTO statements, 6-37
      IF statement, 7-60
      IFDEF pseudo-operator, 5-36
      IFIX FORTRAN function, 7-110
      IFNDIF pseudo-operator, 5-36
      IFNZIF pseudo-operator, 5-36
      IFZERO pseudo-operator, 5-36
      Indirect,
        addressing, 5-29
        commands (@ symbol), 2-20
        INIT option (SET command), 3-62
        Input file table, D-4
        INPUT# statement, 6-65, 6-65
        input/output options, 2-15
        INQUIRE function (USR), C-13
        Instruction field,
          buffer, 5-30
          register, 5-30
          Instructions,
            memory reference, 5-24
            PAL8, 5-11
        INT function,
          BASIC, 6-47
          FORTRAN, 7-110
        Integer function, 6-47
        INTEGER, 7-27
        IOR function, 6-58
        IOT microinstructions, 5-28
        ISIGN FORTRAN function, 7-110
        Iteration,
          BASIC, 6-39
          FORTRAN, 7-62

  G,
    Editor get command, 4-6
    field descriptor, 7-93
    ODT command, 9-5
  /G,
    BASIC option, 6-79
    COMPILE command option, 6-79,
    7-5
    FORTRAN Loader option, 7-10
    LOAD command option, 3-43, 3-45
    PAL8 option, 5-9
    General-purpose dash options, 2-16
    GET command, 3-37
    GET# statement, 6-71
    Global subroutine calls, 5-22
    GOSUB statement, 6-42
    GOTO statement, 6-36, 7-57

  H field descriptor, 7-96
  /H,
    PAL8 option, 5-9, 5-9
    SUBMIT command option, 8-2
    Handler - see device handler,
    HANDLER option (SET command), 3-62
    Hardware configuration summary,
    J-1
    HEIGHT option (SET command), 3-62
    HELP command, 2-22, 3-38
    error messages, 3-39
    HELP.SV, 3-38
    .HN file name extension, 2-12
    Hollerith constants, 7-31

  J Editor command, 4-10
  /J PAL8 option, 5-9
  $JOB BATCH command, 3-72, 8-3
  Job status word, 3-60

  K Editor command, 4-8
  /K PAL8 option, 5-9
  KEYS function, 6-59, 6-77
  Keyboard errors, 2-15
  KT8A Memory Management Control,
INDEX (Cont.)

