July 1978

This manual describes the use of the MACRO assembler on TRAX systems.

TRAX
MACRO Reference Manual

Order No. AA-D340A-TC

OPERATING SYSTEMS AND VERSIONS: TRAX Version 1.0

To order additional copies of this document, contact the Software Distribution Center, Digital Equipment Corporation, Maynard, Massachusetts 01754

digital equipment corporation • maynard, massachusetts

First Printing, April 1978

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.

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	DECsystem-10	MASSBUS
DEC	DECtape	OMNIBUS
PDP	DIBOL	OS/8
DECUS	EDUSYSTEM	PHA
UNIBUS	FLIP CHIP	RSTS
COMPUTER LABS	FOCAL	RSX
COMTEX	INDAC	TYPESET-8
DDT	LAB-8	TYPESET-11
DECCOMM	DECSYSTEM-20	TMS-11
ASSIST-11	RTS-8	ITPS-10

CONTENTS

			PAGE
PREFACE			ix
	0.1	MANUAL OBJECTIVES AND READER ASSUMPTIONS	ix
	0.2	STRUCTURE OF THE DOCUMENT	ix
	0.3	ASSOCIATED DOCUMENTS	x
	0.4	DOCUMENT CONVENTIONS	x
PART	I	INTRODUCTION TO MACRO	
CHAPTER	1	MACRO FEATURES	1-1
	1.1	OVERVIEW OF MACRO	1-1
	1.1.1	Assembly Pass 1	1-1
	1.1.2	Assembly Pass 2	1-2
CHAPTER	2	SOURCE PROGRAM FORMAT	2-1
	2.1	PROGRAMMING STANDARDS AND CONVENTIONS	2-1
	2.2	STATEMENT FORMAT	2-1
	2.2.1	Label Field	2-2
	2.2.2	Operator Field	2-4
	2.2.3	Operand Field	2-4
	2.2.4	Comment Field	2-5
	2.3	FORMAT CONTROL	2-6
PART	II	PROGRAMMING IN MACRO ASSEMBLY LANGUAGE	
CHAPTER	3	SYMBOLS AND EXPRESSIONS	3-1
	3.1	CHARACTER SET	3-1
	3.1.1	Separating and Delimiting Characters	3-2
	3.1.2	Illegal Characters	3-3
	3.1.3	Unary and Binary Operators	3-4
	3.2	MACRO SYMBOLS	3-5
	3.2.1	Permanent Symbols	3-5
	3.2.2	User-Defined and Macro Symbols	3-5
	3.3	DIRECT ASSIGNMENT STATEMENTS	3-7
	3.4	REGISTER SYMBOLS	3-9
	3.5	LOCAL SYMBOLS	3-10
	3.6	CURRENT LOCATION COUNTER	3-11
	3.7	NUMBERS	3-13
	3.8	TERMS	3-14
	3.9	EXPRESSIONS	3-15
CHAPTER	4	RELOCATION AND LINKING	4-1
CHAPTER	5	ADDRESSING MODES	5-1
	5.1	REGISTER MODE	5-1
	5.2	REGISTER DEFERRED MODE	5-2
	5.3	AUTOINCREMENT MODE	5-2

PAGE

5.4 AUTOINCREMENT DEFERRED MODE 5-3
5.5 AUTODECREMENT MODE 5-3
5.6 AUTODECREMENT DEFERRED MODE 5-3
5.7 INDEX MODE 5-4
5.8 INDEX DEFERRED MODE 5-4
5.9 IMMEDIATE MODE 5-4
5.10 ABSOLUTE MODE 5-5
5.11 RELATIVE MODE 5-6
5.12 RELATIVE DEFERRED MODE 5-6
5.13 SUMMARY OF ADDRESSING FORMS 5-7
5.14 BRANCH INSTRUCTION ADDRESSING 5-8
5.15 USING TRAP INSTRUCTIONS 5-8

PART III MACRO DIRECTIVES

CHAPTER 6 GENERAL ASSEMBLER DIRECTIVES 6-1

6.1 LISTING CONTROL DIRECTIVES 6-1
6.1.1 .LIST and .NLIST Directives 6-1
6.1.2 Page Headings 6-8
6.1.3 .TITLE Directive 6-11
6.1.4 .SBTTL Directive 6-11
6.1.5 .IDENT Directive 6-12
6.1.6 .PAGE Directive/Page Ejection 6-12
6.2 FUNCTION DIRECTIVES: .ENABL AND .DSABL 6-13
6.3 DATA STORAGE DIRECTIVES 6-17
6.3.1 .BYTE Directive 6-17
6.3.2 .WORD Directive 6-18
6.3.3 ASCII Conversion Characters 6-19
6.3.4 .ASCII Directive 6-20
6.3.5 .ASCIZ Directive 6-21
6.3.6 .RAD50 Directive 6-22
6.3.7 Temporary Radix-50 Control Operator: ^R 6-23
6.4 RADIX AND NUMERIC CONTROL FACILITIES 6-24
6.4.1 Radix Control and Unary Control Operators 6-24
6.4.1.1 .RADIX Directive 6-24
6.4.1.2 Temporary Radix Control Operators: ^D, ^O, and ^B 6-25
6.4.2 Numeric Directives and Unary Control Operators 6-26
6.4.2.1 .FLT2 and .FLT4 - Floating-Point Storage Directives 6-27
6.4.2.2 Temporary Numeric Control Operators: ^C and ^F 6-27
6.5 LOCATION COUNTER CONTROL DIRECTIVES 6-29
6.5.1 .EVEN Directive 6-29
6.5.2 .ODD Directive 6-29
6.5.3 .BLKB and .BLKW Directives 6-30
6.6 TERMINATING DIRECTIVES 6-30
6.6.1 .END Directive 6-31
6.6.2 .EOT Directive 6-31
6.7 PROGRAM BOUNDARIES DIRECTIVE: .LIMIT 6-31
6.8 PROGRAM SECTIONING DIRECTIVES 6-32
6.8.1 .PSECT Directive 6-32
6.8.1.1 Creating Program Sections 6-36
6.8.1.2 Code or Data Sharing 6-37
6.8.1.3 Memory Allocation Considerations 6-38
6.8.2 .ASECT and .CSECT Directives 6-38
6.9 SYMBOL CONTROL DIRECTIVE: .GLOBL 6-39
6.10 CONDITIONAL ASSEMBLY DIRECTIVES 6-40
6.10.1 Conditional Assembly Block Directives: .IF, .ENDC 6-40
6.10.2 Subconditional Assembly Block Directives: .IFF, .IFT, .IFTF 6-43

			PAGE
	6.10.3	Immediate Conditional Assembly Directive: .IIF	6-45
CHAPTER	7	MACRO DIRECTIVES	7-1
	7.1	DEFINING MACROS	7-1
	7.1.1	.MACRO Directive	7-1
	7.1.2	.ENDM Directive	7-2
	7.1.3	.MEXIT Directive	7-3
	7.1.4	MACRO Definition Formatting	7-3
	7.2	CALLING MACROS	7-3
	7.3	ARGUMENTS IN MACRO DEFINITIONS AND MACRO CALLS	7-4
	7.3.1	Macro Nesting	7-5
	7.3.2	Special Characters in Macro Arguments	7-6
	7.3.3	Passing Numeric Arguments as Symbols	7-6
	7.3.4	Number of Arguments in Macro Calls	7-7
	7.3.5	Creating Local Symbols Automatically	7-7
	7.3.6	Keyword Arguments	7-9
	7.3.7	Concatenation of Macro Arguments	7-10
	7.4	MACRO ATTRIBUTE DIRECTIVES: .NARG, .NCHR, AND .NTYPE	7-11
	7.4.1	.NARG Directive	7-11
	7.4.2	.NCHR Directive	7-12
	7.4.3	.NTYPE Directive	7-13
	7.5	.ERROR AND .PRINT DIRECTIVES	7-14
	7.6	INDEFINITE REPEAT BLOCK DIRECTIVES: .IRP AND .IRPC	7-15
	7.6.1	.IRP Directive	7-16
	7.6.2	.IRPC Directive	7-16
	7.7	REPEAT BLOCK DIRECTIVE: .REPT, .ENDR	7-18
	7.8	MACRO LIBRARY DIRECTIVE: .MCALL	7-18
PART	IV	OPERATING PROCEDURES	
CHAPTER	8	OPERATING PROCEDURES	8-1
	8.1	TRAX OPERATING PROCEDURES	8-1
	8.1.1	Invoking MACRO Under TRAX	8-1
	8.1.2	TRAX Command String Format	8-1
	8.1.3	TRAX Macro Qualifiers	8-2
	8.1.4	TRAX File Specification Qualifiers	8-3
	8.1.5	Cross-Reference Processor (CREF)	8-4
	8.1.6	TRAX MACRO in Batch Mode	8-6
	8.1.7	TRAX Indirect Command Files	8-6
	8.2	TRAX FILE SPECIFICATION FORMAT	8-7
	8.3	MACRO ERROR MESSAGES	8-8
APPENDIX	A	MACRO CHARACTER SETS	A-1
	A.1	ASCII CHARACTER SET	A-1
	A.2	RADIX-50 CHARACTER SET	A-4
APPENDIX	B	MACRO ASSEMBLY LANGUAGE AND ASSEMBLER DIRECTIVES	B-1
	B.1	SPECIAL CHARACTERS	B-1
	B.2	SUMMARY OF ADDRESS MODE SYNTAX	B-1
	B.3	ASSEMBLER DIRECTIVES	B-2
APPENDIX	C	PERMANENT SYMBOL TABLE (PST)	C-1
	C.1	OP CODES	C-1
	C.2	MACRO DIRECTIVES	C-4

			PAGE
APPENDIX D		DIAGNOSTIC ERROR MESSAGE SUMMARY	D-1
	D.1	MACRO ERROR CODES	D-1
APPENDIX E		SAMPLE CODING STANDARD	E-1
	E.1	INTRODUCTION	E-1
	E.2	LINE FORMAT	E-1
	E.3	COMMENTS	E-2
	E.4	NAMING STANDARDS	E-2
	E.4.1	Register Standards	E-2
	E.4.1.1	General Purpose Registers	E-2
	E.4.1.2	Hardware Registers	E-2
	E.4.1.3	Device Registers	E-2
	E.4.2	Processor Priority	E-3
	E.4.3	Other Symbols	E-3
	E.4.4	Using the Standard Symbolics	E-3
	E.4.5	Symbols	E-3
	E.4.5.1	Global Symbols	E-3
	E.4.5.2	Symbol Examples	E-4
	E.4.5.3	Program-Local Symbols	E-4
	E.4.5.4	Macro Names	E-5
	E.5	PROGRAM MODULES	E-5
	E.5.1	General Comments on Programs	E-5
	E.5.2	The Module Preface	E-5
	E.5.3	Formatting the Module Preface	E-7
	E.5.4	Modularity	E-8
	E.5.4.1	Calling Conventions (Inter-Module)	E-8
	E.5.4.2	Exiting	E-9
	E.5.4.3	Intra-Module Calling Conventions	E-9
	E.5.4.4	Success/Failure Indication	E-9
	E.5.4.5	Module Checking Routines	E-9
	E.6	FORMATTING STANDARDS	E-9
	E.6.1	Program Flow	E-9
	E.6.2	Common Exits	E-11
	E.6.3	Code with Interrupts Inhibited	E-12
	E.7	PROGRAM SOURCE FILES	E-12
	E.8	FORBIDDEN INSTRUCTION USAGE	E-12
	E.9	RECOMMENDED CODING PRACTICE	E-13
	E.9.1	Conditional Branches	E-13
	E.10	PDP-11 VERSION NUMBER STANDARD	E-13
	E.10.1	Displaying the Version Identifier	E-14
	E.10.2	Use of the Version Number in the Program	E-15
APPENDIX F		SAMPLE ASSEMBLY AND CROSS REFERENCE LISTING	F-1
INDEX			Index-1

FIGURES

FIGURE	3-1	Assembly Listing Showing Local Symbol Block	3-11
	3-2	Sample Assembly Results	3-12
	6-1	Example of Line Printer Assembly Listing	6-6
	6-2	Example of Terminal Assembly Listing	6-7
	6-3	Listing Produced With Listing Control Directives	6-9
	6-4	Assembly Listing Table of Contents	6-12
	6-5	Example of .ENABL and .DSABL Directives	6-16
	6-6	Example of .BLKB and .BLKW Directives	6-30
	7-1	Example of .IRP and .IRPC Directives	7-17
	8-1	Sample CREF Listing	8-5

PAGE

TABLES

TABLE	3-1	Special Characters Used in MACRO	3-1
	3-2	Legal Separating Characters	3-3
	3-3	Legal Argument Delimiters	3-3
	3-4	Legal Unary Operators	3-4
	3-5	Legal Binary Operators	3-5
	6-1	Symbolic Arguments of Listing Control Directives	6-3
	6-2	Symbolic Arguments of Function Control Directives	6-13
	6-3	Symbolic Arguments of .PSECT Directive	6-33
	6-4	Non-TRAX Program Section Default Values	6-38
	6-5	Legal Condition Tests for Conditional Assembly Directives	6-41
	6-6	Subconditional Assembly Block Directives	6-43
	8-1	File Specification Default Values	8-8

PREFACE

0.1 MANUAL OBJECTIVES AND READER ASSUMPTIONS

MACRO is supported in TRAX to enable Application Programmers to write transaction step tasks (TSTs). In addition it may be used to write Support Environment subroutines which do not perform. This manual provides a reference for the MACRO language. No prior knowledge of the MACRO Relocatable Assembler is assumed.

Although the description of the assembly language is wholly self-contained within this manual, the reader is assumed to be familiar with the PDP-11 processors and related terminology, as presented in the PDP-11 Processor Handbooks. No attempt is made in this document to describe the PDP-11 hardware or the functions of the various PDP-11 instructions.

The development of transaction step tasks (TSTs) also requires knowledge of the library of RMS-11 macros as presented in the TRAX RMS MACRO Reference Manual.

In presenting MACRO, a tutorial bias has been adopted to enlarge upon the reference material. This posture s reflected in the examples and the accompanying commentary describing MACRO language elements in typical applications.

NOTE

Utilization of MACRO will eliminate compatibility of user applications with future members of a planned family of TRAX systems.

0.2 STRUCTURE OF THE DOCUMENT

This manual contains four parts. Part I, consisting of two chapters, briefly introduces MACRO. Chapter 1 lists the key features of MACRO, and Chapter 2 identifies the advantages of following programming standards and conventions. Also described is the format used in coding MACRO source programs.

Part II, consisting of three chapters, presents general information essential to programming with the MACRO assembly language. Chapter 3 describes the symbols, terms, and expressions that form the elements of MACRO instructions. The character set is listed, and the types of programming symbols that may be defined by the user are discussed.

Chapter 4 describes the output of MACRO and presents concepts essential to the proper relocation and linking of object modules by the Linker. Chapter 5 briefly describes how data stored in memory can be accessed and manipulated using the addressing modes recognized by the PDP-11 hardware.

Part III, consisting of two chapters, describes the MACRO directives that control the processing of source statements during assembly. Chapter 6 discusses directives which accomplish generalized MACRO functions, while Chapter 7 deals with directives used in the definition and expansion of macros.

Part IV, consisting only of Chapter 8, presents the operating procedures essential to the assembly, linking, and initiating of MACRO programs.

Finally, several appendixes are provided, supplying additional information of interest to the MACRO programmer.

Appendix A lists the ASCII and Radix-50 character sets that may be used in MACRO programs. Appendix B lists the special characters recognized by MACRO, summarizes the syntax of the various addressing modes used in PDP-11 processors, and briefly describes the MACRO directives in alphabetical order. The permanent symbols that have been defined for use with MACRO are listed alphabetically in Appendix C.

The diagnostic error codes produced by MACRO to identify various types of errors detected during the assembly process are listed alphabetically in Appendix D. Appendix E contains a sample coding standard that is recommended practice in preparing MACRO programs.

0.3 ASSOCIATED DOCUMENTS

The reader should refer to the TRAX documentation directory for descriptions of documents associated with this manual.

0.4 DOCUMENT CONVENTIONS

The symbols defined below are used throughout this manual.

Symbol	Definition
[]	Brackets indicate that the enclosed argument is optional.
|	Vertical bars indicate that a single choice must be made from a list of arguments.
...	Ellipsis indicates optional continuation of an argument list in the form of the last specified argument.
UPPER-CASE CHARACTERS	Upper-case characters indicate elements of the language that must be used exactly as shown.
lower-case characters	Lower-case characters indicate elements of the language that are supplied by the programmer.
(n)	In some instances the symbol (n) is used following a number to indicate the radix. For example, 100(8) indicates that 100 is an octal value, while 100(10) indicates a decimal value.

PART I

INTRODUCTION TO MACRO

CHAPTER 1

MACRO FEATURES

The MACRO Assembler provides the following features:

1. Program and command string control of assembly functions
2. Device and filename specifications for input and output files
3. Error listing on command output device
4. Alphabetized, formatted symbol table listing; optional cross-reference listing of symbols
5. Relocatable object modules
6. Global symbols for linking independent object modules
7. Conditional assembly directives
8. Program sectioning directives
9. User-defined macros and macro libraries
10. Comprehensive system macro library
11. Extensive program and command string control of listing functions
12. An indirect command file facility for controlling the assembly process.

1.1 OVERVIEW OF MACRO

MACRO is a 2-pass assembler. The functions and operations relevant to each assembly pass are described in the following sections.

1.1.1 Assembly Pass 1

The main purpose of assembly pass 1 is to locate and read all required macros from libraries; to build symbol tables and program section tables for the program; while also performing a rudimentary assembly of each source statement.

The first stage of assembly pass 1 is the initialization of all impure data areas that MACRO uses internally for the assembly process. These areas include all dynamic storage areas and buffer areas used as file storage regions.

After initializing memory areas, MACRO issues a call to a system subroutine which transfers a command line into memory. This command line contains the specifications of the files to be used during assembly. After scanning the command line for proper syntax, MACRO initializes the specified output files. These files are opened to determine if valid output file specifications have been passed in the command line. They are then closed to minimize requirements for active file space.

As the assembly process begins, MACRO initiates a routine which retrieves source lines from the input file. If no such file is currently open, as is the case at the beginning of assembly, MACRO opens the next input file specified in the command line previously read and begins to assemble the source statements. MACRO determines the length of each instruction and assembles it accordingly as one word, two words, or three words.

At the end of assembly pass 1, MACRO reopens the output files described above and writes out information that is to be used later by the Linker in linking the object modules. Such information as the object module name, the program version number, and the global symbol directory (GSD) entries for each program section are output to the object file. After writing out the GSD entries for a given program section, MACRO scans through the symbol tables to find all the global symbols that are bound to that particular program section. MACRO then writes out GSD records to the object file for these symbols. This process continues for each program section, bringing to a close assembly pass 1.

1.1.2 **Assembly Pass 2**

As an integral part of pass 2, MACRO simultaneously writes the object records to the output file and generates the assembly listing, followed by the symbol table listing for the program. A cross-reference listing may also be generated. See Section 8.1.4.

Basically, assembly pass 2 consists of the same steps performed in assembly pass 1, except that all source statements containing MACRO-detected errors are flagged with an error code as the assembly listing file is created. The object file that is created as the final consequence of pass 2 contains all the object records, together with relocation records containing information necessary for subsequent Linker linking of the object file.

The information thus passed to the Linker enables the global symbols in the object modules to be associated with absolute or virtual memory addresses, thereby forming an executable body of code.

The user may wish to become familiar with the macro object file format and description. This information is presented in the TRAX Linker Reference Manual (see Section 0.3 in the Preface).

CHAPTER 2

SOURCE PROGRAM FORMAT

2.1 PROGRAMMING STANDARDS AND CONVENTIONS

Assembly level programming deals directly with the host hardware. Hence, great care must be exercised in establishing programming standards and conventions to enable code written by one group to be interchanged easily with another group. Standards provide a number of advantages. When applied to the program development process, standards make the programming effort easier to:

Plan
Comprehend
Test
Modify
Convert.

Even though standards must accommodate local requirements, many aspects of the program development process have universal applicability. The standards common to all of DIGITAL's PDP-11 software products are presented in Appendix E as a model for users. Observance of these standards is beneficial to DIGITAL and its users, by simplifying both communications and the continuing task of software maintenance and enhancement.

2.2 STATEMENT FORMAT

A source program is composed of a sequence of source coding lines. Each line contains a single assembly-language statement. MACRO will accept a source line of 132 characters, but 80 characters is the recommended length, because of constraints imposed by listing format and terminal line size.

A MACRO statement may consist of as many as four fields. These fields are identified by their order of appearance within the statement and/or by specified separating characters between fields. The general format of a MACRO statement is:

```
Label:  Operator Operand        ;Comment(s)
```

The label and comment fields are optional. The operator and operand fields are interdependent, i.e., when both fields are present in a source statement, each field is evaluated by MACRO in the context of the other.

A statement may contain an operator field and no operand field, but the reverse is not true. A statement containing an operand with no operator does not conform to established MACRO coding conventions; such a statement is currently interpreted by MACRO during assembly as an implicit .WORD directive (see Section 6.3.2).

MACRO interprets and processes source program statements one by one, generating one or more binary instructions or data words, or performing a specified assembly process. Blank lines, although legal, have no significance in the source program.

An assembly-language statement must be completed on one source line; no continuation lines are allowed in MACRO.

The tab character can be used in the source statement to format the fields into aligned columns in accordance with DIGITAL's standard source program format, as shown below:

Label - begins in column 1

Operator - begins in column 9

Operand(s) - begin(s) in column 17

Comment(s) - begin(s) in column 33.

For example, the following statement should be formatted in the source program into specific columns, increasing its readability in the assembly listing:

```
REGTST:BIT#MASK,VALUE;COMPARES BITS IN OPERANDS.

1       9       17              33      (columns)

REGTST: BIT     #MASK,VALUE     ;COMPARES BITS IN OPERANDS.
```

The above formatting conventions are not mandatory in coding MACRO programs (free-field coding is permissible). However, it is recommended that source programs be prepared in accordance with these conventions for consistency and clarity.

2.2.1 Label Field

A label is a means of symbolically referring to a location in a program.

A label is a user-defined symbol which is assigned the value of the current location counter and entered into the user-defined symbol table. The current location counter is the means by which MACRO assigns memory addresses to the source program statements as they are encountered during the assembly process. The address value of the label is absolute or relocatable, depending on whether the current program section being assembled is absolute or relocatable. (The concept of program sections and the attributes that may be specified for them are discussed in detail in Section 6.8.)

In the case of an absolute program section, the value of the current location counter is likewise absolute, i.e., its value references an absolute virtual memory address (such as location 100). Similarly, the value of the current location counter in a relocatable program section is also relocatable; however, a relocation bias calculated by the Linker will be added to the apparent value of the current location counter to establish its effective absolute virtual address at execution time.

If present, a label always appears as the first field in a source statement and must be terminated by a colon. For example, if the current location counter value is absolute 100(8), the statement:

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

assigns the value 100(8) to the label ABCD. Subsequent references to this label would then yield a value of absolute 100(8). In this example, if the location counter value were relocatable, the final value of ABCD would be 100(8)+K, where K represents the relocation bias of the program section, as calculated by the Linker at link time.

More than one label may appear within a single label field. Each label so specified is assigned the same address value. For example, if the current location counter value is 100(8), the multiple labels in the following statement:

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

are each assigned the value 100(8).

Multiple labels may also appear on successive lines. For example, the statements

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

likewise cause the same current location counter value to be assigned to all three labels.

Of the two methods of assigning multiple labels shown above, the second is preferred, because consistency of field positioning within the source program improves readability.

A double colon (::) defines the label as a global symbol. Such a label can be referenced by independently-assembled object modules. References to this label in other modules will be resolved by the Linker when the modules are linked as a composite executable task. For example, the statement

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

establishes the label ABCD as a global symbol. The distinguishing attribute of a global symbol is that it can be referenced from within an object module other than the module in which the symbol is defined (see Section 6.9).

The legal characters for defining labels are:

A through Z
0 through 9
. (Period)
$ (Dollar Sign)

NOTE

By convention, the dollar sign ($) and period (.) are reserved for use in defining DIGITAL system software symbols. Therefore these characters should not be used in defining labels in MACRO source programs.

A label may be any length; however, only the first six characters are significant and, therefore, must be unique among all the labels in the source program. All labels are terminatd by a colon (:), which is not considered part of the label. It is a mandatory delimiter. An error code (M) is generated in the assembly listing if the first six characters in two or more labels are the same (see Appendix D).

A symbol used as a label must not be redefined within the source program. If the symbol is redefined, a label with a multiple definition results, causing MACRO to generate an error code (M) in the assembly listing (see Appendix D). Furthermore, any statement in the source program which references a multi-defined label results in an additional diagnostic message; in this case, an error code (D) is generated in the assembly listing (see Appendix D).

2.2.2 Operator Field

The operator field specifies the action to be performed. It may consist of an instruction mnemonic (op code), an assembler directive, or a macro call.

The operator field follows the label field in a source statement. Chapters 6 and 7 describe these three types of operator field entries.

When the operator is an instruction mnemonic, the mnemonic op code specifies the machine instruction to be generated. MACRO then continues with the evaluation of the address(es) of the operand(s) which follow(s). When the operator is a directive, the directive causes MACRO to perform certain control actions or processing operations during the assembly of the source program. When the operator is a macro call, MACRO inserts the code generated by the macro expansion.

The operator field need not be preceded by a label; but it may be preceded by one or more labels and followed by one or more operands and/or a comment. Furthermore, leading and trailing spaces or tabs in the operator field have no significance; such characters serve only to separate the operator field from the preceding and following fields.

An operator is terminated by a space, tab, or any non-RAD50 character, as in the following examples:

```
MOV A,B                 ;THE SPACE TERMINATES THE OPERATOR
                        ;MOV.

MOV     A,B             ;THE TAB TERMINATES THE OPERATOR MOV.

MOV@A,B                 ;THE @ CHARACTER TERMINATES THE
                        ;OPERATOR MOV.
```

Although the statements above are all equivalent in function, the second statement is the recommended form because it conforms to MACRO coding conventions.

2.2.3 Operand Field

When the operator field contains an instruction mnemonic (op code), the operand field specifies those program variables that are to be evaluated/manipulated by the operator. The operand field may also be

used to supply arguments to MACRO directives and macro calls, as described in Chapters 6 and 7, respectively.

Operands may be expressions or symbolic arguments (within the context of the specified operation). Multiple expressions used in the operand field of a MACRO statement must be separated by a comma; multiple symbolic arguments similarly used may be delimited by any legal separator, i.e., a comma, tab, and/or space. An operand should be preceded by an operator field; if it is not, the statement is treated by MACRO as an implicit .WORD directive (see Section 6.3.2).

When the operator field contains an op code, associated operands are always expressions, as shown in the following statement:

```
        MOV     R0,A+2(R1)
```

On the other hand, when the operator field contains a MACRO directive or a macro call, associated operands are normally symbolic arguments, as shown in the following statement:

```
        .MACRO  ALPHA  ARG1,ARG2
```

Refer to the description of each MACRO directive to determine the type and number of operands required in issuing the directive.

The operand field is terminated by a semicolon when the field is followed by a comment. For example, in the following statement:

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

the tab between MOV and A terminates the operator field and defines the beginning of the operand field; a comma separates the operands A and B; and a semicolon terminates the operand field and defines the beginning of the comment field. When no comment field follows, the operand field is terminated by the end of the source line.

2.2.4 Comment Field

The comment field normally begins in column 33 and extends through the end of the line. This field is optional and may contain any ASCII characters except null, RUBOUT, carriage-return, line-feed, vertical-tab or form-feed. All other characters appearing in the comment field, even special characters reserved for use in MACRO, are checked only for ASCII legality and then included in the assembly listing as they appear in the source text.

All comment fields must begin with the semicolon character(;). When lengthy comments extend beyond the end of the source line (column 80), the comment may be resumed in a following line. Such a line must contain a leading semicolon, and it is suggested that the body of the comment be continued in the same columnar position in which the comment began. A comment line can also be included as an entirely separate line within the code body.

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

2.3 FORMAT CONTROL

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

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

No spaces or tabs have been used to separate the fields in this statement. Note the difficulty in recognizing where one field ends and the next begins.

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

This statement conforms to the standard horizontal formatting conventions, i.e., the statement elements are separated into four distinct fields and are therefore easily discernible.

Page formatting and assembly listing considerations are discussed in Chapter 6 in the context of MACRO directives that may be specified to accomplish desired formatting operations. Appendix E describes the coding conventions used in all DIGITAL PDP-11 operating system software.

PART II

PROGRAMMING IN MACRO ASSEMBLY LANGUAGE

CHAPTER 3

SYMBOLS AND EXPRESSIONS

This chapter describes the components of MACRO instructions. The character set, the conventions observed in constructing symbols, and the use of numbers, operators, terms and expressions are discussed as they relate to MACRO programming.

3.1 CHARACTER SET

The following characters are legal in MACRO source programs:

1. The letters A through Z. Both upper- and lower-case letters are acceptable, although, upon input, lower-case letters are converted to upper-case (see Section 6.2, .ENABL LC).

2. The digits 0 through 9.

3. The characters . (period) and $ (dollar sign). These characters are reserved for use as Digital Equipment Corporation system program symbols.

4. The special characters listed in Table 3-1.

Table 3-1
Special Characters Used in MACRO

Character	Designation	Function
:	Colon	Label terminator.
::	Double colon	Label terminator; defines the label as a global label.
=	Equal sign	Direct assignment operator; and macro keyword indicator.
==	Double equal sign	Direct assignment operator; defines the symbol as a global symbol.
%	Percent sign	Register term indicator.
	Tab	Item or field terminator.
	Space	Item or field terminator.

(continued on next page)

Table 3-1 (Cont.)
Special Characters Used in MACRO

Character	Designation	Function
#	Number sign	Immediate expression indicator.
@	At sign	Deferred addressing indicator.
(	Left parenthesis	Initial register indicator.
)	Right parenthesis	Terminal register indicator.
.	Period	Current location counter
,	Comma	Operand field separator.
;	Semicolon	Comment field indicator.
<	Left angle bracket	Initial argument or expression indicator.
>	Right angle bracket	Terminal argument or expression indicator.
+	Plus sign	Arithmetic addition operator or autoincrement indicator.
-	Minus sign	Arithmetic subtraction operator or autodecrement indicator.
*	Asterisk	Arithmetic multiplication operator.
/	Slash	Arithmetic division operator.
&	Ampersand	Logical AND operator.
!	Exclamation point	Logical inclusive OR operator.
"	Double quote	Double ASCII character indicator.
'	Single quote	Single ASCII character indicator; or concatenation indicator.
^	Up arrow or circumflex	Universal unary operator or argument indicator.
\	Backslash	Macro call numeric argument indicator.

3.1.1 Separating and Delimiting Characters

Legal separating characters and legal argument delimiters are defined below in Tables 3-2 and 3-3 respectively.

Table 3-2
Legal Separating Characters

Character	Definition	Usage
Space	One or more spaces and/or tabs	A space is a legal separator between instruction fields and between symbolic arguments within the operand field. Spaces within expressions are ignored (see Section 3.9).
,	Comma	A comma is a legal separator between symbolic arguments within the operand field. Multiple expressions used in the operand field must be separated by a comma.

Table 3-3
Legal Argument Delimiters

Character	Definition	Usage
<...>	Paired angle brackets	Paired angle brackets may be used anywhere in a program to enclose an expression for treatment as a single term. Paired angle brackets are also used to enclose a macro argument, particularly when that argument contains separating characters (see Section 7.3).
^x...x	Up-arrow (unary operator) construction, where the up-arrow is followed by an argument that is bracketed by any paired printing characters (x).	This construction is equivalent in function to the paired angle brackets described above and is generally used only where the argument itself contains angle brackets.

3.1.2 Illegal Characters

A character is determined to be illegal for one of two reasons:

1. A character is not an element of the recognized MACRO character set. A character of this kind is replaced in the listing by a question mark, and an error code (I) is printed in the assembly listing (see Appendix D). The exception to this is an embedded null which, when detected, terminates the scan of the current line.

2. A legal MACRO character is illegal in the context of its usage within the source statement, i.e., its syntax is illegal or questionable. Such a character causes an error code (Q) to be printed in the assembly listing.

3.1.3 Unary and Binary Operators

Legal MACRO unary operators are described in Table 3-4. Unary operators are used in connection with single terms (arguments or operands) to indicate an action to be performed on that term during assembly. A term preceded by a unary operator is considered to contain that operator. The term so specified thus becomes a value which can be used alone or as an element of an expression.

Table 3-4
Legal Unary Operators

Unary Operator	Explanation	Example	Effect
+	Plus sign	+A	Produces the positive value of A.
-	Minus sign	-A	Produces the negative (2's complement) value of A.
^	Up-arrow, universal unary operator. (This usage is described in detail in Section 6.4.)	^C24	Produces the 1's complement value of 24(8).
		^D127	Interprets 127 as a decimal number.
		^F3.0	Interprets 3.0 as a 1-word, floating-point number.
		^O34	Interprets 34 as an octal number.
		^B11000111	Interprets 11000111 as a binary number.
		^RABC	Evaluates ABC in Radix-50 form.

Unary operators can be used adjacent to each other or in constructions involving multiple terms, as shown below:

```
-^D50    (Equivalent to -<^D50>)
^C^O12   (Equivalent to ^C<^O12>)
```

Legal MACRO binary operators are described in Table 3-5. In contrast to unary operators, binary operators specify actions to be performed on multiple items or terms within an expression. Table 3-5 shows the relationships that can be established between expression terms through the use of binary operators.

Table 3-5
Legal Binary Operators

Binary Operator	Explanation	Example
+	Addition	A+B
-	Subtraction	A-B
*	Multiplication	A*B (16-bit product returned)
/	Division	A/B (16-bit quotient returned)
&	Logical AND	A&B
!	Logical inclusive OR	A!B

All binary operators have equal priority. Items or terms can be grouped for evaluation within an expression by enclosing them within angle brackets. Terms so enclosed are evaluated first, and remaining operations are performed from left to right, as shown in the examples below:

```
.WORD   1+2*3           ;EQUALS 11(8).
.WORD   1+<2*3>         ;EQUALS 7(8).
```

3.2 MACRO SYMBOLS

Three types of symbols may be defined for use within MACRO source programs: permanent symbols, user-defined symbols, and macro symbols. MACRO maintains three types of symbol tables: the Permanent Symbol Table (PST), the User Symbol Table (UST), and the Macro Symbol Table (MST). The PST contains all the permanent symbols defined within (and thus automatically recognized by) MACRO and is part of the MACRO task image. The UST and MST are constructed as the source program is assembled.

