PDP-15
MACRO-15 ASSEMBLER PROGRAMMER'S REFERENCE MANUAL

To obtain additional copies of this manual, order number DEC-15-AMZA-D from the Program Library Digital Equipment Corporation, Maynard, Mass. 01754 Price $ 2.50

DIGITAL EQUIPMENT CORPORATION □ MAYNARD, MASSACHUSETTS

1st Printing October 1969

CONTENTS

CONTENTS (Cont)

CONTENTS (Cont)

PREFACE

OVERALL PDP-15 DOCUMENTATION STRUCTURE

A tree-type block diagram of the overall "PDP-15 Family of Manuals" is illustrated on page viii. A brief description of the contents and the order number of each manual shown in the diagram are presented on page ix.

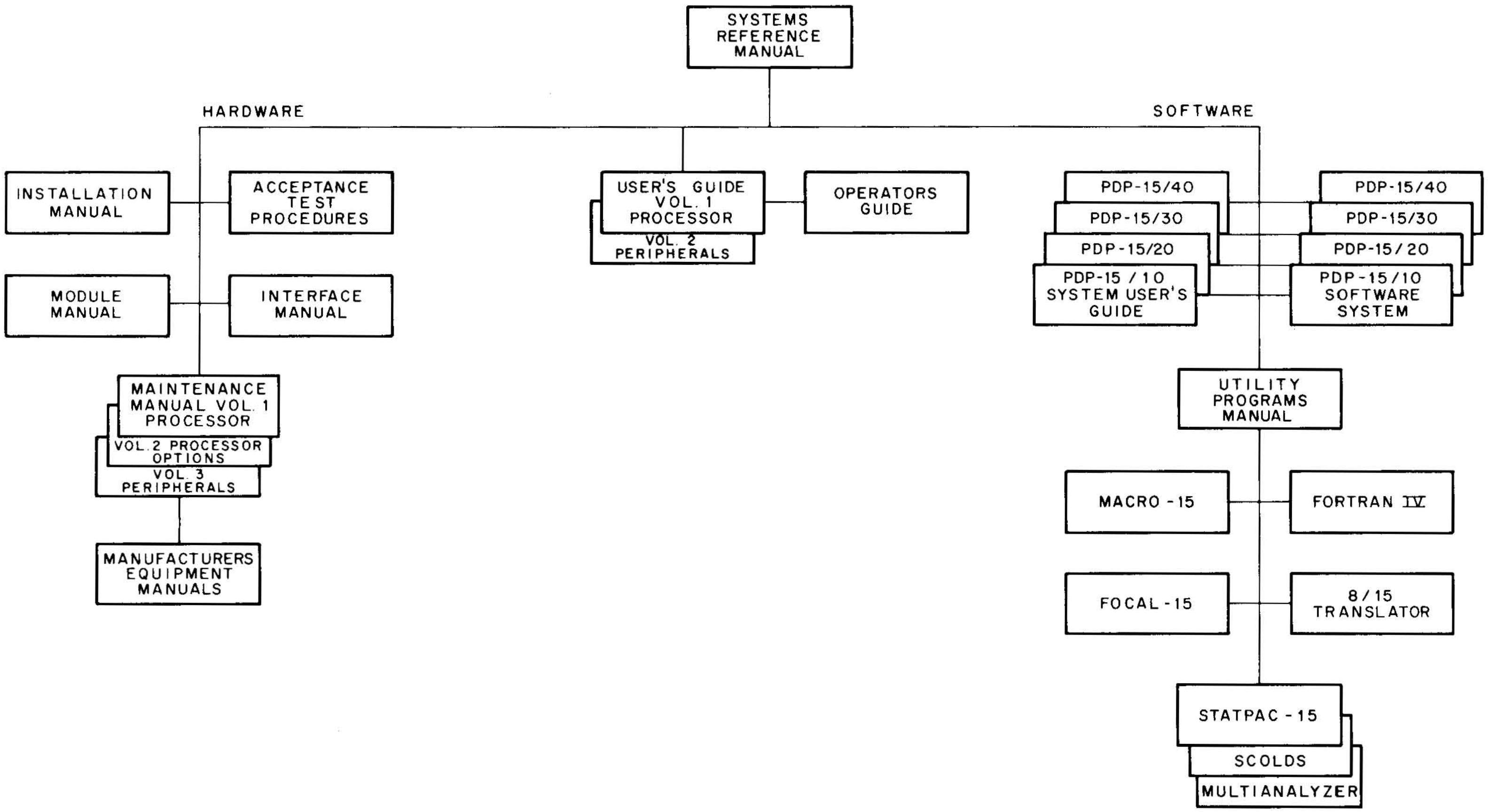
ORGANIZATION OF PDP-15 SOFTWARE MANUALS

There are two basic catagories of PDP-15 software manuals:

a. Unique, single-system, manuals which contain information concerning only one of the four available PDP-15 systems. This catagory consists of detailed software system descriptive manuals, each with an associated operational command summary. An example of this class of manual would be the "PDP-15/10 Software System" manual and its associated "PDP-15/10 Users' Guide".

b. Common, multi-system, manuals that describe utility, language, application and other PDP-15 programs which may be employed in one or more of the four available PDP-15 systems. Some examples of this type of manual are the PDP-15 "Utility", "MACRO-15 Assembler" and "STATPAC" manuals.

