MACRO-9 ASSEMBLER

Programmer's Reference Manual

PDP-9 ADVANCED SOFTWARE SYSTEM

Order No. DEC-9A-AMZA-D from the Program Library, Digital Equipment Corporation, Price $2.00
Maynard, Mass. Direct comment concerning this manual to Software Quality Control, Maynard, Mass.

DIGITAL EQUIPMENT CORPORATION □ MAYNARD, MASSACHUSETTS

1st Edition August 1967
2nd Edition Revised December 1967
3rd Edition Revised November 1968

PREFACE

It is assumed that the reader is familiar with the PDP-9 User's Handbook and Supplement (F-95), and with the Monitors manual (DEC-9A-MAD0-D). In this manual, frequent references are made to the Linking Loader, which is described in Utility Programs manual (DEC-9A-GUBA-D).

CONTENTS

CONTENTS (Cont)

This is a complete sample Program Listing. The last column (comments) contains the paragraph number in this manual where full explanations may be found.

```
↑P

MACRO

>B,L,S←SAMPLE


MD      00101;00102
H       00107
 PASS 1 COMPLETED
↑P↑P


SAMPLE    PAGE   1


                                                    .TITLE SAMPLE          /3.1
                                                    .DEC                   /3.4
                    000013  A       A=11;           100;       200         /2.2.3
     00000  R       000144  A
     00001  R       000310  A
                                                    .OCT                    3.4
                    000011  A       A=11;           100;       200
     00002  R       000100  A
     00003  R       000200  A
     00004  R       200010  A       START           LAC 10
     00005  R               A       BUFF            .BLOCK 12              /3.5
     00017  R               A       C               .BLOCK 0
     00017  R       200106  R                       LAC TEMP#              /2.2.2
     00020  R       000000  A       D               0; 0; 0;               /2.2.3
     00021  R       000000  A
     00022  R       000000  A
     00023  R       406050  A       TAG             .ASCII 'ABC'           /3.8
     00024  R       300000  A
     00025  R       010203  A                       .SIXBT 'ABC'            3.8
     00026  R       406050  A       ADDRES          .ASCII 'ABCD'EFGE
     00027  R       342214  A
     00030  R       434000  A
     00031  R       000000  A
     00032  R       406041  A                       .ASCII 'AB'<11>
     00033  R       100000  A
                                                    .IFDEF A               /3.13
U    00034  R       200107  R                       LAC H
                                                    .ENDC
     00073  R                                       .LOC TAG+50
     00073  R       040400  A       DEP             DAC 400
     00074  R       040401  A       DEPT            DAC 401
                                                    .GLOBL X,Y,Z           /3.9
     00075  R       200100  A       X               LAC 100
     00076  R       120110  E                       JMS* Y
     00077  R       120111  E                       JMS* Z
                                                    .IODEV 1,2,-3,-4,5     /3.10
     00100  R       000026  R                       .DSA ADDRES            /3.11
M    00101  R       200075  R       MD              LAC X
DM   00102  R       200101  R       MD              LAC MD
                                                    .REPT 3,1              /3.12
     00103  R       200010  A                       LAC 10
     00104  R       200011  A *RPT
     00105  R       200012  A *RPT
                    000004  R                       .END START             /3.6
     00110  R       000110  E *ETV
     00111  R       000111  E *ETV

                                                    THERE ARE   3 ERROR LINES


SAMPLE    PAGE   2                      SAMPLE    PAGE   3

A          000011    A                  START       00004    R
ADDRES       00026   R                  BUFF        00005    R
BUFF         00005   R                  A          000011    A
C            00017   R                  C           00017    R
D            00020   R                  D           00020    R
DEP          00073   R                  TAG         00023    R
DEPT         00074   R                  ADDRES      00026    R
H            00107   R                  DEP         00073    R
MD           00101   R                  DEPT        00074    R
START        00004   R                  X           00075    R
TAG          00023   R                  MD          00101    R
TEMP         00106   R                  TEMP        00106    R
X            00075   R                  H           00107    R
Y            00110   E                  Y           00110    E
Z            00111   E                  Z           00111    E
```

CHAPTER 1
INTRODUCTION

MACRO-9 is the symbolic assembly program for the PDP-9 ADVANCED Software System. Operating under control of one of the Monitor systems, which handles I/O functions, the MACRO-9 Assembler processes input source programs in two passes, and requires less than 6K* of core memory.

MACRO-9 makes machine language programming on the PDP-9 much easier, faster and more efficient. It permits the programmer to use mnemonic symbols to represent instruction operation codes, locations, and numeric quantities. By using symbols to identify instructions and data in his program, the programmer can easily refer to any point in his program, without knowing actual machine locations.

The standard output of the Assembler is a relocatable binary object program that can be loaded for debugging or execution by the Linking Loader. MACRO-9 prepares the object program for relocation, and the Linking Loader sets up linkages to external subroutines. Optionally, the binary program may be output either with absolute addresses (non-relocatable) or in the full binary mode.

The programmer may direct MACRO-9 processing by using a powerful set of pseudo-operation (pseudo-op) instructions. These pseudo-ops are used to set the radix for numerical interpretation by the Assembler, to reserve blocks of storage locations, to repeat object code, to handle strings of text characters in 7-bit ASCII code, or a special 6-bit code, to assemble certain coding elements if specific conditions are met, and other functions which are explained in detail in Chapter 3.

The most advanced feature of MACRO-9 is its powerful macro instruction generator. This permits easy handling of recursive sequences, changing only the arguments. Programmers can use macro instructions to create new language elements, adapting the Assembler to their specific programming applications. Macro instructions may be called up to three levels, nested to n levels, and redefined within the program. The technique of defining and calling macro instructions is discussed in Chapter 4.

An output listing, showing both the programmer's source coding and the object program produced by MACRO-9, is printed if desired. This listing includes all the symbols used by the programmer with their assigned values. If assembly errors are detected, erroneous lines are marked with specific letter error codes, which may be interpreted by referring to the error list in Chapter 5 of this manual.

Operating procedures for MACRO-9 are contained in the I/O Monitor Guide (DEC-9A-MIPA-D) for paper tape systems, and in the Keyboard Monitor Guide (DEC-9A-MKFA-D) for bulk storage systems.

---

*An abbreviated version, called MACROA, described in Appendix G, is available for 8K PDP-9 systems with DECtape.

1.1 HARDWARE REQUIREMENTS AND OPTIONS

MACRO-9 operates in PDP-9 systems with the I/O Monitor and the following minimum hardware configurations:

8K core memory

Console Teletype (KSR33 or KSR35)

Paper tape reader and paper tape punch

With the addition of bulk storage (such as 2 DECtapes, 2 magnetic tapes, a drum, or a disk) to the hardware configuration, MACRO-9 operates with the Keyboard Monitor, which allows the user flexibility in assigning I/O devices at assembly time and provides true device independence.

With the addition of bulk storage, 8K of memory, the memory protection option, and one external Teletype, MACRO-9 operates with the Background/Foreground Monitor where assembly may be accomplished as a normal BACKGROUND job.

1.2 ASSEMBLER PROCESSING

The MACRO-9 Assembler processes source programs in two passes; that is, it reads the same source code twice, outputting the object code (and producing printed listing, if requested) during the second pass. The two passes are resident in memory at the same time. PASS1 and PASS2 are almost identical in their operations, but object code is produced only during PASS2. The main function of PASS1 is to resolve locations that are to be assigned to symbols and to build up a symbol table. PASS2 uses the information computed by PASS1 (and left in memory) to produce the final output.

The standard object code produced by MACRO-9 is in a relocatable format which is acceptable to the PDP-9 Linking Loader. Relocatable programs that are assembled separately and use identical global symbols* where applicable, can be combined by the Linking Loader into an executable object program.

Some of the advantages of having programs in relocatable format are as follows.

a. Reassembly of one program, which at object time was combined with other programs, does not necessitate a reassembly of the entire system.

b. Library routines (in relocatable object code) can be requested from the system device or user library device.

c. Only global symbol definitions must be unique in a group of programs that operate together.

*Symbols which are referenced in one program and defined in another.

CHAPTER 2
ASSEMBLY LANGUAGE ELEMENTS

2.1 PROGRAM STATEMENTS

A single statement may be written on a 72-character Teletype line, in which case the carriage-return line-feed sequence characters delimit the statement. Such a statement actually begins with a line-feed character and is terminated by a carriage-return character. Since these form-control characters are not printed, they are represented as) (carriage return) and ↓ (line feed). In the examples of statements in this manual, only the carriage return is shown:

STATEMENT)

Several statements may be written on a single line, separated by semicolons:

STATEMENT;STATEMENT;STATEMENT)

In this case, the statement line begins with a line-feed character and ends with a carriage-return character, but semicolons are used as internal statement delimiters. Thus, if a statement is followed by another statement on the same line, it ends with a semicolon.

A statement may contain up to four fields that are separated by a space, spaces, or a tab character. These four fields are the label (or tag) field, the operation field, the address field, and the comments field. Because the space and tab characters are not printed, the space is represented by ␣, and the tab by →| in this manual. Tabs are normally set 10 spaces apart on most Teletype machines, and used to line up the fields in columns in the source program listing.

This is the basic statement format:

LABEL →| OPERATION →| ADDRESS →| /COMMENTS)

where each field is delimited by a tab or space, and each statement is terminated by a semicolon or carriage-return. The comments field is preceded by a tab (or space) and a slash (/).

Note that a combination of a space and a tab will be interpreted by the MACRO-9 assembler as <u>two</u> field delimiters.

Example:

TAG →| OP ␣ →| ADR)	both are
TAG ␣ →| OP →| ADR)	incorrect

These errors will not show on the listing because the space is hidden in the tab.

A MACRO-9 statement may have an entry in each of the four fields, or three, or two, or only one field. The following forms are acceptable:

TAG)

TAG →| OP)

TAG →| OP →| ADDR)

TAG →| OP →| ADDR ␣ (s) / comments ↵

TAG →| OP ␣ (s) / comments ↵

TAG →| →| ADDR ↵

TAG →| →| ADDR ␣ (s) / comments ↵

TAG →| (s) / comments ↵

→| OP ↵

→| OP →| ADDR ↵

→| OP →| ADDR →| (s) / comments ↵

→| OP →| (s) / comments ↵

→| →| ADDR ↵

→| →| ADDR →| (s) / comments ↵

/ comments ↵

→| (s) / comments ↵

Note that when a label field is not used, its delimiting tab is written, except for lines containing only comments. When the operation field is not used, its delimiting tab is written if an address field follows, except in label only and comments only statements.

A label (or tag) is a symbolic address created by the programmer to identify the statement. When a label is processed by the Assembler, it is said to be defined. A label can be defined only once. The operation code field may contain a machine mnemonic instruction code, a MACRO-9 pseudo-op code, a macro name, a number, or a symbol. The address field may contain a symbol, number, or expression which is evaluated by the assembler to form the address portion of a machine instruction. In some pseudo-operations, and in macro instructions, this field is used for other purposes, as will be explained in this manual. Comments are usually short explanatory notes which the programmer adds to a statement as an aid in analysis and debugging. Comments do not affect the object program or assembly processing. They are merely printed in the program listing. Comments must be preceded by a slash (/). The slash must be preceded by:

a. Space
b. Tab
c. Semicolon
d. First character of line

2.2 SYMBOLS

The programmer creates symbols for use in statements, to represent addresses, operation codes and numeric values. A symbol contains one to six characters from the following set:

The letters A through Z

The digits 0 through 9

Two special characters, period (.) and the percent sign (%).

The first character of a symbol must be a letter, a period, or percent sign. A period may not be used alone as a symbol. If the first character is a period, it cannot be followed immediately by a digit. The first character of a symbol must not be a digit.

The following symbols are legal:

MARK1	..1234	.A
A%	%50.99	.%
P9.3	INPUT	

The following symbols are illegal:

TAG:1	L@ B1	:and @ are illegal characters.
5ABC		First character may not be a digit.
.5A		If first character is a period, it cannot be followed by a digit.