L,
   Editor list command, 4-6
   field descriptor, 7-94
   ODT command, 9-6
   -L dash option, 2-17
/L,
   COMPILE command option, 7-5
   PAL9 option, 5-9
   QUEUE command option, H-2
   SET HANDLER command option, 3-67
LA34/38, 2-6
LA36, 2-6
LA129, 2-6
Labels, 5-11
   .LD file name extension, 2-12
   LEN function, 6-50
   LET statement, 6-23
   LINE FEED, 2-7
   Editor command, 4-12
   PAL8 special character, 5-24
   ODT command, 9-3
   Link generation and storage, 5-38
   LIST command, 3-40
   BASIC, 6-5
   LISTNH command, 6-6
   Literals,
      FORTRAN, 7-97
      PAL8 current page, 5-22
      PAL8 storage, 5-39
   LOAD command, 3-42
   LOAD.SV, 3-11, 3-34
   Loading PAL8 programs, 5-6
   Loading,
      diskettes, 2-1
      RL81K disk packs, 2-3
   Location counter - see current location counter
   Locking USR in memory, C-12
   LOG function function, 6-47
   LOGICAL statement, 7-27
   Logical,
      devices,
         assigning names, 3-2
         deassigning names, 3-22
         permanent names, 2-10
         expressions, 7-42
         unit numbers, 7-77
   LOOKUP,
      function (USR), C-6
   Looping,
      BASIC, 6-39
      FORTRAN statements, 7-64
      nested (BASIC), 6-40
   Lower-case characters,
      conversion, 6-54
      line printer use, 3-64
      PAL8 restrictions, 5-11
   LPT device handler, 3-67, F-3
   LQP device handler, 3-67, F-3
   -LS dash option, 2-17
   .LS file name extension, 2-12
   M command (ODT), 9-7, 9-8
   /M,
      CREF command option, 3-19
      DIRECT command option, 3-25
      DUPLICATE command option, 3-30
      SUBMIT command option, 3-72
      Machine instruction set, 5-24
      MAP command, 3-46
         option summary, 3-48
         error messages, 3-49
      Mathematical subroutines, B-1
      MAX0 function, 7-111
      MAX1 function, 7-111
      MEMORY command, 3-50
   Memory,
      fields, 3-50
      map, 3-46, 5-8
      page references, 5-38
      reference instructions, 5-24
      reserving with ZBLOCK, 5-33
      Memory-image files,
         creation, 3-59
         execution, 3-71
         loading, 3-37
         loading and executing, 3-55
      Memory Management Control, 5-31
      Microinstructions, 5-25
      IOT, 5-28
         operate, 5-26
      MIN0 function, 7-111
      MIN1 function, 7-111
      MIN12 function, 7-111
      Minus (-),
         BASIC, 6-17, 6-34
         FORTRAN, 7-28, 7-39
         PAL8, 5-18
      MOD function, 7-111
      Monitor,
         issuing commands from BASIC, 6-55
         period (.) prompting symbol, 2-6
   -MP dash option, 2-17
   .MP file name extension, 2-12
   $MSG command (BATCH), 8-3
   Multiplication,
      FORTRAN, 7-39
      PAL8, 5-18
   N Editor next command, 4-8
   /N,
      COMPILE command option, 7-5
      COPY command option, 3-13
      DELETE command option, 3-24
      DUPLICATE command option, 3-30
      PAL8 option, 5-9
      QUEUE command option, H-2

Index-9
INDEX (Cont.)

/n,
FORMAT command option, 3-35
MAP command option, 3-48
NAME command, 6-10
Natural logarithm, 6-47
-NB dash option, 2-17
Nested loops, 6-40
NEW command, 6-4
NEXT statement, 6-39
nnnn+ and nnnn- ODT commands, 9-4
Non-system device, 2-23
.NOT. FORTRAN operator, 7-42
Number sign (#), 6-33
Numbers,
   BASIC string, 6-20
PAL8, 5-13
Numeric,
   constants, 6-13
   fields, 6-33, 6-36
   functions, 6-44
   variables, 6-15
Numeric-to-string conversion, 6-54

/O,
DELETE command option, 3-24
DIRECT command option, 3-25
LIST command option, 3-41
PAL8 option, 5-9, 5-30
RENAME command option, 3-56
TYPE command option, 3-76
OCSS$ function, 6-55
OC$ function, 6-54
OC$AL,
   pseudo-operator, 5-36
   statement, 7-27
Octal constants, 7-30
   3-52, 9-1
Octal Debugging Technique (ODT),
   Octal-to-decimal conversion, 6-54
ODT command, 3-52
   command summary, 9-9
   commands, 9-3
   error messages, 9-8
   illegal characters, 9-5
   introduction, 9-1
   programming notes, 9-8
   setting breakpoints, 9-6
   setting search limits, 9-7
   special characters, 9-3
   word searches, 9-8
OLD command, 6-4
ON GOTO statement, 6-38
ON GOSUB statement, 6-43
OPEN# statement, 6-67
Operands,
   as PAL8 symbols, 5-18
   PAL8, 5-12
Operate microinstructions, 5-26
Operators,
   PAL8 arithmetic and logical, 5-18
   PAL8 conditional assembly, 5-36
OR,
   BASIC function, 6-58
PAL8, 5-18
.OR. FORTRAN operator, 7-42
OS/78,
   bootstrapping, 2-6
   command argument retention, 2-17
   Command Decoder, D-1
   commands, 2-11
      summary, 2-26
   devices,
      handlers, 3-6, F-3
      permanent names,
      error message summary, G-1
      file names and extensions, 2-10
      file-structured devices, 2-23
      full word command options, I-1
      hardware configurations, J-1
      introduction to, 1-1
      logical device names, 2-10
      making software backup, 2-8
      monitor commands from BASIC, 6-55
      startup procedure, 2-5
      system demonstration, 2-10
   User Service Routine, C-1
Output file table, D-4
Overlay files (BASIC), 6-80