PDP-15 FAMILY OF MANUALS



15-0040

SYSTEM REFERENCE MANUAL - Overview of PDP-15 hardware and software systems and options; instruction repertoire, expansion features and descriptions of system peripherals.
DEC-15-GRZA-D

USERS GUIDE VOLUME 1, PROCESSOR - Principal guide to system hardware includes system and subsystem features, functional descriptions, machine-language programming considerations, instruction repertoire and system expansion data.
DEC-15-H2DA-D

VOLUME 2 PERIPHERALS - Features functional descriptions and programming considerations for peripheral devices.
DEC-15-H2DA-D

OPERATOR'S GUIDE - Procedural data, including operator maintenance, for using the operator's console and the peripheral devices associated with PDP-15 Systems.
DEC-15-H2CA-D

PDP-15/10 SYSTEM USER'S GUIDE - COMPACT and BASIC I/O Monitor operating procedures.
DEC-15-GG1A-D

PDP-15/20 SYSTEM USER'S GUIDE - Advanced monitor system operating procedures.
DEC-15-MG2A-D

PDP-15/30 SYSTEM USER'S GUIDE - Background/Foreground monitor system operating procedures.
DEC-15-MG3A-D

PDP-15/40 SYSTEM USER'S GUIDE - Disk-oriented background/foreground monitor system operating procedures.
DEC-15-MG4A-D

PDP-15/10 SOFTWARE SYSTEM - COMPACT software system and BASIC I/O Monitor system descriptions.
DEC-15-GR1A-D

PDP-15/20 ADVANCED Monitor Software System - ADVANCED Monitor System descriptions; programs include system monitor and language, utility and application types; operation, core organization and input/output operations within the monitor environment are discussed.
DEC-15-MR2A-D

PDP-15/30 BACKGROUND/FOREGROUND Monitor Software System - Background/Foreground Monitor description including the associated language, utility and applications programs.
DEC-15-MR3A-D

PDP-15/40 Disk-Oriented BACKGROUND/ FOREGROUND Monitor Software System - Background/Foreground Monitor in a disk-oriented environment is described; programs include language, utility, and application types.
DEC-15-MR4A-D

MAINTENANCE MANUAL VOLUME 1, PROCESSOR - Block diagram and functional theory of operation of the processor logic. Preventive and corrective maintenance data.
DEC-15-HB2A-D

VOLUME 2, PROCESSOR OPTIONS - Block diagram and functional theory of operation of the processor options. Preventive and corrective maintenance data.
DEC-15-HB2A-D

VOLUME 3 PERIPHERALS (Set of Manuals) - Block diagram and functional theory of operation of the peripheral devices. Preventive and corrective maintenance data.
DEC-15-HB2A-D

INSTALLATION MANUAL - Power specifications, environmental considerations, cabling and other information pertinent to installing PDP-15 Systems.
DEC-15-H2AA-D

ACCEPTANCE TEST PROCEDURES - Step-by-step procedures designed to insure optimum PDP-15 Systems operation.

MODULE MANUAL - Characteristics, specifications, timing and functional descriptions of modules used in PDP-15 Systems.

INTERFACE MANUAL - Information for interfacing devices to a PDP-15 System.
DEC-15-H0AA-D

UTILITY PROGRAMS MANUAL - Utility programs common to PDP-15 Monitor systems.
DEC-15-YWZA-D

MACRO-15 - MACRO assembly language for the PDP-15.
DEC-15-AMZA-D

FORTRAN IV - PDP-15 version of the FORTRAN IV compiler language.
DEC-15-KFZA-D

FOCAL-15 - An algebraic interactive compiler-level language developed by Digital Equipment Corporation.
DEC-15-KJZA-D

CHAPTER 1
INTRODUCTION

1.1 MACRO-15 LANGUAGE

MACRO-15 is a basic PDP-15 symbolic assembler language which makes machine language programming on the PDP-15 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.

Assembled MACRO-15 programs may be run on any PDP-15 system; however, MACRO-15 symbolic programs can be assembled only on systems which have at least 8K of memory and a monitor-type software system.

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-15 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 (see Chapter 3 for a description of the binary output modes).

The programmer directs MACRO-15 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 to perform other functions which are explained in detail in Chapter 3.

The most advanced features of MACRO-15 is its powerful macro instruction generator. This generator permits easy handling of recursive instruction 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-15, 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 assembly are described in detail in Chapter 5. These procedures are also summarized in the "Users' Guide" for each Monitor Software system.