Only the first six characters of a symbol are meaningful to the Assembler, but the programmer may use more for his own information. If he writes,

SYMBOL1

SYMBOL2

SYMBOL3

as the symbolic labels on three different statements in his program, the Assembler will recognize only SYMBOL and type error flags on the lines containing SYMBOL1, SYMBOL2 and SYMBOL3 because to the Assembler they are duplicates of SYMBOL.

2.2.1 Evaluation of Symbols

When the Assembler encounters a symbol during processing of a source language statement, it evaluates the symbol by reference to two tables: the user's Symbol Table and the Permanent Symbol Table. The user's Symbol Table contains all symbols defined by the user. The user defines symbols by using them as labels, as variables, as macro names, and by direct assignment statements. A label is defined when first used, and cannot be redefined. (When a label is defined by the user, it is given the current value of the Location Counter, as will be explained later in this chapter.)

All permanently defined system symbols, including Monitor commands and all Assembler pseudo-instructions use a period (.) as their first character. (In some cases the "." may be used as the last character of a Monitor I/O symbol). The Assembler has, in its Permanent Symbol Table, definitions of the symbols for all of the PDP-9 memory reference instructions, operate instructions, EAE instructions, and some input/output transfer instructions. (See Appendix A for a complete list of these instructions.)

PDP-9 instruction mnemonic symbols may be used in the operation field of a statement without prior definition by the user.

Example

→| LAC ␣ A)

LAC is a symbol whose appearance in the operation field of a statement causes the Assembler to treat it as an op code rather than a symbolic address. It has a value of 200000_8 which is taken from the operation code definition in the Permanent Symbol Table.

The user can use instruction menmonics or the pseudo-instruction mnemonics code as symbol labels. For example,

DZM →|DZM␣ Y)

where the label DZM is entered in the Symbol Table and given the current value of the Location Counter, and the op code DZM is given the value 140000 from the Permanent Symbol Table. The user must be careful, however, in using these dual purpose (field dependent) symbols. Symbols in the operation field are interpreted as either instruction codes or pseudo-ops, not as a symbolic label, if they are in the Permanent Symbol Table. Monitor command op-code symbols cannot be duplicated by the user. In the following example, several symbols with values have been entered in the user's Symbol Table and the Permanent Symbol Table. The sample coding shows how the Assembler uses these tables to form object program storage words.

User Symbol Table		Permanent Symbol Table	
Symbol	Value	Symbol	Value
TAG1	100	LAC	200000
TAG2	200	DAC	040000
DAC	300	JMP	600000

If the following statements are written,	the following code is generated by the Assembler.
TAG1 →| DAC →| TAG2	040200
TAG2 →| LAC →| DAC	200300
DAC →| JMP →| TAG1	600100
→| TAG1	000100

2.2.2 Variables

A variable is a symbol that is defined in the Symbol Table by using it in an address field or operation field, with the number sign (#). A variable reserves a single storage word, which may be referenced by using the symbol at other points in the program, with or without the number sign. If the variable duplicates a user defined label, the variable is flagged as an error during assembly.

Variables are assigned memory locations at the end of the program. The initial contents of variable locations are unspecified.

Examples

Location Counter	Source Statements	Generated Code
	→| .LOC ␣ 100	
100	→| LAC ␣ TA#G1	200105
101	→| DAC ␣ TAG3	040107
102	→| LAC ␣ TAG2#	200106
103	→| DAC ␣ TAG3#	040107
104	→| LAC ␣ #TAG2	200106
	→| .END	

Storage words can be set to zero as follows:

→|A→|0; →|0; →0 ↵

In this way, three words are set to zero starting at A. Storage words can also be set to zero by statements containing only labels

A; B; C; D; E ↵

When the programmer defines a macro instruction, the macro name is entered in the Symbol Table. Macros are fully described in Chapter 4.

2.2.3 Direct Assignment Statements

The programmer may define a symbol directly in the Symbol Table by means of a direct assignment statement, written in the form:

SYMBOL=n
or
SYM1=SYM2

where n is any number or expression. There should be no spaces between the symbol and the equal sign, or between the equal sign and the assigned value, or symbol. MACRO-9 enters the symbol in the

Symbol Table, along with the assigned value. Symbols entered in this way may redefined. These are legal direct assignment statements:

```
X=28; A=1; B=2↵
```

A symbol can also be assigned a symbolic value:

```
A=4
B=A
```

The symbol B is given the value 4. Direct assignment statements do not generate storage words in the object program.

In general, it is good programming practice to define symbols before using them in statements which generate storage words. The Assembler will interpret the following sequence without trouble.

```
Z=5
Y=Z
X=Y
→| LAC ␣ X ␣ /SAME AS LAC 5 ↵
```

A symbol may be defined after use. For example,

```
LAC Y↵
Y=1 ↵
```

This is called a forward reference, and is resolved properly in PASS2. When first encountered in PASS1, the LAC Y statement is incomplete because Y is not yet defined. Later in PASS1, Y is given the value 1. In PASS2, the Assembler finds that Y = 1 in the Symbol Table, and forms the complete storage word.

Since MACRO-9 operates in two passes, only one-step forward references are allowed. The following forward reference is illegal:

```
LAC Y ↵
Y=Z  ↵
Z=1  ↵
```

In the listing, during PASS1, the line which contains Y = Z will be printed as a warning.

2.2.4 Undefined Symbols

If any symbols, except global symbols, remain undefined at the end of PASS1 of assembly, they are automatically defined as the addresses of successive registers following the block reserved for variables at the end of the program. All statements that referenced the undefined symbol are flagged as undefined. One memory location is reserved for each undefined symbol with the initial contents of the reserved location being unspecified.

Location Counter	Source Statements	Generated Code	Comments
	→| .LOC ␣ 100 ↵		
100	→| LAC ␣ UNDEF1 ↵	200106	Flagged as an error
101	→| LAC ␣ TAG# ↵	200104	
102	→| LAC ␣ TAG#1 ↵	200105	
103	→| LAC ␣ UNDEF2 ↵	200107	Flagged as an error
	→| .END ↵		

2.3 NUMBERS

The initial radix (base) used in all number interpretation by the Assembler is octal (base 8). In order to allow the user to express decimal values, and then restore to octal values, two radix-setting pseudo-ops (.OCT and .DEC) are provided. These pseudo-ops, described in Chapter 3, must be coded in the operation field of a statement. If any other information is written in the same statement, the Assembler treats it as a comment. All numbers are decoded in the current radix until a new radix control pseudo-op is encountered. The programmer may change the radix at any point in a program.

Examples

Source Program	Generated Value (Octal)	Radix in Effect
→| LAC →| 100	200100	8 } initial value is
→| 25	000025	8 } assumed to be octal
→| .DEC		
→| LAC →| 100	200144	10
→| 275	000423	10
→| .OCT		
→| 76	000076	8

2.3.1 Integer Values

An integer is a string of digits, with or without a leading sign. Negative numbers are represented in two's complement form. The range of integers is as follows.

Unsigned	$0 \rightarrow 262143_{10}$	(777777_8) or $2^{18}-1$
Signed	$\pm 0 \rightarrow 131071_{10}$	(377777_8) or $\pm 2^{17}-1$

An octal integer* is a string of digits (0-7), signed or unsigned. If a non-octal digit is encountered (8 or 9) the string of digits will be assembled as if the decimal radix was in effect and it will be flagged as a possible error.

Examples

Coded Value	Generated Value (Octal)	Comment
-5	777773	two's complement
3347	003347	
3779	007303	possible error, decimal assumed

A decimal integer** is a string of digits (0-9), signed or unsigned.

Examples

Coded Value	Generated Value (Octal)	Comment
-8	777770	two's complement
+256	000400	
-136098	000000	Error, greater than $-2^{17}-1$

2.3.2 Expressions

Expressions are strings of symbols and numbers separated by arithmetic or Boolean operators. Expressions represent unsigned numeric values ranging from 0 to $2^{18}-1$. All arithmetic is performed in unsigned integer arithmetic (two's complement), modulo 2^{18}. Division by zero is regarded as division by one and results in the original dividend. Fractional remainders are ignored; this condition is not regarded as an error. The value of an expression is calculated by substituting the numeric values for each element (symbol) of the expression and performing the specified operations.

The following are the allowable operators to be used with expressions:

---

*Initiated by .OCT pseudo-op and is also the initial assumption if no radix control pseudo-op was encountered.

**Initiated by .DEC pseudo-op.

Character		Function	
Name	Symbol		
Plus	+	Addition (two's complement)	
Minus	-	Subtraction (convert to two's complement and add)	
Asterisk	*	Multiplication (unsigned)	
Slash	/	Division (unsigned)	
Ampersand	&	Logical AND	Boolean
Exclamation point	!	Inclusive OR	
Back slash	\	Exclusive OR	

Operations are performed from left to right (i.e., in the order in which they are encountered). For example, the assembly language statement A+B*C+D/E-F*G is equivalent to the following algebraic expression (((((A+B)*C)+D)/E)-F)*G.

Examples

Assume the following symbol values:

Symbol	Value (Octal)
A	000002
B	000010
C	000003
D	000005

The following expressions would be evaluated.

Expression	Evaluation (Octal)
A + B - C	000007
A/B + A * C	000006
B/A -2*A -1	000003
A & B	000000

(The remainder of A/B is lost)

Expression	Evaluation (Octal)
C + A & D	000005
B * D/A	000024
B*C/A*D	000074

2.4 ADDRESS ASSIGNMENTS

As source program statements are processed, the Assembler assigns consecutive memory locations to the storage words of the object program. This is done by reference to the Location Counter, which is initially set to zero, and incremented by one each time a storage word is formed in the object program. Some statements, such as machine instructions, cause only one storage word to be generated, incrementing the Location Counter by one. Other statements, such as those used to enter data or text, or to reserve blocks of storage words, cause the Location Counter to be incremented by the number of storage words generated.

2.4.1 Referencing the Location Counter

The programmer may directly reference the Location Counter by using the symbol, period (.), in the address field. He can write,

→| JMP ␣ .-1

which will cause the program to jump to the storage word whose address was previously assigned by the Location Counter. The Location Counter may be set to another value by using the .LOC pseudo-op, described in Chapter 3.

2.4.2 Indirect Addressing

To specify an indirect address, which may be used in memory reference instructions, the programmer writes an asterisk immediately following the operation field symbol. This sets the Defer bit (Bit 4) of the storage word.

If an asterisk suffixes either a non-memory reference instruction, or appears with a symbol in the address field, an error will result.

Two examples of legal indirect addressing follow.

→| TAD* →| A

→| LAC* →| B

The following examples are illegal.

→| LAC →| TAD*	The asterisk is not allowed in an address field.
→| CLA*)	Indirect addressing may not be specified in non-memory reference instructions.

2.4.3 Literals

Symbolic data references in the operation and address fields may be replaced with direct representation of the data enclosed in parentheses*. This inserted data is called a literal. The Assembler sets up the address link, so one less statement is needed in the source program. The following examples show how literals may be used, and their equivalent statements. The information contained within the parentheses, whether it be a number, symbol, expression, or machine instruction is assembled and assigned consecutive memory locations after the locations used by the program. The address of the generated word will appear in the statement that referenced the literal. Duplicate literals, completely defined when scanned in the source program during PASS1, are stored only once so that many uses of the same literal in a given program result in only one (1) memory location being allocated for that literal.

Usage of Literal	Equivalent Statements
→| ADD ␣ (1)	→| ADD ␣ ONE ONE →| 1
→| LAC ␣ (TAG)	→| LAC ␣ TAGAD TAGAD →| TAG
→| LAC ␣ (DAC →| TAG)	→| LAC ␣ INST INST →| DAC →| TAG
→| LAC ␣ (JMP →| .+2)	HERE →| LAC ␣ INST INST →| JMP ␣ HERE+2

*The opening parenthesis [( ] is mandatory while the closing parenthesis [ )] is optional.

The following sample program illustrates how the Assembler handles literals.