3.2.1 Permanent Symbols

Permanent symbols consist of the instruction mnemonics (see Appendix C) and MACRO directives (see Chapters 6 and 7 and Appendix B). These symbols are a permanent part of the MACRO task image and need not be defined before being used in the operator field of a MACRO source statement (see Section 2.2.2).

3.2.2 User-Defined and Macro Symbols

User-defined symbols are those symbols treated by the programmer as labels (see Section 2.2.1) or that are equated to a specific value through a direct assignment statement (see Section 3.3) or appear as macro names or dummy arguments. These symbols are added to the User Symbol Table as they are encountered during assembly. Macro symbols are those symbols used as macro names (see Section 7.1). Similarly, these symbols are added to the Macro Symbol Table as they are encountered during assembly.

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

NOTE

The dollar sign ($) and period (.) characters are reserved for use in defining Digital Equipment Corporation system software symbols. For example, READ$ is a file-processing system macro. The user is cautioned not to employ these characters in constructing user-defined symbols or macro symbols in order to avoid possible conflicts with existing or future Digital Equipment Corporation system software symbols.

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

1. The first character of a symbol must not be a number (except in the case of local symbols; see Section 3.5).

2. The first six characters of a symbol must be unique.

3. A symbol can be written with more than six legal characters, but the seventh and subsequent characters are checked only for ASCII legality and are not otherwise evaluated or recognized by MACRO.

4. Spaces, tabs, and illegal characters must not be embedded within a symbol. The legal MACRO character set is defined in Section 3.1.

The value of a symbol depends upon its use in the program. When a symbol appears in the operator field, it may be any one of the three symbol types described above i.e., permanent, user-defined, macro. To determine the value of an operator-field symbol, MACRO searches the symbol tables in the following order:

1. Macro Symbol Table

2. Permanent Symbol Table

3. User-Defined Symbol Table

This search order allows redefinition of Permanent Symbol Table entries as macro symbols. That is, permanent symbols may be used as macro symbols. But the user must keep in mind the sequence in which the search for symbols is performed in order to avoid incorrect interpretation of the symbol's use.

When a symbol appears in the operand field, the User-Defined Symbol Table is searched first, then the Permanent Symbol Table is searched.

Depending on their use in the source program, user-defined symbols have either a local (internal) attribute or a global (external) attribute.

Normally, MACRO treats all user-defined symbols as local, that is, their definition is limited to the module in which they appear.

However, symbols can be explicitly declared to be global symbols through one of three methods:

1. Use of the .GLOBL directive (see Section 6.9).

2. Use of the double colon (::) in defining a label (see Section 2.2.1).

3. Use of the double equal (==) sign in a direct assignment statement (see Section 3.3).

All symbols within a module that remain undefined at the end of assembly are treated as default global references.

NOTE

Undefined symbols at the end of assembly are assigned a value of 0 and placed into the user-defined symbol table as undefined default global references. If the .DSABL GBL directive is in effect, however, (see Section 6.2), the automatic global reference default function of MACRO is inhibited, causing the statement containing the undefined symbol to be flagged with an error code (U) in the assembly listing (see Appendix D).

Global symbols provide linkages between independently-assembled object modules within the task image. A global symbol defined as a label, for example, may serve as an entry-point address to another section of code within the task image. Such symbols are referenced from other source modules in order to transfer control throughout the task's execution. These global symbols are resolved by the Linker at link time, ensuring that the resulting task image is a logically coherent and complete body of code.

3.3 DIRECT ASSIGNMENT STATEMENTS

A direct assignment statement allows you to equate a symbol to a specific value. When a direct assignment statement is first used to define a symbol, that symbol is entered into the User-Defined Symbol Table. A symbol defined in this manner may be redefined in a subsequent direct assignment statement by assigning a new value to the previously-defined symbol.

The general format for a direct assignment statement is:

symbol=expression

or

symbol==expression

where: expression - can have only one level of forward reference (see 5. below).

- cannot contain an undefined global reference.

A direct assignment statement embodying the double equal (==) sign, as shown above, defines the symbol as global (see Section 6.9).

The following examples illustrate the coding of direct assignment statements:

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

B=A-1&MASKLOW                      ;THE SYMBOL B IS EQUATED TO THE
                                   ;VALUE OF THE ENTIRE EXPRESSION
                                   ;WHICH FOLLOWS.

C:
D=.                                ;THE SYMBOL D IS EQUATED TO ., AND
E:      MOV     #1,ABLE            ;THE LABELS C AND E ARE ASSIGNED A
                                   ;VALUE THAT IS EQUAL TO THE LOCATION
                                   ;OF THE MOV INSTRUCTION.
```

The last of the three examples above is provided only to illustrate the performance of MACRO in such situations. See Section 3.6 for a description of the period (.) as the current location counter symbol.

The following conventions apply to the coding of direct assignment statements:

1. An equal sign (=) or double equal sign (==) must separate the symbol from the expression defining the symbol's value. Spaces preceding and/or following the direct assignment operators, although permissible, have no significance in the resulting value.

2. The symbol being assigned in a direct assignment statement is placed in the label field.

3. Only one symbol can be defined in a single direct assignment statement.

4. A direct assignment statement may be followed only by a comment field.

5. Only one level of forward referencing is allowed, as shown in the following example:

```
X=Y     (Illegal forward reference)

y=z     (Legal forward reference)

Z=1
```

The above example would result in the generation of an error code (U) in the assembly listing on the line containing the illegal forward reference.

Although one level of forward referencing is allowed for local symbols, a global symbol defined in a direct assignment statement must not contain a forward reference, i.e., the global assignment expression must not itself contain an undefined reference to another symbol. Such a forward reference is illegal, causing an error code (A) to be generated in the assembly listing.

3.4 REGISTER SYMBOLS

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

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

where % indicates a reference to a register rather than a location. The digit specifying the register can be replaced by any legal, absolute term that can be evaluated during the first assembly pass. Use standard symbolic names for all register references.

The register definitions listed below are automatically assigned by MACRO, i.e., these definitions are the normal default values and remain valid for all register references within the source program.

```
R0=%0                           ;REGISTER 0 DEFINITION.
R1=%1                           ;REGISTER 1 DEFINITION.
R2=%2                           ;REGISTER 2 DEFINITION.
R3=%3                           ;REGISTER 3 DEFINITION.
R4=%4                           ;REGISTER 4 DEFINITION.
R5=%5                           ;REGISTER 5 DEFINITION.
SP=%6                           ;STACK POINTER DEFINITION.
pc=%7                           ;PROGRAM COUNTER DEFINITION.
```

Note that registers 6 and 7 are given special names because of their unique system functions.

A register symbol may be defined in a direct assignment statement appearing in the program. The defining expression of a register symbol must be a legal, absolute value. Although you can reassign the standard register symbols through the use of the .DSABL REG directive (see Section 6.2), this practice is not recommended. An attempt to redefine a default register symbol without first specifying the .DSABL REG directive to override the normal register definitions causes that assignment statement to be flagged with an error code (R) in the assembly listing. The symbolic default names assigned to the registers, as listed above, are the conventional names used in all DIGITAL-supplied PDP-11 system programs. For this reason, you are well advised to follow these conventions.

All non-standard register symbols must be defined before they are referenced in the source program. A register expression less than 0 or greater than 7 is flagged with an error code (R) in the assembly listing.

The % character may be used with any legal term or expression to specify a register. For example, the statement

```
        CLR     %3+1
```

is equivalent in function to the statement

```
        CLR     %4
```

and clears the contents of register 4.

In contrast, the statement

```
        CLR     4
```

clears the contents of virtual memory location 4.

3.5 LOCAL SYMBOLS

Local symbols are specially formatted symbols used as labels within a block of coding that has been delimited as a local symbol block. Local symbols are of the form n$, where n is a decimal integer from 1 to 65535, inclusive. Examples of local symbols are:

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

A local symbol block is delimited in one of three ways:

1. The range of a local symbol block usually consists of those statements between two normally-constructed symbolic labels (see Figure 3-1). Note that a statement of the form:

   ALPHA=expression

   is a direct assignment statement (see Section 3.3), but does not create a label and thus does not delimit the range of a local symbol block.

2. The range of a local symbol block is normally terminated upon encountering a .PSECT, .CSECT, or .ASECT directive in the source program (see Figure 3-1).

3. The range of a local symbol block is delimited through MACRO directives, as follows:

   Starting delimiter: .ENABL LSB (see Section 6.2)

   Ending delimiter: .ENABL LSB

   or

   .DSABL LSB (see Section 6.2)

   followed by one of: Symbolic label

   .PSECT (see Section 6.8.1)
   .CSECT (see Section 6.8.2)
   .ASECT (see Section 6.8.2)

Local symbols provide a convenient means of generating labels for branch instructions and other such references within a local symbol block. Using local symbols reduces the possibility of symbols with multiple definitions appearing within a user program. In addition, the use of local symbols differentiates entry-point labels from local labels, since local symbols cannot be referenced from outside their respective local symbol block. Thus, local symbols of the same name can appear in other local symbol blocks without conflict. Local symbols do not appear in cross-reference listings.

Local symbols require less symbol table space than other types of symbols. Their use is recommended. When defining local symbols, use the range from 1$ to 63$ first, then the range from 128$ to 65535$. Local symbols within the range 64$ through 127$, inclusive, can be generated automatically as a feature of MACRO. Such local symbols are useful in the expansion of macros during assembly and are described in detail in this context in Section 7.3.5.

Be sure to avoid multiple definitions of local symbols within the same local symbol block. For example, if the local symbol 10$ is defined two or more times within the same local symbol block, each symbol represents a different address value. Such a multi-defined symbol causes an error code (P) to be generated in the assembly listing.

For examples of local symbols and local symbol blocks as they appear in a source program, see Figure 3-1.

```
121                                     ;
122                                     ; PROGRAM INITIALIZATION CODE
123                                     ;
124
125 000000                                      .PSECT  XCTPRG,GBL
126 000000  012700  000000'     XCTPRG::MOV     #IMPURE,R0      ;IMPURE DATA INITIALIZATION
127 000004  005020              1$:     CLR     (R0)+
128 000006  022700  000000'             CMP     #IMPURT,R0
129 000012  101374                      BHI     1$
130
131 000000                                      .PSECT  XCTPAS,GBL
132 000000  012700  000000'     XCTPAS::MOV     #IMPPAS,R0      ;PASS INITIALIZATION
133 000004  005020              1$:     CLR     (R0)+
134 000006  022700  000000'             CMP     #IMPPAT,R0
135 000012  101374                      BHI     1$
136
137 000000                                      .PSECT  XCTLIN,GBL
138 000000  012700  000000'     XCTLIN::MOV     #IMPLIN,R0      ;LINE INITIALIZATION
139 000004  005020              1$:     CLR     (R0)+
140 000006  022700  000000'             CMP     #IMPLIT,R0
141 000012  101374                      BHI     1$
142
```

Figure 3-1 Assembly Listing Showing Local Symbol Block

3.6 CURRENT LOCATION COUNTER

The period (.) is the symbol for the current location counter. When used in the operand field of an instruction, it represents the address of the first word of the instruction, as shown in the first example below. When used in the operand field of a MACRO directive, it represents the address of the current byte or word, as shown in the second example below.

```
A:      MOV     #.,R0           ;THE PERIOD (.) REFERS TO THE ADDRESS
                                ;OF THE MOV INSTRUCTION.
```

(The function of the # symbol is explained in Section 5.9.)

```
SAL=0
        .WORD   177535,.+4,SAL  ;THE OPERAND .+4 IN THE .WORD
                                ;DIRECTIVE REPRESENTS A VALUE
                                ;THAT IS STORED AS THE SECOND
                                ;OF THREE WORDS DURING
                                ;ASSEMBLY.
```

Assume that the current value of the location counter is 500. During assembly, MACRO reserves storage in response to the .WORD directive (see Section 6.3.2), beginning with location 500. The operands accompanying the .WORD directive determine the values so stored. The

value 177535 is thus stored in location 500. The value represented by .+4 is stored in location 502; this value is derived as the current value of the location counter (which is now 502), plus the absolute value 4, thereby depositing the value 506 in location 502. Finally, the value of SAL, previously equated to 0, is deposited in location 504.

Figure 3-2 illustrates the result of the example.

LOCATION	CONTENTS
500	177535
502	506
504	0

Figure 3-2 Sample Assembly Results

At the beginning of each assembly pass, MACRO resets the location counter. Normally, consecutive memory locations are assigned to each byte of object data generated. However, the value of the location counter can be changed through a direct assignment statement of the following form:

.=expression

Similar to other MACRO symbols, the current location counter symbol (.) has an attribute of relocatability associated with it: it is either absolute or relocatable, depending on the specific such attribute of the current program section. (A program section and its attributes are defined through the use of the .PSECT directive described in Section 6.8.1.) The existing attribute (or mode) of the current location counter cannot be changed by specifying a defining expression having a different attribute.

Furthermore, such a defining expression must not force the location counter into another program section (.PSECT area), even though the program sections so involved may both be absolute or relocatable. The expression defining the location counter value must not contain a forward reference, i.e., the expression must not contain a reference to a symbol that is not previously defined. Such violations constitute a general assembly error, resulting in an error code (A) in the assembly listing.

Thus, the attribute (or mode) of the current location counter takes on the attribute of the current program section. Therefore, its attribute from program section to program section can be changed only through the program sectioning directives (.PSECT, .ASECT, and .CSECT), as described in Section 6.8.

The following coding illustrates the use of the current location counter:

```
        .ASECT
.=500                           ;SET LOCATION COUNTER TO
                                ;ABSOLUTE 500(OCTAL).
FIRST:  MOV     .+10,COUNT      ;THE LABEL "FIRST" HAS THE VALUE
                                ;500(OCTAL).
                                ;.+10 EQUALS 510(OCTAL). THE
                                ;CONTENTS OF THE LOCATION
```

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

Storage areas may be reserved in the program by advancing the location counter. For example, if the current value of the location counter is 1000, each of the following statements:

```
.=.+40
```

or

```
.BLKB 40
```

or

```
.BLKW 20
```

reserves 40(8) bytes of storage space in the source program. The .BLKB and .BLKW directives, however, are recommended as the preferred ways to reserve storage space (see Section 6.5.3).

3.7 NUMBERS

MACRO assumes that all numbers in the source program are to be interpreted in octal radix, unless otherwise specified. An exception to this is that operands associated with Floating Point Processor instructions and Floating Point Data directives are treated as decimal (see Section 6.4.2). This default radix can be altered with the .RADIX directive (see Section 6.4.1.1). Also, individual numbers can be designated as decimal, binary, or octal numbers through temporary radix control operators (see Section 6.4.1.2).

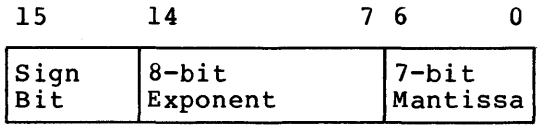
For every statement in the source program that contains a digit that is not in the current radix, an error code (N) is generated in the assembly listing. However, MACRO continues with the scan of the statement and evaluates each such number encountered as a decimal value.

Negative numbers must be preceded by a minus sign; MACRO translates such numbers into two's complement form. Positive numbers may (but need not) be preceded by a plus sign.

A number containing more than 16 significant bits, i.e., greater than 177777(8), is truncated from the left and flagged with an error code (T) in the assembly listing.

Numbers are always considered to be absolute values, i.e., they are not relocatable.

Single-word floating-point numbers may be generated with the ^F operator (see Section 6.4.2.2) and are stored in the following format:



Refer to the appropriate PDP-11 Processor Handbook for details of the floating-point number format.

3.8 TERMS

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

1. A number, as defined in Section 3.7, whose 16-bit value is used.

2. A symbol, as defined in Section 3.2. Symbols are evaluated as follows:

   a. A period (.) specified in an expression causes the value of the current location counter to be used.

   b. A defined symbol is located in the User-Defined Symbol Table (UST) and its value is used.

   c. A permanent symbol's basic value is used, with zero substituted for the addressing modes. (Appendix C lists all op codes and their values.)

   d. An undefined symbol is assigned a value of zero and inserted in the User-Defined Symbol Table as an undefined default global reference. If the .DSABL GBL directive (see Section 6.2) is in effect, the automatic global reference default function of MACRO is inhibited, in which case, the statement containing the undefined symbol is flagged with an error code (U) in the assembly listing.

3. An ASCII conversion operation is performed, using either a single quote followed by a single ASCII character, or a double quote followed by two ASCII characters. This type of expression construction is explained in detail in Section 6.3.3.

4. A term may also be an expression enclosed in angle brackets (<>). Any expression so enclosed is evaluated and reduced to a single term before the remainder of the expression in which it appears is evaluated. Angle brackets, for example, may be used to alter the left-to-right evaluation of expressions (as in A*B+C versus A*<B+C>), or to apply a unary operator to an entire expression (as in -<A+B>).

3.9 EXPRESSIONS

Expressions are combinations of terms joined together by binary operators (see Table 3-5) and which reduce to a 16-bit expression value. The evaluation of an expression includes the determination of its attributes. A resultant expression value may be any one of four types (as described later in this section): absolute, relocatable, external, or complex relocatable.

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

```
-+-A
```

is equivalent to:

```
-<+<-A>>
```

A missing term, expression, or external symbol is interpreted as a zero. A missing or illegal operator terminates the expression analysis, causing an error code (A) or (Q), or both, to be generated in the assembly listing, depending on the context of the expression itself. For example, the expression:

```
TAG ! LA 177777
```

is evaluated as

```
TAG ! LA
```

because the first non-blank character following the symbol LA is not a legal binary operator, an expression separator (i.e., a comma), or an operand field terminator (i.e., a semicolon or the end of the source line). It should be noted that spaces within expressions are ignored.

The value of an external expression is equal to the value of the absolute part of that expression. For example, the expression EXTERN+A, where "EXTERN" is an external symbol, has a value at assembly-time that is equal to the value of the internal symbol A. This expression, however, when evaluated by the Linker at link time takes on the resolved value of the symbol EXTERN, plus the value of symbol A.

Expressions, when evaluated by MACRO, are determined to be one of four types: absolute, relocatable, external (or global), or complex relocatable. The following distinctions are important:

1. An expression is absolute if its value is fixed. An expression whose terms are numbers and ASCII conversion characters will reduce to an absolute value. A relocatable expression or term minus a relocatable term, where both elements being evaluated belong to the same program section, are also absolute, since such an expression is reduced to a single term by MACRO upon completion of the expression scan. For example, the expression TAG2-TAG1, where both TAG1 and TAG2 are defined in the same program section, is an absolute expression.

2. An expression is relocatable if its value is fixed relative to the base address of the program section in which it appears, but it will have an offset value added at task-build time. Expressions whose terms contain labels defined in relocatable program sections will have a relocatable value; similarly, a period (.) in a relocatable program section, representing the value of the current location counter, will also have a relocatable value.

3. An expression is external (or global) if it contains a single global reference (plus or minus an absolute expression value) that is not defined within the current program. Thus, an external expression is only partially defined following assembly and must be resolved by the Linker at link time.

4. An expression is complex relocatable if any of the following conditions applies:

   - It contains a global reference and a relocatable symbol.

   - It contains more than one global reference.

   - It contains relocatable terms belonging to different program sections.

   - The value resulting from the expression has more than one level of relocation. For example, if the relocatable symbols TAG1 and TAG2 associated with the same program section are specified in an expression construction in the form TAG1+TAG2, two levels of relocation would be introduced, since each symbol is evaluated in terms of the relocation bias in effect for the program section.

   - An operation other than addition is specified on an undefined global symbol.

   - An operation other than addition, subtraction, negation, or complementation is specified for a relocatable value.

The evaluation of relocatable, external, and complex relocatable expressions is completed at link time.

CHAPTER 4

RELOCATION AND LINKING

The output of MACRO is an object module which must be processed by the Linker before it can be loaded and executed. Essentially, the Linker fixes (i.e., makes absolute) the values of external or relocatable symbols in the object module, thus transforming the object module, or several such object modules, into an executable task image. This process is called linking.

To enable the Linker to fix the value of an expression, MACRO issues certain directives to the Linker, together with other required parameters. In the case of relocatable expressions in the object module, the Linker adds the base of the associated relocatable program section to the value of the relocatable expression provided by MACRO. In the case of external expression values, the Linker determines the value of the external term in the expression (since the external symbol must be defined in one of the other object modules being linked together) and then adds it to the absolute portion of the external expression, as provided by MACRO.

All instructions that require modification by the Linker are flagged in the assembly listing, as illustrated in the example below. The apostrophe (') following the octal expansion of the instruction indicates that simple relocation is required; the letter G indicates that the value of an external symbol must be added to the absolute portion of an expression; and the letter C indicates that complex relocation analysis by the Linker is required in order to fix the value of the expression.

EXAMPLE:

```
005065  CLR     EXTERN(R5)          ;THE VALUE OF THE "EXTERN" SYMBOL IS
000000G                             ;ASSEMBLED AS ZERO AND IS TO BE
                                    ;RESOLVED BY THE TASK BUILDER.

005065  CLR     EXTERN+6(R5)        ;THE VALUE OF THE SYMBOL "EXTERN"
000006G                             ;IS TO BE RESOLVED BY
                                    ;THE TASK BUILDER AND ADDED TO
                                    ;THE ABSOLUTE PORTION (+6) OF
                                    ;THE EXPRESSION.

005065  CLR     RELOC(R5)           ;ASSUMING THAT THE VALUE OF THE
000040'                             ;SYMBOL "RELOC" IS RELOCATABLE
                                    ;40, THE TASK BUILDER WILL ADD A
                                    ;RELOCATION BIAS TO THIS VALUE.

005065  CLR     -<EXTERN+RELOC>(R5) ;THIS EXPRESSION IS COMPLEX
000000C                             ;RELOCATABLE BECAUSE IT REQUIRES
                                    ;THE NEGATION OF AN EXPRESSION
                                    ;THAT CONTAINS A GLOBAL (EXTERN)
                                    ;REFERENCE AND A RELOCATABLE TERM.
```

For a complete description of object records output by MACRO, refer to the TRAX Linker Reference Manual (see Section 0.3 in the Preface).

CHAPTER 5

ADDRESSING MODES

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

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

> "whenever the processor implicitly uses the program counter (PC) to fetch a word from memory, the program counter is automatically incremented by 2 after the fetch operation is completed."

In the case of 2- or 3-word instructions, the processor uses the PC to fetch the following words as well.

The following symbols are used in describing addressing modes throughout this chapter:

1. E is any expression, as defined in Chapter 3.

2. R is a register expression, i.e., any expression containing a term preceded by a percent sign (%) or a symbol previously equated to such a term, as shown in the examples below:

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

   The symbol R may also represent any of the normal default register definitions (see Section 3.4).

3. ER is a register expression or an absolute expression in the range 0 to 7, inclusive.

4. A is a general addressing specification which produces a 6-bit mode address field, as described in the PDP-11 Processor Handbooks. The addressing specification, A, is described in terms of E, R, and ER, as defined above. Each addressing specification within this section is illustrated using either the single operand instruction CLR or the double operand instruction MOV.

5.1 REGISTER MODE

The register itself (R) contains the operand to be manipulated by the instruction.

Format for A: R

Example:

```
        CLR     R3              ;CLEARS REGISTER 3.
```

5.2 REGISTER DEFERRED MODE

The register (R) contains the address of the operand to be manipulated by the instruction.

Format for A: @R or (ER)

Examples:

```
        CLR     @R1             ;ALL THESE INSTRUCTIONS CLEAR
        CLR     (R1)            ;THE WORD AT THE ADDRESS
        CLR     (1)             ;CONTAINED IN REGISTER 1.
```

5.3 AUTOINCREMENT MODE

The contents of the register (ER) are incremented immediately after being used as the address of the operand (see Note below).

Format for A: (ER)+

Examples:

```
        CLR     (R0)+           ;EACH INSTRUCTION CLEARS
        CLR     (R4)+           ;THE WORD AT THE ADDRESS
        CLR     (R2)+           ;CONTAINED IN THE SPECIFIED
                                ;REGISTER AND INCREMENTS
                                ;THAT REGISTER'S CONTENTS
                                ;BY TWO.
```

NOTE

Certain special instruction/address mode combinations, which are rarely or never used, do not operate exactly the same on all PDP-11 processors, as described below.

In the autoincrement mode, both the JMP and JSR instructions autoincrement the register before its use on the PDP-11/40, but not on the PDP-11/45 or 11/10.

In double operand instructions having the addressing form Rn,(Rn)+ or Rn,-(Rn), where the source and destination registers are the same, the source operand is evaluated as the autoincremented or autodecremented value, but the destination register, at the time it is used, still contains the originally-intended effective address. In the following example, as executed on

the PDP-11/40, Register 0 originally contains 100(8):

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

MOV     R0,-(R0)        ;THE QUANTITY 76 IS MOVED
                        ;TO LOCATION 100.
```

The use of these forms should be avoided, since they are not compatible with the entire family of PDP-11 processors.

An error code (Z) is printed in the assembly listing with each instruction which is not compatible among all members of the PDP-11 family.

5.4 AUTOINCREMENT DEFERRED MODE

The register (ER) contains a pointer to the address of the operand. The contents of the register are incremented after being used as a pointer.

Format for A: @(ER)+

Example:

```
        CLR     @(R3)+          ;THE CONTENTS OF REGISTER 3 POINT
                                ;TO THE ADDRESS OF A WORD TO BE
                                ;CLEARED BEFORE THE CONTENTS OF THE
                                ;REGISTER ARE INCREMENTED BY TWO.
```

5.5 AUTODECREMENT MODE

The contents of the register (ER) are decremented before being used as the address of the operand (see Note above in Section 5.3).

Format for A: -(ER)

Examples:

```
        CLR     -(R0)           ;DECREMENT THE CONTENTS OF THE SPECI-
                                ;FIED REGISTER (0, 3, OR 2) BY TWO
        CLR     -(R3)           ;BEFORE USING ITS CONTENTS
        CLR     -(R2)           ;AS THE ADDRESS OF THE WORD TO BE
                                ;CLEARED.
```

5.6 AUTODECREMENT DEFERRED MODE

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

Format for A: @-(ER)

Example:

```
        CLR     @-(R2)          ;DECREMENT THE CONTENTS OF
                                ;REGISTER 2 BY TWO BEFORE
                                ;USING ITS CONTENTS AS A POINTER
                                ;TO THE ADDRESS OF THE WORD TO BE
                                ;CLEARED.
```

5.7 INDEX MODE

The value of an expression (E) is stored as the second or third word of the instruction. The effective address of the operand is calculated as the value of E, plus the contents of register ER. The value E is the offset of the instruction, and the contents of register ER form the base.

Format for A: E(ER)

Examples:

```
        CLR     X+2(R1)         ;THE EFFECTIVE ADDRESS OF THE WORD
                                ;TO BE CLEARED IS X+2, PLUS THE
                                ;CONTENTS OF REGISTER 1.
        MOV     R0,-2(R3)       ;THE EFFECTIVE ADDRESS OF THE
                                ;DESTINATION LOCATION IS -2, PLUS
                                ;THE CONTENTS OF REGISTER 3.
```

5.8 INDEX DEFERRED MODE

An expression (E), plus the contents of a register (ER), yields a pointer to the address of the operand. As in index mode above, the value E is the offset of the instruction, and the contents of register ER form the base.

Format for A: @E(ER)

Example:

```
        CLR     @114(R4)        ;IF REGISTER 4 CONTAINS 100, THIS
                                ;VALUE, PLUS THE OFFSET 114, YIELDS
                                ;THE POINTER 214.  IF LOCATION 214
                                ;CONTAINS THE ADDRESS 2000, LOCATION
                                ;2000 WOULD BE CLEARED.
```

5.9 IMMEDIATE MODE

Immediate mode allows the operand itself (E) to be stored as the second or third word of the instruction. This mode is assembled as an autoincrement of the PC.

Format for A: #E

Examples:

```
        MOV     #100,R0         ;MOVE THE VALUE 100 INTO REGISTER 0.
        MOV     #X,R0           ;MOVE THE VALUE OF SYMBOL X INTO
                                ;REGISTER 0.
```

The number sign (#) in the MACRO character set has special significance as an addressing mode indicator. When this character appears in the operand field, as shown above, it specifies the immediate addressing mode, indicating to MACRO that the operand itself immediately follows the instruction word.

The operation of this mode can be shown through the first example, MOV #100,R0, which assembles as two words:

```
Location 20:  0 1 2 7 0 0

Location 22:  0 0 0 1 0 0

Location 24:  Next instruction
```

Note that the source operand (the value 100) is assembled immediately following the instruction word, i.e., as the second word in the instruction. Upon execution of the instruction, the processor fetches the first word (MOV) and increments the PC by 2 so that it points to location 22 (which contains the source operand).

After the next fetch and increment cycle, the source operand (100) is moved into register 0, leaving the PC pointing to location 24 (the next instruction).

5.10 ABSOLUTE MODE

Absolute mode is the equivalent of immediate mode deferred. The address expression @#E specifies an absolute address which is stored as the second or third word of the instruction. In other words, the value immediately following the instruction word is taken as the absolute address of the operand. Absolute mode is assembled as an autoincrement deferred of the PC.

Format for A: @#E

Examples:

```
        MOV     @#100,R0        ;MOVE THE CONTENTS OF LOCATION 100
                                ;INTO REGISTER R0.
        CLR     @#X             ;CLEAR THE CONTENTS OF THE LOCATION
                                ;WHOSE ADDRESS IS SPECIFIED BY
                                ;THE SYMBOL X.
```

The operation of this mode can be shown through the first example, MOV @#100,R0, which assembles as two words:

```
Location 20:  0 1 3 7 0 0

Location 22:  0 0 0 1 0 0

Location 24:  Next instruction
```

Note that the absolute address 100 is assembled immediately following the instruction word, i.e., as the second word in the instruction. Upon execution of the instruction, the processor fetches the first word (MOV) and increments the PC by 2 so that it points to location 22 (which contains the absolute address of the source operand). After the next fetch and increment cycle, the contents of absolute address 100 (the source operand) are moved into register 0, leaving the PC pointing to location 24 (the next instruction).

5.11 RELATIVE MODE

Relative mode is the normal mode for memory references within your program. It is assembled as index mode, using the PC as the index register.

Format for A: E

Examples:

```
        CLR     100             ;CLEAR LOCATION 100, RELATIVE TO
                                ;THE CONTENTS OF THE PC.
        MOV     R0,Y            ;MOVE THE CONTENTS OF REGISTER 0
                                ;TO LOCATION Y, RELATIVE TO THE
                                ;CONTENTS OF THE PC.
```

In relative mode, the offset for the address calculation is assembled as the second or third word of the instruction. This value is added to the contents of the PC (the base register) to yield the address of the source operand.

The operation of relative mode can be shown with the statement MOV 100,R3, which assembles as two words:

```
Location 20:  0 1 6 7 0 3

Location 22:  0 0 0 0 5 4

Location 24:  Next instruction
```

Note that the constant 54 is assembled immediately following the instruction word, i.e., as the second word in the instruction. Upon execution of the instruction, the processor fetches the first word (MOV) and increments the PC by 2 so that it points to location 22 (containing the value 54). After the next fetch and increment cycle, the processor calculates the effective address of the source operand by taking the contents of location 22 (the offset) and adding it to the current value of the PC, which now points to location 24 (the next instruction). Thus, the source operand address is the result of the calculation OFFSET+PC = 54+24 = 100(8), causing the contents of location 100 to be moved into register 3.

Since MACRO considers the contents of the current location counter (.) as the address of the first word of the instruction, an equivalent index mode statement is shown below:

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

This instruction has a relative addressing mode because the operand address is calculated relative to the current value of the location counter. The offset is the distance (in bytes) between the operand and the current value of the location counter.

5.12 RELATIVE DEFERRED MODE

The relative deferred mode is similar in operation to the relative mode above, except that the expression E is used as a pointer to the address of the operand. In other words, the operand following the instruction word is added to the contents of the PC to yield a pointer to the address of the operand.

Format for A: @E

Example:

```
MOV     @X,R0           ;RELATIVE TO THE CURRENT VALUE OF
                        ;THE PC, MOVE THE CONTENTS OF THE
                        ;LOCATION WHOSE ADDRESS IS POINTED
                        ;TO BY LOCATION X INTO REGISTER 0.
```

5.13 SUMMARY OF ADDRESSING FORMS

Each PDP-11 instruction takes at least one word. Operands of the form listed below do not increase the length of an instruction.

Form	Meaning
R	Register mode
@R or (ER)	Register deferred mode (see Note below)
(ER)+	Autoincrement mode
@(ER)+	Autoincrement deferred mode
-(ER)	Autodecrement mode
@-(ER)	Autodecrement deferred mode

Operands of the following forms add one word to the instruction length for each occurrence of an operand of that form:

Form	Meaning
E(ER)	Index mode
@E(ER)	Index deferred mode
#E	Immediate mode
@#E	Absolute mode (see Note below)
E	Relative mode
@E	Relative deferred mode

The syntax of the addressing modes is summarized in Appendix B. Additional discussion of addressing modes is provided in the applicable PDP-11 Processor Handbook.

NOTE

An alternate form for @R is (ER). However, the form @(ER) is only logically, but not physically equivalent to the expression @0(ER). The addressing form @#E differs from form E in that the second or third word of the instruction contains the absolute address of the operand, rather than the relative distance between the operand and the PC. Thus, the instruction CLR @#100 clears absolute location 100, even if the instruction is moved from the

point at which it was assembled. See the description of the .ENABL AMA function in Section 6.2, which causes all relative mode addresses to be assembled as absolute mode addresses.

5.14 BRANCH INSTRUCTION ADDRESSING

The branch instructions are 1-word instructions. The high-order byte contains the operator, and the low-order byte contains an 8-bit signed offset (seven bits, plus sign), which specifies the branch address relative to the current value of the PC. The hardware calculates the branch address as follows:

1. Extends the sign of the offset through bits 8-15.
2. Multiplies the result by 2, creating a byte offset rather than a word offset.
3. Adds the result to the current value of the PC to form the effective branch address.

MACRO performs the reverse operation to form the word offset from the specified address. Remember that when the offset is added to the current value of the PC, the PC is pointing to the word following the branch instruction; hence, the factor -2 in the following calculation:

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

Since the value of the PC = .+2, we have:

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

In using branch instructions, you must exercise care to avoid the following error conditions:

1. Branching from one program section to another;
2. Branching to a location that is defined as an external (global) symbol; or
3. Specifying a branch address that is out of range, i.e., the branch offset is a value that does not lie within the range -128(10) to +127(10).

The above conditions cause an error code (A) to be generated in the assembly listing for the statement in error.

5.15 USING TRAP INSTRUCTIONS

The EMT and TRAP instructions do not use the low-order byte of the instruction word, allowing information to be transferred to the trap handlers in the low-order byte. If the EMT or TRAP instruction is followed by an expression, the value of the expression is stored in the low-order byte of the word. However, if the expression is greater than 377(8), it is truncated to eight bits and an error code (T) is generated in the assembly listing.