/P,
Editor,
   output command, 4-7
   page command, 4-4
FORTRAN scale factor symbol, 7-99

/P,
CREF command option, 3-19
DUPLICATE command option, 3-30
FORMAT command option, 3-35
LOAD command option, 3-44
-PA dash option, 2-17
.PA file name extension, 2-12
Page zero addressing, 5-29
PAGE,
   pseudo-operator, 5-33
SET command option, 3-62
PAL command, 3-53
PAL8, 3-34, 3-53
   assembly, 3-18, 5-5
   chaining programs, C-10
   character set, 5-10
   restrictions, 5-11
   command string examples, 5-2
   comments, 5-12

Index-10
INDEX (Cont.)

PAL8 (cont.)
conditional, assembly operators, 5-36
expressions, 5-23
cross-reference listing, 5-7
current location counter, 5-14
current page literals, 5-22
diagnostic messages, 5-39
direct assignment statements, 5-16
error messages, 5-10
expressions, 5-18
file extensions, 5-1
FORM FEED character, 5-12
formatting characters, 5-12
instructions, 5-11
introduction to, 5-1
label, 5-11
link generation and storage, 5-38
listing control, 5-34
literals, 5-39
loading and saving a program, 5-6
math subroutines, B-1
memory map, 5-8
numbers, 5-13
off-page references, 5-38
operands, 5-12
option summary, 5-9
page zero addressing
pseudo-operator, 5-29
permanent symbols, 5-14
program,
execution, 3-37
loading, 3-42
pseudo-operators, 5-29
radix control, 5-36
special characters, 5-21
statements, 5-11, 5-13
symbol table, 5-16
alteration, 5-35
symbolic instructions, 5-17
symbolic operands, 5-18
symbols, 5-13
TAB character, 5-12
text strings, 5-37
user-defined symbols, 5-14
using Command Decoder, D-1
PAL8.SV, 3-19, 3-34, 3-53
Parentheses ( ( ), 2-16
FORTRAN, 7-44
PAL8, 5-22
Patching a memory-image (SAVE) file, 3-43
PAUSE,
pseudo-operator, 5-37
SET command option, 3-62
statement, 7-67
PDP-8 instruction set, 5-24
Percent (%), 5-18
Period (.),
BASIC, 6-34
Editor current line symbol, 4-11
monitor prompting symbol, 2-6
PAL8 special character, 5-21
Permanent file lookup, C-6
Permanent symbols, 5-14
PIP.SV, 3-70
Plus sign (+),
BASIC operator, 6-17
FORTNAN, 7-28
carriage control character, 7-101
operator, 7-39
PAL8 operator, 5-18
PMT$ function, 6-60
PNT function, 6-31, 6-60, 6-76
POS function, 6-51
Postdeletion of files, 3-13
Predeletion of files, 3-13
Print head position function, 6-60
PRINT statement, 6-29
PRINT# statement, 6-65
PRINT USING statement, 6-32
Program chaining, C-10
Prompt function, 6-60
Pseudo-operators,
DECIMAL, 5-36
DEVICE, 5-37
EXPUNGE, 5-35
EJECT, 5-34
FIELD, 5-29
FILENAME, 5-37
IFDEF, 5-36
IFNDEF, 5-36
IFNZERO, 5-36
IFZERO, 5-36
OCTAL, 5-36
PAGE, 5-33
PAL8, 5-29
PAUSE, 5-37
RELOC, 5-34
TEXT, 5-36
XLIST, 5-34
ZBLOCK, 5-33
PUT# statement, 6-71
Q,
Editor quit command, 4-8
/Q,
COMPILE command option, 7-5
COPY command option, 3-15
DELETE command option, 3-24
LIST command option, 3-41
SUBMIT command option, 3-72, 8-1
TYPE command option, 3-76
INDEX (Cont.)