Location Counter	Source Statement	Generated Code
	→| .LOC ␣ 100	
100	TAG 1 →| LAC ␣ (100)	200110
101	→| DAC ␣ 100	040100
102	→| LAC ␣ (JMP ␣ .+5)	200111
103	→| LAC ␣ (TAG1)	200110
104	→| LAC ␣ (JMP ␣ TAG1)	200112
105	→| LAC ␣ (JMP ␣ TAG2)	200113
	TAG2=TAG1	
106	→| LAC ␣ (JMP)	200114
107	DAC →| LAC ␣ (DAC →| DAC)	200115
	→| .END	
	Generated Literals	
110		000100
111		600107
112		600100
113		600100
114		600000
115		040107

2.5 STATEMENT FIELDS

The following paragraphs provide a detailed explanation of statement fields, including how symbols and numbers may be used in each field.

2.5.1 Label Field

If the user wishes to assign a symbolic label to a statement, to facilitate references to the storage word generated by the Assembler, he may do so by beginning the source statement with any desired symbol. The symbol must not duplicate a system or user defined macro symbol and must be terminated by a space or tab, or a statement terminating semicolon, or carriage-return/line-feed sequence.

Examples

TAG ␣ any value

TAG ␣ (s) any value

TAG →| ␣ (s) any value

TAG;

TAG ↵

TAG ␣ (s) (no more data on line)

These examples are equivalent to coding

TAG →| 0 ↵

in that a word of all 0s is output with the symbol TAG associated with it.

Symbols used as labels are defined in the Symbol Table with a numerical value equal to the present value of the Location Counter. A label is defined only once. If it was previously defined by the user, the current definition of the symbol will be flagged in error as a multiple definition. All references to a multiply defined symbol will be converted to the first value encountered by the Assembler.

Example

Location Counter	Statement	Storage Word Generated	Notes
100	A →| LAC →| B	200103	
101	A →| LAC →| C	200104	error, multiple definition
102	→| LAC →| A	200100	first value of A referenced
103	B →| 0	000000	
104	C →| 0	000000	

Anything more than a single symbol to the left of the label-field delimiter is an error; it will be flagged and ignored. The following statements are illegal.

TAG+1 →| LAS ↵

LOC*2 →| RAR ↵

Redefinition of certain symbols can be accomplished by using direct assignments; that is, the value of a symbol can be modified. If an Assembler permanent symbol or user symbol (which was defined by a direct assignment) is redefined, the value of the symbol can be changed without causing an error message. If a user symbol, which was first defined as a label, is redefined by either a direct assignment or by using it again in the label field, it will cause an error. Variables also cannot be redefined by a direct assignment.

Examples

Coding	Generated Value (Octal)	Comments
A=3		sets current value of A to 3
→| LAC →| A	200003	
→| DAC →| A	040003	
A=4		redefines value of A to 4
→| LAC →| A	200004	
B →| DAC →| A*	040004	
B=A		illegal usage; a label cannot be redefined
→| DAC →| B	040105	
PSF=700201		to redefine possibly incorrect permanent symbol definition.

*Assume that this instruction will occupy location 105.

2.5.2 Operation Field

Whether or not a symbol label is associated with the statement, the operation field must be delimited on its left by a space(s) or tab. If it is not delimited on its left, it will be interpreted as the label field. The operation field may contain any symbol, number, or expression which will be evaluated as an 18-bit quantity using unsigned arithmetic modulo 2^{18}. In the operation field, machine instruction op codes and pseudo-op mnemonic symbols take precedence over identically named user defined symbols. The operation field must be terminated by one of the following characters:

(1) →\| or ␣ (s) (field delimiters)
(2) ↵ or ; (statement delimiters)

Examples

TAG →\| ISZ
→\| .+3 ␣ (s)
␣ (s)CMA!CML ↵
→\| TAG/5+TAG2; →\| TAG3 ↵

The asterisk (*) character appended to a memory reference instruction symbol, in the operation field, causes the setting of the Defer bit (Bit 4) of the instruction word; that is, the reference will

be an indirect reference. If the asterisk (*) is appended on either a non-memory reference instruction or appended to any symbol in the address field, it will cause an error condition.

Examples

legal	illegal
→| TAD* →| A	→| LAC →| TAD*
→| LAC* →| B	→| CLA*

However, the asterisk (*) may be used anywhere as a multiplication operator.

Examples

legal	illegal
→| LAC →| TAG*5	→| LAC →| TAG*4+TAD*
→| TAG*TAG1	→| A*

2.5.3 Address Field

The address field, if used in a statement, must be separated from the operation field by a tab, or space(s). The address field may contain any symbol, number, or expression which will be evaluated as an 18-bit quantity using unsigned arithmetic, modulo 2^{18}. If op code or pseudo-op code symbols are used in the address field, they must be user defined, otherwise they will be undefined by the Assembler and cause an error message. The address field must be terminated by one of the following characters:

(1) →\| or ␣ (s) (field delimiters)
(2) ↵ or ; (statement delimiters)

Examples

TAG2 →\| DAC →\| .+3
→\| →\| TAG2/5+3 ␣ (s)

In the last example, the rest of the line will be automatically treated as a comment and ignored by the Assembler.

The address field may also be terminated by a semicolon, or a carriage-return/line-feed sequence.

Examples

→\| JMP →\| BEGIN ↵
→\| TAD →\| A; →\| DAC →\| B ↵

In the last example, a tab or space(s) is required after the semicolon in order to have the Assembler interpret DAC as being the operation field rather than the label field.

When the address field is a relocatable expression, an error condition may exist. The size of a relocatable program is restricted to 8K (8192_{10} words) and cannot be loaded across memory banks. Therefore, any relocatable address field whose value exceeds 17777_8 is meaningless and will be flagged in error.

When the address field is an absolute expression, an error condition will exist if the extended memory bits (3 & 4) of the address do not match the extended memory bits of the bank currently being assembled into and the extended memory bits of the address are not 0.

Examples

Location (octal)	Instruction	Comments
20000	→| LAC ␣ 20100	will not cause error messages
20001	→| DAC ␣ 101	
20002	→| JMS ␣ 250	
20005	→| ISZ ␣ 40146	will cause an error message

The Linking Loader will not relocate any absolute addresses; thus, absolute addresses within a relocatable program are relative to that bank in memory in which the program is loaded.

Example

Assume that the following source line is part of a relocatable program that was loaded into bank 1 ($20000_8 \rightarrow 37777_8$)

Source Statement	Effective Address
→| LAC ␣ 300 ↵	20300

An exception to the above rule is the auto-index registers, which occupy locations 10_8 - 17_8 in memory bank 0. The hardware will always ensure that indirect references to 10_8 - 17_8 in any bank will always access 10_8 - 17_8 of bank 0.

2.5.4 Comments Field

Comments may appear anywhere in a statement. They must begin with a slash (/) that is immediately preceded by a

a. ␣ (s)	space(s)
b. →|	tab
c. ↵	carriage return/line feed (end of previous line)
d. ;	semicolon

Comments are terminated only by a carriage-return/line-feed sequence or when 72_{10} characters have been encountered.

Examples

␣ (s)/THIS IS A COMMENT (rest of line is blank)
TAG1 →| LAC ␣ /after the ; is still a comment
/THIS IS A COMMENT
→| RTR ␣ /COMMENT ↵
→| RTR; →| RTR;/THIS IS A COMMENT

Observe that;→| A/COMMENT ↵ is not a comment, but rather an operation field expression. A line that is completely blank; that is, between two sets of ↵↓ (s) is treated as a comment by the Assembler.

Example

␣ (72 blanks)

A statement is terminated as follows:

↵↓ or ; or rest of line is completely blank.

Examples

→| LAC ↵
→| DAC (the rest of the line is blank)
→| TAG+3
→| RTR; →| RTR; →| RTR ↵

In the last example, the statement-terminating character, which is a semicolon (;) enables one source line to represent more than one word of object code. A tab or space is required after the semicolons in order to have the second and third RTR's interpreted as being in the operation field and not in the label field.

2.6 STATEMENT EVALUATION

When MACRO-9 evaluates a statement, it checks for symbols or numbers in each of the three evaluated fields: label, operation, and address. (Comment fields are not evaluated.)

2.6.1 Numbers

Numbers are not field dependent. When the Assembler encounters a number (or expression) in the operation or address fields (numbers are illegal in the label field), it uses those values to form the storage word. The following statements are equivalent:

→| 200000␣10↵
→| 10+LAC ↵
→| LAC␣10 ↵

All three statements cause the Assembler to generate a storage word containing 200010. A statement may consist of a number or expression which generates a single 18-bit storage word; for example:

→| 23;␣ 45;␣357;␣62

This group of four statements generates four words interpreted under the current radix. Zero words are generated by statements containing only labels. For example,

A; B; C; D; E↵

generates five words set to zero, which may be referenced by the labels defined.

2.6.2 Word Evaluation

When the Assembler encounters a symbol in a statement field, it determines the value of the symbol by reference to the user's symbol table and the permanent symbol table, according to the priority list shown below. The value of a storage word is computed by combining the 18-bit operation field quantity with the 18-bit address field quantity in the following manner.

$$\left[\underset{0-17}{(\text{OPERATION FIELD}} + \underset{0-17}{\text{ADDRESS FIELD})} \,\&\, 017777_8\right] + \left[\underset{0-17}{\text{OPERATION FIELD}} \,\&\, 760000_8\right] = \text{Value of Word}$$

The Assembler performs a validity check to verify that meaningful results were produced. As long as

$$\left[\underset{0-17}{\text{ADDRESS FIELD}} \,\&\, 76000_8\right] = 760000_8 \text{ or } 000000$$

then meaningful results were produced.

Examples

→| TAD →| A	Where A can range from 0→ 77777_8 and combined with TAD (340000_8) results in 340000_8 →| 357777_8.
→| LAW →| -1	Where -1 (777777_8) is combined with LAW (760000_8) and results in 777777_8.

If the ADDRESS FIELD & 760000_8 does not equal 760000_8 or 000000, erroneous results may have been produced and it will be flagged by the Assembler. This validity check is performed only if an operations field and an address field is present.

If numbers are found in the operation and address fields, they are combined in the same manner as defined symbols. For example,

→|2→|5 →| /GENERATES 000007

The value of a symbol depends on whether it is in the label field, the operation field, or the address field. The Assembler attempts to evaluate each symbol by running down a priority list, depending on the field, as shown below.

Label Field	Operation Field	Address Field
Current Value of Location Counter	1. Pseudo-op 2. User macro in user symbol table 3. System macro table 4. Direct assignment in user symbol table 5. Permanent symbol table 6. User symbol table 7. Undefined	1. User symbol table, (including) direct assignments) 2. Undefined

This means that if a symbol is used in the address fields, it must be defined in the user's symbol table before the word is formed during PASS1; otherwise, it is undefined.

In the operation field, pseudo-ops take precedence and may not be redefined. Direct assignments allow the user to redefine machine op codes, as shown in the example below.

Example:

```
DAC = DPOSIT
```

System macros may be redefined as a user macro name, but may not be redefined as a user symbol by direct assignment or by use as a statement label.

The user may use machine instruction codes and MACRO-9 pseudo-op codes in the label field and refer to them later in the address field.

CHAPTER 3
PSEUDO OPERATIONS

In the discussion of symbols in the previous chapter, it was mentioned that the assembler has, in its permanent symbol table, definitions of the symbols for all the PDP-9 memory reference instructions, operate instructions, EAE instructions, and many IOT instructions which may be used in the operation field without prior definition by the user. Also contained in the permanent symbol table are a class of symbols called pseudo-operations (pseudo-ops) which, instead of generating instructions or data, direct the assembler on how to proceed with the assembly.

By convention, the first character of every pseudo-op symbol is a period (.). This convention is used in an attempt to prevent the programmer from inadvertently using, in the operation field, a pseudo-instruction symbol as one of his own. Pseudo-ops may be used only in the operation field.

3.1 PROGRAM IDENTIFICATION (.TITLE)

The program name may be written in a .TITLE statement as shown below. The assembler treats this statement as a comment.

→| .TITLE␣ NAME OF PROGRAM

3.2 OBJECT PROGRAM OUTPUT (.ABS and .FULL)

The normal object code produced by MACRO-9 is relocatable binary which is loaded at run time by the Linking Loader. In addition to relocatable output, the user may specify two other types of output code to be generated by the Assembler.