PART III

MACRO DIRECTIVES

Chapters 6 and 7 describe all the directives used with MACRO. Directives are statements that cause MACRO to perform certain operations during assembly. Chapter 6 describes several types of directives, including those which control symbol interpretation, listing header material, program sections, data storage formats, and assembly listings. Chapter 7 describes those directives concerning macros, macro arguments, and repetitive coding sequences.

MACRO directives can be preceded by a label (subject to any restrictions associated with specific directives) and followed by a comment. A MACRO directive occupies the operator field of a source statement. Only one directive can be included in any given source line. The operand field may be occupied by one or more operands or left blank; legal operands differ with each directive specified.

CHAPTER 6

GENERAL ASSEMBLER DIRECTIVES

This category of directives includes:

1. Listing control
2. Function control
3. Data storage
4. Radix and numeric control
5. Location counter control
6. Terminators
7. Program boundaries
8. Program sectioning
9. Symbol control
10. Conditional assembly
11. PAL-11R conditional assembly.

Each is described in its own section of this chapter.

6.1 LISTING CONTROL DIRECTIVES

Listing control directives control the content, format, and pagination of all line printer and teleprinter listing output generated during assembly. Facilities also exist for creating object module names and other identification information in the listing output.

6.1.1 .LIST and .NLIST Directives

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

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

where: arg represents one or more of the optional symbolic arguments defined in Table 6-1.

As indicated above, the listing control directives may be used without arguments, in which case the listing directives alter the listing level count. The listing level count is initialized to zero. At each occurrence of a .LIST directive, the listing level count is incremented; at each occurrence of an .NLIST directive, the listing level count is decremented. When the listing level count is negative, the listing is suppressed (unless the line contains an error). Conversely, when the listing level count is greater than zero, the listing is always generated. Finally, when the count is zero, the line is either listed or suppressed, contingent upon the other listing controls currently in effect for the program. For example, the following macro definition employs the .LIST and .NLIST directives to selectively list portions of the macro body when the macro is expanded:

```
        .MACRO  LTEST           ;LIST TEST
; A-THIS LINE SHOULD LIST       ;LISTING LEVEL COUNT IS 0.
        .NLIST                  ;LISTING LEVEL COUNT IS -1.
; B-THIS LINE SHOULD NOT LIST
        .NLIST                  ;LISTING LEVEL COUNT IS -2.
; C-THIS LINE SHOULD NOT LIST
        .LIST                   ;LISTING LEVEL COUNT IS -1.
; D-THIS LINE SHOULD NOT LIST
        .LIST                   ;LISTING LEVEL COUNT IS 0.
; E-THIS LINE SHOULD LIST       ;LISTING LEVEL COUNT IS BACK TO 0.
        .ENDM
          .
          .
          .
        .LIST   ME              ;LIST MACRO EXPANSION.
        LTEST                   ;CALL THE MACRO
; A-THIS LINE SHOULD LIST       ;LISTING LEVEL COUNT IS 0.
; E-THIS LINE SHOULD LIST       ;LISTING LEVEL COUNT IS BACK TO 0.
```

An important purpose of the level count is to allow macro expansions to be listed selectively and yet exit with the listing level count restored to the value existing prior to the macro call.

When used with arguments, the listing directives do not alter the listing level count; however, the .LIST and .NLIST directives can be used to override current listing control, as shown in the example below:

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

        .NLIST  ME              ;DO NOT LIST MACRO EXPANSIONS.
        XX
X=.
```

The symbolic arguments allowed for use with the listing directives are described in Table 6-1. These arguments can be used singly or in combination with each other. If multiple arguments are specified in a listing directive, each argument must be separated by a comma, tab, or space. For any argument not specifically included in a listing

control statement, the associated default assumption (List or No list) is applicable throughout the source program. The default assumptions for the listing control directives also appear in Table 6-1.

Table 6-1
Symbolic Arguments of Listing Control Directives

Argument	Default	Function
SEQ*	List	Controls the listing of source line sequence numbers. MACRO assigns sequence number 1 to the first source line in a file, and increments the sequence number for each additional line in the file. If this field is suppressed through an .NLIST SEQ directive, MACRO generates a tab, effectively allocating space for the field, but fills the field with blanks. Thus, the inter-positional relationships of subsequent fields in the listing remain undisturbed. During the assembly process, MACRO examines each source line for possible error conditions. For any line in error, an appropriate error flag is printed preceding the line sequence number field (see Appendix D). MACRO does not assign sequence numbers for files that have had sequence numbers assigned by other programs, such as an editor.
LOC*	List	Controls the listing of the current location counter field. Normally, this field is not suppressed. However, if it is suppressed through the .NLIST LOC directive, MACRO does not generate a tab, nor does it allocate space for the field, as is the case with the source line sequence number field (SEQ) described above. Thus, the suppression of the current location counter (LOC) field effectively left-justifies all subsequent fields (while preserving inter-positional relationships) to that position otherwise normally occupied by this field.
BIN*	List	Controls the listing of generated binary code. If this field is suppressed through an .NLIST BIN directive, left-justification of the source code field occurs in the same manner described above for the current location counter (LOC) field.

(continued on next page)

* If the .NLIST arguments SEQ, LOC, BIN, and SRC are in effect at the same time, i.e., if all four significant fields in the listing are to be suppressed, the printing of the resulting blank line is inhibited.

Table 6-1 (Cont.)
Symbolic Arguments of Listing Control Directives

Argument	Default	Function
BEX	List	Controls the listing of binary extensions, i.e., the locations and binary contents beyond those that will fit on the source statement line. This is a subset of the BIN argument.
SRC*	List	Controls the listing of source lines.
COM	List	Controls the listing of comments. This is a subset of the SRC argument. The .NLIST COM directive reduces listing time and space when comments are not desired.
MD	List	Controls the listing of macro definitions and repeat range expansions.
MC	List	Controls the listing of macro calls and repeat range expansions.
ME	No list	Controls the listing of macro expansions.
MEB	No list	Controls the listing of macro expansion binary code. A .LIST MEB directive causes only those macro expansion statements that generate binary code to be listed. This is a subset of the ME argument.
CND	List	Controls the listing of unsatisfied conditional coding and associated .IF and .ENDC directives in the source program. This argument permits conditional assemblies to be listed without including unsatisfied conditional coding.
LD	No list	Controls the listing of all listing directives having no arguments, i.e., those listing directives that alter the listing level count.
TOC	List	Controls the listing of the table of contents during assembly pass 1 (see Section 6.1.4 describing the .SBTTL directive). This argument does not affect the printing of the full assembly listing during assembly pass 2.

(continued on next page)

* If the .NLIST arguments SEQ, LOC, BIN, and SRC are in effect at the same time, i.e., if all four significant fields in the listing are to be suppressed, the printing of the resulting blank line is inhibited.

Table 6-1 (Cont.)
Symbolic Arguments of Listing Control Directives

Argument	Default	Function
SYM	List	Controls the listing of the symbol table resulting from the assembly of the source program.
TTM	List	Controls the listing output format. The default can be set by the system manager. If the system manager does not set a default, it is set to line printer format. Figure 6-1 illustrates the line printer output format. Figure 6-2 illustrates the teleprinter output format.

An example of an assembly listing, as sent to a 132-column line printer, is shown in Figure 6-1. Note that binary extensions for statements generating more than one word are formatted horizontally on the source line.

An example of an assembly listing, as sent to a teleprinter (in the same format as for an 80-column line printer), is shown in Figure 6-2. Notice that binary extensions for statements generating more than one word are printed on subsequent lines. There is no explicit truncation of output to 80 characters by the assembler.

Any argument specified in a .LIST/.NLIST directive other than those listed in Table 6-1 causes the directive to be flagged with an error code (A) in the assembly listing.

The listing control options can also be specified at assembly time through switches included in the command string to MACRO (see Chapter 8). The use of these switches overrides all corresponding listing control (.LIST or .NLIST) directives specified in the source program.

```
CSITST -- TEST OF CSI1 AND CSI2 MACRO M0707  09-JUL-74 15:47  PAGE 5
READ AND PARSE COMMAND LINES

  209                                           .SBTTL  READ AND PARSE COMMAND LINES
  210
  211 001230                              GETLN:  GCML$   #GCLBLK          ;GET LINE VIA GCML
  212 001244  103003                              BCC     1$               ;SKIP IF NO ERROR
  213 001246                                      EXIT$S                   ;ELSE, EXIT
  214 001254                              1$:     TYPE    G.CMLD+2(R0),G.CMLD(R0),#!0     ;SEND OUT THE INPUT LINE
  215 001300                                      CSI$1   #CSIBLK,GCLBLK+G.CMLD+2,GCLBLK+G.CMLD
  216 001324  103064                              BCC     2$               ;BRANCH IF NO ERROR DETECTED
  217 001326  016046  000020                      MOV     C.FILD+2(R0),-(SP)       ;PUT STRING ERROR ADDR IN STK
  218 001332  166016  000004                      SUB     C.CMLD+2(R0),(SP)        ;CALCULATE LENGTH OF FIRST PART
  219 001336                                      TYPE    C.CMLD+2(R0),(SP),#!$    ;SEND OUT FIRST PART OF STRING
  220 001360                                      TYPE    C.FILD+2(R0),C.FILD(R0),#!$     ;SEND OUT SECOND PART
  221 001404  066060  000016  000020              ADD     C.FILD(R0),C.FILD+2(R0) ;CALC ADDR OF LAST PART OF STRING
  222 001412  162660  000002                      SUB     (SP)+,C.CMLD(R0)         ;DEDUCT LENGTH OF FIRST PART
  223 001416  166060  000016  000002              SUB     C.FILD(R0),C.CMLD(R0)    ;CALC LENGTH OF LAST PART
  224 001424                                      TYPE    C.FILD+2(R0),C.CMLD(R0),#40     ;SEND OUT LAST PART
  225 001450                                      TYPEM   STX,40           ;SEND SYNTAX ERROR MESSAGE
  226 001474  000655                              BR      GETLN            ;TRY FOR MORE
  227
  228 001476  005760  000002              2$:     TST     C.CMLD(R0)       ;CHECK LENGTH OF LINE
  229 001502  001652                              BEQ     GETLN            ;IF NULL, SKIP BACK FOR NEXT LINE
  230 001504  112767  000060  176432              MOVB    #!0,EQUBIT       ;ASSUME EQUAL SIGN NOT FOUND
  231 001512  132760  000040  000001              BITB    #CS.EQU,C.STAT(R0)  ;CHECK STATUS
  232 001520  001402                              BEQ     10$              ;SKIP IF EQUAL SIGN NOT SEEN
  233 001522  105267  176416                      INCB    EQUBIT           ;ELSE, INDICATE EQUAL SIGN FOUND
  234 001526                              10$:    TYPEM   EQU,40           ;SEND EQUAL SIGN STATUS MESSAGE
  235 001552                                      TYPEM   OPT,40           ;SEND OUTPUT SCAN MESSAGE
  236 001576                              OPARSE: CALL    INIT2            ;INIT LOCNS FOR CSI2 CALL/TEST
  237 001602                                      CSI$2   ,OUTPUT,#SWTBL   ;PARSE OUTPUT SPEC
  238 001620  103441                              BCS     CS2ERR           ;SKIP ON ERROR
  239 001622                                      CALL    EVALU8           ;EVALUATE RESULTS OF SEMANTIC PARSE
  240 001626  132760  000020  000001              BITB    #CS.MOR,C.STAT(R0)  ;ADDITIONAL OUTPUT SPECS?
  241 001634  001360                              BNE     OPARSE           ;YES, CONTINUE WITH OUTPUT SCAN
  242 001636                                      TYPEM   IPT,40           ;SEND INPUT SCAN MESSAGE
  243 001662                              IPARSE: CALL    INIT2            ;INIT LOCNS FOR CSI2 CALL/TEST
  244 001666                                      CSI$2   ,INPUT,#SWTBL    ;PARSE INPUT SPEC
  245 001704  103407                              BCS     CS2ERR           ;SKIP ON ERROR
  246 001706                                      CALL    EVALU8           ;EVALUATE RESULTS OF SEMANTIC PARSE
  247 001712  132760  000020  000001              BITB    #CS.MOR,C.STAT(R0)  ;ADDITIONAL INPUT SPECS?
  248 001720  001360                              BNE     IPARSE           ;YES, CONTINUE WITH INPUT SCAN
  249 001722  000412                              BR      JMPGET           ;GET ANOTHER COMMAND LINE
```

Figure 6-1 Example of Line Printer Assembly Listing

```
CSITST -- TEST OF CSI1 AND CSI2 MACRO M0707  09-JUL-74 15:59  PAGE 5
READ AND PARSE COMMAND LINES

                                .SBTTL  READ AND PARSE COMMAND LINES
                        GETLN:  GCML$   #GCLBLK          ;GET LINE VIA GCML
001244  103003                BCC     1$               ;SKIP IF NO ERROR
001246                          EXIT$S                   ;ELSE, EXIT
001254                  1$:     TYPE    G.CMLD+2(R0),G.CMLD(R0),#'0     ;SEND OUT THE INPUT LINE
001300                          CSI$1   #CSIBLK,GCLBLK+G.CMLD+2,GCLBLK+G.CMLD
001324  103064                BCC     2$               ;BRANCH IF NO ERROR DETECTED
001326  016046                MOV     C.FILD+2(R0),-(SP)       ;PUT STRING ERROR ADDR IN STK
        000020
001332  166016                SUB     C.CMLD+2(R0),(SP)        ;CALCULATE LENGTH OF FIRST PART
        000004
001336                          TYPE    C.CMLD+2(R0),(SP),#'$    ;SEND OUT FIRST PART OF STRING
001360                          TYPE    C.FILD+2(R0),C.FILD(R0),#'$      ;SEND OUT SECOND PART
001404  066060                ADD     C.FILD(R0),C.FILD+2(R0) ;CALC ADDR OF LAST PART OF STRING
        000016
        000020
001412  162660                SUB     (SP)+,C.CMLD(R0)         ;DEDUCT LENGTH OF FIRST PART
        000002
001416  166060                SUB     C.FILD(R0),C.CMLD(R0)    ;CALC LENGTH OF LAST PART
        000016
        000002
001424                          TYPE    C.FILD+2(R0),C.CMLD(R0),#40      ;SEND OUT LAST PART
001450                          TYPEM   STX,40           ;SEND SYNTAX ERROR MESSAGE
001474  000655                BR      GETLN            ;TRY FOR MORE

001476  005760          2$:   TST     C.CMLD(R0)       ;CHECK LENGTH OF LINE
        000002
001502  001652                BEQ     GETLN            ;IF NULL, SKIP BACK FOR NEXT LINE
001504  112767                MOVB    #'0,EQUBIT       ;ASSUME EQUAL SIGN NOT FOUND
        000060
        176432
001512  132760                BITB    #CS.EQU,C.STAT(R0)  ;CHECK STATUS
        000040
        000001
001520  001402                BEQ     10$              ;SKIP IF EQUAL SIGN NOT SEEN
001522  105267                INCB    EQUBIT           ;ELSE, INDICATE EQUAL SIGN FOUND
        176416
001526                  10$:    TYPEM   EQU,40           ;SEND EQUAL SIGN STATUS MESSAGE
001552                          TYPEM   OPT,40           ;SEND OUTPUT SCAN MESSAGE
001576                  OPARSE: CALL    INIT2            ;INIT LOCNS FOR CSI2 CALL/TEST
001602                          CSI$2   ,OUTPUT,#SWTBL   ;PARSE OUTPUT SPEC
001620  103441                BCS     CS2ERR           ;SKIP ON ERROR
001622                          CALL    EVALU8           ;EVALUATE RESULTS OF SEMANTIC PARSE

001626  132760                BITB    #CS.MOR,C.STAT(R0)  ;ADDITIONAL OUTPUT SPECS?
        000020
        000001
001634  001360                BNE     OPARSE           ;YES, CONTINUE WITH OUTPUT SCAN
001636                          TYPEM   IPT,40           ;SEND INPUT SCAN MESSAGE
001662                  IPARSE: CALL    INIT2            ;INIT LOCNS FOR CSI2 CALL/TEST
001666                          CSI$2   ,INPUT,#SWTBL    ;PARSE INPUT SPEC
001704  103407                BCS     CS2ERR           ;SKIP ON ERROR
001706                          CALL    EVALU8           ;EVALUATE RESULTS OF SEMANTIC PARSE
001712  132760                BITB    #CS.MOR,C.STAT(R0)  ;ADDITIONAL INPUT SPECS?
        000020
        000001
001720  001360                BNE     IPARSE           ;YES, CONTINUE WITH INPUT SCAN
```

Figure 6-2 Example of Terminal Assembly Listing

Figure 6-3 shows a listing, produced in line printer format, reflecting the use of the .LIST and .NLIST directives in the source program and the effects such directives have on the assembly listing output.

6.1.2 Page Headings

MACRO prints each assembly page in the format shown in either Figure 6-1 or Figure 6-2, depending on the listing mode (see TTM, Table 6-1). On the first line of each page, MACRO prints the following (from left to right):

1. Title of the object module, as established through the .TITLE directive (see next section).
2. Assembler version identification.
3. Date.
4. Time-of-day.
5. Page number.

The second line of each assembly listing page contains the subtitle text specified in the last-encountered .SBTTL directive (see Section 6.1.4).

```
27
28
29 000062                                  LSTMAC  COM             ;COMMENT LINES TEST
                                           .NLIST  COM
   000062  000001  000002  000003          .WORD   1,2,3,4,5
   000070  000004  000005
                                           .LIST   COM
30
31
32 000074                                  LSTMAC  <COM,BEX>       ;COMMENT LINES AND EXTENDED BINARY TEST
                                           .NLIST  COM,BEX
   000074  000001  000002  000003          .WORD   1,2,3,4,5
                                           .LIST   COM,BEX
```

```
.MAIN.  MACRO M0707  09-JUL-74 16:29  PAGE 1-1

33
34
35
36                              .LIST   TTM             ;NARROW LISTING MODE IS IN EFFECT
37
38 000106                       LSTMAC  SEQ             ;SEQUENCE NUMBERS TEST
                                .NLIST  SEQ
   000106  000001               .WORD   1,2,3,4,5       ;THIS IS A COMMENT
   000110  000002
   000112  000003
   000114  000004
   000116  000005
                                .LIST   SEQ
39
40
41 000120                       LSTMAC  BEX             ;EXTENDED BINARY TEST
                                .NLIST  BEX
   000120  000001               .WORD   1,2,3,4,5       ;THIS IS A COMMENT
                                .LIST   BEX
42
43
44          000001'             .END
```

Figure 6-3 Listing Produced With Listing Control Directives

```
.MAIN.  MACRO M0707  09-JUL-74 16:29  PAGE 1


                                        .NLIST  TTM             ;WIDE LISTING MODE IS IN EFFECT
                                        .LIST   ME              ;LIST MACRO EXPANSIONS

                                ;
                                ; LISTING CONTROL TEST MACRO
                                ;
                                        .MACRO  LSTMAC  ARG
                                        .NLIST  ARG
                                        .WORD   1,2,3,4,5       ;THIS IS A COMMENT
                                        .LIST   ARG
                                        .ENDM

000012                                  LSTMAC  LOC             ;LOCATION COUNTER TEST
                                .NLIST  LOC
000001  000002  000003          .WORD   1,2,3,4,5       ;THIS IS A COMMENT
000004  000005
                                        .LIST   LOC

000024                                  LSTMAC  BIN             ;GENERATED BINARY TEST
                .NLIST  BIN
000024          .WORD   1,2,3,4,5       ;THIS IS A COMMENT
                                        .LIST   BIN

000036                                  LSTMAC  BEX             ;EXTENDED BINARY TEST
                                        .NLIST  BEX
000036  000001  000002  000003          .WORD   1,2,3,4,5       ;THIS IS A COMMENT
                                        .LIST   BEX

000050                                  LSTMAC  SRC             ;SOURCE LINES TEST

000050  000001  000002  000003
000056  000004  000005
                                        .LIST   SRC
```

Figure 6-3 (Cont.) Listing Produced With Listing Control Directives

6.1.3 .TITLE Directive

The .TITLE directive is used to assign a name to the object module as the first entry in the header of each page in the assembly listing. The name so assigned is the first six non-blank characters following the .TITLE directive. This name should be six Radix-50 characters or less in length; any characters beyond the first six are checked for ASCII legality, but they are not used as part of the object module name. For example, the directive:

```
.TITLE  PROGRAM TO PERFORM DAILY ACCOUNTING
```

causes the assembled object module to be named PROGRA. Note that this 6-character name bears no relationship to the filename of the object module, as specified in the command string to MACRO. The name of an object module (specified in the .TITLE directive) appears in the Linker load map. This is also the module name which the Librarian will recognize.

If the .TITLE directive is not specified, MACRO assigns the default name .MAIN. to the object module. If more than one .TITLE directive is specified in the source program, the last .TITLE directive encountered establishes the name for the entire object module.

All spaces and/or tabs up to the first non-space/non-tab character following the .TITLE directive are ignored by MACRO when evaluating the text string.

If the .TITLE directive is specified without an object module name, or if the first non-space/non-tab character in the object module name is not a Radix-50 character, the directive is flagged with an error code (A) in the assembly listing.

Section A.2 of Appendix A contains a table of Radix-50 characters.

6.1.4 .SBTTL Directive

The .SBTTL directive is used to produce a table of contents immediately preceding the assembly listing and to further identify each page in the listing. In the latter case, the text following the .SBTTL directive is printed as the second line of the header of each page in the listing, continuing until altered by a subsequent .SBTTL directive in the program. For example, the directive:

```
.SBTTL  CONDITIONAL ASSEMBLIES
```

causes the text

```
CONDITIONAL ASSEMBLIES
```

to be printed as the second line in the header of the assembly listing.

During assembly pass 1, a table of contents is printed for the assembly listing, containing the line sequence number, the page number, and the text accompanying each .SBTTL directive. The listing of the table of contents is suppressed whenever an .NLIST TOC directive is encountered in the source program (see Table 6-1). An example of a table of contents listing is shown in Figure 6-4.

```
CSITST -- TEST OF CSI1 AND CSI2 MACRO M0707  09-JUL-74 15:47
TABLE OF CONTENTS

   2- 55        MACRO DEFINITIONS
   3- 74        MESSAGE STRINGS
   4-153        MISCELLANEOUS DATA
   5-209        READ AND PARSE COMMAND LINES
   6-255        EVALUATE THE SEMANTIC ANALYSIS
   7-345        SUBROUTINES
```

Figure 6-4 Assembly Listing Table of Contents

6.1.5 .IDENT Directive

The .IDENT directive provides an additional means of labeling the object module produced by MACRO. In addition to the name assigned to the object module with the .TITLE directive (see Section 6.1.3), a character string up to six Radix-50 characters can be specified between paired printing delimiters to label the object module with the program version number. This directive takes the following form:

```
.IDENT  /string/
```

where: string represents six legal Radix-50 characters or less which establish the program identification or version number. This number is included in the global symbol directory of the object module.

/ / represent delimiting characters. These delimiters may be any paired printing characters, other than the equal sign (=), the left angle bracket (<), or the semicolon (;), as long as the delimiting character is not contained within the text string itself. If the delimiting characters do not match, or if an illegal delimiting character is used, the .IDENT directive is flagged with an error code (A) in the assembly listing.

An example of the .IDENT directive is shown below:

```
.IDENT  /V05A/
```

The character string V05A is converted to Radix-50 representation and included in the global symbol directory of the object module. This character string also appears in the Linker load map and the Librarian directory listings.

When more than one .IDENT directive is encountered in a given program, the last such directive encountered establishes the character string which forms part of the object module identification.

6.1.6 .PAGE Directive/Page Ejection

Page ejection is accomplished in one of four ways:

1. After reaching a count of 58 lines in the listing, MACRO automatically performs a page eject to skip over page perforations on line printer paper and to formulate teleprinter output into pages.

2. In addition, the .PAGE directive is used within the source program to perform a page eject at desired points in the listing. The format of this directive is:

   .PAGE

   This directive takes no arguments and causes a skip to the top of the next page when encountered. It also causes the page number to be incremented. The .PAGE directive does not appear in the listing.

   When used within a macro definition, the .PAGE directive is ignored during the assembly of the macro definition. Rather, the page eject operation is performed as the macro itself is expanded. In this case, the page number is also incremented.

3. A page eject is performed when a form-feed character is encountered. If the form-feed character appears within a macro definition, a page eject occurs during the assembly of the macro definition, but not during the expansion of the macro itself. A page eject resulting from the use of the form-feed character likewise causes the page number to be incremented.

4. Encountering a new source file causes the page number to be incremented and the line sequence count to be reset.

6.2 FUNCTION DIRECTIVES: .ENABL AND .DSABL

Several function control options are provided by MACRO through the .ENABL and .DSABL directives. These directives are included in a source program to invoke or inhibit certain MACRO functions and operations incidental to the assembly process itself. These directives take the following form:

```
.ENABL  arg
.DSABL  arg
```

where: arg represents one or more of the optional symbolic arguments defined in Table 6-2.

Table 6-2
Symbolic Arguments of Function Control Directives

Argument	Default	Function
AMA	Disable	Enabling this function causes all relative addresses (address mode 67) to be assembled as absolute addresses (address mode 37). This function is useful during the debugging phase of program development.

(continued on next page)

Table 6-2 (Cont.)
Symbolic Arguments of Function Control Directives

Argument	Default	Function
CDR	Disable	Enabling this function causes source columns 73 and greater, i.e., to the end of the line, to be treated as a comment. The most common use of this feature is to permit sequence numbers in card columns 73-80.
CRF	Enable	Disabling this function inhibits the generation of cross-reference output. This function only has meaning if cross-reference output generation is specified in the command string.
FPT	Disable	Enabling this function causes floating-point truncation; disabling this function causes floating-point rounding.
LC	Disable	Enabling this function causes MACRO to accept lower-case ASCII input instead of converting it to upper-case. If this function is not enabled, all text is converted to upper-case.
LSB	Disable	This argument permits the enabling or disabling of a local symbol block. Although a local symbol block is normally established by encountering a new symbolic label or a .PSECT directive in the source program, an .ENABL LSB directive establishes a new local symbol block which is not terminated until (1) another .ENABL LSB is encountered, or (2) another symbolic label or .PSECT directive is encountered following a paired .DSABL LSB directive. Although the .ENABL LSB directive permits a local symbol block to cross .PSECT boundaries, local symbols cannot be defined in a program section other than the one that was in effect when the block was entered. The basic function of this directive with regard to .PSECT's is limited to those instances where it is desirable to leave a program section temporarily to store data, followed by a return to the original program section. Attempts to define local symbols in an alternate program section are flagged with an error code (P) in the assembly listing. An example of the .ENABL LSB and .DSABL LSB directives, as typically used in a source program, is shown in Figure 6-5.

(continued on next page)

Table 6-2 (Cont.)
Symbolic Arguments of Function Control Directives

Arugment	Default	Function
PNC	Enable	Disabling this function inhibits binary output until an .ENABL PNC statement is encountered within the same module.
REG	Enable	When specified, the .DSABL REG directive inhibits the normal MACRO default register definitions; if not disabled, the default definitions listed below remain in effect. R0=%0 R1=%1 R2=%2 R3=%3 R4=%4 R5=%5 SP=%6 PC=%7 The .ENABL REG statement may be used as the logical complement of the .DSABL REG directive. The use of these directives, however, is not recommended. For logical consistency, use the normal default register definitions listed above.
GBL	Enable	When the .ENABL GBL directive is specified, MACRO treats all symbol references that are undefined at the end of assembly pass 1 as default global references; when the .DSABL GBL directive is specified, MACRO treats all such references as undefined symbols. In assembly pass 2, if the .DSABL GBL function is still in effect, these undefined symbols are flagged with an error code (U) in the assembly listing; otherwise, they continue to be regarded by MACRO as global references.

Specifying any argument in an .ENABL/.DSABL directive other than those listed in Table 6-2 causes that directive to be flagged with an error code (A) in the assembly listing.

```
SQUEEZE MACRO M0707  09-JUL-74 15:13  PAGE 4


                                                .ENABL  LSB

003142  010103                        FNDSMI: MOV     R1,R3           ;PUT ADDR OF LINE IN R3
003144  060203                                ADD     R2,R3           ;POINT R3 PAST LAST CHAR IN LINE
003146  020301                        1$:     CMP     R3,R1           ;DOES R3 POINT TO START OF LINE?
003150  001422                                BEQ     30$             ;IF SO, LEAVE INDICATING FAILURE
003152  124327  000073                        CMPB    -(R3),#SEMIC    ;IS THE LAST CHARACTER SEMICOLON?
003156  001373                                BNE     1$              ;NO, CONTINUE LOOKING
003160  010302                                MOV     R3,R2           ;YES, POINT R2 PAST NEW END-OF-LINE
003162  000412                                BR      20$             ;LEAVE VIA COMMON SUCCESS CODE

003164  060102                        SKPBLK: ADD     R1,R2           ;POINT R2 PAST END-OF-LINE
003166  020201                        10$:    CMP     R2,R1           ;DOES R2 POINT TO START OF LINE?
003170  001412                                BEQ     30$             ;IF SO, LEAVE WITH FAILURE
003172  124227  000011                        CMPB    -(R2),#TAB      ;IS THE LAST CHARACTER A TAB?
003176  001773                                BEQ     10$             ;IF SO, IGNORE IT
003200  121227  000040                        CMPB    (R2),#BLANK     ;IS IT A BLANK?
003204  001770                                BEQ     10$             ;IF SO, IGNORE IT
003206  005202                                INC     R2              ;NON-BLANK CHARACTER--POINT PAST IT
003210  160102                        20$:    SUB     R1,R2           ;RE-COMPUTE LINE LENGTH
003212  000241                                CLC                     ;INDICATE SUCCESS
003214  000401                                BR      40$             ;BRANCH TO LEAVE
003216  000261                        30$:    SEC                     ;INDICATE FAILURE
003220                                40$:    RETURN                  ;
                                                .DSABL  LSB
```

Figure 6-5 Example of .ENABL and .DSABL Directives

6.3 DATA STORAGE DIRECTIVES

A wide range of data and data types can be generated with the following directives, ASCII conversion characters, and radix-control operators:

```
.BYTE
.WORD
'
"
.ASCII
.ASCIZ
.FLT2
.FLT4
.RAD50
^B
^C
^D
^F
^O
^R
```

These MACRO facilities are described in the following sections.

6.3.1 .BYTE Directive

The .BYTE directive is used to generate successive bytes of binary data in the object module. The directive is of the form:

```
.BYTE   exp             ;STORES THE BINARY VALUE OF THE
                        ;EXPRESSION "EXP" IN THE NEXT BYTE.

.BYTE   expl,exp2,expn  ;STORES THE BINARY VALUES OF THE LIST
                        ;OF EXPRESSIONS IN SUCCESSIVE BYTES.
```

A legal expression must reduce to eight bits of data or less. The operands of a .BYTE directive are evaluated as word expressions before being truncated to the low-order eight bits. The 16-bit value of the specified expression must have a high-order byte (which is truncated) that is either all zeros (0) or all ones (1). Each expression value is stored in the next byte of the object module. Multiple expressions, which must be separated by commas, are stored in successive bytes, as described below:

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

If the high-order byte of the expression reduces to a value other than 0 or -1, the value is truncated to the low-order eight bits and flagged with an error code (T) in the assembly listing.

The construction ^D in the first operand of the .BYTE directive above illustrates the use of a temporary radix-control operator. The function of such special unary operators is described in Section 6.4.1.2.

At link time, it is likely that a relocatable expression will result in a value having more than eight bits, in which case the Linker

issues a truncation diagnostic for the object module in question. For example, the following statements create such a possibility:

```
        .BYTE   23              ;STORES OCTAL 23 IN NEXT BYTE.
A:
        .BYTE   A               ;RELOCATABLE VALUE A WILL PROBABLY
                                ;CAUSE LINKER TRUNCATION
                                ;DIAGNOSTIC.
```

If an expression following the .BYTE directive is null, it is interpreted as a zero, as described below:

```
.=420
        .BYTE   ,,,             ;ZEROS ARE STORED IN BYTES 420, 421,
                                ;422, AND 423.
```

Note that in the above example, four bytes of storage result from the .BYTE directive. The three commas in the operand field represent an implicit declaration of four null values, each separated from the other by a comma. Hence, four bytes, each containing a value of zero (0), are reserved in the object module.

6.3.2 .WORD Directive

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

```
        .WORD   exp             ;STORES THE BINARY EQUIVALENT OF THE
                                ;EXPRESSION EXP IN THE NEXT WORD.