Question mark (?) wild character, 2-18
QUEUE command, 3-54, H-2
Quote ("),
   double - PAL8 special character, 5-22

R,
   Editor read command, 4-5, 4-6
   monitor command, 3-55
/R,
   DIRECT command option, 3-25
   DUPLICATE command option, 3-30
   MAP command option, 3-48
   RA file name extension, 2-12
   Radix control, 5-36
   RALF.SV, 3-11
   Random number function, 6-48
   READ statement, 6-28, 7-80
   READONLY option (SET command), 3-62
   Real constants, 7-28
   REAL statement, 7-27
   Reassigning device names, 2-11
   Record,
      field definition, 6-70
      size statement, 6-69
   Reference Documents, Preface
   Relational,
      expressions,
         BASIC, 6-18
         FORTRAN, 7-41
      operators,
         BASIC strings, 6-22
   RELOC pseudo-operator, 5-34
   Relocatable binary files, 3-44
   REM statement, 6-26
   RENAME command, 3-56
   RENAME command option summary, 3-56
   REQUEST command, 3-57, H-1, H-2
   RESEQ program, 6-11
   RESET function (USR), C-14
   Reseting BASIC files, 6-67
   RESTORE statement, 6-28, 6-67
   RETURN key, 2-7
      BASIC, 6-12
      ODT command, 9-3
      PAL8 special character, 5-24
   RETURN statement, 6-42, 7-75
   REWIND statement, 7-87
   .RL file name extension, 2-12
   RL81K disk pack,
      bootstrap procedure, 2-5
      formatting, 3-35
      making backup copies, 2-8
      storage capacity, 2-23
      use, 2-24
   RLFMT.SV, 3-36
   RLxx device handler, 3-67, F-4
   RND function, 6-48
   Rounding in BASIC, 6-21
   RUN command, 6-5
   RX01/RX02,
      duplicating diskettes, 3-28
      storage capacity, 2-23
      use, 2-24
   RXAx device handler, 3-68, F-4
   RXCOPY.SV, 3-30

S
   Editor search character command, 4-10
   /S,
      BASIC option, 6-79
      COMPARE option, 3-6
      COMPIL command option, 6-79
      DUPLICATE command option, 3-30
      FORTRAN Loader option, 7-10
      LOAD command option, 3-42, 3-44
      MAP command option, 3-48
      PAL8 option, 5-9
   SAVE command, 3-59
   BASIC, 6-10
   Saving,
      PAL8 programs, 5-6
      BASIC programs, 6-79
   SCOPE option (SET command), 3-62
   SCRATCH command, 6-11
   SEGS function, 6-52
   Semicolon (;) ODT command, 9-4
   Semicontingual transfer, 6-38
   SEQUENCE command, 6-8
   SET command, 3-62
      error messages, 3-68
      options, 3-62
   Setting terminal characteristics, 2-5
   SGN function 6-48,
   SIGN FORTRAN function, 7-111
   SIN,
      BASIC, 6-45
      FORTRAN, 7-111
   Single-density diskette formatting, 3-30
   SINH FORTRAN function, 7-112
   Six-bit ASCII, 5-37
   Slash (/), 2-16
      BASIC operator, 6-17
      BATCH command, 8-3
      Editor symbol, 4-12
      FORTRAN operator, 7-39
      FORTRAN use, 7-101
      ODT command, 9-3
      PAL8 comments, 5-12
   SLux device handler, 3-68, F-5
   Source file comparison, 3-6

Index-12
INDEX (Cont.)