The following rules apply to the usage of these optional output codes:

a. The pseudo-op specifying the type of output must appear before any coding appears (excluding title and comments), otherwise it will be ignored.

b. Once an optional output code pseudo-op is specified, the user is not allowed to switch output modes. Any subsequent requests are flagged and ignored.

c. Any options provided for in the address field of the pseudo-ops are useful only if the output device is paper tape.

3.2.1 .ABS Pseudo-op

Label Field	Operation Field	Address Field
Not Used	.ABS	NLD or ␣

The .ABS pseudo-op causes absolute, checksummed binary code to be output (no values are relocatable). If no value is specified in the address field, the Assembler will precede the output with the Absolute Binary Loader which will load the punched output at object time. The loader is loaded, via hardware readin, into location 17720 of any memory bank. (This loader loads only paper tape.) If the address field contains NLD, no loader will precede the output.

NOTE

.ABS output can be written on file-oriented devices. The assembler assumes .ABS NLD for all .ABS output to file-oriented devices and appends an extension of .ABS to the filename. This file can be punched with PIP, using Dump mode. (There will be no absolute loader at the beginning of the tape.)

A description of the absolute output format follows.

Block Heading - (three binary words)

WORD 1 - Starting address to load the block body which follows.

WORD 2 - Number of words in the block body (two's complement).

WORD 3 - Checksum of block body (two's complement). Checksum includes Word 1 and Word 2 of the block heading.

Block Body - (n binary words)

The block body contains the binary data to be loaded under block heading control.

Starting Block - (two binary words)

WORD 1 - Location to start execution of program. It is distinguished from the block heading by having bit 0 set to 1 (negative).

WORD 2 - Dummy word.

If the user requests the Absolute Loader, and the value of the expression of the .END statement is equal to 0, the provided loader halts before transferring control to the object program, thereby allowing manual intervention by the user.

A listing of the Absolute Binary Loader is given in Appendix F.

3.2.2 .FULL Pseudo-op

Label Field	Operation Field	Address Field
Not Used	.FULL	Not Used

(Only useful if output is paper tape)

The .FULL pseudo-op causes a full mode output to be produced. The program is assembled as uncheck-summed absolute code and each physical record of output contains nothing other than 18-bit binary storage words generated by the Assembler. The Assembler will cause the address of the .END statement to contain a punch in channel 7, thereby allowing the output to be loaded via hardware Readin Mode. If no address is specified in the .END statement, a halt (rather than a jump) will be outputted as the last word.

The following specific restrictions apply to programs assembled in .FULL mode output.

.LOC	Should be used only at the beginning of the program.
.BLOCK	May be used only if no literals appear in the program, and must immediately precede .END.

Variables and undefined symbols may be used if no literals appear in the program.

Literals may be used only if the program has no variables and undefined symbols.

3.3 SETTING THE LOCATION COUNTER (.LOC)

Label Field	Operation Field	Address Field
Not Used	.LOC	Pre-defined symbolic expression, or number

The .LOC pseudo-op sets or resets the Location Counter to the value of the expression contained in the address field. The symbolic elements of the expression must have been defined previously; otherwise, phase errors might occur in PASS2. The .LOC pseudo-op may be used anywhere and as many times as required.

Examples

Location Counter	Instruction
100	→| .LOC␣ 100
100	→| LAC␣ TAG1
101	→| DAC␣ TAG2
102	→| .LOC␣ .
102	A →| LAC␣ B
103	→| DAC␣ C
107	→| .LOC␣ A+5

Location Counter	Instruction
107	→| LAC ␣ C
110	→| DAC ␣ D
111	→| LAC ␣ E
112	→| DAC ␣ F

3.4 RADIX CONTROL (.OCT and .DEC)

The initial radix (base) used in all number interpretation by the Assembler is octal (base 8). In order to allow the user to express decimal values, and then restore to octal values, two radix setting pseudo-ops are provided.

Pseudo-op Code	Meaning
.OCT	Interpret all succeeding numerical values in base 8 (octal)
.DEC	Interpret all succeeding numerical values in base 10 (decimal)

These pseudo-instructions must be coded in the operation field of a statement. All numbers are decoded in the current radix until a new radix control pseudo-instruction is encountered. The programmer may change the radix at any point in a program.

Source Program	Generated Value (Octal)	Radix in Effect
→| LAC 100	200100	8 } initial value is
→| 25	000025	8 } assumed to be octal
→| DEC		
→| LAC 100	200144	10
→| 275	000423	10
→| .OCT		
→| 76	000076	8
→| 85	000000	error

3.5 RESERVING BLOCKS OF STORAGE (.BLOCK)

.BLOCK reserves a block of memory equal to the value of the expression contained in the address field. If the address field contains a numerical value, it will be evaluated according to the radix in effect. The symbolic elements of the expression must have been defined previously; otherwise, phase errors might occur in PASS2. The expression is evaluated modulo 2^{15} (77777_8). The user may reference the first location in the block of reserved memory by defining a symbol in the label field. The initial contents of the reserved locations are unspecified.

Label Field	Operation Field	Address Field
User Symbol	. BLOCK	Predefined Expression

Examples

BUFF →| . BLOCK ␣12↵

→| . BLOCK␣ A+B+65↵

3.6 PROGRAM TERMINATION (.END)

One pseudo-op must be included in every MACRO-9 source program. This is the .END statement, which must be the last statement in the main program. This statement marks the physical end of the source program, and also contains the location of the first instruction in the object program to be executed at run-time.

The .END statement is written in the general form:

→|. END ␣ START↵

Where START may be a symbol, number, or expression whose value is the address of the first program instruction to be executed. In relocatable programs, to be loaded by the Linking Loader, only the main program requires a starting address; all other subprogram starting addresses will be ignored.

A starting address must appear in absolute or self-loading programs; otherwise, the program will halt after being loaded and the user must manually start his program.

These are legal .END statements

→|. END ␣ BEGIN+5↵

→|. END ␣ 200↵

3.7 PROGRAM SEGMENTS (.EOT)

If the input source program is physically segmented, each segment except the last must terminate with an .EOT (end-of-tape) statement. The last segment must terminate with an .END statement. For example, if the input source program is prepared on three different tapes, the first two are terminated by .EOT statements, and the last by an .END statement. The .EOT statement is written without label and address fields, as follows.

⇥ .EOT ↵

3.8 TEXT HANDLING (.ASCII and .SIXBT)

The two text handling pseudo-ops enable the user to represent the 7-bit ASCII, or 6-bit trimmed ASCII character sets. The Assembler converts the desired character set to its appropriate numerical equivalents. (See Appendix A.)

Label Field	Operation Field	Address Field
SYMBOL	{ .ASCII .SIXBT }	Delimiter - character string - delimiter - ⟨ expression ⟩ .....

Only the 64 printing characters (including space) may be used in the text pseudo-instructions. See non-printing characters, Section 2.4.5. The numerical values generated by the text pseudo-ops are left-justified in the storage word(s) they occupy with the unused portion (bits) of a word zero filled.

3.8.1 .ASCII Pseudo-op

.ASCII denotes 7-bit ASCII characters. (It is the character set that is the input to and output from Monitor.) The characters are packed five per two words of memory with the rightmost bit of every second word set to zero. An even number of words will always be outputted.

Basic Form

First Word			Second Word		
0 6 7 13 14 17			0 2 3 9 10 16 17		
1st Char.	2nd Char.	3rd Char.	4th Char.	5th Char.	0

3.8.2 .SIXBT Pseudo-op

.SIXBT denotes 6-bit trimmed ASCII characters, which are formed by truncating the leftmost bit of the corresponding 7-bit character. Characters are packed three per storage word.

Basic Form

0 5	6 11	12 17
1st Char.	2nd Char.	3rd Char.

3.8.3 Text Statement Format

The statement format is the same for both of the text pseudo-ops. The format is as follows.

MYTAG →| { .ASCII / .SIXBT } →| delimiter | character string | delimiter | <expression>.....

3.8.4 Text Delimiter

Spaces or tabs prior to the first text delimiter or angle bracket (<) will be ignored; afterwards, if they are not enclosed by delimiters or angle brackets, they will terminate the pseudo-instruction. Also, ↵ will terminate the pseudo-instruction.

Any printing character may be used as the text delimiter, except those listed below.

a. < - as it is used to indicate the start of an expression.

b. ↵ - as it terminates the pseudo-instruction.

(The apostrophe (') is the recommended text delimiting character.) The text delimiter must be present on both the left-hand and the right-hand sides of the text string, otherwise the user may get more characters than desired; however, ↵ may be used to terminate the pseudo-instruction.

3.8.5 Non-Printing Characters

The octal codes for non-printing characters may be entered in .ASCII statements by enclosing them in angle bracket delimiters. In the following statement, five characters are stored in two storage words.

→|. ASCII 'AB'<015>'CD'↵

Octal numbers enclosed in angle brackets will be truncated to 7 bits (.ASCII) or 6 bits (.SIXBT).

Examples

Source Line	Recognized Text	Comments
TAG →|. ASCII ␣ 'ABC'	ABC	
→|. SIXBT ␣ 'ABC'	ABC	
→|. SIXBT ␣ 'ABC'#'/#	ABC'/	The # is used as a delimiter in order that (') may be interpreted as text.
→|. ASCII ␣ 'ABCD'EFGE	ABCDFG	
→|. ASCII ␣ 'AB'⟨11⟩	AB →|	⟨11⟩ used to represent tab.
→|. ASCII ␣ 'AB⟨11⟩	AB⟨11⟩	There is no delimiter after B, therefore, (⟨11⟩) is treated as text.
→|. ASCII ␣ ⟨15⟩⟨012⟩'ABC'	↓↓ ABC	
→|. ASCII ␣ ⟨15⟩⟨12⟩ABC ␣␣(s)	↓↓ BC ␣ (s)	A ↓ is interpreted as the text delimiter. Also, since ↓ was not used to terminate the text, the ␣␣ (s) are interpreted as text characters.

The following example shows the binary word format which MACRO-9 generates for a given line of text.

Example:

→\|. ASCII→\|'ABC'⟨015⟩⟨12⟩'DEF

Generated Coding

Word Number	Octal	Binary			
Word 1	406050	1000001	1000010	1000	
Word 2	306424	011	0001101	0001010	0
Word 3	422130	1000100	1000101	1000	
Word 4	600000	110	0000000	0000000	0

3.9 LOADER CONTROL (.GLOBL)

Label Field	Operation Field	Address Field
Not Used	. GLOBL	A, B, C, D, E. . . .

The standard output of the Assembler is a relocatable object program. The Linking Loader joins relocatable programs by supplying definitions for global symbols which are referenced in one program and defined in another. The pseudo-op .GLOBL, followed by a list of symbols, is used to define to the Assembler those global symbols which are either.

a. internal globals - defined in the current program and referenced by other programs

b. external symbols - referenced in the current program and defined in another program

The loader uses this information to load and then link the relocatable programs to each other.

All references to external symbols should be indirect references as memory banks may have to be crossed.

Examples

```
→|. GLOBL→|A,B,C
A →|LAC→|100          /A is an internal global
  →|JMS*→|B           /These two instructions reference
  →|JMS*→|C           /External symbols indirectly
```

The .GLOBL statement may appear anywhere within the program.

3.10 REQUESTING I/O DEVICES (.IODEV)

The .IODEV pseudo-op appears anywhere in the program and is used to cause the Assembler to output code for the Linking Loader which specifies the slots in the Monitor's Device Assignment Table (DAT) whose associated device handlers are required by the program (see Monitors manual, DEC-9A-MADO-D).

Label Field	Operation Field	Address Field
Not Used	.IODEV	1,2,3...

3.11 DEFINING A SYMBOLIC ADDRESS (.DSA)

.DSA (define symbol address) is used in the operation field when it is desired to create a word composed of just an address field. It is especially useful when a user symbol is also an instruction or pseudo-op symbol.

Label Field	Operation Field	Address Field
User Symbol	.DSA	Any Expression

Examples

```
JMP →|LAC →|TAG
    →|.DSA→| JMP      Equivalent methods of defining the user symbol JMP
    →|     →| JMP      to be in the address field.
```

3.12 REPEATING OBJECT CODING (.REPT)

Label Field	Operation Field	Address Field
Not Used	.REPT	Count, { Increment or ␣

The .REPT pseudo causes the object code of the next sequential object code generating instruction to be repeated Count times. Optionally, the object code may be incremented for each time it is repeated by specifying an Increment. The Count and Increment are numerical values (signed or unsigned) which will be evaluated according to the radix in effect. The repeated instruction may contain a label, which will be associated with the first statement generated.