        .WORD   expl,exp2,expn  ;STORES THE BINARY EQUIVALENTS OF THE
                                ;LIST OF EXPRESSIONS IN SUCCESSIVE
                                ;WORDS.
```

A legal expression must result in 16 bits of data or less. Each expression is stored in the next word of the object program. Multiple expressions must be separated by commas and stored in successive words, as shown in the following example:

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

If an expression following the .WORD directive contains a null value, it is interpreted as a zero, as shown in the following example:

```
.=500
        .WORD   ,5,             ;STORES THE VALUES 0, 5, AND 0 IN
                                ;LOCATION 500, 502, AND 504,
                                ;RESPECTIVELY.
```

A statement containing a blank operator field, i.e., a symbol that is not recognized by MACRO as a macro call, an instruction nmemonic, a MACRO directive, or a semicolon is interpreted during assembly as an implicit .WORD directive, as shown in the example below:

```
.=440
LABEL:  100,LABEL               ;STORES THE VALUE 100 IN LOCATION 440
                                ;AND THE VALUE 440 IN LOCATION 442.
```

CAUTION

You should not use this technique to generate .WORD directives because it may not be included in future PDP-11 assemblers.

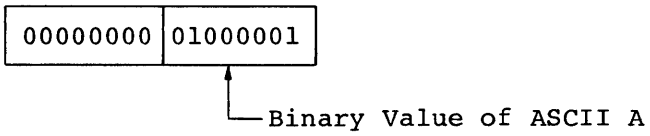
6.3.3 ASCII Conversion Characters

The single quote (') and the double quote (") characters are unary operators that can appear in any MACRO expression. When so used, these characters cause a 16-bit expression value to be generated.

When the single quote is used, MACRO takes the next character in the expression and converts it from its 7-bit ASCII value to a 16-bit expression value. The 16-bit value is then used as an absolute term within the expression. For example, the statement:

```
MOV     #'A,R0
```

results in the following 16-bit expression value being moved into register 0:



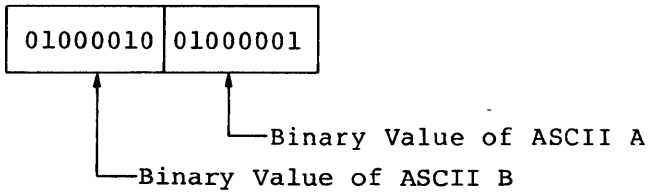
Thus, in the example above, the expression 'A results in a value of 101(8). Note that the high-order byte is always zero (0) in the resulting expression value when the single quote unary operator is used.

The ' character must not be followed by a carriage-return, null, RUBOUT, line-feed, or form-feed character; if it is, an error code (A) is generated in the assembly listing.

When the double quote is used, MACRO takes the next two characters in the expression and converts them to a 16-bit binary expression value from their 7-bit ASCII values. This 16-bit value is then used as an absolute term within the expression. For example, the statement:

```
MOV     #"AB,R0
```

results in the following 16-bit expression value being moved into register 0:



Thus, in the example above, the expression "AB results in a value of 041101(8).

The " character also must not be followed by a carriage-return, null, RUBOUT, line-feed, or form-feed character; if it is, an error code (A) is likewise generated in the assembly listing.

The ASCII character set is listed in Section A.1, Appendix A.

6.3.4 .ASCII Directive

The .ASCII directive translates character strings into their 7-bit ASCII equivalents and stores them in the object module. The format of the .ASCII directive is as follows:

```
.ASCII  /string 1/.../string n/
```

where: string is a string of printable ASCII characters. All printable ASCII characters are legal. The vertical-tab, null, line-feed, RUBOUT, and all other non-printable ASCII characters, except carriage-return and form-feed, are illegal characters. Such an illegal non-printing character is flagged with an error code (I) in the assembly listing. The carriage-return and form-feed characters terminate the scan of the source line. This premature termination of the .ASCII statement results in the generation of an error code (A) in the assembly listing, because MACRO is unable to complete the scan of the matching delimiter at the end of the character string.

/ / represent delimiting characters. These delimiters may be any paired printing characters, other than the equal sign (=), the left angle bracket (<), or the semicolon (;), as long as the delimiting character is not contained within the text string itself. If the delimiting characters do not match, or if an illegal delimiting character is used, the .ASCII directive is flagged with an error code (A) in the assembly listing.

A non-printing character can be expressed in an .ASCII statement only by enclosing its equivalent octal value within angle brackets. Each set of angle brackets so used represents a single character. For example, in the following statement:

```
.ASCII  <15>/ABC/<A+2>/DEF/<5><4>
```

the expressions <15>, <A+2>, <5>, and <4> represent the values of non-printing characters. Furthermore, the expressions must reduce to eight bits of absolute data or less, subject to the same rules for generating data as with the .BYTE directive (see Section 6.3.1).

Angle brackets can be embedded between delimiting characters in the character string, but angle brackets so used do not take on their usual significance as delimiters for non-printing characters. For example, the statement:

```
.ASCII  /ABC<expression>DEF/
```

contains a single ASCII character string, and performs no evaluation of the embedded, bracketed expression. This use of the angle brackets is shown in the third example of the .ASCII directive below:

```
        .ASCII  /HELLO/             ;STORES THE BINARY REPRESENTATION
                                    ;OF THE LETTERS HELLO IN FIVE
                                    ;CONSECUTIVE BYTES.

        .ASCII  /ABC/<15><12>/DEF/  ;STORES THE BINARY REPRESENTATION
                                    ;OF THE CHARACTERS A,B,C,CARRIAGE
                                    ;RETURN,LINE FEED,D,E,F IN EIGHT
                                    ;CONSECUTIVE BYTES.

        .ASCII  /A<15>B/            ;STORES THE BINARY REPRESENTATION
                                    ;OF THE CHARACTERS A, <, 1, 5, >,
                                    ;AND B IN SIX CONSECUTIVE BYTES.
```

The semicolon (;) and equal sign (=) can be used as delimiting characters in an ASCII string, but care must be exercised in so doing because of their significance as a comment indicator and assignment operator, respectively, as illustrated in the examples below:

```
        .ASCII  ;ABC;/DEF/      ;STORES THE BINARY REPRESENTATION OF
                                ;THE CHARACTERS A, B, C, D, E, AND F
                                ;IN SIX CONSECUTIVE BYTES; NOT
                                ;RECOMMENDED PRACTICE.

        .ASCII  /ABC/;DEF;      ;STORES THE BINARY REPRESENTATIONS OF
                                ;THE CHARACTERS A, B, AND C IN THREE
                                ;CONSECUTIVE BYTES; THE CHARACTERS D,
                                ;E, F, AND ; ARE TREATED AS A COMMENT.

        .ASCII  /ABC/=DEF=      ;STORES THE BINARY REPRESENTATION
                                ;OF THE CHARACTERS A, B, C, D, E, AND
                                ;F IN SIX CONSECUTIVE BYTES; NOT
                                ;RECOMMENDED PRACTICE.
```

An equal sign is treated as an assignment operator when it appears as the first character in the ASCII string, as illustrated by the following example:

```
        .ASCII  =DEF=           ;THE DIRECT ASSIGNMENT OPERATION
                                ;.ASCII=DEF IS PERFORMED, AND A Q
                                ;(SYNTAX) ERROR IS GENERATED UPON
                                ;ENCOUNTERING THE SECOND = SIGN.
```

6.3.5 .ASCIZ Directive

The .ASCIZ directive is equivalent to the .ASCII directive described above, except that a zero byte is automatically inserted as the final character of the string. Thus, when a list or text string has been created with an .ASCIZ directive, a search for the null character in the last byte can effectively determine the end of the string, as reflected by the coding below:

```
CR=15
LF=12
HELLO:  .ASCIZ  <CR><LF>/MACRO V01A/<CR><LF>  ;INTRODUCTORY MESSAGE
        .EVEN
        .
        .
        .
```

```
        MOV     #HELLO,R1       ;GET ADDRESS OF MESSAGE.
        MOV     #LINBUF,R2      ;GET ADDRESS OF OUTPUT BUFFER.
10$:    MOVB    (R1)+,(R2)+     ;MOVE A BYTE TO OUTPUT BUFFER.
        BNE     10$             ;IF NOT NULL, MOVE ANOTHER BYTE.
        .
        .
        .
```

The .ASCIZ directive is subject to the same checks for character legality and proper character string construction as described above for the .ASCII directive.

6.3.6 .RAD50 Directive

The .RAD50 directive allows the user to generate data in Radix-50 packed format. Radix-50 form allows three characters to be packed into sixteen bits (one word); therefore, any 6-character symbol can be stored in two consecutive words. The form of the directive is:

```
.RAD50  /string 1/.../string n/
```

where: string — represents a series of characters to be packed (three characters per word). The string must consist of the characters A through Z, 0 through 9, dollar sign ($), period (.) and space ( ). An illegal printing character causes an error flag (Q) to be printed in the assembly listing.

If fewer than three characters are to be packed, the string is packed left-justified within the word, and trailing spaces are assumed.

As with the .ASCII directive described in Section 6.3.4, the vertical-tab, null, line-feed, RUBOUT, and all other non-printing characters, except carriage-return and form-feed, are illegal characters, resulting in an error code (I) in the assembly listing. Similarly, the carriage-return and form-feed characters result in an error code (A) because these characters end the scan of the line, preventing MACRO from detecting the terminating matching delimiter.

/ / — represent delimiting characters. These delimiters may be any paired printing characters, other than the equal sign (=), the left angle bracket (<), or the semicolon (;), provided that the delimiting character is not contained within the text string itself. If the delimiting characters do not match, or if an illegal delimiting character is used, the .RAD50 directive is flagged with an error code (A) in the assembly listing.

Examples of .RAD50 directives are shown below:

```
.RAD50  /ABC/           ;PACKS ABC INTO ONE WORD.
.RAD50  /AB/            ;PACKS AB (SPACE) INTO ONE WORD.
.RAD50  /ABCD/          ;PACKS ABC INTO FIRST WORD AND
                        ;D (SPACE) (SPACE) INTO SECOND WORD.
.RAD50  /ABCDEF/        ;PACKS ABC INTO FIRST WORD, DEF INTO
                        ;SECOND WORD.
```

Each character is translated into its Radix-50 equivalent, as indicated in the following table:

Character	Radix-50 Octal Equivalent
(space)	0
A-Z	1-32
$	33
.	34
(undefined)	35
0-9	36-47

The Radix-50 equivalents for characters 1 through 3 (C1,C2,C3) are combined as follows:

```
Radix-50 Value = ((C1*50)+C2)*50+C3
```

For example:

```
Radix-50 Value of ABC = ((1*50)+2)*50+3 = 3223
```

Refer to Section A.2 in Appendix A for a table of Radix-50 equivalents.

Angle brackets (<>) must be used in the .RAD50 directive whenever special codes are to be inserted in the text string, as shown in the example below:

```
        .RAD50  /AB/<35>        ;STORES 3255 IN ONE WORD.

CHR1=1
CHR2=2
CHR3=3
          .
          .
          .
        .RAD50  <CHR1><CHR2><CHR3> ;EQUIVALENT TO .RAD50 /ABC/.
```

6.3.7 Temporary Radix-50 Control Operator: ^R

The ^R operator specifies that an argument is to be converted to Radix-50 format. This allows up to three characters to be stored in one word. The ^R operator is coded as follows:

```
^Rccc
```

where ccc represents a maximum of three characters to be converted to a 16-bit Radix-50 value. If more than three characters are specified, any following the third character are ignored. If fewer than 3 are specified, it is assumed that the trailing characters are blanks. The following example shows how the ^R operator might be used to pack a 3-character file type specifier (MAC) into a single 16-bit word.

```
MOV  #^RMAC,FILEXT        ;STORE RAD50 MAC AS FILE EXTENSION
```

The number sign (#) is used to indicate immediate data, i.e., data to be assembled directly into object code. ^R specifies that the characters MAC are to be converted to Radix-50. This value is then stored in location FILEXT.

6.4 RADIX AND NUMERIC CONTROL FACILITIES

6.4.1 Radix Control and Unary Control Operators

The normal default assumption for numeric values or expression values appearing in a MACRO source program is octal. However, numerous instances may occur where an alternate radix is useful for portions of a program or for variables within a given statement. It may be useful, for example, to declare a given radix for applicability throughout a program or to specify a numeric value or expression value in a manner that causes it to be interpreted as a binary, octal, or decimal value during assembly. In other such instances, it may be useful to complement numeric values or expression values. These MACRO facilities are described in the following sections.

NOTE

When two or more unary operators appear together, modifying the same term, the operators are applied, from right to left, to the term.

6.4.1.1 **.RADIX Directive** - Numbers used in a MACRO source program are initially considered to be octal values; however, you can declare any one of the following radices for applicability throughout the source program or within specific portions of the program:

2, 8, 10

This is accomplished via a .RADIX directive of the form:

```
.RADIX  n
```

where: n represents one of the three acceptable radices listed above. If the argument n is not specified, the octal default radix is assumed.

The argument in the .RADIX directive is always interpreted as a decimal value. Any alternate radix declared in the source program through the .RADIX directive remains in effect until altered by the occurrence of another such directive, i.e., a given radix declaration is valid throughout a program until changed. For example, the statement:

```
.RADIX  10              ;BEGINS A SECTION OF CODE HAVING A
                        ;DECIMAL RADIX.
    .
    .
    .
.RADIX                  ;REVERTS TO OCTAL RADIX.
```

Any value other than null, 2, 8, or 10 specified as an argument in the .RADIX directive causes an error code (A) to be generated in the assembly listing.

In general, macro definitions should not contain or rely on radix settings established with the .RADIX directive. Rather, temporary radix control operators should be used within a macro definition. Where a possible radix conflict exists within a macro definition or in

possible future uses of that code, it is recommended that the user specify numeric or expression values using the temporary radix control operators described below.

6.4.1.2 **Temporary Radix Control Operators: ^D, ^O, and ^B** - Once the user has specified a given radix for a section of code or has decided to use the default octal radix, he may discover a number of cases where an alternate radix is more convenient or desirable (particularly within macro definitions). The creation of a mask word, for example, might best be accomplished through the use of a binary radix.

MACRO has three unary operators that allow the user to establish an alternate radix, as shown below:

```
^D"number"   ("number" is evaluated as a decimal number)
^O"number"   ("number" is evaluated as an octal number)
^B"number"   ("number" is evaluated as a binary number)
```

Thus, an alternate radix can be declared temporarily to meet a localized requirement in the source program. Such a declaration can be made at any time, regardless of the existence of the default octal radix or another specific radix declaration elsewhere in the program. In other words, the effect of a temporary radix control operator is limited to the term or expression immediately following the operator. Any value specified in connection with a temporary radix control operator is evaluated during assembly as a 16-bit entity. Temporary radix control declarations can be included in the source program anywhere a numeric value is legal.

The expressions below are representative of the methods of specifying temporary radix control operators:

```
^D123          Decimal radix
^O 47          Octal Radix
^B 00001101    Binary Radix
^O<A+13>       Octal Radix
```

Note that the up-arrow and the radix control operator may not be separated, but the radix control operator and the following term or expression can be physically separated by spaces or tabs for legibility or formatting purposes. A multi-element term or expression that is to be interpreted in an alternate radix should be enclosed within angle brackets, as shown in the last of the four temporary radix control expressions above.

The following example also illustrates the use of angle brackets to delimit an expression that is to be interpreted in an alternate radix:

```
        .RADIX  10
A=10
        .WORD   ^O<A+10>*10
```

When the temporary radix expression in the .WORD directive above is evaluated, it effectively yields the following equivalent statement:

```
        .WORD   180.
```

MACRO also allows a temporary radix change to decimal by specifying a number, immediately followed by a decimal point (.), as shown below:

```
 100.     Equivalent to 144(8)
1376.     Equivalent to 2540(8)
 128.     Equivalent to 200(8)
```

The above expression forms are equivalent in function to those listed below:

```
^D100
^D1376
^D128
```

6.4.2 Numeric Directives and Unary Control Operators

Two storage directives and two numeric control operators are available to simplify the use of the floating-point hardware on the PDP-11. These facilities allow floating-point data to be created in the program, and numeric values to be complemented or treated as floating-point numbers.

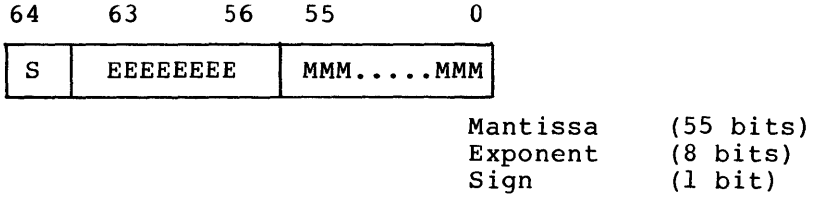
A floating-point number is represented by a string of decimal digits.]xe string (which can be a single digit in length) may optionally contain a decimal point, and may be followed by an optional exponent indicator in the form of the letter E and a signed decimal integer exponent. The number may not contain embedded blanks, tabs or angle brackets and may not be an expression. Such a string will result in one or more errors (A or Q) in the assembly listing.

The list of numeric representations below contains seven distinct, valid representations of the same floating-point number:

```
3
3.
3.0
3.0E0
3E0
.3El
300E-2
```

As can be inferred, the list could be extended indefinitely (e.g., 3000E-3, .03E2, etc.). A leading plus sign is optional (e.g., 3.0 is considered to be +3.0). A leading minus sign complements the sign bit. No other operators are allowed (e.g., 3.0+N is illegal).

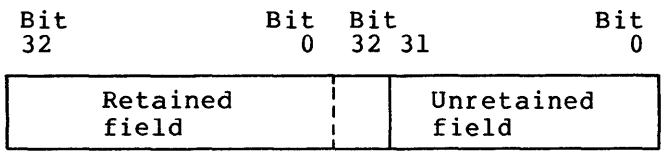
All floating-point numbers are evaluated as 64 bits in the following format:



MACRO returns a value of the appropriate size and precision via one of the floating-point directives. The values returned may be truncated or rounded (see Section 6.2).

Floating-point numbers are normally rounded. That is, when a floating-point number exceeds the limits of the field in which it is to be stored, the high-order bit of the unretained word is added to the low-order bit of the retained word, as shown below. For example, if the number is to be stored in a 2-word field, but more than 32 bits are needed to express its exact value, the highest bit (32) of the unretained field is added to the least significant bit (0) of the

retained field (see illustration below). The .ENABL FPT directive is used to enable floating-point truncation; .DSABL FPT is used to return to floating-point rounding (see Table 6-2).



Note that all numeric operands associated with Floating Point Processor instructions are automatically evaluated as single-word, decimal, floating-point values unless a temporary radix control operator is specified. For example, to add (floating) the octal constant 41040 to the contents of floating accumulator zero, the following instruction must be used:

```
ADDF    #^041040,F0
```

where: F0 is assumed to represent floating accumulator zero.

Floating-point numbers are described in greater detail in the applicable PDP-11 Processor Handbook.

6.4.2.1 **.FLT2 and .FLT4 - Floating-Point Storage Directives** - MACRO supports two directives that evaluate successive floating-point numbers and store the results in the object module. These directives are similar to the .WORD directive and are of the form:

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

where: arg1,arg2,... represent one or more floating point numbers as described in Section 6.4.2. Multiple arguments must be separated by commas.

.FLT2 causes two words of storage to be generated for each argument, while .FLT4 generates four words of storage for each argument.

6.4.2.2 **Temporary Numeric Control Operators: ^C and ^F** - The ^C unary operator allows you to specify an argument that is to be complemented as it is evaluated during assembly. The ^F unary operator allows you to specify an argument consisting of a 1-word floating-point number.

As with the radix control operators described above, the numeric control operator (^C) can be used anywhere in the source program that an expression value is legal. Such a construction is evaluated by MACRO as a 16-bit binary value before being complemented. For example, the following statement:

```
TAG4:   .WORD   ^C151
```

causes the 1's complement of the value 151 (octal) to be stored as a 16-bit value in the program. The resulting value expressed in octal form is 177626(8).

Because the ^C construction is a unary operator, the operator and its argument are regarded as a term. Thus, more than one unary operator may be applied to a single term. For example, the following construction:

```
^C^D25
```

causes the decimal value 25 to be complemented during assembly. The resulting binary value, when expressed in octal form, reduces to 177746(octal).

The term created through the use of the temporary numeric control operator thus becomes an entity that can be used alone or in combination with other expression elements. For example, the following construction:

```
^C2+6
```

is equivalent in function to:

```
<^C2>+6
```

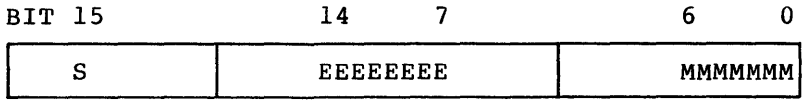
This expression is evaluated during assembly as the 1's complement of 2, plus the absolute value of 6. When these terms are combined, the resulting expression value generates a carry beyond the most significant bit, leaving 000003(8) as the reduced value.

As shown above, when the temporary numeric control operator and its argument are coded as a term within an expression, angle brackets should be used as delimiters to ensure precise evaluation and readability.

MACRO also supports a unary operator for numeric control which allows you to specify an argument consisting of a 1-word floating-point number. For example, the following statement:

```
A:      MOV   #^F3.7,R0
```

creates a 1-word floating-point number at location A+2 containing the value 3.7 formatted as shown below.



Sign (bit 15) Exponent (bits 14-7) Mantissa (bits 6-0)

The importance of ordering with respect to unary operators is shown below.

```
^F1.0    = 020400
^F-1.0   = 120400
-^F1.0   = 157400
-^F-1.0  = 057400
```

The value created by the ^F unary operator and its argument is then a term that can be used)b^Q0itself or in an expression. For example:

```
^C^F6.2
```

is equivalent to:

```
^C<^F6.2>
```

For this reason, the use of angle brackets is advised. Expressions used as terms or arguments of a unary operator must be explicitly grouped.

6.5 LOCATION COUNTER CONTROL DIRECTIVES

The directives used in controlling the value of the current location counter and in reserving storage space in the object program are described in the following sections.

In this connection, it should be noted that several MACRO statements may cause an odd number of bytes to be allocated, as listed below:

1. .BYTE directive
2. .BLKB directive
3. .ASCII or .ASCIZ directive
4. .ODD directive
5. A direct assignment statement of the form .=.+expression, which results in the assignment of an odd address value.

In cases that yield an odd address value, the next word-boundaried instruction automatically forces the location counter to an even value, but that instruction is flagged with an error code (B) in the assembly listing.

6.5.1 .EVEN Directive

The .EVEN directive ensures that the current location counter contains an even value by adding 1 if the current value is odd. If the current location counter is already even, no action is taken. Any operands following an .EVEN directive are flagged with an error code (Q) in the assembly listing.

The .EVEN directive is used as follows:

```
        .ASCIZ  /THIS IS A TEST/
        .EVEN                   ;ENSURES THAT THE NEXT STATEMENT WILL
                                ;BEGIN ON A WORD BOUNDARY.
        .WORD   XYZ
```

6.5.2 .ODD Directive

The .ODD directive ensures that the current location counter contains an odd value by adding 1 if the current value is even. If the current location counter is already odd, no action is taken. Any operands following an .ODD directive are also flagged with an error code (Q) in the assembly listing.

6.5.3 .BLKB and .BLKW Directives

Blocks of storage can be reserved in the object program by means of the .BLKB and .BLKW directives. The .BLKB directive is used to reserve byte blocks; similarly, the .BLKW directive reserves word blocks. The two directives are of the form:

```
.BLKB   exp
.BLKW   exp
```

where: exp represents the specified number of bytes or words to be reserved in the object program. If no argument is present, a default value of 1 is assumed. These directives should not be used without arguments. Any expression that is completely defined at assembly-time and that reduces to an absolute value is legal. If the expression specified in either of these directives is not an absolute value, the statement is flagged with an error code (A) in the assembly listing.

Figure 6-6 illustrates the use of the .BLKB and .BLKW directives.

```
166 000000                      .PSECT  IMPURE,D
167 000000          PASS::      .BLKW   1               ;PASS FLAG
168                                                     ;NEXT GROUP MUST STAY TOGETHER
169 000000                      .PSECT  IMPPAS,D,GBL
170 000000          SYMBOL::    .BLKW   2               ;SYMBOL ACCUMULATOR
171 000004          MODE::                              ;MODE/FLAGS BYTE
172 000004          FLAGS::     .BLKB   1               ;
173 000005          SECTOR::    .BLKB   1               ;SYMBOL/EXPRESSION TYPE
174 000006          VALUE::     .BLKW   1               ;EXPRESSION VALUE
175 000010          RELLVL::    .BLKW   1               ;RELOCATION LEVEL
176         000003              .REPT   MAXXMT-<<.-SYMBOL>/2>
177                             .BLKW   1
178                             .ENDR
179
180 000020          CLCNAM::    .BLKW   2               ;CURRENT LOCATION COUNTER NAME
181 000024          CLCFGS::    .BLKB   1               ;
182 000025          CLCSEC::    .BLKB   1               ;
183 000026          CLCLOC::    .BLKW   1               ;
184 000030          CLCMAX::    .BLKW   1               ;END OF GROUPED DATA
185 000032          CHRPNT::    .BLKW   1               ;CHARACTER POINTER
186 000034          SYMBEG::    .BLKW   1               ;POINTER TO START OF SYMBOL
187 000036          ENDFLG::    .BLKW   1               ;
188 000000                      .PSECT
```

Figure 6-6 Example of .BLKB and .BLKW Directives

The .BLKB directive in a source program has the same effect as the following statement:

```
.=.+expression
```

which causes the value of the expression to be added to the current value of the location counter. The .BLKB directive, however, is easier to interpret in the context of the source code in which it appears and is therefore recommended.

6.6 TERMINATING DIRECTIVES

6.6.1 .END Directive

The .END directive indicates the logical end of the source input, and takes the following form:

```
.END    exp
```

where: exp represents an optional expression value which, if present, indicates the program-entry point, i.e., the transfer address at which program execution is to begin.

When MACRO encounters a valid occurrence of the .END directive, it terminates the current assembly pass. Any additional text beyond this point in the current source file, as well as in additional source files identified in the command line, will be ignored.

When creating a task image consisting of several object modules, only one object module may be terminated with an .END exp statement specifying the starting address. All other object modules must be terminated with an .END statement without an address argument; otherwise, the Linker will issue a diagnostic message. If no starting address is specified in any of the object modules, task execution will begin at location 1 of the task and immediately fault because of an odd addressing error.

The .END statement must not be used within a macro expansion or a conditional assembly block; if it is so used, it is flagged with an error code (O) in the assembly listing. The .END statement may be used, however, in an immediate conditional statement (see Section 6.10.2).

If the source program input is not terminated with an .END directive, an error code (E) results in the assembly listing.

6.6.2 .EOT Directive

Under the TRAX operating system, the MACRO .EOT directive is ignored and simply treated as a directive without effect, i.e., as a no-op.

6.7 PROGRAM BOUNDARIES DIRECTIVE: .LIMIT

It is often desirable to know the upper and lower address boundaries of the task image. When the .LIMIT directive is specified in the source program, MACRO effectively generates the following instruction:

```
.BLKW    2
```

causing two storage words to be reserved in the object module. Later, at link time, the address of the bottom of the task's stack is inserted into the first reserved word, and the address of the first free word following the task image is inserted into the second reserved word.

During linking, the size of the task image is rounded upward to the nearest 2-word boundary.

For a discussion of task memory allocation and mapping, refer to the applicable Linker reference manual (see Section 0.3 in the Preface).

6.8 PROGRAM SECTIONING DIRECTIVES

The MACRO program sectioning directives are used to declare names for program sections and to establish certain program section attributes essential to Linker processing.

6.8.1 .PSECT Directive

The .PSECT directive allows absolute control over the memory allocation of a program at link time, because any program attributes established through this directive are passed to the Linker.

For example, if you are writing programs for a multi-user environment, a program section containing pure code (instructions only) or a program section containing impure code (data only) may be explicitly declared through the .PSECT directive. Furthermore, these program sections may be explicitly declared as read-only code, qualifying them for use as protected, reentrant programs.

In addition, program sections exhibiting the global (GBL) attribute can be explicitly allocated in a task's overlay structure at link time.

The advantages gained through sectioning programs in this manner therefore relate primarily to control of memory allocation, program modularity, and more effective partitioning of memory. Refer to the applicable Linker reference manual for a discussion of memory allocation (see Section 0.3 in the Preface).

The .PSECT directive is formatted as follows:

```
.PSECT  name,arg1,arg2,...argn
```

where:

name	represents the symbolic name of the program section, as described in Table 6-3.
,	represents any legal separator (comma, tab and/or space).
arg1, arg2,... argn	represent one or more of the legal symbolic arguments defined for use with the .PSECT directive, as described in Table 6-3. The slash separating each pair of symbolic arguments listed in the table indicates that these optional arguments are mutually exclusive, i.e., one or the other, but not both, may be specified. Multiple arguments must be separated by a legal separating character. Any symbolic argument specified in the .PSECT directive other than those listed in Table 6-3 will cause that statement to be flagged with an error code (A) in the assembly listing.

Table 6-3
Symbolic Arguments of .PSECT Directive

Argument	Default	Meaning
NAME	Blank	Establishes the program section name, which is specified as one to six Radix-50 characters. If this argument is omitted, a comma must appear in place of the name parameter. The Radix-50 character set is listed in Section A.2 of Appendix A.
RO/RW	RW	Defines which type of access is permitted to the program section: RO=Read-Only Access RW=Read/Write Access NOTE IAS and RSX-11D set hardware protection for RO program sections. TRAX does not provide such protection.
I/D	I	Defines the program section as containing either instructions (I) or data (D). These attributes allow the Linker to differentiate global symbols that are program entry-point instructions (I) from those that are data values (D).
GBL/LCL	LCL	Defines the scope of the program section, as subsequently interpreted by the Linker. In building single-segment programs, the GBL/LCL arguments have no meaning, because the total memory allocation for the program will go into the root segment of the task. The GBL/LCL arguments apply only in the case of overlays. If an object module contains a local program section, then the storage allocation for that module will occur within the segment in which the module resides. Many modules can reference this same program section, and the memory allocation for each module is either concatenated or overlaid within the segment, depending on the argument of the program section (.PSECT) defining

(continued on next page)

Table 6-3 (Cont.)
Symbolic Arguments of .PSECT Directive

Argument	Default	Meaning
GBL/LCL (cont'd)	LCL	its allocation requirements (see CON/OVR below). If an object module contains a global program section, the contributions to this program section are collected across segment boundaries, and the allocation of memory for that section will go into the segment nearest the root in which the first contribution to this program section appeared. (The term contribution implies an allocation of memory to the program section.)
ABS/REL	REL	Defines the relocatability attribute of the program section: ABS=Absolute (non-relocatable). When the ABS argument is specified, the program section is regarded by the Linker as an absolute module, thus requiring no relocation. The program section is assembled and loaded, starting at absolute virtual address 0. The location of data in absolute program sections must fall within the virtual memory limits of the segment containing the program section; otherwise, an error results at link time. For example, the following code, although valid at during assembly, may generate a Linker error message if virtual location 100000 is outside the segment's virtual address space: .PSECT ALPHA,ABS .=.+100000 .WORD X The above coding assembles properly, but the resulting load address may be outside the respective segment's boundaries. In such cases, the Linker recognizes this as an attempt to load data outside the task image and rwith an error message. REL=Relocatable. When the REL argument is specified, the Linker calculates a relocation bias and adds it to all references to locations within the program section, i.e., all references to the program section must have a relocation bias added to them to make them absolute.

(continued on next page)

Table 6-3 (Cont.)
Symbolic Arguments of .PSECT Directive

Argument	Default	Meaning
CON/OVR	CON	Defines the allocation requirements of the program section: CON=Concatenated. All program section contributions are to be concatenated with other references to this same program section in order to determine the total memory allocation requirement for this program section. OVR=Overlaid. All program section contributions are to be overlaid. Thus, the total allocation requirement for the program section is equal to the largest allocation request made by any individual contribution to this program section.

The only argument in the .PSECT directive that is position-dependent is NAME. If it is omitted, a comma must be used in its place. For example, the directive:

```
.PSECT  ,GBL
```

shows a .PSECT directive with a blank name argument and the GBL argument. Default values (see Table 6-3) are assumed for all other unspecified arguments.

Once the attributes of a program section are declared through a .PSECT directive, MACRO assumes that these attributes remain in effect for all subsequent .PSECT directives of the same name that are encountered within the module.

MACRO provides for 256(10) program sections, as listed below:

1. One default absolute program section (. ABS.)
2. One default unnamed relocatable program section
3. Two-hundred-fifty-four named program sections.

The .PSECT directive enables the user to:

1. Create program sections (see Section 6.8.1.1)
2. Share code and data among program sections (see Section 6.8.1.2).

For each program section specified or implied, MACRO maintains the following information:

1. Program section name
2. Contents of the current location counter

3. Maximum location counter value encountered

4. Program section attributes, i.e., the .PSECT arguments described in Table 6-3 above.

6.8.1.1 **Creating Program Sections** - MACRO automatically begins assembling source statements at relocatable zero of the unnamed program section, i.e., the first statement of a source program is always an implied .PSECT directive.

The first occurrence of a .PSECT directive with a given name assumes that the current location counter is set at relocatable zero. The scope of this directive then extends until a directive declaring a different program section is specified. Further occurrences of a program section name in subsequent .PSECT statements cause the resumption of assembly where that section previously ended. For example:

```
        .PSECT                  ;DECLARES UNNAMED RELOCATABLE PROGRAM
A:      .WORD   0               ;SECTION ASSEMBLED AT RELOCATABLE
B:      .WORD   0               ;ADDRESSES 0, 2, AND 4.
C:      .WORD   0
        .PSECT  ALPHA           ;DECLARES RELOCATABLE PROGRAM SECTION
X:      .WORD   0               ;NAMED ALPHA ASSEMBLED AT RELOCATABLE
Y:      .WORD   0               ;ADDRESSES 0 AND 2.
        .PSECT                  ;RETURNS TO UNNAMED RELOCATABLE
D:      .WORD   0               ;PROGRAM SECTION AND CONTINUES ASSEM-
                                ;BLY AT RELOCATABLE ADDRESS 6.
```

A given program section may be defined completely upon encountering its first .PSECT directive. Thereafter, the section can be referenced by specifying its name only, or by completely respecifying its attributes. For example, a program section can be declared through the directive:

```
        .PSECT  ALPHA,ABS,OVR
```

and later referenced through the equivalent directive:

```
        .PSECT  ALPHA
```

which requires no arguments.

By maintaining separate location counters for each program section, MACRO allows the user to write statements that are not physically contiguous within the program, but that can be loaded contiguously following assembly, as shown in the following example.

```
        .PSECT  SEC1,REL,RO     ;START A RELOCATABLE PROGRAM SECTION
A:      .WORD   0               ;NAMED SEC1 ASSEMBLED AT RELOCATABLE
B:      .WORD   0               ;ADDRESSES 0, 2, AND 4.
C:      .WORD   0
ST:     CLR     A               ;ASSEMBLE CODE AT RELOCATABLE
        CLR     B               ;ADDRESSES 6 THROUGH 12.
        CLR     C
        .PSECT  SECA,ABS        ;START AN ABSOLUTE PROGRAM SECTION
                                ;NAMED SECA.  ASSEMBLE CODE AT
        .WORD   .+2,A           ;ABSOLUTE ADDRESSES 0 AND 2.
        .PSECT  SEC1            ;RESUME RELOCATABLE PROGRAM SECTION
        INC     A               ;SEC1.  ASSEMBLE CODE AT RELOCATABLE
        BR      ST              ;ADDRESSES 14 AND 16.
```

All labels in an absolute program section are absolute; likewise, all labels in a relocatable section are relocatable. The current location counter symbol (.) is also relocatable or absolute when referenced in a relocatable or absolute program section, respectively.

Any labels appearing on a line containing a .PSECT (or .ASECT or .CSECT) directive are assigned the value of the current location counter before the .PSECT (or other) directive takes effect. Thus, if the first statement of a program is:

```
A:      .PSECT  ALT,REL
```

the label A is assigned to relocatable address zero of the unnamed (or blank) program section.

It is not known during assembly where relocatable program sections will be loaded, therefore all references between relocatable sections in a single assembly are translated by MACRO to references relative to the base of the referenced section. Thus, MACRO provides the Linker with the necessary information to resolve the linkages between various program sections. Such information is not necessary, however, when referencing an absolute program section, because all instructions in an absolute program section are associated with an absolute virtual address.

In the following example, references to the symbols X and Y are translated into references relative to the base of the relocatable program section named SEN.