Space character,
  FORTRAN carriage control, 7-101
  PAL8 operator, 5-18
  PAL8 special character, 5-24
Spooler (symbiont) commands, H-2
Spooling, 3-73, 8-1, H-1
SPOLR.SV, 3-54, H-1
SQRT function, 6-46
SQR function, 6-46
SQR function, 7-112
Square brackets ([]), 2-16,
  Preface
Square root function, 6-46
SQUISH command, 3-70
SRCCOM.SV, 3-9
START button, 2-6
START command, 3-71
Statement,
  format,
    BASIC, 6-22
    FORTRAN, 7-24
  terminators,
    OS/78 monitor, 2-7
    PAL8, 5-13
Statements,
  FORTRAN, 7-21
  PAL8, 5-11, 5-16
STEP statement, 6-39
STOP statement,
  BASIC, 6-41
  FORTRAN, 7-68
Storage capacity of devices, 2-23
STRS function, 6-54
String,
  arithmetic, 6-19
  concatenation, 6-18
  constants, 6-14
  integer function, 6-55
  length function, 6-50
  numbers, 6-20
  to numeric conversion, 6-53
  relations, 6-22
  variables, 6-15
SUBMIT command, 3-72,
  option summary, 8-2
Subprogram declaration, 7-48
SUBROUTINE statement, 7-73
Subroutines,
  BASIC, 6-42
  FORTRAN, 7-73
  mathematical, B-1
  PAL8 global calls, 5-22
  semiconditional BASIC, 6-43
Subscripts,
  BASIC, 6-16
  FORTRAN, 7-36
Substring function, 6-51, 6-52
Subtraction,
  BASIC, 6-17
  FORTRAN, 7-39
  PAL8, 5-18
.SV extension, 2-12, 3-59
Symbiont operation,
  CANCEL command, 3-5
  commands, H-1
  CUSP coding conventions, H-2
  REQUEST command, 3-57
  spooler program, 3-54
  writing your own, H-3
Symbol table,
  PAL8, 5-16, 5-35, 5-41
Symbolic editor, 4-1
Symbols,
  FORTRAN, 7-26
  PAL8, 5-13, 5-41
    use as instructions, 5-17
    use as operands, 5-18
    permanent, 5-14
    user-defined, 5-14
SYS device handler, F-4
System Hardware, 1-1
System information,
  additional documentation,
    Preface
  HELP command, 2-22
System date, 3-21
System device, 2-23
System table reset via USR, C-14

T field descriptor, 7-97
  T TTY dash option, 2-17
/T,
  COMPARE option, 3-6
  COPY command option, 3-14
  MAP command option, 3-48
  PAL8 option, 5-9
  RENAME command option, 3-56
  SUBMIT command option, 3-72, 8-2
TAB,
  BASIC function, 6-31
  character,
    PAL8 operator, 5-18
    PAL8 use, 5-12
  FORTRAN, 7-24
    function, 7-110
TAN BASIC function, 6-46
TANH FORTRAN function, 7-112
Tentative file lookup, C-7
Terminals,
  BASIC ESCape sequences, 6-76
  changing attributes, 2-5, 3-62
  control in BASIC, 5-59
  conventions, 2-7
  LA36, 2-6
  operating parameters, 2-6
  VT52, 2-6
  VT100, 2-5
TERMINATE command, 3-75
TEXT pseudo-operator, 5-37
INDEX (Cont.)