Examples

Source Code	Generated Object Code
→| . REPT ␣ 5	
→| 0	000000
	000000
	000000
	000000
	000000
→| .REPT ␣ 4,1	
→| 1	000001
	000002
	000003
	000004
→| .REPT ␣ 3, -1	
→| 5	000005
	000004
	000003

Source Code	Generated Object Code
TAG=50	
→| .REPT ␣ 4,1	
→| JMP ␣ TAG	600050
	600051
	600052
	600053

NOTE

If the statement to be repeated generates more than one location of code, the .REPT will repeat only the last location. For example,

→|.REPT ␣3

→.ASCII ␣'A'

will generate the following:

```
404000  5/7  A
000000
000000  last word is
000000  repeated
```

3.13 CONDITIONAL ASSEMBLY (.IF xxx and .ENDC)

It is often useful to assemble some parts of the source program on an optional basis. This is done in MACRO-9 by means of conditional assembly statements, of the form:

→| .IF...→| expression

The pseudo-op may be any of the eight conditional pseudo-ops shown below, and the address field may contain any number, symbol, or expression. If there is a symbol, or an expression containing symbolic elements, such a symbol must have been previously defined in the source program.

If the condition is satisfied, that part of the source program starting with the statement immediately following the conditional statement and up to but not including an .ENDC (end conditional) pseudo-op, is assembled. If the condition is not satisfied, this coding is not assembled.

The eight conditional pseudo-ops (sometimes called IF statements) and their meanings are shown below.

	Assemble IF x is:
→| .IFPNZ ␣ x	Positive and non-zero
→| .IFNEG ␣ x	Negative
→| .IFZER ␣ x	Zero
→| .IFPOZ ␣ x	Positive or zero
→| .IFNOZ ␣ x	Negative or zero
→| .IFNZR ␣ x	Not zero
→| .IFDEF ␣ x	A defined symbol
→| .IFUND ␣ x	An undefined symbol

In the following sequence, the pseudo-op .IFZER is satisfied, and the source program coding between .IFZER and .ENDC is assembled.

```
SUBTOT=48
TOTALL=48
→| .IFZER →| SUBTOT-TOTALL
→| LAC ␣ A
→| DAC ␣ B
→| .ENDC
```

Conditional statements may be nested. For each IF statement there must be a terminating .ENDC statement. If the outermost IF statement is not satisfied, the entire group is not assembled. If the first IF is satisfied, the following coding is assembled. If another IF is encountered, however, its condition is tested, and the following coding is assembled only if the second IF statement is satisfied. Logically, nested IF statements are like AND circuits. If the first, second and third conditions, are satisfied, then the coding that follows the third nested IF statement is assembled.

Example

```
→| .IFPOS ␣ X          conditional 1 initiator
→| LAC →| TAG
→| .IFNZR ␣ Y          conditional 2 initiator
→| DAC →| TAG1
→| .ENDC               conditional 2 terminator
→| .IFDEF ␣ Z          conditional 3 initiator
→| DAC →| TAG2
→| .ENDC               conditional 3 terminator
→| .ENDC               conditional 1 terminator
```

Conditional statements can be used in a variety of ways. One of the most useful is in terminating recursive macro calls (to be described in the next chapter). In general, a counter is changed each time through the loop, or recursive call, until the condition is not satisfied. This process concludes assembly of the loop or recursive call.

3.14 LISTING CONTROL (.EJECT)

The following Assembler listing controls are effective only when a listing is requested by Assembler control keyboard request.

Label Field	Operation Field	Address Field
Not Used	.EJECT	Not Used

When .EJECT is encountered anywhere in the source program, it causes the listing device that is being used to skip to head-of-form.

3.15 PROGRAM SIZE (.SIZE)

Label Field	Operation Field	Address Field
User Symbol	.SIZE	Not Used

When the Assembler encounters .SIZE, it outputs, at that point, the address of the last location plus one occupied by the object program. This is normally the length of the object program (in octal).

3.16 DEFINING MACROS (.DEFIN, .ETC, and .ENDM)

The .DEFIN pseudo-op is used to define macros (described in Chapter 4). The address field in the .DEFIN statement contains the macro name, followed by a list of dummy arguments. If the list of dummy arguments will not fit on the same line as the .DEFIN pseudo-op, it may be continued by means of the .ETC pseudo-op in the operation field and additional arguments in the address field of the next line. The coding that is to constitute the body of the macro follows the .DEFIN statement. The body of the macro definition is terminated by an .ENDM pseudo-op in the operation field. (See Chapter 4 for more details on the use of macros.)

CHAPTER 4
MACROS

When a program is being written, it often happens that certain coding sequences are repeated several times with only the arguments changed. It would be convenient if the entire repeated sequence could be generated by a single statement. To accomplish this, it is first necessary to define the coding sequence with dummy arguments as a macro instruction and then use a single statement referring to the macro name, along with a list of real arguments which will replace the dummy arguments and generate the desired sequence.

Consider the following coding sequence.

```
→| LAC →| A
→| TAD →| B
→| DAC →| C
    ⋮
→| LAC →| D
→| TAD →| E
→| DAC →| F
```

The sequence

```
→| LAC →| x
→| TAD →| y
→| DAC →| z
```

is the model upon which the repeated sequence is based. The characters x, y, and z are called dummy arguments and are identified as such by being listed immediately after the macro name when the macro instruction is defined.

4.1 DEFINING A MACRO

Macros must be defined before they are used. The process of defining a macro is as follows.

```
                                  (Macro Name)   (Dummy Arguments)
(Definition Line)     →| .DEFIN →| MACNME,ARG1,ARG2,ARG3      /comment
                      →| LAC    →| ARG1
(Body)                →| TAD    →| ARG2
                      →| DAC    →| ARG3
(Terminating Line)    →| .ENDM
```

The pseudo-op .DFIN in the operation field defines the symbol following it as the name of the macro. Next, follow the dummy arguments, as required, separated by commas and terminated by any of the following symbols.

a. space (␣)

b. tab (→|)

c. carriage return (↵)

The macro name, as well as the dummy arguments must be legal MACRO-9 symbols. Any previous definition of a dummy argument is ignored while in a macro definition. Comments after the dummy argument list in a definition are legal.

If the list of dummy arguments cannot fit on a single line (that is, if the .DEFIN statement requires more than 72_{10} characters) it may be continued on the succeeding line or lines by the usage of the .ETC pseudo-op, as shown below.

```
→|.DEFIN→| MACNME,ARG1, ARG2, ARG3   /comment
→|.ETC  →| ARG4, ARG5    /argument continuation
    ⋮
→|.DEFIN →| MACNME
→|.ETC  →| ARG1
→|.ETC  →| ARG2
→|.ETC  →| ARG4
→|.ETC  →| ARG5
```

4.2 MACRO BODY

The body of the macro definition follows the .DEFIN statement. Appearances of dummy arguments are marked, and the character string of the body is stored 5 characters per 2 words in the macro definition table, until the macro terminating pseudo-op .ENDM is encountered. Comments within the macro definition are not stored.

Dummy arguments may appear in the definition lines only as symbols or elements of an expression. They may appear in the label field, operation field, or address field. Dummy arguments may appear within a literal or defined as variables. They will not be recognized if they appear within a comment.

The following restrictions apply to the usage of the ·DEFIN, ·ETC and ·ENDM pseudo-ops:

a. If they appear in other than the operation field within the body of a macro definition, they will cause erroneous results.

b. If ·ENDM or ·ETC appears outside the range of a macro definition, they will be flagged as undefined.

If .ASCII or .SIXBT is used in the body of a macro, a slash (/) or number (#) must not appear as part of the text string or as a delimiter (use <57> to represent a slash and <43> to represent a number sign). Also a dummy argument name should not inadvertently be used as part of the text string.

Definition	Comments
→I . DEFIN →I MAC, A, B, C, D, E, F	
→I LAC→I A#	
→I SPA	
→I JMP→I B	
→I ISZ →ITMP →I /E	E is not recognized as an argument
→I LAC→I (C	
→I DAC→I D + 1	
→I F	
→I . ASCII →I E	
B=.	
→I . ENDM	

4.3 MACRO CALLS

A macro call consists of the macro name, which must be in the operation field, followed, if desired, by a list of real arguments separated by commas and terminated by one of the characters listed below.

a. space (␣)
b. tab (→I)
c. carriage return (↵)

If the real arguments cannot fit on one line of coding, they may be continued on succeeding lines by terminating the current line with a dollar sign ($). When they are continued on succeeding lines they must start in the tag field.

Example:

```
        →IMAC →IREAL1,REAL2,REAL3,$
REAL4,REAL5
```

If there are n dummy arguments in the macro definition, all real arguments in the macro call beyond the nth dummy argument will be ignored. A macro call may have a label associated with it, which will be assigned to the current value of the location counter.

Example:

```
(Definition)    →| . DEFIN →| UPDATE, LOC, AMOUNT
                →| LAC →| LOC
                →| TAD →| AMOUNT
                →| DAC →| LOC
                →| . ENDM
(Call)    TAG →| UPDATE →| CNTR, (5      /TAG ENTERED INTO SYMBOL TABLE
                                          /WITH CURRENT VALUE OF LOCATION COUNTER
(Expansion) TAG →| LAC →| CNTR
                →| TAD →| (5
                →| DAC →| CNTR
```

The prevailing radix will be saved prior to expansion, and restored after expansion takes place. Default assumption will be octal for the macro call. It is not necessary for the macro definition to have any dummy arguments associated with it.

Example

```
                →| .DEFIN␣  TWOS
                →| CMA
                →| TAD␣  (1
                →| .ENDM
(Call)          →| TWOS
(Expansion)     →| CMA
                →| TAD →| (1
```

4.3.1 Argument Delimiters

It was stated that the list of arguments is terminated by any of the following symbols.

a. comma (,)

b. space (␣)

c. tab (→|)

d. carriage return (↵)

These characters may be used within real arguments only by enclosing them in angle brackets. Angle brackets will not be recognized if they appear within a comment.

Example

```
(Definition)    →|.DEFIN ␣ MAC, A, B, C
                →|LAC ␣  A
                →|TAD ␣  B
                →|DAC␣  C
                →|.ENDM
```

```
(Call)        →|MAC ␣ TAG1,<TAG2   /comment
              →|TAD ␣ (1)>, TAG3
(Expansion)   →|LAC ␣ TAG1
              →|TAD ␣ TAG2
              →|TAD ␣ (1)
              →|DAC ␣ TAG3
```

All characters within a matching pair of angle brackets are considered to be one argument, and the entire argument, with the delimiters (<>) removed, will be substituted for the dummy argument in the original definition.

MACRO-9 recognizes the end of an argument only on seeing a terminating character not enclosed within angle brackets.

If brackets appear within brackets, only the outermost pair is deleted. If angle brackets are required within a real argument, they must be enclosed by argument delimiter angle brackets.

Example

```
(Definition)  →| .DEFIN →|ERRMSG, TEXT
              →| JMS →|PRINT
              →| .ASCII →|TEXT
              →| .ENDM
(Call)        →| ERRMSG →|</ERROR  IN  LINE/ <15>>
(Expansion    →| JMS →|PRINT
              →| .ASCII →|/ERROR  IN  LINE/ <15>
```

4.3.2 Created Symbols

Often, it is desirable to attach a symbolic tag to a line of code within a macro definition. As this tag is defined each time the macro is called, a different symbol must be supplied at each call to avoid multiply defined tags.

This symbol can be explicitly supplied by the user or the user can implicitly request MACRO-9 to replace the dummy argument with a created symbol which will be unique for each call of the macro. For example,

```
→|.DEFIN →|MAC, A, ?B
```

The question mark (?) prefixed to the dummy argument B indicates that it will be supplied from a created symbol if not explicitly supplied by the user when the macro is called for.

The created symbols are of the form ..0000→ ..9999. As they are required they are entered into the symbol table like any other symbol.