```
        .PSECT  ENT,ABS
.=.+1000
A:      CLR     X               ;ASSEMBLED AS CLR BASE OF
                                ;RELOCATABLE SECTION + 10.
        JMP     Y               ;ASSEMBLED AS JMP BASE OF
                                ;RELOCATABLE SECTION + 6.
        .PSECT  SEN,REL
        MOV     R0,R1
        JMP     A               ;ASSEMBLED AS JMP 1000.
Y:      HALT
X:      .WORD   0
```

NOTE

In the preceding example, using a constant in conjunction with the current location counter symbol (.) in the form .=1000 would result in an error, because constants are always absolute and are always associated with the program's .ASECT (. ABS.). If the form .=1000 were used, a program section incompatibility would be detected. See Section 3.6 for a discussion of the current location counter.

6.8.1.2 **Code or Data Sharing** - Named relocatable program sections with the arguments GBL and OVR operate in the same manner as FORTRAN COMMON, i.e., program sections of the same name with the arguments GBL and OVR from different assemblies are all loaded at the same location by the Linker. All other program sections, i.e., those with the argument CON, are concatenated.

Note that no conflict exists between internal symbolic names and program section names, i.e., it is legal to use the same symbolic name for both purposes. Considering FORTRAN again, using the same symbolic name is necessary to accommodate the following statement:

```
COMMON  /X/ A,B,C,X
```

where the symbol X represents the base of the program section and also the fourth element of that section.

6.8.1.3 **Memory Allocation Considerations** - The assembler does not generate an error when a module ends at an odd location. This allows you to place odd length data at the end of a module. However, when several modules contain object code contributions to the same program section having the concatenate attribute (see Table 6-3), odd length modules (except the last) may cause the Linker to link succeeding modules starting at odd locations, thereby making the linked program unexecutable. To avoid this problem, code and data should be separated from each other and be placed in separately named program sections. This permits the Linker to automatically begin each program section on an even address. Refer to the applicable Linker reference manual for further information on memory allocation of tasks (see Section 0.3 in the Preface).

6.8.2 .ASECT and .CSECT Directives

TRAX assembly-language programs use the .PSECT and .ASECT directives exclusively, since the .PSECT directive provides all the capabilities of the .CSECT directive defined for other PDP-11 assemblers. MACRO will accept both .ASECT and .CSECT directives, but assembles them as though they were .PSECT directives with the default attributes listed in Table 6-4. Also, compatibility exists between other MACRO programs and the TRAX Linkers, since the respective Linkers recognize the .ASECT and .CSECT directives that appear in such programs and likewise assign the default values listed in Table 6-4.

Table 6-4
Non-TRAX Program Section Default Values

Attribute	Default Value		
	.ASECT	.CSECT (named)	.CSECT (unnamed)
Name	. ABS.	name	Blank
Access	RW	RW	RW
Type	I	I	I
Scope	GBL	GBL	LCL
Relocation	ABS	REL	REL
Allocation	OVR	OVR	CON

The allowable syntactical forms of the .ASECT and .CSECT directives are:

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

Note that the statement:

```
.CSECT  JIM
```

is identical to the statement:

```
.PSECT  JIM,GBL,OVR
```

because the .CSECT default values GBL and OVR are assumed for the named program section.

6.9 SYMBOL CONTROL DIRECTIVE: .GLOBL

MACRO produces a relocatable object module and a listing file containing the assembly listing and symbol table. The Linker joins separately-assembled object modules into a single executable task image. During linking, object modules are relocated as a function of the specified base of the module. The object modules are then linked via global symbols, such that a global symbol in one module, defined either by a global assignment operator (==), a global label operator (::), or the .GLOBL directive can be referenced from another module. Thus, all symbols which will be referenced by other program modules must be singled out as global symbols in the defining modules.

The .GLOBL directive is provided to define (and thus provide linkage to) symbols not otherwise defined as global symbols within a module. For example, if the .DSABL GBL directive is in effect (see Section 6.2), .GLOBL directives might be included in a source program to effect linkage to library routines. For a global symbol definition, the directive .GLOBL A,B,C is equivalent to:

```
A==expression (or A::)
B==expression (or B::)
C==expression (or C::)
```

Thus, the general form of the .GLOBL directive is:

```
.GLOBL  sym1,sym2,...symn
```

where:

sym1, sym2,... symn	represent legal symbolic names. When multiple symbols are specified, they are separated by any legal separator (comma, space, and/or tab).

A .GLOBL directive may also embody a label field and/or a comment field.

At the end of assembly pass 1, MACRO determines whether a given global symbol is defined within the current program module or whether it is to be treated as an external symbol. All internal symbols appearing within a given program must be defined at the end of assembly pass 1 or they will be assumed to be default global references. Refer to Section 6.2 for a description of enabling/disabling of global references.

In the example below, A and B are entry-point symbols. The symbol A has been explicitly defined as a global symbol by means of the .GLOBL directive, and the symbol B has been explicitly defined as a global label by means of the double colon (::). Since the symbol C is not defined as a label within the current assembly, it is an external (global) reference.

```
;
; DEFINE A SUBROUTINE WITH 2 ENTRY POINTS WHICH CALLS AN
; EXTERNAL SUBROUTINE
;
        .PSECT                  ;DECLARE THE UNNAMED PROGRAM SECTION.
        .GLOBL  A               ;DEFINE A AS A GLOBAL SYMBOL.
A:      MOV     @(R5)+,R0       ;DEFINE ENTRY POINT A.
        MOV     #X,R1
X:      JSR     PC,C            ;CALL EXTERNAL SUBROUTINE C.
        RTS     R5              ;EXIT.
B::     MOV     (R5)+,R1        ;DEFINE ENTRY POINT B.
        CLR     R2
        BR      X
```

External symbols can appear in the operand field of an instruction or MACRO directive as a direct reference, as shown in the examples below:

```
        CLR     EXT
        .WORD   EXT
        CLR     @EXT
```

External symbols may also appear as a term within an expression, as shown below:

```
        CLR     EXT+A
        .WORD   EXT-2
        CLR     @EXT+A(R1)
```

It should be noted that an undefined external symbol cannot be used in the evaluation of a direct assignment statement or as an argument in a conditional assembly directive (see Sections 6.10.1 and 6.10.3).

6.10 CONDITIONAL ASSEMBLY DIRECTIVES

Conditional assembly directives allow you to include or exclude blocks of source code during the assembly process, based on the evaluation of stated condition tests within the body of the program. This capability allows several variations of a program to be generated from the same source code.

6.10.1 Conditional Assembly Block Directives: .IF, .ENDC

The general form of a conditional assembly block is as follows:

```
        .IF     cond,argument(s) ;START CONDITIONAL ASSEMBLY BLOCK.
          .
          .
          .
        range                    ;RANGE OF CONDITIONAL ASSEMBLY BLOCK.
          .
          .
          .
        .ENDC                    ;END OF CONDITIONAL ASSEMBLY BLOCK.
```

where: cond — represents a specified condition that must be met if the block is to be included in the assembly. The conditions that may be tested by the conditional assembly directives are defined in Table 6-5.

represents any legal separator (comma, space, and/or tab).

argument(s) — represent(s) the symbolic argument(s) or expression(s) of the specified conditional test. These arguments are thus a function of the specified condition to be tested (see Table 6-5).

range — represents the body of code that is either included in the assembly or excluded, depending upon whether the specified condition is met.

.ENDC — terminates the conditional assembly block. This directive must be present to end the conditional assembly block.

A condition test other than those listed in Table 6-5, an illegal argument, or a null argument specified in an .IF directive causes that line to be flagged with an error code (A) in the assembly listing.

Table 6-5
Legal Condition Tests for Conditional Assembly Directives

Conditions		Arguments	Assemble Block If:
Positive	Complement		
EQ	NE	Expression	Expression is equal to 0 (or not equal to 0).
GT	LE	Expression	Expression is greater than 0 (or less than or equal to 0).
LT	GE	Expression	Expression is less than 0 (or greater than or equal to 0).
DF	NDF	Symbolic argument	Symbol is defined (or not defined).
B	NB	Macro-type argument	Argument is blank (or non-blank).
IDN	DIF	Two macro-type arguments	Arguments are identical (or different).
Z	NZ	Expression	Same as EQ/NE.
G	L	Expression	Same as GT/LT.

NOTE

A macro-type argument (which is a form of symbolic argument), as shown below, is enclosed within angle brackets or denoted with an up-arrow construction (as described in Section 7.3.1).

```
<A,B,C>
^/124/
```

An example of a conditional assembly directive follows:

```
.IF EQ  ALPHA+1          ;ASSEMBLE BLOCK IF ALPHA+1=0.
  .
  .
  .
.ENDC
```

The two operators & and ! have special meaning within DF and NDF conditions, in that they are allowed in grouping symbolic arguments.

&	Logical AND operator
!	Logical inclusive OR operator

For example, the conditional assembly statement:

```
.IF DF  SYM1 & SYM2
  .
  .
  .
.ENDC
```

results in the assembly of the conditional block if the symbols SYM1 and SYM2 are both defined.

Nested conditional directives take the form:

```
Conditional Assembly Directive
Conditional Assembly Directive
  .
  .
  .
.ENDC
.ENDC
```

For example, the following conditional directives:

```
.IF DF  SYM1
.IF DF  SYM2
  .
  .
  .
.ENDC
.ENDC
```

can govern whether assembly is to occur. In the example above, if the outermost condition is unsatisfied, no deeper level of evaluation of nested conditional statements within the program occurs.

Each conditional assembly block must be terminated with an .ENDC directive. An .ENDC directive encountered outside a conditional assembly block is flagged with an error code (O) in the assembly listing.

MACRO permits a nesting depth of 16(10) conditional assembly levels. Any statement that attempts to exceed this nesting level depth is flagged with an error code (O) in the assembly listing.

6.10.2 Subconditional Assembly Block Directives: .IFF, .IFT, .IFTF

Subconditional directives may be placed within conditional assembly blocks to indicate:

1. The assembly of an alternate body of code when the condition of the block tests false.

2. The assembly of a non-contiguous body of code within the conditional assembly block, depending upon the result of the conditional test in entering the block.

3. The unconditional assembly of a body of code within a conditional assembly block.

The subconditional directives are described in detail in Table 6-6. If a subconditional directive appears outside a conditional assembly block, an error code (O) is generated in the assembly listing.

Table 6-6
Subconditional Assembly Block Directives

Subconditional Directive	Function
.IFF	If the condition tested upon entering the conditional assembly block is false, the code following this directive, and continuing up to the next occurrence of a subconditional directive or to the end of the conditional assembly block, is to be included in the program.
.IFT	If the condition tested upon entering the conditional assembly block is true, the code following this directive, and continuing up to the next occurrence of a subconditional directive or to the end of the conditional assembly block, is to be included in the program.
.IFTF	The code following this directive, and continuing up to the next occurrence of a subconditional directive or to the end of the conditional assembly block, is to be included in the program, regardless of the result of the condition tested upon entering the conditional assembly block.

The implied argument of a subconditional directive is the condition test specified upon entering the conditional assembly block, as reflected by the initial directive in the conditional coding examples

below. Conditional or subconditional directives in nested conditional assembly blocks are not evaluated if the previous (or outer) condition in the block is not satisfied. Examples 3 and 4 below illustrate nested directives that are not evaluated because of previous unsatisfied conditional coding.

EXAMPLE 1: Assume that symbol SYM is defined.

```
        .IF DF  SYM             ;TESTS TRUE, SYM IS DEFINED. ASSEMBLE
          .                     ;THE FOLLOWING CODE.
          .
          .
        .IFF                    ;TESTS FALSE. SYM IS DEFINED. DO NOT
          .                     ;ASSEMBLE THE FOLLOWING CODE.
          .
          .
        .IFT                    ;TESTS TRUE. SYM IS DEFINED. ASSEM-
          .                     ;BLE THE FOLLOWING CODE.
          .
          .
        .IFTF                   ;ASSEMBLE FOLLOWING CODE UNCONDITION-
          .                     ;ALLY.
          .
          .
        .IFT                    ;TESTS TRUE. SYM IS DEFINED. ASSEM-
          .                     ;BLE REMAINDER OF CONDITIONAL ASSEM-
          .                     ;BLY BLOCK.
          .
        .ENDC
```

EXAMPLE 2: Assume that symbol X is defined and that symbol Y is not defined.

```
        .IF DF  X               ;TESTS TRUE, SYMBOL X IS DEFINED.
        .IF DF  Y               ;TESTS FALSE, SYMBOL Y IS NOT DEFINED.
        .IFF                    ;TESTS TRUE, SYMBOL Y IS NOT DEFINED,
          .                     ;ASSEMBLE THE FOLLOWING CODE.
          .
          .
        .IFT                    ;TESTS FALSE, SYMBOL Y IS NOT DEFINED.
          .                     ;DO NOT ASSEMBLE THE FOLLOWING CODE.
          .
          .
        .ENDC
        .ENDC
```

EXAMPLE 3: Assume that symbol A is defined and that symbol B is not defined.

```
        .IF DF  A               ;TESTS TRUE.  A IS DEFINED.
                                ;ASSEMBLE THE FOLLOWING CODE.
        MOV     A,R1
          .
          .
          .
        .IFF                    ;TESTS FALSE.  A IS DEFINED.  DO NOT
                                ;ASSEMBLE THE FOLLOWING CODE.
        MOV     R1,R0
          .
          .
          .
        .IF NDF B               ;NESTED CONDITIONAL DIRECTIVE IS NOT
```

```
          .                         ;EVALUATED.
          .
          .
        .ENDC
        .ENDC
```

EXAMPLE 4: Assume that symbol X is not defined and that symbol Y is defined.

```
        .IF DF  X                   ;TESTS FALSE. SYMBOL X IS NOT DEFINED.
                                    ;DO NOT ASSEMBLE THE FOLLOWING CODE.
        .IF DF  Y                   ;NESTED CONDITIONAL DIRECTIVE IS NOT
          .                         ;EVALUATED.
          .
          .
        .IFF                        ;NESTED SUBCONDITIONAL DIRECTIVE IS
          .                         ;NOT EVALUATED.
          .
          .
        .IFT                      ;NESTED SUBCONDITIONAL DIRECTIVE IS
          .                         ;NOT EVALUATED.
          .
          .
        .ENDC
        .ENDC
```

6.10.3 Immediate Conditional Assembly Directive: .IIF

An immediate conditional assembly directive provides a means for writing a 1-line conditional assembly block. In using this directive, no terminating .ENDC statement is required, and the condition to be tested is completely expressed within the line containing the directive. Immediate conditional assembly directives are of the form:

```
        .IIF    cond,arg,statement
```

where:

- **cond** — represents one of the legal condition tests defined for conditional assembly blocks in Table 6-5.
- **,** — represents any legal separator (comma, space, and/or tab).
- **arg** — represents the argument associated with the immediate conditional directive, i.e., an expression, symbolic argument, or macro-type argument, as described in Table 6-5.
- **,** — represents the separator between the conditional argument and the statement field. If the preceding argument is an expression, then a comma must be used; otherwise, a comma, space, and/or tab may be used.
- **statement** — represents the specified statement to be assembled if the condition is satisfied.

For example, the immediate conditional statement:

```
        .IIF    DF FOO,BEQ ALPHA
```

generates the code

```
        BEQ     ALPHA
```

if the symbol FOO is defined within the source program.

As with the .IF directive, a condition test other than those listed in Table 6-5, an illegal argument, or a null argument specified in an .IIF directive results in an error code (A) in the assembly listing.

CHAPTER 7

MACRO DIRECTIVES

7.1 DEFINING MACROS

In assembly-language programming, it is often convenient and desirable to generate a recurring coding sequence by invoking a single statement within the program. In order to do this, the desired coding sequence is first established with dummy arguments as a macro definition. Once a macro has been defined, a single statement calling the macro by name with a list of real arguments (replacing the corresponding dummy arguments in the macro definition) generates the desired coding sequence. This sequence is called the macro expansion.

7.1.1 .MACRO Directive

The first statement of a macro definition must be a .MACRO directive. This directive takes the form:

```
label:    .MACRO  name, dummy argument list
```

where:

label	represents an optional statement label.
name	represents the programmer-assigned symbolic name of the macro. This name may be any legal symbol and may be used as a label elsewhere in the program.
'	represents any legal separator (comma, space, and/or tab).
dummy argument list	represents a number of legal symbols (see 3.2.2) that may appear anywhere in the body of the macro definition, even as a label. These dummy symbols can be used elsewhere in the program with no conflict of definition. Multiple dummy arguments specified in this directive may be separated by any legal separator. The detection of a duplicate or an illegal symbol in a dummy argument list terminates the scan and causes an error code to be generated.

A comment may follow the dummy argument list in a .MACRO directive, as shown below:

```
.MACRO  ABS  A,B          ;DEFINES MACRO ABS WITH TWO ARGUMENTS.
```

NOTE

Although it is legal for a label to appear on a .MACRO directive, this practice is discouraged, especially in the case of nested macro definitions, because invalid. labels or labels constructed with the concatenation
@ character will cause the macro directive to be ignored. This may result in improper termination of the macro definition. This NOTE also applied to .IRP, .IRPC, and .REPT.

7.1.2 .ENDM Directive

The final statement of every macro definition must be an .ENDM directive of the form:

```
.ENDM   name
```

where: name represents an optional argument specifying the symbolic name of the macro being terminated by the directive, as shown in the following example:

```
.ENDM                   ;TERMINATES THE CURRENT
                        ;MACRO DEFINITION.

.ENDM   ABS             ;TERMINATES THE CURRENT
                        ;MACRO DEFINITION NAMED ABS.
```

If specified, the symbolic name in the .ENDM statement must match the name specified in the corresponding .MACRO directive. Otherwise, the statement is flagged with an error code (A) in the assembly listing (see Appendix D). In either case, the current macro definition is terminated. Specifying the macro name in the .ENDM statement thus permits MACRO to detect missing .ENDM statements or improperly-nested macro definitions.

The .ENDM directive may be followed by a comment field, but must not contain a label, as shown below:

```
.MACRO  TYPMSG MESSGE   ;TYPE A MESSAGE.
JSR     R5,TYPMSG
.WORD   MESSGE
.ENDM                   ;END OF TYPMSG MACRO.
```

An .ENDM statement encountered by MACRO outside a macro definition is flagged with an error code (O) in the assembly listing (see Appendix D).

NOTES

1. Labels on .ENDM directives are ignored.

2. Illegal labels will cause the directive to be bypassed.

7.1.3 .MEXIT Directive

The .MEXIT directive may be used to terminate a macro expansion before the end of the macro is encountered. This directive is also legal within repeat blocks (see Sections 7.6 and 7.7). It is most useful in the context of nested macros. The .MEXIT directive terminates the current macro as though an .ENDM directive had been encountered. Using the .MEXIT directive bypasses the complexities of nested conditional directives and alternate assembly paths, as shown in the following example:

```
.MACRO  ALTR N,A,B
   .
   .
   .
.IF EQ  N               ;START CONDITIONAL ASSEMBLY BLOCK.
   .
   .
   .
.MEXIT                  ;TERMINATE MACRO EXPANSION.
.ENDC                   ;END CONDITIONAL ASSEMBLY BLOCK.
   .
   .
   .
.ENDM                   ;NORMAL END OF MACRO.
```

Considering the above macro, in an assembly where the real argument for the dummy symbol N is equal to zero (see Table 6-5), the conditional block would be assembled, and the macro expansion would be terminated by the .MEXIT directive. When macros are nested, a .MEXIT directive causes an exit to the next higher level of macro expansion.

A .MEXIT directive encountered outside a macro definition is flagged with an error code (O) in the assembly listing.

7.1.4 MACRO Definition Formatting

A form-feed character used within a macro definition causes a page eject during the assembly of the macro definition. A page eject, however, is not performed when the macro is expanded.

Conversely, when the .PAGE directive is specified within a macro definition, it is ignored during the assembly of the macro definition, but a page eject is performed when that macro is expanded.

7.2 CALLING MACROS

A macro definition must be established by means of the .MACRO directive (see Section 7.1.1) before the macro can be expanded within the source program. Macro calls are of the general form:

```
label:  name    real arguments
```

where:

- label — represents an optional statement label.
- name — represents the name of the macro, as specified in the .MACRO directive (see Section 7.1.1).

real arguments	represent symbolic arguments which replace the dummy arguments specified in the .MACRO directive. When multiple arguments are specified, they are separated by any legal separator. Arguments to the macro call are treated as character strings whose usage is determined by the macro definition. Note that MACRO accepts the ASCII value of lower-case alphabetic characters when .ENABL LC has been specified.

When a macro name is the same as a user label, the appearance of the symbol in the operator field designates the symbol as a macro call; the appearance of the symbol in the operand field designates it as a label, as shown below:

```
ABS:    MOV     (R0),R1         ;ABS IS DEFINED AS A LABEL.
         .
         .
         .
        BR      ABS             ;ABS IS CONSIDERED TO BE A LABEL.
         .
         .
         .
        ABS     #4,ENT,LAR      ;ABS IS A MACRO CALL.
```

7.3 ARGUMENTS IN MACRO DEFINITIONS AND MACRO CALLS

Arguments within a macro definition or macro call are separated from other arguments by any of the legal separating characters described in Section 3.1.1.

Macro definition arguments (dummy) and macro call arguments (real) normally maintain a strict positional relationship. That is, the first real argument in a macro call corresponds with the first dummy argument in a macro definition. Only the use of keyword arguments in a macro call can override this correspondence (see Section 7.3.6).

For example, the following macro definition and its associated macro expansion contain multiple arguments:

```
        .MACRO  REN A,B,C
         .
         .
         .
        REN     ALPHA,BETA,<C1,C2>
```

Arguments which themselves contain separating characters must be enclosed in paired angle brackets, as shown above. For example, the macro call:

```
        REN     <MOV    X,Y>,#44,WEV
```

causes the entire expression

```
        MOV     X,Y
```

to replace all occurrences of the symbol A in the macro definition. Real arguments within a macro call are considered to be character strings and are treated as a single entity during the macro expansion.

The up-arrow (^) construction is provided to allow angle brackets to be passed as part of the argument. This construction, for example, could have been used in the above macro call, as follows:

```
REN     ^/<MOV X,Y>/,#44,WEV
```

causing the entire character string <MOV X,Y> to be passed as an argument.

The following macro call:

```
REN     #44,WEV^/MOV X,Y/
```

however, contains only two arguments (#44 and WEV^/MOV X,Y/), because the up-arrow is a unary operator (see Section 3.1.3) and it is not preceded by an argument separator.

As shown in the examples above, spaces can be used within bracketed argument constructions to increase the legibility of such expressions.

7.3.1 Macro Nesting

The nesting of macros, where the expansion of one macro includes a call to another, causes one set of angle brackets in the macro definition to be removed from an argument with each nested call. The depth of nesting allowed is dependent upon the amount of dynamic memory used by the source program being assembled.

To pass an argument containing legal argument delimiters to nested macros, the argument in the macro definition should be enclosed within one set of angle brackets for each level of nesting, as shown in the coding sequence below. It should be noted that this extra set of angle brackets for each level of nesting is required in the macro definition, not in the macro call.

```
.MACRO  LEVEL1 DUM1,DUM2
LEVEL2  <DUM1>
LEVEL2  <DUM2>
.ENDM

.MACRO  LEVEL2 DUM3
DUM3
ADD     #10,R0
MOV     R0,(R1)+
.ENDM
```

A call to the LEVEL1 macro, as shown below, for example:

```
LEVEL1  <MOV    X,R0>,<MOV      R2,R0>
```

causes the following macro expansion to occur:

```
MOV     X,R0
ADD     #10,R0
MOV     R0,(R1)+
MOV     R2,R0
ADD     #10,R0
MOV     R0,(R1)+
```

When macro definitions are nested, i.e., when a macro definition is contained entirely within the definition of another macro, the inner definition is not a callable macro until the outer macro has been called and expanded. For example, in the following coding:

```
        .MACRO  LV1 A,B
           .
           .
           .
        .MACRO  LV2 C
           .
           .
           .
        .ENDM
        .ENDM
```

the LV2 macro cannot be called and expanded until the LV1 macro has been so invoked. Likewise, any macro defined within the LV2 macro definition cannot be called and expanded until LV2 has also been invoked.

7.3.2 Special Characters in Macro Arguments

An argument may include special characters without enclosing them in a bracketed construction if that argument does not contain spaces, tabs, semicolons, or commas. For example, the macro definition:

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

causes the following code to be generated:

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

7.3.3 Passing Numeric Arguments as Symbols

When macro arguments are passed, an absolute symbol value can be passed which is treated by the macro as a numeric string. An argument preceded by the unary operator backslash (\) is treated as a numeric value in the current program radix. The ASCII characters representing this value are inserted in the macro expansion, and their function is defined in the context of the resulting code, as shown in the following example:

```
        .MACRO  INC A,B
        CON     A,\B            ;B IS TREATED AS A NUMBER IN CURRENT
B=B+1                           ;PROGRAM RADIX.
        .ENDM
        .MACRO  CON A,B
A'B:    .WORD   4               ;A'B IS DESCRIBED IN SECTION 7.3.6.
        .ENDM
          .
          .
          .
C=0     INC     X,C
```

The above macro call (INC) would thus expand to:

```
X0:     .WORD   4
```

Note in this expanded code that the label X0: is the result of the concatenation of two real arguments. The single quote (') character in the label A'B: causes the real arguments X and 0 to be concatenated as they are passed during the expansion of the macro. This type of argument construction is described in further detail in Section 7.3.6.

A subsequent call to the same macro would generate the following code:

```
X1:     .WORD   4
```

and so on, for later calls. The two macro definitions are necessary because the symbol associated with dummy argument B (i.e., C) cannot be updated in the CON macro definition, because its numeric value has already been substituted for its symbolic name, i.e., the character 0 has replaced C in the argument string. In the CON macro definition, the number passed is treated as a string argument. (Where the value of the real argument is 0, only a single 0 character is passed to the macro expansion.)

Passing numeric values in this manner is useful in identifying source listings. For example, versions of programs created through conditional assemblies of a single source program can be identified through such coding as that shown below. Assume, for example, that the symbol ID in the macro call (IDT) has been equated elsewhere in the source program to the value 6.

```
        .MACRO  IDT  SYM         ;ASSUME THAT THE SYMBOL ID TAKES
        .IDENT  /V05A'SYM/       ;ON A UNIQUE 2-DIGIT VALUE.
        .ENDM                    ;WHERE V05A IS THE UPDATE
            .                    ;VERSION OF THE PROGRAM.
            .
            .
        IDT     \ID
```

The above macro call would then expand to:

```
        .IDENT  /V05A6/
```

where 6 is the numeric value of the symbol ID.

7.3.4 Number of Arguments in Macro Calls

If more arguments appear in the macro call than in the macro definition, an error code (Q) is generated in the assembly listing. If fewer arguments appear in the macro call than in the macro definition, missing arguments are assumed to be null values. The conditional directives .IF B and .IF NB (see Table 6-5) can be used within the macro to detect missing arguments. The number of arguments can also be specified using the .NARG directive (Section 7.4.1). Note that a macro can be defined with no arguments.

7.3.5 Creating Local Symbols Automatically

A label is often required in an expanded macro. In the conventional macro facilities thus far described, such a label must be explicitly

specified as an argument with each macro call. Be careful in issuing subsequent calls to the same macro, to avoid specifying a duplicate label as a real argument. This concern can be eliminated through a feature of MACRO which creates a unique symbol where a label is required in an expanded macro.

As noted in Section 3.5, MACRO can automatically create local symbols of the form n$, where n is a decimal integer within the range 64 through 127, inclusive. Such local symbols are created by MACRO in numerical order, as shown below:

```
64$
65$
 .
 .
 .
126$
127$
```

This automatic facility is invoked on each call of a macro whose definition contains a dummy argument preceded by the question mark (?) character, as shown in the macro definition below:

```
        .MACRO  ALPHA, A,?B     ;CONTAINS DUMMY ARGUMENT B PRECEDED BY
                                ;QUESTION MARK.
        TST     A
        BEQ     B
        ADD     #5,A
B:
        .ENDM
```

A local symbol is generated automatically by MACRO only when a real argument of the macro call is either null or missing, as shown in Example 1 below, which reflects the expansion of the ALPHA macro defined above.

If the real argument is specified in the macro call, however, MACRO inhibits the generation of a local symbol and normal argument replacement occurs, as shown in Example 2 below.

EXAMPLE 1: Generate a Local Symbol for the Missing Argument:

```
        ALPHA   R1              ;SECOND ARGUMENT IS MISSING.
        TST     R1
        BEQ     64$             ;LOCAL SYMBOL IS GENERATED.
        ADD     #5,R1
64$:
```

EXAMPLE 2: Do Not Generate a Local Symbol:

```
        ALPHA   R2,XYZ          ;SECOND ARGUMENT XYZ IS SPECIFIED.
        TST     R2
        BEQ     XYZ             ;NORMAL ARGUMENT REPLACEMENT OCCURS.
        ADD     #5,R2
XYZ:
```

Automatically-generated local symbols are restricted to the first 16(10) arguments of a macro definition.

Note that automatically-created local symbols resulting local symbols from the expansion of a macro, as described above, do not in any way influence local symbol block boundaries. In other words, such automatically-created local symbols do not establish a local symbol block in their own right.

However, when a macro has several arguments earmarked for automatic local symbol generation, substituting a specific label for one such argument introduces a risk that assembly errors will result. This is because MACRO constructs its argument substitution list at the point of macro invocation. Therefore, the appearance of any label, the .ENABL LSB directive, or the .PSECT directive, in the macro expansion will create a new local symbol block. This could leave local symbol references in the previous block and the symbol definitions in the new one, resulting in error codes in the assembly listing (see Appendix D). Furthermore, a subsequent macro expansion that generates local symbols in the new block may duplicate one of the symbols in question, resulting in an additional error code (P) in the assembly listing.

7.3.6 **Keyword Arguments**

Macros may be defined with and/or invoked with keyword arguments. A keyword argument has the following form:

```
name=string
```

where

name represents the dummy argument,

string represents the real symbolic argument.

The keyword argument may not contain embedded argument separators unless properly delimited as described in section 7.3.

When a keyword argument appears in the dummy argument list of a macro definition, the specified string becomes the default real argument at macro call.

When a keyword argument appears in the real argument list of a macro call, the specified string becomes the real argument for the dummy argument that exactly matches the specified name, whether or not the dummy argument was defined with a keyword. If a match fails, the entire argument specification is treated as the next positional real argument. A keyword argument may be specified anywhere in the dummy argument list of a macro definition and is part of the positional ordering of argument. On the other hand, a keyword argument may be specified anywhere in the real argument list of a macro call but does not affect the positional correspondence of the remaining arguments.