TIME FORTRAN function, 7-112
.TM file name extension, 2-12
TRC function, 6-57
.TRUE. logical constant, 7-30
Truncation BASIC string arithmetic, 6-21
TTY device handler, F-5
TYPE command, 3-76
option summary, 3-76
error messages, 3-77

/V,
COPY command option, 3-12
DELETE command option, 3-24
DIRECT command option, 3-25
LIST command option, 3-41
RENAME command option, 3-56
TYPE command option, 3-76
WAL function, 6-53
Value assignment, BASIC, 6-23
Variables,
BASIC, 6-14
FORTRAN, 7-32
numeric (BASIC), 6-15
string (BASIC), 6-15
subscripted (BASIC), 6-16
VLUX device handler, 3-68, F-5
VT52 terminal, 2-6
VT100 terminal, 2-5
VXA0, device handler, characteristics, F-4
installing with SET, 3-68
use, 2-25

/U,
COPY command option, 3-12
CREF command option, 3-19
DIRECT command option, 3-25
SUBMIT command option, 3-72, 8-2
UA/UB/UC commands, 3-78
Unconditional transfer,
BASIC, 6-36
FORTRAN GOTO, 7-57
Underline (_) ODT command, 9-4
Unformatted FORTRAN I/O, 7-80
Unloading RL01K disk packs, 2-5
Uparrow (^),
BASIC operator, 6-17
echo control, 3-63
ODT command, 9-4
PAL8 operator, 5-18
Updating BASIC records, 6-71
Upper-case conversion, 6-54
User Service Routine (USR), C-1
User-defined functions in BASIC, 6-78
USR, C-1
CHAIN function, C-10
CLOSE function, C-8
DECODE function, C-10
ENTER function, C-7
ERROR function, C-11
FETCH function, C-4
INQUIRE function, C-13
LOOKUP function, C-6
RESET function, C-14
restrictions, C-3
standard call, C-1
summary of functions, C-1
USRIN function, C-12
USRROUT function, C-13
USRIN function (USR), C-12
USRROUT function (USR), C-13

/V,
Editor list buffer command, 4-9

/V,
COPY command option, 3-12
DELETE command option, 3-24
DIRECT command option, 3-25
LIST command option, 3-41
RENAME command option, 3-56
TYPE command option, 3-76
WAL function, 6-53
Value assignment, BASIC, 6-23
Variables,
BASIC, 6-14
FORTRAN, 7-32
numeric (BASIC), 6-15
string (BASIC), 6-15
subscripted (BASIC), 6-16
VLUX device handler, 3-68, F-5
VT52 terminal, 2-6
VT100 terminal, 2-5
VXA0, device handler, characteristics, F-4
installing with SET, 3-68
use, 2-25

/W ODT command, 9-8

/W PAL8 option, 5-9
WIDTH SET command option, 3-62
Wildcards, 2-18
WRITE statement, 7-83

/X FORTRAN field descriptor, 7-97

/X,
COMPARE command option, 3-6
CREF command option, 3-19
XLIST pseudo-operator, 5-34
.XOR. FORTRAN operator, 7-42

/Y Editor yank command, 4-10

/Z PAL8 page zero addressing pseudo-operator, 5-29
ZBLOCK pseudo-operator, 5-33
ZERO command, 3-15, 3-79

Index-14
NOTE: This form is for document comments only. DIGITAL will use comments submitted on this form at the company's discretion. If you require a written reply and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Did you find this manual understandable, usable, and well-organized? Please make suggestions for improvement.

____________________________________________________________________________________________________________________________________________________

____________________________________________________________________________________________________________________________________________________

____________________________________________________________________________________________________________________________________________________

____________________________________________________________________________________________________________________________________________________

____________________________________________________________________________________________________________________________________________________

Did you find errors in this manual? If so, specify the error and the page number.

____________________________________________________________________________________________________________________________________________________

____________________________________________________________________________________________________________________________________________________

____________________________________________________________________________________________________________________________________________________

____________________________________________________________________________________________________________________________________________________

____________________________________________________________________________________________________________________________________________________

Please indicate the type of reader that you most nearly represent.

☐ Assembly language programmer
☐ Higher-level language programmer
☐ Occasional programmer (experienced)
☐ User with little programming experience
☐ Student programmer
☐ Other (please specify)____________________________________________________

Name____________________________________Date____________________

Organization________________________________________________________

Street________________________________________________________________

City____________________State___________Zip Code__________

or

Country