Unsupplied real arguments corresponding to dummy arguments not preceded by a question mark are substituted in as empty strings; and supplied real arguments corresponding to dummy arguments preceded by a question mark suppress the generation of a corresponding created symbol.

Example

```
(Definition)        →| . DEFIN →|MAC, A, B, ?C, ?D, ?E
                    →| LAC →| A
                    →| SZA
                    →| JMP →| D
                    →| LAC →| B
                    →| DAC →| C#
                    →| DAC →| E
                    D=.
                    →| . ENDM
(Call)              →| MAC →| X#,,,,MYTAG
(Expansion)         →| LAC →|X#
                    →| SZA
                    →| JMP →|. . 0000
                    →| LAC
                    →| DAC →|. . 0001
                    →| DAC →| MYTAG
                  . . 0000=.
```

If one of the elements in a real argument string is not supplied, that element must be replaced by a comma, as in the call above. A real argument string may be terminated in several ways as shown below:

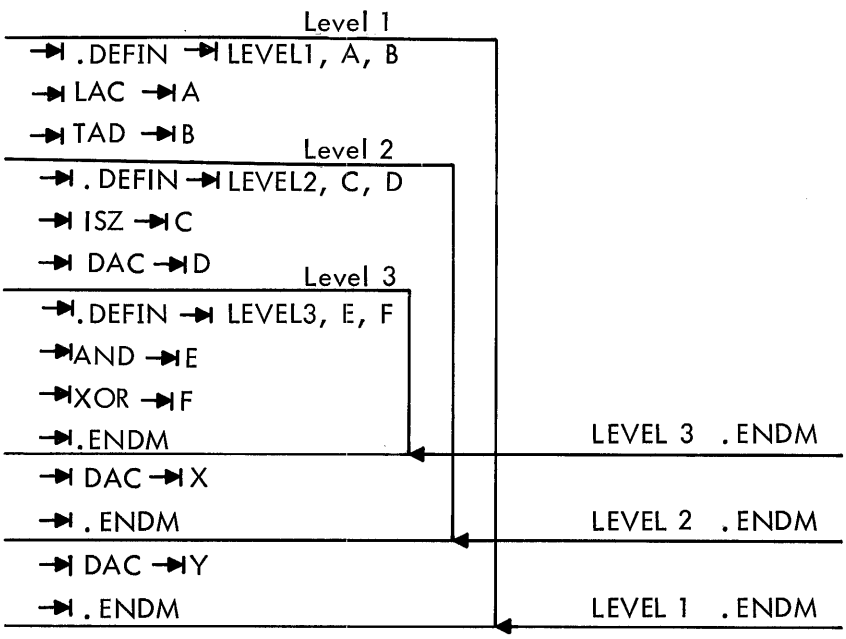
Example

```
→| MAC →| A, B, ␣
→| MAC →| A, B,,↵
→| MAC →| A, B ␣
→| MAC →| A, B↵
→| MAC →| A, B,↵
```

4.4 NESTING OF MACROS

Macros may be nested; that is, macros may be defined within other macros. For ease of discussion, levels may be assigned to these nested macros. The outermost macros (those defined directly) will be called first-level macros. Macros defined within first-level macros will be called second-level macros; macros defined within second-level macros will be called third-level macros, etc. Each nested macro requires an .ENDM pseudo-op to denote its termination.

Example:



At the beginning of processing, first-level macros are defined and may be called in the normal manner. Second and higher level macros are not yet defined. When a first-level macro is called, all its second-level macros are defined. Thereafter, the level of definition is irrevelant and macros may be called in the normal manner. If the second-level macros contain third-level macros, the third-level macros are not defined until the second-level macros containing them have been called.

Using the example above, the following would occur:

Call	Expansion	Comments
LEVEL 1 TAG1, TAG2	LAC TAG1	Causes LEVEL 2 to be defined
	TAD TAG2	
	DAC Y	
LEVEL 2 TAG3, TAG4	ISZ TAG3	Causes LEVEL 3 to be defined
	DAC TAG4	
	DAC X	
LEVEL 3 TAG5, TAG6	AND TAG5	
	XOR TAG6	

If LEVEL 3 is called before LEVEL 2 it would be an error, and the line would be flagged as undefined.

When a macro of level n contains another macro of the level n + 1, to call the macro of level n results in the generation of the body of the macro into the user's program in the normal manner until the .DEFIN statement of the level n + 1 macro is encountered; the level n + 1 macro is then defined and does not appear in the user's program. When the definition of the level n + 1 is completed (.ENDM encountered), the Assembler continues to generate the level n body into the user's program until, or unless, the entire level n macro has been generated.

4.5 REDEFINITION OF MACROS

If a macro name, which has been previously defined, appears within another definition, the macro is redefined and the original definition is eliminated. For example,

```
        →|. DEFIN →|INDXSV
        →|JMS →|SAVE
        →|JMP →|SAVXT
SAVE →|0
        →|LAC →|10
        →|DAC→| TMP#
        →|LAC →|11
        →|DAC→| TMP1#
        →|JMP*→| SAVE
 SAVXT=.
        →|. DEFIN →|INDXSV
        →|JMS →|SAVE
        →|. ENDM
        →|. ENDM
```

When the macro INDXSV is called for the first time, the subroutine calling sequence is generated and followed immediately by the subroutine itself. After the subroutine is generated, a . DEFIN that contains the name INDXSV is encountered. This new macro is defined and takes the place of the original macro INDXSV. All subsequent calls to INDXSV cause only the calling sequence to be generated. The original definition of INDXSV will not be removed until after the expansion is complete.

Call	Expansion
→| INDXSV	→|JMS →| SAVE
	→|JMP →| SAVXT
	SAVE →|0
	→|LAC →|10
	→|DAC→| TMP#
	→|LAC →|11
	→|DAC→| TMP1#
	→|JMP*→| SAVE
	SAVXT=.
→| INDXSV	→|JMS →| SAVE

4.6 MACRO CALLS WITHIN MACRO DEFINITIONS

The body of a macro definition may contain calls for other macros which have not yet been defined. However, the embedded calls must be defined before a call is issued to the macro which contains the embedded call. Embedded calls are allowed only to three levels.

Example

```
→I .DEFIN →IMAC1, A, B, C, D, E
→I LAC →IA
→I TAD →IB
→I MAC2 →I C, D                    /EMBEDDED CALL
→I DAC →I E
→I .ENDM
→I .DEFIN →I MAC2, A, B            /DEFINITION OF EMBEDDED CALL
→I XOR →IA
→I AND →I B
→I .ENDM
```

The call

```
→I MAC1 →ITAG1, TAG2, (400, (777, TAG3
```

causes generation of

```
→I LAC  →ITAG1
→I TAD  →ITAG2
→I MAC2→I (400, (777
→I XOR  →I (400
→I AND  →I (777
→I DAC  →I TAG3
```

4.7 RECURSIVE CALLS

Although it is legal for a macro definition to contain an embedded call to itself, it must be avoided because the expansion will cause more than three levels to occur.

Example

```
→I .DEFIN →IMAC, A, B, C
→I LAC →IA
→I TAD →IB
→I DAC →I C
→I MAC →IA, B, C                   /RECURSIVE CALL
→I .ENDM
```

When a call for MAC is encountered by the Assembler, it searches memory for the definition and expands it. Since there is another call for MAC contained within the definition, the assembler goes back once again to obtain the definition and this process would never cease, if more than three

levels were allowed. A conditional assembly statement could be used, however, to limit the number of levels as in the following example.

Example

```
A = 0
B = 3
        →I .DEFIN→IMAC,C,D
        →ILAC   →IC
        →IDAC   →ID
A = A + 1
        →I .IFNZR→IB-A
        →IMAC   →ISAVE,TEMP          /RECURSIVE CALL
        →I .ENDC
        →I .ENDM
```

Names and arguments of nested macros and arguments of imbedded calls may be substituted and used with perfect generality.

Example

```
        →I . DEFIN →IMAC1, A, B, C, D
        →I LAC→I A
        →I ADD→I B
        →I DAC→I C
        →I . DEFIN →ID, E
        →I AND →I A
        →I DAC →IE
        →I . ENDM
        →I . ENDM
              :
              :
        →I . DEFIN  →I MAC2, M, N, O, P, Q, ?R
        →I ISZ →IM
        →I JMP→I R
        →I MAC1 →IN, O, P, Q
        R=.
        →I . ENDM
```

The call

→| MAC2 →| COUNT, TAG1, TAG2, TAG3, MAC3

causes the generation of

→| ISZ →|COUNT
→| JMP →| . . 0000
→| LAC →| TAG1
→| ADD→| TAG2
→| DAC→| TAG3
. . 0000=.

It also causes the definition of MAC3

CHAPTER 5
ASSEMBLER OPERATION

5.1 OPERATING PROCEDURES

Operating procedures for MACRO-9 are contained in the I/O Monitor Guide (DEC-9A-MIPA-D) for paper tape systems, and in the Keyboard Monitor Guide (DEC-9A-MKFA-D) for bulk storage systems.

5.2 ASSEMBLY LISTINGS

If the user requests it, via the Monitor command string, the Assembler will produce an output listing on the requested output device. The top of the first page of the listing will contain the name of the program as given in the Monitor command string. The body of the listing will be formatted as follows.

ERROR FLAGS	LOCATION	ADDRESS MODE	OBJECT CODE	ADDRESS TYPE	SOURCE STATEMENT
xxx	xxxxx	{R, A}	xxxxxx	{R, A, E}	x x

where

FLAGS = Errors encountered by the Assembler (see paragraph 5.5)

LOCATION = Relative or absolute location assigned to the source

ADDRESS MODE = Indicates the type of user label address

- A = Absolute
- R = Relocatable

OBJECT CODE = The contents of the location (in octal)

ADDRESS TYPE = Indicates the classification of the address portion of the object code

- A = Absolute
- R = Relocatable
- E = External symbol

Locations and object codes assigned for literals and external symbols are listed following the program. The program name may be written in a .TITLE statement, as shown. This is treated as a comment.

Sample Listing

```
MACRO9      PAGE  1

                                                .TITLE  MACRO9  TEST  PROGRAM

                                                /
                                                /
     00000 R    200025 R                TAG1    LAC   A#
     00001 R    200002 R                        LAC   TAG2
     00002 R    000014 R                TAG2    TAG3
     00003 R    000033 R                        TAG4#
     00004 R    000002 R                        TAG2
     00005 R    200036 R                        LAC   (1
     00006 R    000037 R                        (2
     00007 R    000037 R                        (2
     00010 R    000040 R                        (3
     00011 R    000041 R                        (4
     00012 R    000042 R                        (JMP   TAG2
     00013 R    000043 R                        (JMP   TAG3
     00014 R    200042 R                TAG3    LAC (JMP   TAG2
     00015 R    200043 R                        LAC  (JMP   TAG3
     00016 R    000026 R                        B#
     00017 R    000027 R                        C#
     00020 R    000030 R                        D#
U    00021 R    000034 R                        E
     00022 R    000031 R                        F#
U    00023 R    000035 R                        G
     00024 R    000032 R                        H#
                000002 R                        .END   TAG2
     00036 R    000001 A   *LIT
     00037 R    000002 A   *LIT
     00040 R    000003 A   *LIT
     00041 R    000004 A   *LIT
     00042 R    600002 R   *LIT
     00043 R    600014 R   *LIT
     00044 R    600014 R   *LIT
```

5.3 SYMBOL TABLE OUTPUT

After the assembly listing is printed, the Assembler outputs a symbol table, if requested, which lists all user-defined symbols. There are two symbol lists. The first is an alphabetically ordered list of the symbols, and the second is a list in order of numerical value. The symbol table listing is useful in tracing or debugging a program for which the programmer does not have an assembly listing. The symbol table listing shows which symbols are:

A = Absolute

R = Relocatable

E = External symbol

5.4 ERROR DETECTION

MACRO-9 examines each source statement for possible errors. The statement which contains the error will be flagged by one or several letters in the left-hand margin of the line. The following table shows the error flags and their meanings.