```
 1                              .LIST   ME
 2                      ;
 3                      ; DEFINE A MACRO HAVING KEYWORDS IN DUMMY ARGUMENT LIST
 4                      ;
 5
 6                              .MACRO  TEST CONTRL=1,BLOCK,ADDRES=TEMP
 7                              .WORD   CONTRL
 8                              .WORD   BLOCK
 9                              .WORD   ADDRES
10                              .ENDM
11
12
13                      ;
14                      ; NOW INVOKE SEVERAL TIMES
15                      ;
16
17 000000                      TEST    A,B,C
   000000  000000G              .WORD   A
```

```
   000002  000000G          .WORD   B
   000004  000000G          .WORD   C
18
19 000006                   TEST    ADDRES=20,BLOCK=30,CONTRL=40
   000006  000040           .WORD   40
   000010  000030           .WORD   30
   000012  000020           .WORD   20
20
21 000014                   TEST    BLOCK=5
   000014  000001           .WORD   1
   000016  000005           .WORD   5
   000020  000000G          .WORD   TEMP
22
23 000022                   TEST    CONTRL=5,ADDRES=VARIAB
   000022  000005           .WORD   5
   000024  000000           .WORD
   000026  000000G          .WORD   VARIAB
24
25 000030                   TEST
   000030  000001           .WORD   1
   000032  000000           .WORD
   000034  000000G          .WORD   TEMP
26
27 000036                   TEST    ADDRES=JACK!JILL
   000036  000001           .WORD   1
   000040  000000           .WORD
   000042  000000C          .WORD   JACK!JILL
28
29
30         000001           .END
```

7.3.7 Concatenation of Macro Arguments

The apostrophe or single quote character (') operates as a legal delimiting character in macro definitions. A single quote that precedes and/or follows a dummy argument in a macro definition is removed, and the substitution of the real argument occurs at that point. For example, in the following statements:

```
        .MACRO  DEF A,B,C
A'B:    .ASCIZ  /C/
        .BYTE   ''A,''B
        .ENDM
```

when the macro DEF is called through the statement:

```
        DEF     X,Y,<MACRO>
```

it is expanded, as follows:

```
XY:     .ASCIZ  /MACRO/
        .BYTE   'X,'Y
```

In expanding the first line, the scan for the first argument terminates upon finding the first ' character. Since A is a dummy argument, the ' is removed. The scan then resumes with B; B is also noted as another dummy argument. The two real arguments X and Y are then concatenated to form the label XY:. The third dummy argument is noted in the operand field of the .ASCIZ directive, causing the real argument MACRO to be substituted in this field.

When evaluating the arguments to the .BYTE directive during expansion of the second line, the scan begins with the first ' character. Since it is neither preceded nor followed by a dummy argument, this ' character remains in the macro expansion. The scan then encounters the second ' character, which is followed by a dummy argument and is therefore discarded. The scan of argument A is terminated upon encountering the comma (,). The third ' character is neither preceded nor followed by a dummy argument and again remains in the macro expansion. The fourth (and last) ' character is followed by another dummy argument and is likewise discarded. (Note that four ' characters were necessary in the macro definition to generate two ' characters in the macro expansion.)

7.4 MACRO ATTRIBUTE DIRECTIVES: .NARG, .NCHR, AND .NTYPE

Three directives are available in MACRO which allow the user to determine certain attributes of macro arguments. The use of these directives permits selective modifications of a macro expansion, depending on the nature of the arguments being passed. These directives are described separately below.

7.4.1 .NARG Directive

The .NARG directive is used to determine the number of arguments in the macro call currently being expanded. Hence, the .NARG directive can appear only within a macro definition; if it does not, an error code (O) is generated in the assembly listing. This directive takes the form:

```
label:  .NARG   symbol
```

where:

- label — represents an optional statement label.
- symbol — represents any legal symbol. This symbol is equated to the number of arguments in the macro call currently being expanded. If a symbol is not specified, the .NARG directive is flagged with an error code (A) in the assembly listing.

An example of the .NARG directive follows:

```
 1                              .TITLE NARG
 2
 3                              .MACRO  NOPP,NUM
 4                              .NARG   SYM
 5                              .IF     EQ,SYM
 6                              .MEXIT
 7                              .IFF
 8                              .REPT   NUM
 9                              NOP
10                              .ENDM
11                              .ENDC
12                              .ENDM
13
14
15 000000                       NOPP
            000000              .NARG   SYM
                                .IF     EQ,SYM
                                .MEXIT
                                .IFF
                                .REPT
                                NOP
                                .ENDM
                                .ENDC
16
17
18 000000                       NOPP    6
            000001              .NARG   SYM
                                .IF     EQ,SYM
                                .MEXIT
                                .IFF
            000006              .REPT   6
                                NOP
                                .ENDM
   000000   000240              NOP
   000002   000240              NOP
   000004   000240              NOP
   000006   000240              NOP
   000010   000240              NOP
   000012   000240              NOP
                                .ENDC
19
20
21          000001              .END
```

7.4.2 .NCHR Directive

The .NCHR directive, which can appear anywhere in a MACRO program, is used to determine the number of characters in a specified character string. This directive, which is useful in calculating the length of macro arguments, takes the following form:

label: .NCHR symbol,<string>

where: label represents an optional statement label.

symbol represents any legal symbol. This symbol is equated to the number of characters in the specified character string. If a symbol is not specified, the .NCHR directive is flagged with an error code (A) in the assembly listing (see Appendix D).

, represents any legal separator (comma, space, and/or tab).

<string> represents a string of printable characters. The character string need be enclosed within angle brackets (<>) or up-arrows (^) only if the specified character string contains a legal separator (comma, space, and/or tab). If the delimiting characters do not match or if the ending delimiter cannot be detected because of a syntactical error in the character string (thus prematurely terminating its evaluation), the .NCHR directive is flagged with an error code (A) in the assembly listing.

An example of the .NCHR directive follows:

```
 1                                      .TITLE NCHR
 2
 3                                      .MACRO  CHAR,MESS
 4                                      .NCHR   SYM,MESS
 5                                      .WORD   SYM
 6                                      .ASCII  /MESS/
 7                                      .EVEN
 8                                      .ENDM
 9
10
11 000000                      MSG1:   CHAR    <HELLO>
           000005                      .NCHR   SYM,HELLO
   000000  000005                      .WORD   SYM
   000002     110                      .ASCII  /HELLO/
   000003     105
   000004     114
   000005     114
   000006     117
                                      .EVEN
12
13
14         000001                      .END
```

7.4.3 .NTYPE Directive

The .NTYPE directive is used to determine the addressing mode of a specified macro argument. Hence, the .NTYPE directive can appear only within a macro definition; if it appears elsewhere, it is flagged with an error code (O) in the assembly listing. This directive takes the form:

label: .NTYPE symbol,aexp

where: label represents an optional statement label.

symbol represents any legal symbol. This symbol is equated to the 6-bit addressing mode of the following argument. If a symbol is not specified, the .NTYPE directive is flagged with an error code (A) in the assembly listing.

, represents any legal separator (comma, space, and/or tab).

aexp represents any legal address expression, as used with an opcode. If no argument is specified, the result will be zero.

An example of the use of an .NTYPE directive in a macro definition is shown below:

```
 1                              .TITLE NTYPE
 2
 3                              .MACRO  SAVE,ARG
 4                              .NTYPE  SYM,ARG
 5                              .IF     EQ,SYM&70
 6                              MOV     ARG,-(SP)       ;REGISTER MODE
 7                              .IFF
 8                              MOV     #ARG,-(SP)      ;NON-REGISTER MODE
 9                              .ENDC
10                              .ENDM
11
12
13 000000  000000  TEMP:        .WORD   0
14
15
16 000002                      SAVE    R1
           000001              .NTYPE  SYM,R1
                               .IF     EQ,SYM&70
   000002  010146              MOV     R1,-(SP)        ;REGISTER MODE
                               .IFF
                               MOV     #R1,-(SP)       ;NON-REGISTER MODE
                               .ENDC
17
18
19 000004                      SAVE    TEMP
           000067              .NTYPE  SYM,TEMP
                               .IF     EQ,SYM&70
                               MOV     TEMP,-(SP)      ;REGISTER MODE
                               .IFF
   000004  012746              MOV     #TEMP,-(SP)     ;NON-REGISTER MODE
           000000'
                               .ENDC
20
21
22         000001              .END
```

For additional information concerning addressing modes, refer to Chapter 5 and Appendix B, Section B.2.

7.5 .ERROR AND .PRINT DIRECTIVES

The .ERROR directive is used to output messages to the listing file during assembly pass 2. A common use of this directive is to provide a diagnostic announcement of a rejected or erroneous macro call or to alert the user to the existence of an illegal set of conditions specified in a conditional assembly. If the listing file is not specified, the .ERROR messages are output to the command output device. The .ERROR directive takes the form:

```
label:  .ERROR  expr    ;text
```

where: label represents an optional statement label.

expr	represents an optional expression whose value is output when the .ERROR directive is encountered during assembly.
;	denotes the beginning of the text string.
text	represents the specified message associated with the .ERROR directive.

Upon encountering an .ERROR directive anywhere in a source program, MACRO outputs a single line containing:

1. An error code (P)
2. The sequence number of the .ERROR directive statement
3. The value of the current location counter
4. The value of the expression, if one is specified
5. The source line containing the .ERROR directive.

For example, the following directive:

```
        .ERROR  A     ;INVALID MACRO ARGUMENT
```

causes a line in the following form to be output to the listing file:

```
    Seq. Loc.    Exp.
    No.  No.     Value                                  Text

P   512 005642  000076          .ERROR  A       ;INVALID MACRO ARGUMENT
```

The .PRINT directive is identical in function to the .ERROR directive, except that it is not flagged with the P error code.

7.6 INDEFINITE REPEAT BLOCK DIRECTIVES: .IRP AND .IRPC

An indefinite repeat block is a structure that is similar to a macro definition; essentially a macro definition that has only one dummy argument. At each expansion of the indefinite repeat range, this dummy argument is replaced with successive elements of a specified real argument list. An indefinite repeat block directive and its associated repeat range are coded in-line within the source program. This type of macro definition and expansion does not require calling the macro by name, as required in the expansion of conventional macros previously described in this section.

An indefinite repeat block can appear either within or outside another macro definition, indefinite repeat block, or repeat block (see Section 7.7). The rules for specifying indefinite repeat block arguments are the same as for specifying macro arguments (see Section 7.3).

7.6.1 .IRP Directive

The .IRP directive is used to replace a dummy argument with successive real arguments specified in an argument string. This replacement process occurs during the expansion of an indefinite repeat block range. This directive takes the following form:

```
label:  .IRP    sym,<argument list>
          .
          .
          .
        (range of indefinite repeat block)
          .
          .
          .
        .ENDM
```

where:

- label — represents an optional statement label.
- sym — represents a dummy argument that is successively replaced with the specified real arguments enclosed within the angle brackets. If no dummy argument is specified, the .IRP directive is flagged with an error code (A) in the assembly listing.
- , — represents any legal separator (comma, space, and/or tab).
- <argument list> — represents a list of real arguments enclosed within angle brackets that is to be used in the expansion of the indefinite repeat range. A real argument may consist of one or more characters; multiple arguments must be separated by any legal separator (comma, space, and/or tab). If no real arguments are specified, no action is taken.
- range — represents the block of code to be repeated once for each occurrence of a real argument in the list. The range may contain other macro definitions and repeat ranges. The .MEXIT directive (see Section 7.1.3) is legal within the range of an indefinite repeat block.
- .ENDM — indicates the end of the indefinite repeat block range.

An example of the use of the .IRP directive is shown in Figure 7-1.

7.6.2 .IRPC Directive

The .IRPC directive is available to permit single character substitution, rather than argument substitution. On each iteration of the indefinite repeat range, the dummy argument is replaced with each successive character in the specified string. The .IRPC directive is specified as follows:

```
label:  .IRPC   sym,<string>
          .
          .
          .
        (range of indefinite repeat block)
```

```
        .
        .
        .
      .ENDM
```

where:

label — represents an optional statement label.

sym — represents a dummy argument that is successively replaced with the specified real arguments enclosed within the angle brackets. If no dummy argument is specified, the .IRPC directive is flagged with an error code (A) in the assembly listing.

, — represents any legal separator (comma, space, and/or tab).

<string> — represents a list of characters enclosed within angle brackets to be used in the expansion of the indefinite repeat range. Although the angle brackets are required only when the string contains separating characters, their use is recommended for legibility.

range — represents the block of code to be repeated once for each occurrence of a character in the list. The range may contain macro definitions and repeat ranges. The .MEXIT directive (see Section 7.1.3) is legal within the range of an indefinite repeat block.

.ENDM — indicates the end of the indefinite repeat block range.

An example of the use of the .IRPC directive is shown in Figure 7-1.

```
 1                                              .TITLE  IRPTST
 2                                              .LIST   ME
 3
 4
 5
 6                                              .IRP    X,<AA,BB,CC,DD,EE,FF>
 7                                              MOV     X,(R0)+
 8                                              .ENDM
   000000  016720  000000G                      MOV     AA,(R0)+
   000004  016720  000000G                      MOV     BB,(R0)+
   000010  016720  000000G                      MOV     CC,(R0)+
   000014  016720  000000G                      MOV     DD,(R0)+
   000020  016720  000000G                      MOV     EE,(R0)+
   000024  016720  000000G                      MOV     FF,(R0)+
 9
10
11                                              .IRPC   X,<ABCDEF>
12                                              MOVB    #'X,-(R1)
13                                              .ENDM
   000030  112741  000000G                      MOVB    #A,-(R1)
   000034  112741  000000G                      MOVB    #B,-(R1)
   000040  112741  000000G                      MOVB    #C,-(R1)
   000044  112741  000000G                      MOVB    #D,-(R1)
   000050  112741  000000G                      MOVB    #E,-(R1)
   000054  112741  000000G                      MOVB    #F,-(R1)
14
15
16         000001'                             .END
```

Figure 7-1 Example of .IRP and .IRPC Directives

7.7 REPEAT BLOCK DIRECTIVE: .REPT, .ENDR

It is sometimes useful to duplicate a block of code a number of times in-line with other source code. This duplication of code is accomplished by creating a repeat block, using a directive in the form:

```
label:  .REPT   exp
          .
          .
          .
        (range of repeat block)
          .
          .
          .
        .ENDM
```

where:

label	represents an optional statement label.
exp	represents any legal expression whose value controls the number of times the block of code is to be assembled within the program. When the expression value is less than or equal to zero (0), the repeat block is not assembled. If this expression is not an absolute value, the .REPT statement is flagged with an error code (A) in the assembly listing.
range	represents the block of code to be repeated the number of times determined by the specified expression value. The repeat block may contain macro definitions, indefinite repeat blocks, or other repeat blocks. The .MEXIT directive is legal within the range of a repeat block.
.ENDM or .ENDR	indicates the end of the repeat block range. The terminating statement in a repeat block can be either an .ENDM directive or an .ENDR directive.

7.8 MACRO LIBRARY DIRECTIVE: .MCALL

The .MCALL directive allows you to indicate in advance those system and/or user-defined macro definitions that are required in the assembly of the source program. The .MCALL directive allows you to specify the names of all system or user macro definitions not defined within the source program but which are required to assemble the program. The .MCALL directive must appear before the first occurrence of a call to any externally-defined macro. The .MCALL directive is of the form:

```
        .MCALL  arg1,arg2,...argn
```

where:

arg1, arg2,... argn	represent the symbolic names of the macro definitions required in the assembly of the source program. The symbolic macro names may be separated by any legal separator (comma, space, and/or tab).

The .MCALL directive thus provides the means to access both user-defined and system macro libraries during assembly.

The /LIBRARY qualifier is specified in connection with an input file specification, to indicate to MACRO that the file is a macro library. When a macro call is encountered in the source program, MACRO first searches the user macro library for the named macro definitions, and, if necessary, continues the search with the system macro library.

Any number of such user-supplied macro files may be designated. In cases of multiple library files, the search for the named macros begins with the last such file specified. The search continues in reverse order until the required macro definitions are found, terminating again, if necessary, with a search of the system macro library.

If any named macro is not found upon completion of the search, i.e., if the macro is not defined, the .MCALL statement is flagged with an error code (U) in the assembly listing. Furthermore, a statement elsewhere in the source program which attempts to expand such an undefined macro is flagged with an error code (O) in the assembly listing.

The TRAX command strings to MACRO, through which a file specification is supplied, are described in detail in Sections 8.1.2 and 8.2.2, respectively.

PART IV

OPERATING PROCEDURES

CHAPTER 8

OPERATING PROCEDURES

MACRO assembles one or more ASCII source files containing MACRO statements into a single relocatable binary object file. The output of MACRO consists of a binary object file and a file containing the table of contents listing, the assembly listing, and the symbol table listing. An optional cross-reference listing of symbols and macros is available. A sample assembly listing is provided in Appendix I.

8.1 TRAX OPERATING PROCEDURES

The following sections describe the use of MACRO under TRAX.

8.1.1 Invoking MACRO Under TRAX

The MACRO command is used under TRAX to begin MACRO assembler operations. The command causes MACRO to assemble one or more ASCII source files containing MACRO statements into a relocatable binary object file. The assembler will also produce an assembly listing, followed by a symbol table listing. A cross-reference listing can also be produced, by means of the /CROSSREFERENCE qualifier (see 8.2.2, below).

The command can be issued whenever the TRAX Support Environment is at command level in interactive mode. The MACRO command can be input either directly from the terminal (interactive mode) or from a batch file (batch mode). When the specified assembly has completed, MACRO terminates operations and returns control to the Support Environment. (Refer to the TRAX Support Environment User's Guide for further information about interactive and batch mode operations.)

8.1.2 TRAX Command String Format

A MACRO command string can be specified using either an interactive mode or a batch mode. A MACRO command string under TRAX has the following format:

```
MACRO[/QUALIFIER[S]]    FILESPEC[/QUALIFIER[S]] [+FILESPEC+...]
```

where:

MACRO - is the invoking command.

[/QUALIFIER[S]] - are the optional qualifiers.

FILESPEC - is the MACRO source file to be assembled.

[/QUALIFIER[S]] - is the optional qualifiers for the source file.

[+FILESPEC+] - is the additional source files to be assembled.

If the FILESPEC parameter is not given the system will respond with the prompt:

>FILES?

The desired file names and optional qualifiers are entered and processing continues.

8.1.3 **TRAX Macro Qualifiers**

The TRAX Macro Qualifiers direct the MACRO assembler to process the source file with the following options:

QUALIFIERS	SPECIFIES ONE OR MORE OF THE FOLLOWING:
/OBJECT[:filespec] output	Produce an object file as specified by filespec (see Section 8.3). The default is a file with the same filename as the last named source file and an .OBJ extension. /OBJECT is always the default condition.
/NOOBJECT	Do not produce an object file.
/LIST[:filespec] output	Produce an assembly listing file according to filespec (see Section 8.3). If filespec is not specified, the listing is printed on the line printer. The default is /NOLIST.
/NOLIST	Do not produce a listing file. The default in interactive mode is /NOLIST and in batch mode is /LIST.

NOTE

When no listing file is specified, any errors encountered in the source program are displayed at the terminal from which MACRO was initiated.

/CROSSREFERENCE[:arg1...arg4]

Produce a cross-reference listing. Arg1 through arg4 are as described in Section 8.1.5.

/SWITCHES	FUNCTION
/LI:arg /NL:arg	Listing control switches; these options accept ASCII switch values (arg) which are equivalent in function and name to and override the arguments of the .LIST

and .NLIST directives specified in the source program (see Section 6.1.1). This switch overrides the arguments and remains in effect for the entire assembly process.

/EN:arg
/DS:arg

Function control switches; these options accept ASCII switch values (arg) which are equivalent in function and name to and override the arguments of the .ENABL and .DSABL directives specified in the source program (see Section 6.2). This switch overrides the arguments and remains in effect for the entire assembly process.

TRAX accepts the MACRO or $MACRO command as input and initializes the MACRO assembler, which in turn processes the specified files according to the options indicated in the command string. When the operation is complete, MACRO returns control to PDS to obtain the next command line either from the terminal or from the batch stream.

8.1.4 TRAX File Specification Qualifiers

The following optional qualifiers may be appended to a file specification:

/PASS:N

If N=1, assemble the associated file on the first pass.

If N=2, assemble the associated file on the second pass.

/LIBRARY

specifies that an input file is a macro library file. As noted in Section 7.8, any macro that is defined externally must be identified in the .MCALL directive before it can be retrieved from a macro library file and assembled with the user program. In locating macro definitions, MACRO initiates a fixed search algorithm, beginning with the last user macro file specified, continuing in reverse order with each such file specified, and terminating, if necessary, with a search of the system macro library file. If a required macro definition is not found upon completion of the search, an error code (U) results in the assembly listing (see Appendix D). This means that a user macro library file must be specified in the command line prior to the source file(s) that uses any macros defined in the library file. If more than one library file is specified, the libraries will be searched in right-to-left order.

8.1.5 Cross-Reference Processor (CREF)

The CREF processor is used to produce a listing that includes cross-references to symbols that appear in the source program. The cross-reference listing is appended to the assembly listing. Such cross-references are helpful in debugging and in reading long programs.

A cross-reference listing can include up to four sections:

1. User-defined symbols
2. Macro symbols
3. Register symbols
4. Permanent symbols

To generate a cross-reference listing, specify the /CR switch in the MACRO command string. Optional arguments can also be specified. The form of the switch is:

```
       SYM
/CR  : MAC
       REG
       PST
```

where:

SYM	specifies user-defined symbols (default)
MAC	specifies macro symbols (default)
REG	specifies register symbols
PST	specifies permanent symbols.

If you wish to generate listings for user-defined and macro symbols only, simply use /CR. No argument is necessary.

However, if an argument is specified, only that type of cross-reference listing is generated. For example:

```
/CR:SYM
```

produces a cross-reference listing of user-defined symbols only. No listing of macro symbols is generated. Thus, to produce all four types of cross-reference listings, you must specify all four arguments (the order in which they are specified is not significant). Use a colon to separate arguments. For example:

```
/CR:REG:SYM:MAC:PST
```

The CREF processor is more fully described in the Utilities Reference Manual supplied with your system.

Figure 8-1 illustrates a complete cross-reference listing.

```
PERMANENT SYMBOL TABLE CROSS REFERENCE                    CREF   V01

SYMBOL          REFERENCES

ADD             2-227   2-256   2-313
BCC             2-241
BCS             2-271
BEQ             2-203   2-236   2-275   2-295   2-300   2-306
BISB            2-237
BIT             2-235   2-305
BITB            2-250   2-274
BLO             2-232
BNE             2-225   2-251   2-273   2-302
BR              2-282
CLR             2-207   2-264   2-267   2-280   2-298
CLRB            2-304
CMP             2-202   2-224   2-231   2-272   2-301   2-314
INC             2-252   2-253
JMP             2-285   2-331
JSR             2-205   2-208   2-210   2-219   2-220   2-222
                2-281   2-311
MOV             2-204   2-206   2-209   2-221   2-223   2-226
                2-245   2-247   2-248   2-249   2-255   2-260
                2-268   2-276   2-277   2-278   2-279   2-284
                2-310   2-312   2-316   2-329   2-330
MOVB            2-230   2-239   2-307
RTS             2-212   2-320
SEC             2-292
TST             2-246   2-318
TSTB            2-299
.BLKB           2-151
.BLKW           2-77    2-78    2-80    2-81    2-82    2-83
                2-122   2-123   2-124   2-125   2-126   2-127
.BYTE           2-152   2-164   2-164   2-165   2-165   2-166
                2-169   2-169   2-170   2-170
.END            2-333
.ENDC           1-179   2-165   2-166   2-167   2-167   2-168
.IDENT          1-3     2-2
.IF             1-174   2-164   2-164   2-165   2-165   2-166
                2-169   2-169   2-170   2-170   2-205   2-205
                2-208   2-208   2-208   2-210   2-210   2-210
                2-220   2-220   2-222   2-222   2-222   2-222
                2-240   2-243   2-243   2-243   2-243   2-243
                2-258   2-258   2-258   2-258   2-258   2-270
                2-281   2-281   2-281   2-281   2-311   2-311
.IFF            2-164   2-164   2-165   2-166   2-168   2-170
.LIST           1-199
.MACRO          1-46    1-75    1-83    1-98    1-123   1-145
                2-40
.NARG           2-205   2-208   2-210   2-220   2-222   2-240
                2-311
.NLIST          1-1
.PSECT          2-75    2-88    2-118   2-129
.RAD50          2-79    2-149   2-164   2-165   2-166   2-167
.TITLE          1-2     2-1
.WORD           2-135   2-138   2-139   2-140   2-141   2-142
```

Figure 8-1 Sample CREF Listing

8.1.6 TRAX MACRO In Batch Mode

MACRO command strings can be processed via the TRAX batch mode facility. The batch mode facility imposes no restrictions on MACRO. Batch mode processing has the following format:

```
SUBMIT  FILESPEC
```

where:

SUBMIT - is the command to invoke the batch facility.

FILESPEC - is the file containing the MACRO command strings.

The file must have the following structure:

```
$JOB
MACRO
$DATA
SOURCE DATA
$EOD
$EOJ
```

where:

$JOB - defines

MACRO - invokes the MACRO assembler.

$DATA - specifies the data to be processed.

SOURCE DATA - is the MACRO command strings.

$EOD - specifies the end of data.

$EOJ - specifies the end of the job.

For a more complete description of the TRAX batch processing facility please see the TRAX System command Language Reference Manual.

8.1.7 TRAX Indirect Command Files

The indirect command file facility can be used with MACRO command strings. This is accomplished by creating an ASCII file that contains the desired command strings (or portions thereof) in the forms shown in Section 8.1.2. When an indirect command file reference is used in a MACRO command string, the contents of the specified file are taken as all or part of the command string. An indirect command file reference is specified in the form:

```
@filespec
```

where:

@ specifies that the name that follows is an indirect file.

filespec is the file specification of a file (see Section 8.2) that contains a command string. The default extension for the file name is .CMD.

An indirect command file reference must always be the rightmost entry in the command.

8.2 TRAX FILE SPECIFICATION FORMAT

The general form for a file specification in TRAX systems is shown below. Detailed information is provided in the applicable system user's guide or operating procedures manual (see Section 0.3 in the Preface).

```
dev:[g,m]name.ext;ver
```

where:

dev: is the name of the physical device where the desired file resides. A device name consists of two characters followed by a 1- or 2-digit device unit number (octal) and a colon (e.g., DP1:, DK0:, DT3:). The default device under TRAX is established initially by the system manager for each user and can be changed through the SET command, or is given a default value as specified in Table 8-1 if none is specified.

[g,m] is the User File Directory (UFD) code. This code consists of a group number (octal), a comma (,), and an owner (member) number (octal) all enclosed in brackets ([]). An example of a UFD code is: [200,30].

The default UFD is equivalent to the User Identification Code (UIC) given at log-in time. Under IAS, this can be changed through the SET DEFAULT command.

name is the filename and consists of one through nine alphanumeric characters. There is no default for a filename.

.ext is a 1- to 3-alphanumeric character filename extension or type that is preceded by a period (.). An extension is normally used to identify the nature of the file. Default values depend on the context of the file specification and are as follows:

```
.CMD = Indirect command (input) file
.LST = A listing (print format) file
.MAC = MACRO source module (input file)
.OBJ = MACRO object module (output file)
.CRF = Intermediate CREF input file created
       by MACRO.
```

;ver is an octal number between 1 and 77777 that is used to differentiate between versions of the same file. This number must be prefixed by a semicolon (;).

For input files, the default value is the highest version number of the file that exists.

For output files, the default value is the highest version number of the file that exists increased by 1. If no version number exists, the value 1 is used.

Table 8-1
File Specification Default Values

File	Device	Default Value		
		Directory	Filename	Type
Object File	System device.	Current.	None	.OBJ
Listing File	Device used for object file.	Directory used in Object file.	None	.LST
Source 1 File	System device.	Current.	None	.MAC
Source 2 to Sourc% n File	Device used for source 1 or last source file specified.	Directory used for source 1 or last source file specified.	None	.MAC
User Macro Library	System device, if macro file is specified first; if not, device used by last source file is used.	Current, if macro file is specified first; if not, directory of last source file is used.	None	.MLB
System Macro Library	System device.	[1,1]	TRAXMAC	.SML
Indirect Command File	System device.	Current.	None	.CMD

8.3 MACRO ERROR MESSAGES

MACRO outputs an appropriate error message to the command output device when one of the error conditions described below is detected. These error messages reflect operational problems and should not be confused with the diagnostic error messages (see Appendix D) produced by MACRO during assembly.

All the error messages listed below, with the exception of the "MAC -- COMMAND I/O ERROR" message, result in the termination of the current assembly; MACRO then attempts to restart by reading another command line. In the case of a command I/O error, however, MACRO exits, since it is unable to obtain additional command line input.

Error Message	Meaning
MAC -- COMMAND FILE OPEN FAILURE	Either the file from which MACRO is reading a command could not be opened initially or between assemblies; or, the indirect command file specified as "@filename" in the MACRO command line could not be opened. See "OPEN FAILURE ON INPUT FILE" for meaning.
MAC -- COMMAND I/O ERROR	An error was returned by the file system during MACRO's attempt to read a command line. This is an unconditionally fatal error, causing MACRO to exit. No MACRO restart is attempted when this message appears.
MAC -- COMMAND SYNTAX ERROR	An error was detected in the syntax of the MACRO command line.
MAC -- ILLEGAL FILENAME	Neither the device name nor the filename was present in the input file specification (i.e., the input file specification is null), or a "wild card" convention (asterisk) was employed in an input or output file specification. "Wild card" options (*) are not permitted in MACRO file specifications.
MAC -- ILLEGAL SWITCH	An illegal switch was specified for a file, an illegal value was specified with a switch, or an invalid use of a switch was detected by MACRO.
MAC -- INDIRECT COMMAND SYNTAX ERROR	The name of the indirect command file (@filename) specified in the MACRO command line is syntactically incorrect.
MAC -- INDIRECT FILE DEPTH EXCEEDED	An attempt to exceed the maximum allowable number of nested indirect command files has occurred. (Only three levels of indirect command files are permitted in MACRO.)

Error Message	Meaning
MAC -- INSUFFICIENT DYNAMIC MEMORY	There is not enough physical memory available for MACRO to page its symbol table. Reinstall MACRO in a larger partition; or see Section F.3.
MAC -- INVALID FORMAT IN MACRO LIBRARY	The library file has been corrupted or it was not produced by the Librarian Utility Program (LBR).
MAC -- I/O ERROR ON INPUT FILE	In reading a record from a source input file or macro library file, an error was detected by the file system, e.g., a line containing more than 132(10) characters is encountered. This message may also indicate that a device problem exists or that either a source file or a macro library file has been corrupted with incorrect data.
MAC -- I/O ERROR ON MACRO LIBRARY FILE	Same meaning as I/O ERROR ON INPUT FILE, except that the file is a macro library file and not a source input file.
MAC -- I/O ERROR ON OUTPUT FILE	In writing a record to the object output file or the listing output file, an error was detected by the file system. This message may also indicate that a device problem exists or that the storage space on a device has been exhausted (i.e., the device is full).
MAC -- I/O ERROR ON WORK FILE	A read or write error occurred on the work file used to store the symbol table. This error is most likely caused by a problem on this device, or by attempting to write to a device that is full.

Error Message	Meaning
MAC -- OPEN FAILURE ON INPUT FILE	1. Specified device does not exist. 2. The volume is not mounted. 3. A problem exists with the device. 4. Specified directory file does not exist. 5. Specified file does not exist. 6. User does not have access to the file directory or the file itself.
MAC -- OPEN FAILURE ON OUTPUT FILE	1. Specified device does not exist. 2. The volume is not mounted. 3. A problem exists with the device. 4. Specified directory file does not exist. 5. User does not have access to the file directory 6. The volume is full or the device is write protected.
MAC -- 64K STORAGE LIMIT EXCEEDED	64K words of work file memory are available to MACRO. This message indicates that the assembler has generated so many symbols (on the order of 13,000 to 14,000), it has run out of space. This means either the source program is too large to start with, or it contains a condition that leads to excessive size, such as a macro expansion that recursively calls itself without a terminating condition.

The MACRO Assembler uses its stack for the following purposes:

1. Symbol Table. Four words for every symbol, including macro names and local symbols. (Local symbol space, however, is reused.)

2. Control Section Information. Five words for each PSECT, .ASECT, or .CSECT.

3. Storage of macro-definition text. Each and every character, including comments, between a .MACRO and the corresponding .ENDM is stored on the stack.

4. Work space (for code conversion, etc.).

If, during its execution, the MACRO Assembler reports a stack overflow, then either the demands of one or more of the above-described categories must be reduced, or the program must be broken up into smaller modules and linked together at Task-Build time.

The default stack size for MACRQ is 4K. This allows approximately 1000 symbols with the trade-offs mentioned above.

APPENDIX A

MACRO CHARACTER SETS

A.1 ASCII CHARACTER SET

EVEN PARITY BIT	7-BIT OCTAL CODE	CHARACTER	REMARKS
0	000	NUL	Null, tape feed, CONTROL/SHIFT/P.
1	001	SOH	Start of heading; also SOM, start of message, CONTROL/A.
1	002	STX	Start of text; also EOA, end of address, CONTROL/B.
0	003	ETX	End of text; also EOM, end of message, CONTROL/C.
1	004	EOT	End of transmission (END); shuts off TWX machines, CONTROL/D.
0	005	ENQ	Enquiry (ENQRY); also WRU, CONTROL/E.
0	006	ACK	Acknowledge; also RU, CONTROL/F.
1	007	BEL	Rings the bell. CONTROL/G.
1	010	BS	Backspace; also FEO, format effector. backspaces some machines, CONTROL/H.
0	011	HT	Horizontal tab. CONTROL/I.
0	012	LF	Line feed or Line space (new line); advances paper to next line, duplicated by CONTROL/J.
1	013	VT	Vertical tab (VTAB). CONTROL/K.
0	014	FF	Form Feed to top of next page (PAGE). CONTROL/L.
1	015	CR	Carriage return to beginning of line; duplicated by CONTROL/M.
1	016	SO	Shift out; changes ribbon color to red. CONTROL/N.
0	017	SI	Shift in; changes ribbon color to black. CONTROL/O.
1	020	DLE	Data link escape. CONTROL/P (DC0).
0	021	DC1	Device control 1; turns transmitter (READER) on, CONTROL/Q (X ON). 0 022 DC2 Device control 2; turns punch or auxiliary on. CONTROL/R (TAPE, AUX ON).
1	023	DC3	Device control 3; turns transmitter (READER) off, CONTROL/S (X OFF).
0	024	DC4	Device control 4; turns punch or auxiliary off. CONTROL/T (AUX OFF).

EVEN PARITY BIT	7-BIT OCTAL CODE	CHARACTER	REMARKS
1	025	NAK	Negative acknowledge; also ERR, ERROR. CONTROL/U.
1	026	SYN	Synchronous file (SYNC). CONTROL/V.
0	027	ETB	End of transmission block; also LEM, logical end of medium. CONTROL/W.
0	030	CAN	Cancel (CANCL). CONTROL/X.
1	031	EM	End of medium. CONTROL/Y.
1	032	SUB	Substitute. CONTROL/Z.
0	033	ESC	Escape. CONTROL/SHIFT/K.
1	034	FS	File separator. CONTROL/SHIFT/L.
0	035	GS	Group separator. CONTROL/SHIFT/M.
0	036	RS	Record separator. CONTROL/SHIFT/N.
1	037	US	Unit separator. CONTROL/SHIFT/O.
1	040	SP	Space.
0	041	!	
0	042	"	
1	043	#	
0	044	$	
1	045	%	
1	046	&	
0	047	'	Accent acute or apostrophe.
0	050	(	
1	051	)	
1	052	*	
0	053	+	
1	054	,	
0	055	-	
0	056	.	
1	057	/	
0	060	0	
1	061	1	
1	062	2	
0	063	3	
1	064	4	
0	065	5	
0	066	6	
1	067	7	
1	070	8	
0	071	9	
0	072	:	
1	073	;	
0	074	<	
1	075	=	
1	076	>	
0	077	?	
1	100	@	
0	101	A	
0	102	B	
1	103	C	
0	104	D	
1	105	E	
1	106	F	
0	107	G	
0	110	H	
0	111	I	

EVEN PARITY BIT	7-BIT OCTAL CODE	CHARACTER	REMARKS
1	112	J	
0	113	K	
1	114	L	
0	115	M	
0	116	N	
1	117	O	
0	120	P	
1	121	Q	
1	122	R	
0	123	S	
1	124	T	
0	125	U	
0	126	V	
1	127	W	
1	130	X	
0	131	Y	
0	132	Z	
1	133	[	shift/k.
0	134	\	shift/l.
1	135	]	shift/m.
1	136	^	*
0	137	_	**
0	140	`	Accent grave.
1	141	a	
1	142	b	
0	143	c	
1	144	d	
0	145	e	
0	146	f	
1	147	g	
1	150	h	
0	151	i	
0	152	j	
1	153	k	
0	154	l	
1	155	m	
1	156	n	
0	157	o	
1	160	p	
0	161	q	
0	162	r	
1	163	s	
0	164	t	
1	165	u	
1	166	v	
0	167	w	
0	170	x	
1	171	y	
1	172	z	
0	173		
1	174		
0	175		This code generated by ALTMODE.
0	176		This code generated by prefix key (if present).
1	177		Delete, Rubout.

* ^ Appears as # or ↑ on some machines.

** _ Appears as ← on some machines.

A.2 RADIX-50 CHARACTER SET

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

The maximum Radix-50 value is, thus,

```
47*50**2+47*50+47=174777
```

The following table provides a convenient means of translating between the ASCII character set and its Radix-50 equivalents. For example, given the ASCII string X2B, the Radix-50 equivalent is (arithmetic is performed in octal):

```
  X=113000
  2=002400
  B=000002
X2B=115402
```

Single Char. or First Char.		Second Character		Third Character	
Space	000000	Space	000000	Space	000000
A	003100	A	000050	A	000001
B	006200	B	000120	B	000002
C	011300	C	000170	C	000003
D	014400	D	000240	D	000004
E	017500	E	000310	E	000005
F	022600	F	000360	F	000006
G	025700	G	000430	G	000007
H	031000	H	000500	H	000010
I	034100	I	000550	I	000011
J	037200	J	000620	J	000012
K	042300	K	000670	K	000013
L	045400	L	000740	L	000014
M	050500	M	001010	M	000015
N	053600	N	001060	N	000016
O	056700	0	001130	0	000017
P	062000	P	001200	P	000020
Q	065100	Q	001250	Q	000021
R	070200	R	001320	R	000022
S	073300	S	001370	S	000023
T	076400	T	001440	T	000024
U	101500	U	001510	U	000025
V	104600	V	001560	V	000026
W	107700	W	001630	W	000027
X	113000	X	001700	X	000030
Y	116100	Y	001750	Y	000031
Z	121200	Z	002020	Z	000032
$	124300	$	002070	$	000033

SINGLE CHAR. OR FIRST CHAR.		SECOND CHARACTER		THIRD CHARACTER	
.	127400	.	002140	.	000034
Unused	132500	Unused	002210	Unused	000035
0	135600	0	002260	0	000036
1	140700	1	002330	1	000037
2	144000	2	002400	2	000040
3	147100	3	002450	3	000041
4	152200	4	002520	4	000042
5	155300	5	002570	5	000043
6	160400	6	002640	6	000044
7	163500	7	002710	7	000045
8	166600	8	002760	8	000046
9	171700	9	003030	9	000047

APPENDIX B

MACRO ASSEMBLY LANGUAGE AND ASSEMBLER DIRECTIVES

B.1 SPECIAL CHARACTERS

Character	Function
:	Label terminator
=	Direct assignment operator
%	Register term indicator
tab	Item terminator or field terminator
space	Item terminator or field terminator
#	Immediate expression indicator
@	Deferred addressing indicator
(	Initial register indicator
)	Terminal register indicator
, (comma)	Operand field separator
;	Comment field indicator
+	Arithmetic addition operator or auto increment indicator
-	Arithmetic subtraction operator or auto decrement indicator
*	Arithmetic multiplication operator
/	Arithmetic division operator
&	Logical AND operator
!	Logical OR operator
"	Double ASCII character indicator
' (apostrophe)	Single ASCII character indicator or concatenation indicator
.	Assembly location counter
<	Initial argument indicator
>	Terminal argument indicator
^	Universal unary operator or argument indicator
\	Macro call numeric argument indicator
vertical tab	Source line terminator

B.2 SUMMARY OF ADDRESS MODE SYNTAX

Address mode syntax is expressed in the summary below using the following symbols: n is an integer between 0 and 7 representing a register number; R is a register expression; E is an expression; and ER is either a register expression or an expression in the range 0 to 7.

Format	Address Mode Name	Address Mode Number	Meaning
R	Register	0n	Register R contains the operand.
@R or (ER)	Register deferred	1n	Register R contains the address of the operand.
(ER)+	Autoincrement	2n	The contents of the register specified as (ER) are incremented after being used as the address of the operand.
@(ER)+	Autoincrement Deferred	3n	The register specified as (ER) contains the pointer to the address of the operand; the register (ER) is incremented after use.
-(ER)	Autodecrement	4n	The contents of the register specified as (ER) are decremented before being used as the address of the operand.
@-(ER)	Autodecrement Deferred	5n	The contents of the register specified as (ER) are decremented before being used as the pointer to the address of the operand.
E(ER)	Index	6n	The expression E, plus the contents of the register specified as (ER), form the address of the operand.
@E(ER)	Index Deferred	7n	The expression E, plus the contents of the register specified as (ER), yield a pointer to the address of the operand.
#E	Immediate	27	The expression E is the operand itself.
@#E	Absolute	37	The expression E is the address of the operand.
E	Relative	67	The address of the operand E, relative to the instruction, follows the instruction.
@E	Relative Deferred	77	The address of the operand is pointed to by E whose address, relative to the instruction, follows the instruction.

B.3 ASSEMBLER DIRECTIVES

The MACRO assembler directives are summarized in the following table. For a detailed description of each directive, the table contains

references to the appropriate sections in the body of the manual.

Form	Section Reference	Operation
'	6.3.3 7.3.6	A single quote (apostrophe) followed by one ASCII character generates a word which contains the 7-bit ASCII representation of the character in the low-order byte and zero in the high-order byte. This character is also used as a concatenation indicator in the expansion of macro arguments (see Section 7.3.6).
"	6.3.3	A double quote followed by two ASCII characters generates a word which contains the 7-bit ASCII representation of the two characters. The first character is stored in the low-order byte; the second character is stored in the high-order byte.
^Bn	6.4.1.2	Temporary radix control; causes the value n to be treated as a binary number.
^Cexpr	6.4.2.2	Temporary numeric control; causes the expression's value to be ones-complemented.
^Dn	6.4.1.2	Temporary radix control; causes the value n to be treated as a decimal number.
^Fn	6.4.2.2	Temporary numeric control; causes the value n to be treated as a sixteen-bit floating-point number.
^On	6.4.1.2	Temporary radix control; causes the value n to be treated as an octal number.
^Rccc	6.3.7	Convert ccc to Radix-50 form.
.ASCII /string/	6.3.4	Generates a block of data containing the ASCII equivalent of the character string (enclosed in delimiting characters), one character per byte.

Form	Section Reference	Operation
.ASCIZ /string/	6.3.5	Generates a block of data containing the ASCII equivalent of the character string (enclosed in delimiting characters), one character per byte, with a zero byte terminating the specified string.
.ASECT	6.8.2	Begin or resume the absolute program section.
.BLKB exp	6.5.3	Reserves a block of storage space whose length in bytes is determined by the specified expression.
.BLKW exp	6.5.3	Reserves a block of storage space whose length in words is determined by the specified expression.
.BYTE expl,exp2,..	6.3.1	Generates successive bytes of data; each byte contains the value of the corresponding specified expression.
.CSECT [name]	6.8.2	Begin or resume named or unnamed relocatable program section. This directive is provided for compatibility with other PDP-11 assemblers.
.DSABL arg	6.2	Disables the function specified by the argument.
.ENABL arg	6.2	Enables (invokes) the function specified by the argument.
.END [exp]	6.6.1	Indicates the logical end of the source program. The optional argument specifies the transfer address where program execution is to begin.
.ENDC	6.10.1	Indicates the end of a conditional assembly block.
.ENDM [name]	7.1.2	Indicates the end of the current repeat block, indefinite repeat block, or macro definition. The optional name, if used, must be identical to the name specified in the macro definition.
.ENDR	7.7	Indicates the end of the current repeat block. This directive is provided for compatibility with other PDP-11 assemblers.
.EOT	6.6.2	Ignored; indicates end-of-tape (which is detected automatically by the hardware). It is included for compatibility with earlier assemblers.

Form	Section Reference	Operation
.ERROR exp;text	7.5	User-invoked error directive; causes output to the listing file or the command output device containing the optional expression and the statement containing the directive.
.EVEN	6.5.1	Ensures that the current location counter contains an even address by adding 1 if it is odd.
.FLT2 arg1,arg2,...	6.4.2.1	Generates successive 2-word floating-point equivalents for the floating-point numbers specified as arguments.
.FLT4 arg1,arg2,...	6.4.2.1	Generates successive 4-word floating-point equivalents for the floating-point numbers specified as arguments.
.GLOBL sym1,sym2,...	6.9	Defines the symbol(s) specified as global symbol(s).
.IDENT /string/	6.1.5	Provides a means of labeling the object module with the program version number. The version number is the Radix-50 string appearing between the paired delimiting characters.
.IF cond,arg1	6.10.1	Begins a conditional assembly block of source code which is included in the assembly only if the stated condition is met with respect to the argument(s) specified.
.IFF	6.10.2	Appears only within a conditional assembly block, indicating the beginning of a section of code to be assembled if the condition upon entering the block tests false.
.IFT	6.10.2	Appears only within a conditional assembly block, indicating the beginning of a section of code to be assembled if the condition upon entering the block tests true.
.IFTF	6.10.2	Appears only within a conditional assembly block, indicating the beginning of a section of code to be assembled unconditionally.
.IIF cond,arg, statement	6.10.3	Acts as a 1-line conditional assembly block where the condition is tested for the argument specified. The statement is assembled only if the condition tests true.

Form	Section Reference	Operation
.IRP sym, <arg1,arg2,...>	7.6.1	Indicates the beginning of an indefinite repeat block in which the symbol specified is replaced with successive elements of the real argument list enclosed within angle brackets.
.IRPC sym,<string>	7.6.2	Indicates the beginning of an indefinite repeat block in which the specified symbol takes on the value of successive characters, optionally enclosed within angle brackets.
.LIMIT	6.7	Reserves two words into which the Task Builder inserts the low and high addresses of the task image.
.LIST [arg]	6.1.1	Without an argument, the .LIST directive increments the listing level count by 1. With an argument, this directive does not alter the listing level count, but formats the assembly listing according to the argument specified.
.MACRO name,arg1, arg2,...	7.1.1	Indicates the start of a macro definition having the specified name and the following dummy arguments.
.MCALL arg1,arg2,...	7.8	Specifies the symbolic names of the user or system macro definitions required in the assembly of the current user program, but which are not defined within the program.
.MEXIT	7.1.3	Causes an exit from the current macro expansion or indefinite repeat block.
.NARG symbol	7.4.1	Can appear only within a macro definition; equates the specified symbol to the number of arguments in the macro call currently being expanded.
.NCHR symbol,<string>	7.4.2	Can appear anywhere in a source program; equates the symbol specified to the number of characters in the specified string.
.NLIST [arg]	6.1.1	Without an argument, the .NLIST directive decrements the listing level count by 1. With an argument, this directive suppresses that portion of the listing specified by the argument.

Form	Section Reference	Operation
.NTYPE symbol,aexp	7.4.3	Can appear only within a macro definition; equates the symbol to the 6-bit addressing mode of the specified address expression.
.ODD	6.5.2	Ensures that the current location counter contains an odd address by adding 1 if it is even.
.PAGE	6.1.6	Causes the assembly listing to skip to the top of the next page, and to increment the page count.
.PRINT exp;text	7.5	User-invoked message directive; causes output to the listing file or the command output device containing the optional expression and the statement containing the directive.
.PSECT name,attl,... attn	6.8.1	Begin or resume a named or unnamed program section having the specified attributes.
.RADIX n	6.4.1.1	Alters the current program radix to n, where n is 2, 8, or 10.
.RAD50 /string/	6.3.6	Generates a block of data containing the Radix-50 equivalent of the character string enclosed within delimiting characters.
.REPT exp	7.7	Begins a repeat block; causes the section of code up to the next .ENDM or .ENDR directive to be repeated the number of times specified as exp.
.SBTTL string	6.1.4	Causes the specified string to be printed as part of the assembly listing page header. The string component of each .SBTTL directive is collected into a table of contents at the beginning of the assembly listing.
.TITLE string	6.1.3	Assigns the first six Radix-50 characters in the string as an object module name and causes the string to appear on each page of the assembly listing.
.WORD expl,exp2,..	6.3.2	Generates successive words of data; each word contains the value of the corresponding specified expression.

APPENDIX C

PERMANENT SYMBOL TABLE (PST)

The permanent symbol table (PST) contains those symbols which are automatically recognized by MACRO. These symbols consist of both op codes and assembler directives. The op codes (i.e., the instruction set) are listed first, followed by the directives which cause specific actions during assembly.

For a detailed description of the instruction set, see the appropriate PDP-11 Processor Handbook.

C.1 OP CODES

MNEMONIC	OCTAL VALUE	FUNCTIONAL NAME
ADC	005500	Add Carry
ADCB	105500	Add Carry (Byte)
ADD	060000	Add Source To Destination
ASH	072000	Shift Arithmetically
ASHC	073000	Arithmetic Shift Combined
ASL	006300	Arithmetic Shift Left
ASLB	106300	Arithmetic Shift Left (Byte)
ASR	006200	Arithmetic Shift Right
ASRB	106200	Arithmetic Shift Right (Byte)
BCC	103000	Branch If Carry Is Clear
BCS	103400	Branch If Carry Is Set
BEQ	001400	Branch If Equal
BGE	002000	Branch If Greater Than Or Equal
BGT	003000	Branch If Greater Than
BHI	101000	Branch If Higher
BHIS	103000	Branch If Higher Or Same
BIC	040000	Bit Clear
BICB	140000	Bit Clear (Byte)
BIS	050000	Bit Set
BISB	150000	Bit Set (Byte)
BIT	030000	Bit Test
BITB	130000	Bit Test (Byte)
BLE	003400	Branch If Less Than Or Equal
BLO	103400	Branch If Lower
BLOS	101400	Branch If Lower Or Same
BLT	002400	Branch If Less Than

PERMANENT SYMBOL TABLE (PST)

MNEMONIC	OCTAL VALUE	FUNCTIONAL NAME
BMI	100400	Branch If Minus
BNE	001000	Branch If Not Equal
BPL	100000	Branch If Plus
BPT	000003	Breakpoint Trap
BR	000400	Branch Unconditional
BVC	102000	Branch If Overflow Is Clear
BVS	102400	Branch If Overflow Is Set
CALL	004700	Jump To Subroutine (JSR PC,xxx)
CCC	000257	Clear All Condition Codes
CLC	000241	Clear C Condition Code Bit
CLN	000250	Clear N Condition Code Bit
CLR	005000	Clear Destination
CLRB	105000	Clear Destination (Byte)
CLV	000242	Clear V Condition Code Bit
CLZ	000244	Clear Z Condition Code Bit
CMP	020000	Compare Source To Destination
CMPB	120000	Compare Source To Destination (Byte)
COM	005100	Complement Destination
COMB	105100	Complement Destination (Byte)
DEC	005300	Decrement Destination
DECB	105300	Decrement Destination (Byte)
DIV	071000	Divide
EMT	104000	Emulator Trap
FADD	075000	Floating Add
FDIV	075030	Floating Divide
FMUL	075020	Floating Multiply
FSUB	075010	Floating Subtract
HALT	000000	Halt
INC	005200	Increment Destination
INCB	105200	Increment Destination (Byte)
IOT	000004	Input/Output Trap
JMP	000100	Jump
JSR	004000	Jump To Subroutine
MARK	006400	Mark
MFPI	006500	Move From Previous Instruction Space
MFPS	106700	Move from PS (LSI-11)
MOV	010000	Move Source To Destination
MOVB	110000	Move Source To Destination (Byte)
MTPI	006600	Move To Previous Instruction Space
MTPS	106400	Move to PS (LSI-11)
MUL	070000	Multiply
NEG	005400	Negate Destination
NEGB	105400	Negate Destination (Byte)
NOP	000240	No Operation
RESET	000005	Reset External Bus
RETURN	000207	Return From Subroutine (RTS PC)
ROL	006100	Rotate Left
ROLB	106100	Rotate Left (Byte)
ROR	006000	Rotate Right

MNEMONIC	OCTAL VALUE	FUNCTIONAL NAME
RORB	106000	Rotate Right (Byte)
RTI	000002	Return From Interrupt (Permits a trace trap)
RTS	000200	Return From Subroutine
RTT	000006	Return From Interrupt (inhibits trace trap)
SBC	005600	Subtract Carry
SBCB	105600	Subtract Carry (Byte)
SCC	000277	Set All Condition Code Bits
SEC	000261	Set C Condition Code Bit
SEN	000270	Set N Condition Code Bit
SEV	000262	Set V Condition Code Bit
SEZ	000264	Set Z Condition Code Bit
SOB	077000	Subtract One And Branch
SUB	160000	Subtract Source From Destination
SWAB	000300	Swap Bytes
SXT	006700	Sign Extend
TRAP	104400	Trap
TST	005700	Test Destination
TSTB	105700	Test Destination (Byte)
WAIT	000001	Wait For Interrupt
XOR	074000	Exclusive OR

OP CODES FLOATING POINT PROCESSOR ONLY

MNEMONIC	OCTAL VALUE	FUNCTIONAL NAME
ABSD	170600	Make Absolute Double
ABSF	170600	Make Absolute Floating
ADDD	172000	Add Double
ADDF	172000	Add Floating
CFCC	170000	Copy Floating Condition Codes
CLRD	170400	Clear Double
CLRF	170400	Clear Floating
CMPD	173400	Compare Double
CMPF	173400	Compare Floating
DIVD	174400	Divide Double
DIVF	174400	Divide Floating
LDCDF	177400	Load And Convert From Double To Floating
LDCFD	177400	Load And Convert From Floating To Double
LDCID	177000	Load And Convert Integer To Double
LDCIF	177000	Load And Convert Integer To Floating
LDCLD	177000	Load And Convert Long integer To Double
LDCLF	177000	Load And Convert Long Integer To Floating
LDD	172400	Load Double
LDEXP	176400	Load Exponent

MNEMONIC	OCTAL VALUE	FUNCTIONAL NAME
LDF	172400	Load Floating
LDFPS	170100	Load FPPs Program Status
MFPD	106500	Move From Previous Data Space
MODD	171400	Multiply And Integerize Double
MODF	171400	Multiply And Integerize Floating
MTPD	106600	Move To Previous Data Space
MULD	171000	Multiply Double
MULF	171000	Multiply Floating
NEGD	170700	Negate Double
NEGF	170700	Negate Floating
SETD	170011	Set Double Mode
SETF	170001	Set Floating Mode
SETI	170002	Set Integer Mode
SETL	170012	Set Long Integer Mode
SPL	000230	Set Priority Level
STCDF	176000	Store And Convert From Double To Floating
STCDI	175400	Store And Convert From Double To Integer
STCDL	175400	Store And Convert From Double To Long Integer
STCFD	176000	Store And Convert From Floating To Double
STCFI	175400	Store And Convert From Floating To Integer
STCFL	175400	Store And Convert From Floating To Long Integer
STD	174000	Store Double
STEXP	175000	Store Exponent
STF	174000	Store Floating
STFPS	170200	Store FPPs Program Status
STST	170300	Store FPPs Status
SUBD	173000	Subtract Double
SUBF	173000	Subtract Floating
TSTD	170500	Test Double
TSTF	170500	Test Floating

C.2 MACRO DIRECTIVES

DIRECTIVE	FUNCTIONAL SIGNIFICANCE
.ASCII	Translates character string to ASCII equivalents.
.ASCIZ	Translates character string to ASCII equivalents; inserts zero byte as last character.
.ASECT	Begins absolute program section (provided for compatibility with other PDP-11 assembliers).
.BLKB	Reserves byte block in accordance with value of specified argument.
.BLKW	Reserves word block in accordance with value of specified argument.
.BYTE	Generates successive byte data in accordance with specified arguments.
.CSECT	Begins relocatable program section (provided for compatibility with other PDP-11 assemblers).

DIRECTIVE	FUNCTIONAL SIGNIFICANCE
.DSABL	Disables specified function.
.ENABL	Enables specified function.
.END	Defines logical end of source program.
.ENDC	Defines end of conditional assembly block.
.ENDM	Defines end of macro definition, repeat block, or indefinite repeat block.
.ENDR	Defines end of current repeat block (provided for compatibility with other PDP-11 assemblers).
.EOT	Define End of Tape condition (ignored).
.ERROR	Outputs diagnostic message to listing file or command output device.
.EVEN	Word-aligns the current location counter.
.FLT2	Causes two words of storage to be generated for each floating-point argument.
.FLT4	Causes four words of storage to be generated for each floating-point argument.
.GLOBL	Declares global attribute for specified symbol(s).
.IDENT	Labels object module with specified program version number.
.IF	Begins conditional assembly block.
.IFF	Begins subconditional assembly block (if conditional assembly block test is false).
.IFT	Begins subconditional assembly block (if conditional assembly block test is true).
.IFTF	Begins subconditional assembly block (whether conditional assembly block test is true or false).
.IIF	Assembles immediate conditional assembly statement (if specified condition is satisfied).
.IRP	Begins indefinite repeat block; replaces specified symbol with specified successive real arguments.
.IRPC	Begins indefinite repeat block; replaces specified symbol with value of successive characters in specified string.
.LIMIT	Reserves two words of storage for high and low addresses of task image.
.LIST	Controls listing level count and format of assembly listing. .MACRO Denotes start of macro definition.
.MCALL	Identifies required macro definition(s) for assembly.
.MEXIT	Exit from current macro definition or indefinite repeat block.
.NARG	Equates specified symbol to the number of arguments in the macro expansion.
.NCHR	Equates specified symbol to the number of characters in the specified character string.
.NLIST	Controls listing level count and suppresses specified portions of the assembly listing.
.NTYPE	Equates specified symbols to the addressing mode of the specified argument.
.ODD	Byte-aligns the current location counter.
.PAGE	Advances form to top of next page.
.PRINT	Prints specified message on command output device.
.PSECT	Begins specified program section having specified attributes.
.RADIX	Changes current program radix to specified radix.
.RAD50	Generates data block having Radix-50 equivalents of specified character string.
.REPT	Begins repeat block and replicates it according to the value of the specified expression.

DIRECTIVE	FUNCTIONAL SIGNIFICANCE
.SBTTL	Prints specified subtitle text as the second line of the assembly listing page header.
.TITLE	Prints specified title text as object module name in the first line of the assembly listing page header.
.WORD	Generates successive word data in accordance with specified arguments.

The MACRO directives listed above are summarized in greater detail in Appendix B.

APPENDIX D

DIAGNOSTIC ERROR MESSAGE SUMMARY

D.1 MACRO ERROR CODES

A diagnostic error code is printed as the first character in a source line which contains an error detected by MACRO. This error code identifies a syntactical problem or some other type of error condition detected during the processing of a source line. An example of such a source line is shown below:

```
Q     26 000236  010102          MOV R1,R2,A
```

The extraneous argument A in the MOV instruction above causes the line to be flagged with a Q (syntax) error.

Error Code	Meaning
A	Assembly error. Because many different types of error conditions produce this diagnostic message, all the possible directives which may yield a general assembly error have been categorized below to reflect specific classes of error conditions:

CATEGORY 1: ILLEGAL ARGUMENT SPECIFIED.

.RADIX -- A value other than 2, 8, or 10 is specified as a new radix.

.LIST/.NLIST -- Other than a legally defined argument (see Table 6-1) is specified with the directive.

.ENABL/.DSABL -- Other than a legally defined argument (see Table 6-2) is specified with the directive.

.PSECT -- Other than a legally-defined argument (see Table 6-3) is specified with the directive.

.IF/.IIF -- Other than a legally defined conditional test (see Table 6-5) or an illegal argument expression value is specified with the directive.

.MACRO -- An illegal or duplicate symbol found in dummy argument list.

Error Code	Meaning
A (Cont'd)	CATEGORY 2: NULL ARGUMENT OR SYMBOL SPECIFIED. .TITLE -- Program name is not specified in the directive, or first non-blank character following the directive is a non-Radix-50 character. .IRP/.IRPC -- No dummy argument is specified in the directive. .NARG/.NCHAR/.NTYPE -- No symbol is specified in the directive. .IF/.IIF -- No conditional argument is specified in the directive. CATEGORY 3: UNMATCHED DELIMITER/ILLEGAL ARGUMENT CONSTRUCTION. .ASCII/.ASCIZ/.RAD50/.IDENT -- Character string or argument string delimiters do not match, or an illegal character is used as a delimiter, or an illegal argument construction is used in the directive. .NCHAR -- Character string delimiters do not match, or an illegal character is used as a delimiter in the directive. CATEGORY 4: GENERAL ADDRESSING ERRORS. This type of error results from one of several possible conditions: 1. Permissible range of a branch instruction, i.e., from -128(10) to +127(10) words, has been exceeded. 2. A statement makes invalid use of the current location counter, e.g., a ".=expression" statement attempts to force the current location counter to cross program section (.PSECT) boundaries. 3. A statement contains an invalid address expression. In cases where an absolute address expression is required, specifying a global symbol, a relocatable value, or a complex relocatable value (see Section 3.9) results in an invalid address expression. Similarly, in cases where a relocatable address expression is required, either a relocatable or absolute value is permissible, but a global symbol or a complex relocatable value in the statement likewise results in an invalid address expression. Specific cases of this type of error are those which follow:

Error Code	Meaning
	.BLKB/.BLKW/.REPT -- Other than an absolute value or an expression which reduces to an absolute value has been specified with the directive. 4. Multiple expressions are not separated by a comma. This condition causes the next symbol to be evaluated as part of the current expression. CATEGORY 5: ILLEGAL FORWARD REFERENCE. This type of error results from either of two possible conditions: 1. A global assignment statement (symbol==expression) contains a forward reference to another symbol. 2. An expression defining the value of the current location counter contains a forward reference.
B	Bounding error. Instructions or word data are being assembled at an odd address. The location counter is incremented by 1.
D	Doubly-defined symbol referenced. Reference was made to a symbol which is defined more than once.
E	End directive not found. When the end-of-file is reached during source input and the .END directive has not yet been encountered, MACRO generates this error code, ends assembly pass 1, and proceeds with assembly pass 2.
I	Illegal character detected. Illegal characters which are also non-printable are replaced by a question mark (?) on the listing. The character is then ignored.
L	Input line is greater than 132(10) characters in length. Currently, this error condition is caused only through excessive substitution of real arguments for dummy arguments during the expansion of a macro.
M	Multiple definition of a label. A label was encountered which was equivalent (in the first six characters) to a label previously encountered.
N	A number contains a digit that is not in the current program radix. The number is evaluated as a decimal value.
O	Opcode error. Directive out of context. Permissible nesting level depth for conditional assemblies has been exceeded. Attempt to expand a macro which was unidentified after .MCALL search.

Error Code	Meaning
P	Phase error. A label's definition of value varies from one assembly pass to another or a multiple definition of a local symbol has occurred within a local symbol block. Also, when in a local symbol block defined by the .ENABL LSB directive, an attempt has occurred to define a local symbol in a program section other than that which was in effect when the block was entered. A P error code also appears if an .ERROR directive is assembled.
Q	Questionable syntax. Arguments are missing, too many arguments are specified, or the instruction scan was not completed.
R	Register-type error. An invalid use of or reference to a register has been made, or an attempt has been made to redefine a standard register symbol without first issuing the .DSABL REG directive.
T	Truncation error. A number generated more than 16 bits in a word, or an expression generated more than 8 significant bits during the use of the .BYTE directive or trap (EMT or TRAP) instruction.
U	Undefined symbol. An undefined symbol was encountered during the evaluation of an expression; such an undefined symbol is assigned a value of zero. Other possible conditions which result in this error code include unsatisfied macro names in the list of .MCALL arguments and a direct assigment (symbol=expression) statement which contains a forward reference to a symbol whose definition also contains a forward reference; also, a local symbol may have been referenced that does not exist in the current local symbol block.
Z	Instruction error. The instruction so flagged is not compatible among all members of the PDP-11 family. See Section 5.3 for details.

APPENDIX E

SAMPLE CODING STANDARD

E.1 INTRODUCTION

Standards eliminate variability and the requirement to make a decision. Much of the difficulty in establishing standards stems from the notion that they should be optimal. However, to be successfully applied, standards must represent an agreement on certain aspects of the programming process.

This Appendix contains DIGITAL's PDP-11 Program Coding Standard. It is suggested that this be used as a model to assist users in preparing standards for their own installations.

E.2 LINE FORMAT

All source lines shall consist of from one to a maximum of eight characters. This program is described in the DEC Editor Reference Manual (see Section 0.3 in the Preface).

Assembly language code lines shall have the following format:

1. Label Field - if present, the label shall start at tab stop 0 (column 1).

2. Operation field - the operation field shall start at tab stop 1 (column 9).

3. Operand field - the operand field shall start at tab stop 2 (column 17).

4. Comments field - the comments field shall start at tab stop 4 (column 33) and may continue to column 80.

Comment lines that are included in the code body shall be delimited by a line containing only a leading semicolon. The comment itself contains a leading semicolon and starts in column 3. Indents shall be 1 tab.

If the operand field extends beyond tab stop 4 (column 33) simply leave a space and start the comment. Comments which apply to an instruction but require continuation should always line up with the character position which started the comment.

E.3 COMMENTS

Comment all coding to convey the global role of an instruction, rather than simply a literal translation of the instruction into English. In general this will consist of a comment per line of code. If a particularly difficult, obscure, or elegant instruction sequence is used, a paragraph of comments must immediately precede that section of code.

Preface text, which describes formats, algorithms, program-local variables, etc., will be delimited by the character sequence ;+ at the start of the text and ;- at the end; these delimiters facilitate automated extraction of narrative commentary. The comment itself will start in column 3.

For example:

```
;+
; THE INVERT ROUTINE ACCEPTS
; A LIST OF RANDOM NUMBERS AND
; APPLIES THE KOLMOGOROV ALGORITHM
; TO ALPHABETIZE THEM.
;-
```

E.4 NAMING STANDARDS

E.4.1 Register Standards

E.4.1.1 **General Purpose Registers** - Only the following names are permitted as register names; and may not be used for any other purpose:

```
R0=%0                   ;REG 0
R1=%1                   ;REG 1
R2=%2                   ;REG 2
R3=%3                   ;REG 3
R4=%4                   ;REG 4
R5=%5                   ;REG 5
SP=%6                   ;STACK POINTER (REG 6)
PC=%7                   ;PROGRAM COUNTER (REG 7)
```

E.4.1.2 **Hardware Registers** - These registers must be named identically to the hardware definition. For example, PS and SWR.

E.4.1.3 **Device Registers** - These are symbolically named identically to the hardware notation. For example, the control status register for the RK disk is RKCS. Only this symbolic name may be used to refer to this register.

E.4.2 Processor Priority

Testing or altering the processor priority is done using the symbols

PR0, PR1, PR2, ...PR7

which are equated to their corresponding priority bit pattern.

E.4.3 Other Symbols

Frequently-used bit patterns such as CR and LF will be made conventional symbolics on an as-needed basis.

E.4.4 Using the Standard Symbolics

The register standards will be defined within the assembler. All other standard symbols will appear in a file and will be linked prior to program execution.

E.4.5 Symbols*

E.4.5.1 **Global Symbols** - Global symbols should be easily recognized by their format. The following standards apply and completely define symbol standards for PDP-11 Medium/Large software products.

symbol	pos-1	pos-2	pos-3	pos-4	pos-5	pos-6	length
non-glbl-sym	letter	a-num/ null	a-num/ null	a-num/ null	a-num/ null	a-num/ null	>=l
glbl-sym	$/. ***	a-num null	a-num/ null	a-num/ null	a-num/ null	a-num/ null	>=1
glbl-offset	letter	$/. ***	a-num	a-num/ null	a-num/ null	a-num/ null	>=3
glbl-pit-ptrn	letter	a-num	$/. ***	a-num/	a-num/	a-num/	>=4
local-sym	number **	$					>=2

* Symbols that are branch targets are also called labels, but we will always use the term "symbol".

** Number is in the range 0<number<65535.

*** The use of $ or . for global names is reserved for DEC-supplied software.

where:

a-num	is an alphanumeric character.
non-glbl-sym	are non-global symbols.
local-sym	local symbols, as defined by MACRO.
glbl-sym	are global symbols (addresses).
glbl-offset	are global offsets (absolute quantities).
glbl-bit-ptrn	are global bit patterns.

A program never contains a .GLOBL statement without showing cause.

E.4.5.2 **Symbol Examples**

Non-Global Symbols

```
AlB

ZXCJl

INSRT
```

Global Address Symbols

```
$JIM

.VECTR

$SEC
```

Global Absolute Offset Symbols

```
A$JIM

A$XT

A.ENT
```

Global Bit Pattern Symbols

```
Al$20

B3.6

JI.M
```

Local Symbols

```
37$

271$

6$
```

E.4.5.3 **Program-Local Symbols** - Self-relative address arithmetic (.+n) is absolutely forbidden in branch instructions; its use in other contexts must be avoided if at all possible and practical.

Target symbols for branches that exist solely for positional reference will use local symbols of the form

<num>$:

Use of non-local symbols is restricted, within reason, to those cases where reference to the code occurs external to the code. Local-symbols are formatted such that the numbers proceed sequentially down the page and from page to page.

E.4.5.4 **Macro Names** - The last two characters (with the last character possibly being null) have special significance. The next to last character is a $, the last, a character specifying the mode of the macro.

For example, in the three macro forms in-line, stack, and p-section, the in-line form has no suffix, the stack has an <S>, and the p-section a <C>. Thus the Queue I/O macro can be written as any of

QIO$

QIO$S

QIO$C

depending on the form required. These are not reserved letters. Only the form of the name is standard.

E.5 PROGRAM MODULES

E.5.1 General Comments on Programs

In our software, a program provides a single distinct function. No limits exist on size, but the single function limitation should make modules larger than 1K a rarity. Since any software may eventually exploit the virtual memory capacity of the 11/40 and 11/45, programs should make every attempt to maintain a dense reference locus (don't promiscuously branch over page boundaries or over a large absolute address distance).

All code is read-only. Code and data areas are distinct and each contains explanatory text. Read-only data should be segregated from read-write data.

E.5.2 The Module Preface

Each program module in the system shall exist as a separate file. The filename will reflect the name of the module and the file extension shall be of the form 'NNN'. The 'NNN' signifies the edit number or the version number. The version number shall be changed only when a new base level is created. Furthermore, if no corrections are made to a file from one base level to the next, the version number will not be changed. The availability of File Control Services and File Control Primitives will greatly simplify version number maintenance. Program modules adhere to a strict format. This format adds to the readability and understandability of the module. The following sections are included in each module:

For the Code Section:

1. A .TITLE statement that specifies the name of the module. If a module contains more than one routine, subtitles may be used.

2. An .IDENT statement specifying the version number. The PDP-11 version number standard appears in section E.10.

3. A .PSECT statement that defines the program section in which the module resides.

4. A copyright statement, and the disclaimer.

   COPYRIGHT (C) 1976
   DIGITAL EQUIPMENT CORPORATION, MAYNARD, MASS.

   THIS SOFTWARE IS FURNISHED UNDER A LICENSE FOR USE ONLY ON A SINGLE COMPUTER SYSTEM AND MAY BE COPIED ONLY WITH THE INCLUSION OF THE ABOVE COPYRIGHT NOTICE. THIS SOFTWARE, OR ANY OTHER COPIES THEREOF, MAY NOT BE PROVIDED OR OTHERWISE MADE AVAILABLE TO ANY OTHER PERSON EXCEPT FOR USE ON SUCH SYSTEM AND TO ONE WHO AGREES TO THESE LICENSE TERMS. TITLE TO AND OWNERSHIP OF THE SOFTWARE SHALL AT ALL TIMES REMAIN IN DEC.

   THE INFORMATION IN THIS DOCUMENT IS SUBJECT TO CHANGE WITHOUT NOTICE AND SHOULD NOT BE CONSTRUED AS A COMMITMENT BY DIGITAL EQUIPMENT CORPORATION.

   DEC ASSUMES NO RESPONSIBILITY FOR THE USE OR RELIABLILITY OF ITS SOFTWARE ON EQUIPMENT WHICH IS NOT SUPPLIED BY DEC.

5. The version number of the file.

   The PDP-11 version number standard is described in section E.10.

6. The name of the principal author and the date on which the module was first created.

7. The name of each modifying author and the date of modification. Names and modification dates appear one per line and in chronological order.

8. A brief statement of the function of the module.

   Note: Items 1-8 should appear on the same page.

9. A list of the definitions of all equated local symbols used in the module. These definitions appear one per line and in alphabetical order.

10. All local macro definitions, preferably in alphabetical order by name.

11. All local data. The data should indicate

    a. Description of each element (type, size, etc.)
    b. Organization (functional, alpha, adjacent, etc.)
    c. Adjacency requirements

12. A more detailed definition of the function of the module.

13. A list of the inputs expected by the module. This includes the calling sequence if non-standard, condition code settings, and global data settings.

14. A list of the outputs produced as a result of entering this module. These include delivered results, condition code settings, but not side effects. (All these outputs are visible to the caller.)

15. A list of all effects (including side effects) produced as a result of entering this module. Effects include alterations in the state of the system not explicitly expected in the calling sequence, or those not visible to the caller.

16. The module code.

E.5.3 Formatting the Module Preface

Rules:

1. The first eight items appear on the same page and will not have explicit headings. Item 3 may be omitted if the blank p-section is being used.

2. Headings start at the left margin*; descriptive text is indented 1 tab position.

3. Items 7-14 will have headings which start at the left margin, preceded and followed by lines containing only a leading <;>. Items which do not apply may be omitted.

A template for the module preface follows.

FILE-EXAMPL.S01

```
        .TITLE   EXAMPLE
        .IDENT   /01/
        .PSECT   KERNEL

;
; COPYRIGHT (C) 1976
; DIGITAL EQUIPMENT COPORATION, MAYNARD, MASS.
;
; THIS SOFTWARE IS FURNISHED UNDER A LICENSE FOR USE ONLY ON A
; SINGLE COMPUTER SYSTEM AND MAY BE COPIED ONLY WITH THE
; INCLUSION OF THE ABOVE COPYRIGHT NOTICE.  THIS SOFTWARE, OR
; ANY OTHER COPIES THEREOF, MAY NOT BE PROVIDED OR OTHERWISE
; MADE AVAILABLE TO ANY OTHER PERSON EXCEPT FOR USE ON SUCH
; SYSTEM AND TO ONE WHO AGREES TO THESE LICENSE TERMS.  TITLE
; TO AND OWNERSHIP OF THE SOFTWARE SHALL AT ALL TIMES REMAIN
; IN DEC.
;
; THE  INFORMATION IN THIS DOCUMENT IS SUBJECT TO CHANGE WITHOUT
; NOTICE AND  SHOULD NOT BE CONSTRUED AS A COMMITMENT BY DIGITAL
; EQUIPMENT CORPORATION.
;
```

*The left margin consists of a <;> a <space> then the heading, so the text of the heading begins in column 3.