Flag	Meaning
A	Error in direct Symbol Table assignment, assignment ignored (see paragraph 2.5.1).
B	Memory Bank error (program segment too large).
D	The statement contains a reference to a multiply defined symbol. It is assembled with the first value defined.
E	Erroneous results may have been produced (see paragraph 2.5.3). Will also occur on undefined .END value.
I	Line ignored. (Redundant Pseudo-op)
L	Literal phasing error. Literal encountered in PASS2 does not equal any literal encountered in PASS1.
M	An attempt is made to define a symbol which has already been defined. The symbol retains its original value.
N	Error in number usage.
P	Phase error. PASS1 value does not equal PASS2 value of a symbol. PASS1 value will be used.
Q	Questionable Line.
R	Possible relocation error.
S	Symbol error. An illegal character was encountered and ignored.
U	An undefined symbol was encountered.
W	Line overflow during macro expansion.
X	Illegal usage of macro name.

In addition to flagged lines, there are certain conditions which will cause assembly to be terminated prematurely.

Message	Pass	Cause
TABLE OVERFLOW	1 or 2	Too many symbols and/or macros.
CALL OVERFLOW	1	Too many embedded macro calls.

5.5 PROGRAM RELOCATION

The normal output from the MACRO-9 Assembler is a relocatable object program, which may be loaded into any part of memory regardless of which locations are assigned at assembly time. To accomplish this, the address portion of some instructions must have a relocation constant added to it. This relocation constant, which is added at load time by the Linking Loader, is equal to the difference between the memory location that an instruction is actually loaded into and the location that was assigned to it at assembly time. The Assembler determines which storage words are relocatable (marking them with an R in the listing) and which are absolute (marking these non-relocatable words with an A).

The rules that the Assembler follows to determine whether storage word is absolute or relocatable are as follows.

a. If the address is a number (not a symbol), the instruction is absolute.

b. If an address is a symbol which is defined by a direct assignment statement (i.e.,=) and the right-hand side of the assignment is a number; all references to the symbol will be absolute.

c. If a user label occurs within a block of coding that is absolute, the label is absolute.

d. Variables, undefined symbols, external transfer vectors, and literals get the same relocation as was in effect when .END was encountered in PASS1.

e. .gets current relocatability.

f. All others are relocatable.

The following table depicts the manner in which the Assembler handles expressions which contain both absolute and relocatable elements:

(A = Absolute, R = Relocatable)

A + A = A

A - A = A

A + R = R

A - R = R

R + A = R

R - A = R

R + R = R and flagged as possible error

R - R = A

If multiplication or division is performed on a relocatable symbol, it will be flagged as a possible error.

APPENDIX A

CHARACTER SET

Printing Character	7-bit ASCII	6-bit Trimmed ASCII	Printing Character	7-bit ASCII	6-bit Trimmed ASCII
@	100	00	(Space)	040	40
A	101	01	!	041	41
B	102	02	"	042	42
C	103	03	#	043	43
D	104	04	$	044	44
E	105	05	%	045	45
F	106	06	&	046	46
G	107	07	'	047	47
H	110	10	(	050	50
I	111	11	)	051	51
J	112	12	*	052	52
K	113	13	+	053	53
L	114	14	,	054	54
M	115	15	-	055	55
N	116	16	.	056	56
O	117	17	/	057	57
P	120	20	0	060	60
Q	121	21	1	061	61
R	122	22	2	062	62
S	123	23	3	063	63
T	124	24	4	064	64
U	125	25	5	065	65
V	126	26	6	066	66
W	127	27	7	067	67
X	130	30	8	070	70
Y	131	31	9	071	71
Z	132	32	:*	072	72
[*	133	33	;	073	73
\	134	34	<	074	74
]*	135	35	=	075	75
↑*	136	36	>	076	76
←*	137	37	?	077	77
Null	000				
Horizontal Tab	011				
Line Feed	012				
Vertical Tab	013				
Form Feed	014				
Carriage Return	015				
Rubout	177				

Notes:

(1) All other characters are illegal to MACRO-9 and are flagged and ignored.
(2) * = Illegal as source, except in a comment or text.

APPENDIX B
PERMANENT SYMBOL TABLE

Operate

OPR	740000
NOP	740000
CMA	740001
CML	740002
OAS	740004
RAL	740010
RAR	740020
HLT	740040
XX	740040
SMA	740100
SZA	740200
SNL	740400
SML	740400
SKP	741000
SPA	741100
SNA	741200
SZL	741400
SPL	741400
RTL	742010
RTR	742020
CLL	744000
STL	744002
CCL	744002
RCL	744010
RCR	744020
CLA	750000
CLC	750001
LAS	750004
LAT	750004
GLK	750010
LAW	760000

EAE Type KE09A

EAE	640000
OSC	640001
OMQ	640002
CMQ	640004
DIV	640323
NORM	640444
LRS	640500
LLS	640600
ALS	640700
LACS	641001
LACQ	641002
ABS	644000
DIVS	644323
CLQ	640000
FRDIV	650323
LMQ	652000
MUL	653122
IDIV	653323
FRDIVS	654323
MULS	657122
IDIVS	657323
NORMS	660444
LRSS	660500
LLSS	660600
ALSS	660700
GSM	664000

Interrupt

IOF	700002
ION	700042
CAF	703302

Memory Reference

CAL	000000
DAC	040000
JMS	100000
DZM	140000
LAC	200000
XOR	240000
ADD	300000
TAD	340000
XCT	400000
ISZ	440000
AND	400000
SAD	540000
JMP	600000

I/O States

IOT	700000
IORS	700314

Automatic Priority Interrupt Type KF09A

DBK	703304
DBR	703344
SPI	705501
ISA	705504

Memory Extension Control Type KE09B

SEM	707701
EEM	707702
LEM	707704

APPENDIX C
MACRO-9 CHARACTER INTERPRETATION

Character		Function
Name	Symbol	
Space	␣	Field delimiter. Designated by ␣ in this manual.
Horizontal tab	→|	Field delimiter. Designated by →| in this manual.
Semicolon	;	Statement terminator
Carriage return	↵	Statement terminator
Plus	+	Addition operator (two's complement)
Minus	–	Subtraction operator (addition of two's complement)
Asterisk	*	Multiplication operator or indirect addressing indicator
Slash	/	Division operator or comment initiator
Ampersand	&	Logical AND operator
Exclamation point	!	Inclusive OR operator
Back slash	\	Exclusive OR operator
Opening parenthesis	(	Initiate literal
Closing parenthesis	)	Terminate literal
Equals	=	Direct Assignment
Opening angle bracket	<	Argument delimiter
Closing angle bracket	>	Argument delimiter
Comma	,	Argument delimiter
Question mark	?	Create symbol designator in macros
Quotation marks	"	Text string indicators
Apostrophe	'	Text string indicator
Number sign	#	Variable indicator
Dollar sign	$	Real argument continuation

Character	Function
Line feed Form feed Vertical tab	Ignored if preceded by a carriage return; otherwise they are considered as illegal characters.
Null	Blank character. Ignored by the Assembler
Delete	Rubout character. Ignored by the Assembler

Illegal Characters

Only those characters listed on the preceding table are legal in MACRO-9 source programs, all other characters will be ignored and flagged as errors. The following characters, although they are illegal as source, may be used within comments or in .ASCII and .SIXBT pseudo-ops.

Character Name	Symbol
Commercial at	@
Opening square bracket	[
Closing square bracket	]
Up arrow	↑
Left arrow	←
Colon	:

APPENDIX D
SUMMARY OF MACRO-9 PSEUDO-OPS

Pseudo-op	Section	Format	Function
.ABS	3.2.1	→| .ABS →| NLD ↵ or ␣	Object program is output in absolute, blocked, checksummed format for loading by the Absolute Binary Loader
.ASCII	3.8.1	label →| .ASCII ␣ /text/<octal> ↵	Input text strings in 7-bit ASCII code, with the first character serving as delimiter. Octal codes for nonprinting control characters are enclosed in angle brackets.
.BLOCK	3.5	label →| .BLOCK →| exp ↵	Reserves a block of storage words equal to the expression. If a label is used, it references the first word in the block.
.DEC	3.4	→| .DEC ↵	Sets prevailing radix to decimal.
.DEFIN	3.16	→| .DEFIN ␣ macro name, args ↵	Defines macros.
.DSA	3.11	label →| .DSA ␣ exp ↵	Defines a user symbol which is to be used only in the address field.
.EJECT	3.14	→| .EJECT ↵	Skip to head of form on listing device.
.END	3.6	→| .END ␣ START ↵	Must terminate every source program. START is the address of the first instruction to be executed.
.ENDC	3.13	→| .ENDC ↵	Terminates conditional coding in .IF statements.
.ENDM	3.16	→| .ENDM ↵	Terminates the body of a macro definition.
.EOT	3.7	→| .EOT ↵	Must terminate physical program segments, except the last, which is terminated by .END.
.ETC	3.16	→| .ETC ␣ args, args ↵	Used in macro definitions to continue the list of dummy arguments on succeeding lines.
.FULL	3.2.2	→| .FULL ↵	Produces absolute, unblocked, unchecksummed binary object programs. Used only for paper tape output.
.GLOBL	3.9	→| .GLOBL ␣ sym, sym, sym ↵	Used to declare all internal and external symbols which reference other programs. Needed by Linking Loader.
.IFxxx	3.13	→| .IFxxx ␣ exp ↵	If a condition is satisfied, the source coding following the .IF statement, and terminating with an .ENDC statement, is assembled.
.IODEV	3.10	→| .IODEV ␣ .DAT numbers ↵	Specifies .DAT slots and associated I/O handlers required by this program.
.LOC	3.3	→| .LOC ␣ exp ↵	Sets the Location Counter to the value of the expression.

Pseudo-op	Section	Format	Function
.OCT	3.4	→|.OCT ↵	Sets the prevailing radix to octal. Assumed at start of every program.
.REPT	3.12	→|.REPT ␣ count, n ↵	Repeats the object code of the next object code generating instruction Count times. Optionally, the generated word may be incremented by n each time it is repeated.
.SIXBT	3.8.2	label →|.SIXBT ␣/text/⟨octal⟩ ↵	Input text strings in 6-bit trimmed ASCII, with first character as delimiter. Numbers enclosed in angle brackets are truncated to one 6-bit octal character.
.SIZE	3.15	→|.SIZE ↵	MACRO-9 outputs the address of last location plus one occupied by the object program.
.TITLE	3.1	→|.TITLE ␣ any comments ↵	Optional, typed on listing as a comment. (The program name given in the command string is printed at the top of each listing page, and used by the Loader.) May be used to annotate logical sections of a program.

APPENDIX E
SUMMARY OF SYSTEM MACROS

System macros (Monitor commands) are defined in the Monitor manual, and summarized here for the convenience of the PDP-9 programmers.

System macros are predefined to MACRO-9. To use a system macro, the programmer writes a macro call statement, consisting of the macro name and a string of real arguments.

To initialize a device and device handler

→| .INIT␣a,f,r

where a = .DAT slot number in octal
f = 0 for input files; 1 for output files
r = user restart address*

To read a line of data from a device to a user's buffer

→| .READ␣a,m,l,w

where a = .DAT slot number in octal
m = a number, 0 through 4, specifying the data mode:
0 = IOPS binary
1 = Image binary
2 = IOPS ASCII
3 = Image alphanumeric
4 = Dump mode
l = line buffer address
w = word count of the line buffer in decimal, including the two-word header

To write a line of data from the user's buffer to a device

→| .WRITE ␣ a,m,l,w

where a = .DAT slot number in octal
m = a number, 0 through 4, specifying the data mode:
0 = IOPS binary
1 = Image binary
2 = IOPS ASCII
3 = Image alphanumeric
4 = Dump mode
l = line buffer address
w = word count of line buffer in decimal, including the two-word header

*Meaningful only when device associated with .DAT slot a is the Teletype. Typing CTRLP on the keyboard will force control to location r.

To detect the availability of a line buffer

→| .WAIT␣a

where a = .DAT slot number in octal. After the previous .READ, .WRITE, or .TRAN command is completed; .WAIT returns control to the user at LOC+2

To detect the availability of a line buffer and transfer control to ADDR if not available.

→ .WAITR␣a, ADDR

where a = DAT slot number (octal radix)

ADDR = Address to which control is transferred if buffer is not available.

To close a file

→| .CLOSE␣a

where a = .DAT slot number in octal

To set the real-time clock to n and start it

→| .TIMER␣n,c

where n = number of clock increments in decimal. Each increment is 1/60 second (in 60-cycle systems) or 1/50 (in 50-cycle systems)

c = address of subroutine to handle interrupt at end of interval

To return control to Keyboard Monitor, or halt in I/O Monitor environment

→| .EXIT)

Mass Storage Commands for DECtape, Magnetic Tape, Disk and Drum only

To search for a file, and position the device for subsequent .READ commands

→| .SEEK␣a,d

where a = .DAT slot number in octal

d = address of user directory entry block

To examine a file directory, find a free directory entry block and transfer the block to the device

→| .ENTER␣a,d

where a = .DAT slot number in octal

d = address of user directory entry block

To clear a file directory to zero

→| .CLEAR␣a

where a = .DAT slot number in octal

To rewind, backspace, skip, write end-of-file, or write blank tape on nonfile-oriented magnetic tape

→| .MTAPE ␣ a,xx

where a = .DAT slot number in octal

xx = a number, 00 through 07, specifying one of the functions shown below

00 = Rewind to load point*
02 = Backspace one record*
03 = Backspace one file
04 = Write end-of-file
05 = Skip one record
06 = Skip forward one file
07 = Skip to logical end-of-tape

or a number, 10 through 16, to describe the tape configuration

10 = Even parity, 200 BPI
11 = Even parity, 556 BPI
12 = Even parity, 800 BPI
14 = Odd parity, 200 BPI
15 = Odd parity, 556 BPI
16 = Odd parity, 800 BPI

To read from, or write to any user file-structured mass storage device

→| .TRAN ␣ a,d,b,l,w

where a = .DAT slot number in octal

b = transfer direction:

0 = Input forward
1 = Input reverse
2 = Output forward
3 = Output reverse

b = device address in octal, such as block number for DECtape

l = core starting address

w = word count in decimal

To delete a file

→| .DLETE ␣ a,d

where a = .DAT slot number in octal

d = starting address of the 3-word block of storage in user area containing the file name and extension of file to be deleted from the device.

To rename a file

→| .RENAM ␣ a,d

where a = .DAT slot number in octal

d = starting address of two three-word blocks of storage in user area containing the file names and extensions of the file to be renamed, and the new name, respectively.

*May be used with any nonfile structured mass storage device.

To determine whether a file is present on a device

→| .FSTAT␣a,d

where a = .DAT slot number

d = starting address of three-word block in user area containing the file name and extension of the file whose status is desired.

Background/Foreground Monitor System Commands

To read a line of data from a device to a user's buffer in real-time

→| .REALR␣a, n, l, w, ADDR, p

where a = DAT slot number in octal

m = Data mode specification;

0 = IOPS binary
1 = Image binary
2 = IOPS ASCII
3 = Image Alphanumeric
4 = Dump mode

l = Line buffer address

w = word count of line buffer in decimal, including the two-word leader

ADDR = 15-bit address of closed subroutine that is given control when the request made by .REALR is completed.

p = API priority level at which control is to be transferred to ADDR:

0 = mainstream
4 = level of .REALR
5 = API software level 5
6 = API software level 6
7 = API software level 7

To write a line of data from user's buffer to a device in real time

→| .REALW␣a, m, l, w, ADDR, p

where a = DAT slot number in octal

m = Data mode specification;

0 = IOPS binary
1 = Image binary
2 = IOPS ASCII
3 = Image Alphanumeric
4 = Dump mode

l = line buffer address

w = word count of line buffer in decimal, including the two-word leader

ADDR = 15-bit address of closed subroutine that is given control when the request made by .REALW is completed

p = API priority level at which control is to be transferred to ADDR:

0 = mainstream
4 = level of .REALR
5 = API software level 5
6 = API software level 6
7 = API software level 7

To indicate, in a FOREGROUND job, that control is to be relinquished to a BACKGROUND job

→ .IDLE

To set the real-time clock to n and start it

→ .TIMER␣n, c, p

where n = number of clock increments in decimal. Each increment is 1/60 of of a second (1/50 in 50 Hz systems)

c = address of subroutine to handle interrupt at end of interval

p = API priority level at which control is to be transferred to c:

0 = mainstream
4 = level of .TIMER
5 = API software level 5
6 = API software level 6
7 = API software level 7

APPENDIX F
SOURCE LISTING OF THE ABSOLUTE BINARY LOADER

```
                                /
                                /***ABSOLUTE BINARY LOADER ***
                                /
 LDSTRT=17720
 BINLDR     CAF                 /CLEAR FLAGS
            CLOF                /CLOCK OFF
            IOF                 /INTERRUPT OFF
            CLA
            ISA                 /TURN OFF API
            MPLU                /TURN OFF MEMORY PROTECT
            EEM                 /SET EXTENDED MEMORY LOAD
            RSB                 /START READER UP
LDNXBK=17730
            DZM LDCKSM          /CHECKSUMMING LOCATION
            JMS LDREAD
            DAC LDSTAD          /GET STARTING ADDRESS
            SPA                 /BLOCK HEADING OR
            JMP LDXFR           /START BLOCK
            TAD LDCKSM          /ACCUMULATE CHECKSUM
            DAC LDCKSM
            JMS LDREAD
            DAC LDWDCT          /WORD COUNT (2'S COMPLEMENT)
            TAD LDCKSM
            DAC LDCKSM
            JMS LDREAD
            TAD LDCKSM          /PROGRAM CHECKSUM (2'S COMPLEMENT)
            DAC LDCKSM          /ADDED TO ACCUMULATED CHECKSUM
LDNXWD=17746
            JMS LDREAD
            DAC* LDSTAD         /LOAD DATA INTO APPROPRIATE
            ISZ LDSTAD          /MEMORY LOCATIONS
            TAD LDCKSM
            DAC LDCKSM          /ADD IN TO CHECKSUM
            ISZ LDWDCT          /FINISHED LOADING
            JMP LDNXWD          /NO
            SZA                 /LDCKSM SHOULD CONTAIN 0
            HLT                 /CHECKSUM ERROR HALT
            JMP LDNXBK          /PRESS CONTINUE TO IGNORE
 LDXFR=17760
            DAC LDWDCT
            ISZ LDWDCT
            JMP LDWAIT          /WAIT FOR READER
            HLT                 /NO ADDRESS ON .END STATEMENT
 LDREAD=17764                   /MANUALLY START USER PROGRAM
            0
            RSF
            JMP LDREAD+1        /.-1
            RRB
            RSB
            JMP* LDREAD
LADWAIT=17772
            RSF
            JMP LDWAIT
            JMP* LDSTAD         /EXECUTE START ADDRESS
ENDLDR      JMP LDSTRT          /HARDWARE READIN WORD
LDCKSM=17775
LDSTAD=17776
LDWDCT=17777
```

APPENDIX G
ABBREVIATED MACRO-9 FOR 8K SYSTEMS

A shorter version of MACRO-9, called MACROA (for MACRO-9 Abbreviated), is available on the system tape, especially for user's with 8K machines who are using DECtape for input and output.

The following features have been removed in this shorter version:

.ABS

.FULL

Conditional pseudo-ops

.REPT

.DEFIN (User-defined macros are not allowed, but system macro calls are legal.)

This reduces the assembler by about 850 locations, which is about the same number required by the DECtape I/O service routine. The user's Symbol Table cannot exceed 275 symbols.

If the user of MACROA uses DECtape input and paper tape output, the Symbol Table may contain up to 550 symbols.

Calling MACROA

In response to

MONITOR

$

the abbreviated assembler is called by typing, after $,

MACROA

Both versions of MACRO-9 are available on the system tape. If macro definition and/or conditional capabilities are desired, MACRO with DECtape input and paper tape output can be used, allowing up to 275 user defined symbols.

APPENDIX H
SYMBOL TABLE SIZES

The following symbol table sizes are for 8K systems with the full complement of skip IOT's in the skip chain.

NOTE

Handlers listed are for DAT slots -11, -12, -13, and -10, respectively.

MACRO

a. PRB, TTA, PPC, TTA - 317 symbols (decimal)

b. DTC, TTA, PPC, TTA - 189 symbols (decimal)

for .ABS or .FULL output PPB must be used - delete 60 symbols (decimal) from above counts.

MACROA

a. PRB, TTA, PPC, TTA - 610 symbols (decimal)

b. DTC, TTA, PPC, TTA - 482 symbols (decimal)

c. DTB, TTA, DTB, TTA - 261 symbols (decimal)

APPENDIX I

SUMMARY OF OPERATING PROCEDURES WITH KEYBOARD MONITOR

These procedures are described in the Monitor Manual and are summarized here for the convenience of PDP-9 programmers.

To assemble a program:

a. Mount the system tape on DECtape 0 (Set selector switch to 8)

b. Load paper tape bootstrap (HRM), which loads the Monitor, which types out

```
MONITOR
$
```

c. The user may check the Device Assignment Table by typing:

```
REQUEST MACRO
```

after which Monitor types out the table showing all current device assignments for MACRO-9's logical assignments:

	DAT Slot
1. Secondary input	-10
2. Input source program	-11
3. Listing	-12
4. Output binary program	-13
5. Command string	-2
6. Error messages	-3

At this point, the user may assign devices, if none are assigned to any of the 6 slots used by MACRO-9, or he may change assignments, using the ASSIGN command. If the input source program is on DECtape .1, and he wants to use DECtape handler A, he types,

```
ASSIGN    DTAI    -11
```

To verify this change, he may type REQUEST MACRO again, and Monitor will type out the specified MACRO-9 DAT, showing that DECtape 1 is now assigned to DAT slot -11.

d. To obtain typed out operating instructions for MACRO-9, the user may type

```
INSTRUCT MACRO
```

e. To load MACRO-9 after Monitor types $, the user types

```
MACRO
```

After MACRO-9 is loaded and self-initialized, MACRO-9 types MACRO and the user types;

```
                                                    >
P,B,L,S ←program name (or ALT MODE)
```

where B requests that MACRO-9 output a binary object program, L requests a listing, and S requests that the user's symbol table be printed out. These three letters are all optional, and may be written in any order, but they must be separated by commas. If none of these letters are written, error messages are output on the Teletype.

Whether or not any letters are written, the reverse arrow (←) must follow. This is followed by the program name, identifying the source program to be searched for by the loader on the input file, and this name is written at the top of the first listing page.

When the input medium is file-oriented, MACRO-9 expects the file name extension SRC (source). The Editor in the PDP-9 Advanced Software System provides SRC to a file name automatically if the output medium is file-oriented.

If the source is originally on paper tape (or cards), the file name extension SRC must be included in the command string to PIP, when transferring to a file-oriented medium.

If another program is to be assembled following this one, the command string is terminated by a carriage return. At the conclusion of PASS2, control returns to MACRO-9, whicy types

```
MACRO,
>
```

and the user may then type another command string for the next program to be assembled.

If the user wishes to return control to the Keyboard Monitor after the program is assembled, he terminates the command string by typing ALT MODE.

MACRO-9 PROGRAMMERS
REFERENCE MANUAL
DEC-9A-AMZA-D

READER'S COMMENTS

Digital Equipment Corporation maintains a continuous effort to improve the quality and usefulness of its publications. To do this effectively, we need user feedback: your critical evaluation of this manual and the DEC products described.

Please comment on this publication. For example, in your judgment, is it complete, accurate, well-organized, well-written, usable, etc?

Did you find this manual easy to use?

What is the most serious fault in this manual?

What single feature did you like best in this manual?

Did you find errors in this manual? Please describe.

Please describe your position.

Name Organization

Street State Zip

..... Fold Here .....

..... Do Not Tear - Fold Here and Staple .....

FIRST CLASS
PERMIT NO. 33
MAYNARD, MASS.

BUSINESS REPLY MAIL
NO POSTAGE STAMP NECESSARY IF MAILED IN THE UNITED STATES

Postage will be paid by:

digital

Digital Equipment Corporation
Software Quality Control
Building 12
146 Main Street
Maynard, Mass. 01754