```
; DEC ASSUMES NO RESPONSIBILITY FOR THE USE OR RELIABILITY OF
; ITS  SOFTWARE ON  EQUIPMENT WHICH  IS NOT SUPPLIED BY DEC.
;
; VERSION 01
;
; JOE PASCUSNIK 1-JAN-72
;
; MODIFIED BY:
;
;       RICHARD DOE 21-JAN-73
;
;       SPENCER THOMAS 12-JUN-73
;
;    Brief statement of the module's function
;
; EQUATED SYMBOLS
;

        List equated symbols

;
; LOCAL MACROS
;

        Local Macros

;
; LOCAL DATA
;

        Local data

;+
; Module function-details
;
; INPUTS:
;
;       Description of inputs
;
; OUTPUTS:
;
;       Description of outputs
;
; EFFECTS:
;
;       Description of effects
;-

        Begin Module Code
```

E.5.4 **Modularity**

No other characteristic has more impact on the ultimate engineering success of a system than does modularity. Modularity for PDP-11 Software Engineering's products consists of the application of the single-function philosophy described in section E.5.1, and adherence to a set of calling and return conventions.

E.5.4.1 **Calling Conventions (Inter-Module)** - The following calling conventions must be observed.

Transfer of Control

Macros will exist for call and return. The actual transfer will be via a JSR PC instruction. For register save routines, a JSR Rn,SAVE will be permitted.

The CALL macro is:

```
CALL    subr-name
```

The RETURN macro is:

```
RETURN
```

Register Conventions

On entry, a subroutine minimally saves all registers it intends to alter except result registers. On exit it restores these registers. (State preservation is assumed across calls.)

Argument Passing

Any registers may be used, but their use should follow a coherent pattern. For example, if passing three arguments, pass them in R0, Rl and R2 rather than R0, R2, R5. Saving and restoring occurs in one place.

E.5.4.2 **Exiting** - All subroutine exits occur through a single RETURN macro.

E.5.4.3 **Intra-Module Calling Conventions** - Designer optional, but consistency favors a calling sequence identical to that of the inter-module sequence.

E.5.4.4 **Success/Failure Indication** - The C bit will be used to return the success/failure indicator, where success equals 0, and failure equals 1. The argument registers can be used to return values or additional success/failure data.

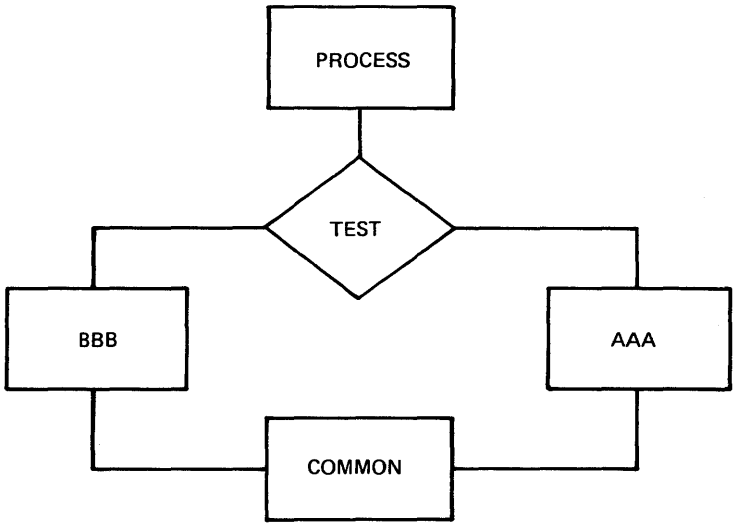
E.5.4.5 **Module Checking Routines** - Modules are responsible for verifying the validity of arguments passed to them. The design of a module's calling sequence should aim at minimizing the validity checks by minimizing invalid combinations. Programmers may add test code to perform additional checks during checkout. All code should aim at discovering an error as close (in terms of instruction executions) to its occurrence as possible.

E.6 FORMATTING STANDARDS

E.6.1 Program Flow

Programs will be organized on the listing such that they flow down the page, even at the cost of an extra branch or jump.

For example:



shall appear on the listing as:

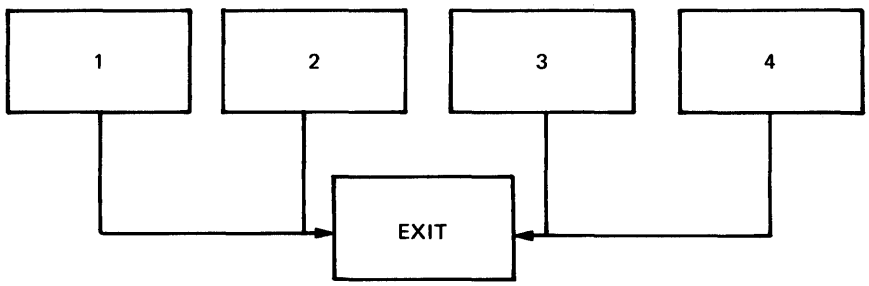
```
        TST
        BNE     BBB
AAA:    ....    ....
        ....    ....
        ....    ....
        ....    ....
        BR      CMN

BBB:    ....    ....
        ....    ....
        ....    ....
CMN:    ....    ....
        ....    ....
        ....    ....
```

Rather than:

```
        TST
        BNE     BBB
AAA:    ....    ....
        ....    ....
        ....    ....
        ....    ....
CMN:    ....    ....
        ....    ....
        ....    ....
        ....    ....
BBB:    ....    ....
        ....    ....
        ....    ....
        ....    ....
        BR      CMN
```

E.6.2 Common Exits

A common exit appears as the last code sequence on the listing. Thus the flow chart:



will appear on the listing as:

```
PR1:    ....    ....
        ....    ....
        ....    ....
        BR      EXIT

PR2:    ....    ....
        ....    ....
        ....    ....
        BR      EXIT

PR3:    ....    ....
        ....    ....
        ....    ....
        BR      EXIT

PR4:    ....    ....
        ....    ....
        ....    ....

EXIT:
```

And not as:

```
PR1:    ....    ....
        ....    ....
        ....    ....

EXIT:   ....    ....
        ....    ....
        ....    ....

PR2:    ....    ....
        ....    ....
        ....    ....
        BR      EXIT

PR3:    ....    ....
        ....    ....
        ....    ....
        BR      EXIT

PR4:    ....    ....
        ....    ....
        ....    ....
        BR      EXIT
```

E.6.3 Code with Interrupts Inhibited

Code that is executed with interrupts inhibited, shall be flagged by a three semicolon (;;;) comment delimiter. For example:

```
..ERTZ:                             ;ENABLE BY RETURNING
                                    ;BY SYSTEM SUBROUTINES,

        BIS     #PR7,PS             ;;; INHIBIT INTERRUPTS
        BIT     #PR7,+2(SP)         ;;; C
        BEQ     10$                 ;;;  O
        RTT                         ;;;   M
                                    ;;;    M
10$:    ....    ....                ;;;     E
        ....    ....                ;;;      N
        ....    ....                ;;;       T
        ....    ....                ;;;        S
```

E.7 PROGRAM SOURCE FILES

Source creation and maintenance shall be done in base levels. A base level is defined as a point at which the program source files have been frozen. From the freeze point to the next base level, corrections will not be made directly to the base level itself. Rather a file of corrections shall be accumulated for each file in the base level. Whenever an updated source file is desired, the correction file will be applied to the base file.

The accumulation of corrections shall proceed until a logical breaking point has occurred (i.e. a milestone or significant implementation point has been reached). At this time all accumulated corrections shall be applied to the previous base level to create a new base level. Correction files will then be started for the new base level.

E.8 FORBIDDEN INSTRUCTION USAGE

1. The use of instructions or index words as literals of the previous instruction. For example:

   ```
   MOV   @PC,Register

   BIC   Src,Dst
   ```

   uses the bit clear instruction as a literal. This may seem to be a very "neat" way to save a word but what about maintaining a program using this trick? To compound the problem, it will not execute properly if I/D space is enabled on the 11/45. In this case @PC is a D bank reference.

2. The use of the MOV instruction instead of a JMP instruction to transfer program control to another location. For example:

   ```
   MOV   #ALPHA,PC
   ```

   transfers control to location ALPHA. Besides taking longer to execute (2.3 microseconds for MOV vs. 1.2 for JMP) the use of MOV instead of JMP makes it nearly impossible to pick up someone else's program and tell where transfers of control

take place. What if one would like to get a jump trace of the execution of a program (a move trace is unheard of)? As a more general issue, perhaps even other operations such as ADD and SUB from PC should be discouraged. Possibly one or two words can be saved by using these operations but how many such occurrences are there?

3. The seemingly "neat" use of all single word instructions where one double-word instruction could be used and would execute faster and would not consume additional memory. Consider the following instruction sequence:

```
CMP   -(R1),(-R1)

CMP   -(R1),-(R1)
```

The intent of this instruction sequence is to subtract 8 from register R1 (not to set condition codes). This can be accomplished in approximately 1/3 the time via a SUB instruction (9.4 vs. 3.8 microseconds) at no additional cost in memory space. Another question here is also, what if R1 is odd? SUB always wins since it will always execute properly and is always faster!

E.9 RECOMMENDED CODING PRACTICE

E.9.1 Conditional Branches

When using the PDP-11 conditional branch instructions, it is imperative that the correct choice be made between the signed and the unsigned branches.

SIGNED	UNSIGNED
BGE	BHIS (BCC)
BLT	BLO
BGT	BHI
BLE	BLOS (BCS)

A common pitfall is to use a signed branch (e.g. BGT) when comparing two memory addresses. All goes well until the two addresses have opposite signs; that is, one of them goes across the 16K (100000(8)) bound. This type of coding error usually shows itself as a result of re-linking at different addresses and/or a change in size of the program.

E.10 PDP-11 VERSION NUMBER STANDARD

The PDP-11 Version Number Standard applies to all modules, parameter files, complete programs, and libraries which are written or caused to be written, as part of the PDP-11 Software Development effort. It is used to provide unique identification of all released, pre-released, and in-house software.

It is limited in that, as currently specified, only six characters of identification are used. Future implementations of the Macro Assembler, Task Builder, and Librarian should provide for at least

nine characters, and possibly twelve. It is expected that this standard will be enhanced as the need arises.

Version Identifier = <form> <version> <edit> <patch>

<form>	Used to identify a particular form of a module or program, where applicable, as in the case of LINK-11. One alphabetic character, if used, and null (i.e., a binary 0) if not used.
<version>	Used to identify the release, or generation, of a program. Two decimal digits, starting at 00, and incremented at the discretion of the project in order to reflect what, in their opinion, is a major change.
<edit>	Used to identify the level to which a particular release, or generation, of a program or module has been edited. An edit is defined to be an alteration to the source form. Two decimal digits, beginning at 01, and incremented with each edit; null if no edits.
<patch>	Used to identify the level to which a particular release, or generation, of a program or module has been patched. A patch is defined as an alteration to a binary form. One alphabetic character, starting at B, and running sequentially toward Z, each time a set of patches is released; null if no patches.

These fields are interrelated. When <version> is changed, then <patch> and <edit> must be reset to nulls. It is intended that when <edit> is incremented, then <patch> will be re-set to null, because the various bugs have been fixed.

E.10.1 Displaying the Version Identifier

The visible output of the version identifier should appear as:

Key <letter> <form> <version> - <edit> <patch>,

where the following Key Letters have been identified:

V	released or frozen version
X	in-house experimental version
Y	field test, pre-release, or in-house release version

Note that 'X' corresponds roughly to individual support, 'Y' to group support, and 'V' to company support.

The dash which separates <version> from <edit> is used only if <edit> and/or <patch> is not null. When a version identifier is displayed as part of program identification, then the format is:

Program Name <space><key-letter><form><version>-<edit><patch>

Examples:

```
PIP X03
LINK VB04-C
MACRO Y05-01
```

E.10.2 Use of the Version Number in the Program

All sources must contain the version number in an .IDENT directive. For programs (or libraries) which consist of more than one module, each individual module will follow this version number standard. The version number of the program or library is not necessarily related to the version numbers of the constituent modules; it is perfectly reasonable, for example, that the first version of a new FORTRAN library, V00, contain an existing SIN routine, say V05-01.

Parameter files are also required to contain the version number in an .IDENT directive. Because the assembler records the last .IDENT seen, parameter files must precede the program.

Entities which consist of a collection of modules or programs, will have an identification module in the first position. An identification module exists solely to provide identification, and normally consists of something like:

```
;OTS IDENTIFICATION
.TITLE FTNLIB
.IDENT /003010/
.END
```

APPENDIX F

SAMPLE ASSEMBLY AND CROSS REFERENCE LISTING

```
;
;
;                    COPYRIGHT (C) 1977 BY
;            DIGITAL EQUIPMENT CORPORATION, MAYNARD, MASS.
;
;
; THIS SOFTWARE IS FURNISHED UNDER A LICENSE AND MAY BE USED AND COPIED
; ONLY IN ACCORDANCE WITH THE TERMS OF SUCH LICENSE AND WITH THE
; INCLUSION OF THE ABOVE COPYRIGHT NOTICE.  THIS SOFTWARE OR ANY OTHER
; COPIES THEREOF MAY NOT BE PROVIDED OR OTHERWISE MADE AVAILABLE TO ANY
; OTHER PERSON.  NO TITLE TO AND OWNERSHIP OF THE SOFTWARE IS HEREBY
; TRANSFERRED.
;
;
; THE INFORMATION IN THIS SOFTWARE IS SUBJECT TO CHANGE WITHOUT NOTICE
; AND SHOULD NOT BE CONSTRUED AS A COMMITMENT BY DIGITAL EQUIPMENT
; CORPORATION.
;
;
; DIGITAL ASSUMES NO RESPONSIBILITY FOR THE USE OR RELIABILITY OF ITS
; SOFTWARE ON EQUIPMENT WHICH IS NOT SUPPLIED BY DIGITAL.
;
;
;
; MODULE:
;
; VERSION:
;
; AUTHOR:
;
; DATE:
;
;
```

```
                                                .TITLE  TST205
                                                .SBTTL  LOCAL DATA STRUCTURES
                                        ;
                                        ;
 000000                                 FPTRPY:
 000000  000004                                 .WORD   4               ; REPLY MESSAGE FPT
 000002  000000                         MSGRPY: .WORD   0               ;    ADDRESS OF MESSAGE
 000004  000012'                                .WORD   SIZE            ;    ADDRESS OF MSG SIZE
 000006  000014'                                .WORD   REPLYN          ;    ADDRESS OF REPLY NUM
 000010  000016'                                .WORD   STATUS          ;    ADDRESS OF STATUS RETURN
 000012                                 SIZE:
 000012  000043                                 .WORD   35.
 000014                                 REPLYN:
 000014  000000                                 .WORD   0
 000016                                 STATUS:
 000016  000000  000000                         .WORD   0,0
 000022                                 FUNC:
 000022  030060                                 .WORD   "00
 000024  030461                                 .WORD   "11
 000026  031062                                 .WORD   "22
 000030  031463                                 .WORD   "33
 000032  025052                                 .WORD   "**
 000034  027056                                 .WORD   "..

TST205 MAIN

                                                .SBTTL  TST205 MAIN
                                        ;
                                        ;       TST205 IS USED IN TRANSACTION TIM08N TO TEST THE TIM
                                        ;       ACTION CODE C.KEY.  THE TST RECEIVES THE TEXT ASSOCIATED
                                        ;       WITH THE FUNCTION KEY DEPRESSED.  THE TEXT IS 35 CHARACTERS
                                        ;       IN LENGTH.  EACH KEY HAS A SPECIFIED TEXT.  DEPENDING
                                        ;       ON THE TEXT, THEN, TST206 SENDS ONE OF 7 REPLIES TO DISPLAY
                                        ;       THE TEXT.  THE MULTIPLE REPLIES ENABLE THE TEXT OF EACH FUNCTION
                                        ;       KEY TO BE DISPLAYED ON A DIFFERENT LINE
                                        ;
 000036                                 TSTEP::
 000036  016504  000002                         MOV     2(R5),R4        ; ADDRESS OF EXCHANGE MSG
                                        ;
                                        ;
 000042  012702  000006                         MOV     #6,R2           ; INITIALIZE
 000046  012701  000022'                        MOV     #FUNC,R1        ; ADDRESS OF POSSIBLE REPLY TEXT
 000052                                 20$:
 000052  022114                                 CMP     (R1)+,(R4)      ; LOOP UNTIL FUNC TEXT FOUND
 000054  001401                                 BEQ     40$
 000056  077203                                 SOB     R2,20$          ; END LOOP
```

```
    45 000060                          40$:
    46 000060  010267  177730                  MOV     R2,REPLYN       ; DEFINE REPLY NUMBER
    47                                 ;
    48                                 ;
    49 000064  010467  177712                  MOV     R4,MSGRPY       ; SAVE ADDRESS OF EM FOR FPT
    50                                 ;
    51                                 ;
    52 000070  012705  000000'                 MOV     #FPTRPY,R5      ; SEND REPLY
    53 000074  004767  000000G                 CALL    REPLY
    54 000100  000207                          RETURN
    55         000001                          .END

SYMBOL TABLE

FPTRPY  000000R         MSGRPY  000002R         REPLYN  000014R         STATUS  000016R         TSTEP   000036RG
FUNC    000022R         REPLY = ****** GX       SIZE    000012R

. ABS.  000000      000
        000102      001
ERRORS DETECTED:  0

VIRTUAL MEMORY USED:  84 WORDS  ( 1 PAGES)
DYNAMIC MEMORY:  16142 WORDS  ( 62 PAGES)
ELAPSED TIME:  00:00:03
TST205,TST205=TST205


SYMBOL TABLE

FPTRPY  000000R         MSGRPY  000002R         REPLYN  000014R         STATUS  000016R         TSTEP   000034RG
FUNC  = ****** GX       REPLY = ****** GX       SIZE    000012R
. ABS.  000000      000
        000100      001
ERRORS DETECTED:  0

VIRTUAL MEMORY USED:  84 WORDS  ( 1 PAGES)
DYNAMIC MEMORY:  3532 WORDS  ( 13 PAGES)
ELAPSED TIME:  00:00:02
LI,OBJ/CRF=TST205.MAC
```

```
SYMBOL CROSS REFERENCE                                  CREF   V01

SYMBOL  VALUE           REFERENCES

FPTRPY    000000 R      #1-5        1-50
FUNC    = ******  GX     1-38
MSGRPY    000002 R      #1-7        *1-47
REPLY   = ******  GX     1-51
REPLYN    000014 R       1-9        #1-13      *1-44
SIZE      000012 R       1-8        #1-11
STATUS    000016 R       1-10       #1-15
TSTEP     000034 RG     #1-33
```

INDEX

Absolute addresses, 6-13
Absolute binary output, 6-13
Absolute expression, 3-16
Absolute mode, 5-5, 5-7
Absolute module, 6-34
Absolute program section, 6-37
Address boundaries, 6-31
Address mode syntax, B-1
Address modes, 5-1
Addressing forms, summary, 5-7
Allocating byte data, 6-17
Allocating word data, 6-18
Allocation requirements, 6-35
Alternate radix, 6-25
Ampersand, 3-2
Angle brackets, 3-3, 3-15, 6-4, 6-25, 6-28, 7-4 to 7-5, 7-16 to 7-17
Apostrophe, 7-10
Argument substitution, 7-16
Arithmetic addition operator or autoincrement indicator, 3-2
Arithmetic division operator, 3-2
Arithmetic multiplication operator, 3-2
Arithmetic subtraction operator or autodecrement indicator, 3-2
ASCII character set, A-1
ASCII conversion, 3-14
ASCII conversion characters, 6-19
.ASCII directive, 6-20
.ASCIZ directive, 6-21
.ASECT directive, 6-38
Assembler directives, 6-1, B-1, B-2
Assembler version, 6-8
Assembly language, B-1
Assembly listing, 2-6
Assembly pass 1, 1-1
Asterisk, 3-2
At sign (@), 3-2
Attribute of the current location counter, 3-12
Autodecrement deferred mode, 5-3, 5-7
Autodecrement mode, 5-7
Autoincrement deferred mode, 5-3, 5-7
Autoincrement mode, 5-2, 5-7
^B operator, 6-25
Backslash, 3-2
Binary operators, 3-15
Blank lines, 2-2
.BLKB directive, 6-30
.BLKW directive, 6-30
Blocks of storage, reserving, 6-30
Branch instruction addressing, 5-8
.BYTE directive, 6-17
^C operator, 6-27
Calling conventions, E-8
Calling macros, 7-3
Changing default radix, 3-13
Changing value of location counter, 3-12
Character set, 3-1
Character substitution, 7-16
Code and data separation, 6-38
Code or data sharing, 6-38
Coding standard, E-1
Colon, 3-1
Comma, 3-2
Command string format, 8-1
Comment, 6-14, E-2
Comment field, 2-5
Comment field indicator, 3-2
Complementing an argument, 6-27
Complex relocatable expression, 3-16
Complex relocation, 4-1
Concatenated, 6-35
Concatenation of macro arguments, 7-10
Conditional assembly block, 6-41
Conditional assembly directive, 6-41, 6-42
Conditional branches, E-13
Continuation lines, 2-2
Creating local symbols automatically, 7-7
Creating program sections, 6-36
Cross-reference listing (CREF), 8-5
Cross-reference processor, 8-4
.CSECT directive, 6-17, 6-38
Current location counter, 2-2, 3-11, 3-14, 5-6, 6-29

^D operator, 6-25
Data storage directives, 6-17
Date, 6-8
Default object module name, 6-11
Default register definitions, 6-15
Deferred addressing indicator, 3-2
Defining macros, 7-1
Device registers, E-2
Diagnostic, 7-14
Diagnostic error message summary, D-1
Direct assignment operator, 3-1
Direct assignment statements, 3-7
Directives, 2-5, 5-9, 6-1
Double ASCII character indicator, 3-2
Double colon, 3-1, 3-7
Double equal sign, 3-1, 3-7
Double quote, 3-2, 3-14, 6-19
.DSABL directive, 3-7, 3-9, 6-13 to 6-15, 6-27
Duplication of code, 7-17

EMT, 5-8
.ENABL directive, 5-8, 6-13 to 6-15, 6-27
.END directive, 6-31
.ENDC directive, 6-41
.ENDM directive, 7-2
End of the source input, 6-31
.ENDR directive, 7-18
Entry-point instructions, 6-33
.EOT directive, 6-31
Equal sign, 3-1
Error codes, D-1
.ERROR directive, 7-14
Error messages, 8-8 to 8-11
Evaluation of expressions, 3-15
.EVEN, 6-29
Exclamation point, 3-2
Exiting, E-9
Expressions, 3-14, 3-15
External expression, 3-15, 3-16
External symbols, 6-40
Externally-defined macro, 7-18

^F operator, 3-14, 6-27
File specification format, 8-7
File Specification Qualifiers TRAX, 8-3
Finding address mode of macro arguments, 7-13
Finding number of characters in strings, 7-12
Floating point,
  data, 6-26
  number, 6-28
  number specification, 6-27
  rounding, 6-14, 6-26
  storage directives, 6-27
  truncation, 6-14, 6-27
.FLT2 directive, 6-27
.FLT4 directive, 6-27
Forbidden instruction usage, E-12
Form-feed, 6-13, 7-3
Format control, 2-6
Formatting standards, E-9
Forward referencing, 3-8
Function directives, 6-13

General purpose registers, E-2
General registers, 3-9
Global,
  label, 6-40
  references, 6-15
  symbol, 2-3, 6-40
  symbol directory, 1-2
.GLOBAL directive, 3-7, 6-39
GSD, 1-2

Hardware registers, E-2
Horizontal formatting, 2-6

.IF directive, 6-40
.IFF directive, 6-43, 6-44
.IFT directive, 6-43
.IFTF directive, 6-43
.IIF directive, 6-45
Illegal characters, 3-3
Immediate conditional assembly, 6-45
Immediate expression indicator, 3-2
Immediate mode, 5-4, 5-7
Immediate mode deferred, 5-5
Implicit .WORD directive, 2-5, 6-18
Indefinite repeat block directives, 7-15
Index deferred mode, 5-4, 5-7
Index mode, 5-4, 5-7

Indirect command files, 8-6
Initial argument or expression indicator, 3-2
Initial register indicator, 3-2
Instruction set, C-1
Invoking MACRO under TRAX, 8-1
.IRP directive, 7-15, 7-16
.IRPC directive, 7-15, 7-16
Item or field terminator, 3-1

Keyword arguments, 7-4, 7-9

Label field, 2-2
Label terminator, 3-1
Left angle bracket, 3-2
Left parenthesis, 3-2
.LIMIT directive, 6-31
Line format, E-1
Linking, 4-1
.LIST directive, 6-1
Listing conditional assemblies, 6-4
Listing control directives, 6-1
Listing control switches, 8-2
Listing level count, 6-2
Listing of binary extensions, 6-4
Listing of comments, 6-4
Listing of generated binary code, 6-3
Listing of macro calls, 6-4
Listing of macro definitions, 6-4
Listing of macro expansion binary code, 6-4
Listing of repeat range expansions, 6-4
Listing of source line sequence numbers, 6-3
Listing of source lines, 6-4
Listing of the current location counter, 6-3
Listing of the symbol table, 6-5
Local symbol block, 6-14
Local symbol block delimiters, 3-10
Local symbols, 3-6, 3-10, 3-11
Location counter, 6-36
Location counter control directives, 6-29
Logical AND operator, 3-2, 6-42
Logical inclusive OR operator, 3-2, 6-42
Lower-case ASCII, 6-14

Macro arguments, 7-6
Macro attribute directives, 7-11
Macro call, 2-5, 7-3, 7-5
Macro call arguments, 7-4
Macro call numeric argument indicator, 3-2
MACRO character sets, A-1
Macro definition, 7-1, 7-15
Macro definition arguments, 7-4
Macro definition formatting, 7-3
Macro definition termination, 7-2
MACRO directives, 5-9, C-4
Macro directives, 7-1
Macro expansion termination, 7-3
Macro library directive, 7-18
Macro name, 7-1, 7-4
Macro names, E-5
Macro nesting, 7-5
Macro qualifiers, 8-2
Macro symbol table, 3-6
MACRO symbols, 3-5
.MCALL directive, 7-18
Memory allocation, 6-32, 6-33, 6-38
Memory allocation and mapping, 6-32
.MEXIT directive, 7-3
Minus sign, 3-2
Modularity, E-8
Module checking routines, E-9
Module preface, E-5
Multi-defined label, 2-4
Multiple definitions of local symbols, 3-11
Multiple labels, 2-4

Naming standards, E-2
.NARG directive, 7-11
.NCHR directive, 7-11, 7-12
Negative numbers, 3-13
Nested conditional directives, 6-43
Nested macros, 7-3, 7-5
.NLIST directive, 6-1, 6-11
.NTYPE directive, 7-11, 7-13
Number of arguments in macro calls, 7-7, 7-11
Number sign, 3-2
Numbers, 3-13
Numeric control, 6-24
Numeric control operators, 6-26, 6-27
Numeric directives, 6-26

^O operator, 6-25
Object module, 4-1
Object module name, 6-11
Octal radix, 3-13
.ODD directive, 6-29
Op codes, 2-4, C-1
Operand field, 2-4
Operand field separator, 3-2
Operating procedures, 8-1
Operator field, 2-4
Order of symbol table search, 3-6
Other symbols, E-3
Overlaid, 6-35
Overlays, 6-33

.PAGE directive, 6-12
Page eject, 7-3
Page ejection, 6-13
Page formatting, 2-6
Page headings, 6-8
Page number, 6-8
Passing numeric arguments as symbols, 6-45
Percent sign, 3-2
Permanent symbol table, 3-5, C-1
Plus sign, 3-2
.PRINT directive, 7-14
Processor priority, E-3
Program boundaries directive, 6-31
Program counter, 3-9, 5-1
Program modules, E-5
Program section access, 6-33
Program section name, 6-33
Program sections, 3-12, 6-32
Program source files, E-12
Program-local symbols, E-4
Programming standards and conventions, 2-1
.PSECT directive, 3-12, 6-32, 6-35

^R operator, 6-23
.RAD50 directive, 3-13, 6-22
Radix control, 6-24
Radix control operators, 6-25
.RADIX directive, 3-13, 6-24
Radix-50 character set, A-4
Radix-50 control operator, 6-23
Radix-50 data, 6-22
Read-only access, 6-33
Read/write access, 6-33
Register deferred mode, 5-2
Register expression, 5-1
Register, mode, 5-1, 5-7
Register standards, E-2
Register symbols, 3-9
Register term indicator, 3-1
Relative addresses, 6-13
Relative addressing mode, 5-6
Relative deferred mode, 5-6, 5-7
Relative mode, 5-6, 5-7
Relocatability, 6-34
Relocatable expressions, 3-16, 4-1
Relocatable module, 6-34
Relocatable program sections, 6-37
Relocation, 4-1
Relocation bias, 2-2, 6-34
Repeat block directive, 7-18
.REPT directive, 7-18
Reserving storage, 6-30
Reserving storage space, 3-13, 6-30
Right parenthesis, 3-2

.SBTTL directive, 6-8, 6-11
Scope of the program section, 6-33
Semicolon, 3-2
Sending messages to listing file, 7-14
Separating and delimiting characters, 3-2
Single ASCII character indicator, 3-2
Single quote, 3-2, 3-14, 6-19, 7-10
Slash, 3-2
Source line sequence numbers, 6-3
Space, 3-1
Special characters, B-1
Special characters in macro arguments, 7-6
Stack pointer, 3-9
Statement format, 2-1
Storing Radix-50 data, 6-23
Subconditional assembly, 6-43
Subtitle, 6-8
Success/failure indication, E-9
Symbol control directive, 6-39
Symbol examples, E-4
Symbol table listing, 1-2
Symbolic arguments of listing control directives, 6-3, 6-4
Symbols, E-3
Symbols and expressions, 3-1
System macro libraries, 7-18

Tab, 3-1
Tab character, 2-2
Table of contents, 6-4, 6-11

Teleprinter mode, 6-5
Terminal argument or expression indicator, 3-2
Terminal register indicator, 3-2
Terminating directives, 6-31
Terms, 3-14
Time-of-day, 6-8
.TITLE directive, 6-11
Title of the object module, 6-8
Translating to ASCII, 6-20, 6-21
Translating to Radix-50, 6-22
Trap instructions, 5-8
TRAX Command String Format, 8-1
TRAX File Specification Format, 8-7
TRAX File Specification Qualifiers, 8-3
TRAX Indirect Command Files, 8-6
TRAX MACRO in Batch Mode, 8-6
TRAX MACRO Qualifiers, 8-2
TRAX Operating Procedures, 8-1

Unary and binary operators, 3-5
Unary control, 6-24
Unary operator ordering, 6-27
Unary operators, 3-15
Unconditional assembly, 6-43
Undefined symbols, 3-7, 3-14
Universal unary operator or argument indicator, 3-2
Up arrow or circumflex, 3-2
Up-arrow, 3-3
Up-arrow (^) construction, 7-5
User symbol table, 3-5
User-defined and macro symbols, 3-5
User-defined macro libraries, 7-18
Using the standard symbolics, E-3

Version number, 6-12
Version number standard, E-13

.WORD directive, 3-11, 6-18

TRAX MACRO
Reference Manual
AA-D340A-TC

READER'S COMMENTS

NOTE: This form is for document comments only. DIGITAL will use comments submitted on this form at the company's discretion. Problems with software should be reported on a Software Performance Report (SPR) form. If you require a written reply and are eligible to receive one under SPR service, submit your comments on an SPR form.

Did you find errors in this manual? If so, specify by page.

_____

_____

_____

_____

_____

_____

Did you find this manual understandable, usable, and well-organized? Please make suggestions for improvement.

_____

_____

_____

_____

_____

_____

Is there sufficient documentation on associated system programs required for use of the software described in this manual? If not, what material is missing and where should it be placed?

_____

_____

_____

_____

_____

_____

Please indicate the type of user/reader that you most nearly represent.

- ☐ Assembly language programmer
- ☐ Higher-level language programmer
- ☐ Occasional programmer (experienced)
- ☐ User with little programming experience
- ☐ Student programmer
- ☐ Non-programmer interested in computer concepts and capabilities

Name _____ Date _____

Organization _____

Street _____

City _____ State _____ Zip Code _____
or
Country

Please cut along this line.

---------- Fold Here ----------

---------- Do Not Tear - Fold Here and Staple ----------

FIRST CLASS
PERMIT NO. 33
MAYNARD, MASS.

BUSINESS REPLY MAIL
NO POSTAGE STAMP NECESSARY IF MAILED IN THE UNITED STATES

Postage will be paid by:

digital

Software Documentation
146 Main Street ML5-5/E39
Maynard, Massachusetts 01754