1.2 HARDWARE REQUIREMENTS AND OPTIONS

The MACRO-15 assembler program may be run on any of the following PDP-15 systems:

a. 15/10 system which has a minimum of 8K of core and optional high-speed paper tape reader and punch units.

b. basic 15/20 system

c. basic 15/30 system

d. basic 15/40 system

1.3 ASSEMBLER PROCESSING

The MACRO-15 assembler processes source programs in either a two-pass or three-pass operation. In the two-pass assembly operation the source program is read twice with the object program (and printed listing when requested) being produced during the second pass. During the first pass (PASS 1), the locations to be assigned the program symbols are resolved and a symbol table is constructed by the assembler. The second pass (PASS 2) uses the information computed during PASS 1 to produce the final object program.

In an optional three-pass assembly operation, PASS 2 will call in a third pass (PASS 3) portion of the assembler program. PASS 3, when called, performs a cross referencing operation during which a listing is produced which contains: (a) all user symbols, (b) where each symbol is defined, and (c) the number of each program line in which a symbol is referenced. On completion of its operation, PASS 3 calls the PASS 1 and PASS 2 portions of the assembler program back into core for further assembly operations.

The standard object code produced by MACRO-15 is in a relocatable format which is acceptable to the PDP-15 Linking Loader Utility program. 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. MACRO-15 reserves one additional word in a program for every external** symbol. This additional word is used as a pointer to the actual data word in another program. The Linking Loader sets up these pointers when the programs are loaded.

---

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

**Symbols which are referenced in the program currently being assembled but which are defined in another 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.

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 set 10 spaces apart on most Teletype machines, and are 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-15 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-15 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 <u>label</u> (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 <u>operation code</u> field may contain a machine mnemonic instruction code, a MACRO-15 pseudo-op code, a macro name, a number, or a symbol. The <u>address</u> 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 (/) may be the first character in a line or may be preceded by:

a. Space (␣)
b. Tab (→|)
c. Semicolon (;)

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. 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.

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 may type error flags on the lines containing SYMBOL1, SYMBOL2 and SYMBOL 3. 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-15 memory reference instructions, operate instructions, EAE instructions, and some input/output transfer instructions. (See Appendix B for a complete list of these instructions.)

PDP-15 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 mnemonics or the pseudo-instruction mnemonics code as symbol labels. For example,

DZM →| DZM␣, Y↵

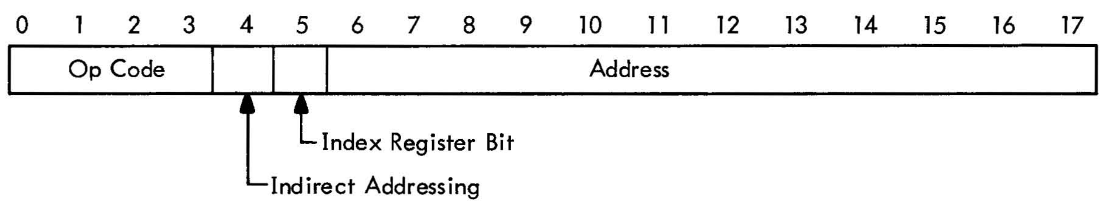
where the label DZM is entered in the symbol table and is given the current value of the location counter, and the op code DZM is given the value 140000 from the permanent symbol table. The user must be careful, however, in using these dual purpose (field dependent) symbols. Symbols in the operation field are interpreted as either instruction codes or pseudo-ops, not as symbolic labels, 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
		X	010000

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
DAC →| TAG1,X	050100
→| TAG1	000100
.	

2.2.1.1 Special Symbols - The symbol X is used to denote index register usage. It is defined in the permanent symbol table as having the value of 10000. The symbol X cannot be redefined and can only be used in the address field.

2.2.1.2 Memory Referencing Instruction Format - The PDP-15 uses 12 bits for addressing, 1 bit to indicate index register usage, 1 bit to indicate indirect addressing, and 4 bits for the op code.



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 (#). Symbols with the # may appear more than once in a program (see items 1, 3, 4, and 5 of example given below). 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 #. If the variable duplicates a user-defined label, it is multiply defined and 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.

Example:

Sequence	Location Counter	Source Statements	Generated Code
		⇥ .LOC ␣ 100	
1	100	⇥ LAC ␣ TA#G1	200105
2	101	⇥ DAC ␣ TAG3	040107
3	102	⇥ LAC ␣ TAG2#	200106
4	103	⇥ DAC ␣ T#AG3,X	050107
5	104	⇥ LAC ␣ #TAG2	200106
		⇥ .END	

2.2.3 Setting Storage Locations to Zero

Storage words can be set to zero as follows:

⇥ A ⇥ 0; ⇥ 0; ⇥ 0 ↵

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

A; B; C; D; E ↵

2.2.4 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-15 enters the symbol in the symbol table, along with the assigned value. Symbols entered in this way may be redefined. These are legal direct assignment statements:

```
X=28; A=1; B=2↵
```

A symbol can also be assigned a symbolic value; e.g., A=4, B=A, or

```
SET=ISZ␣SWITCH
```

In the above example, the symbol B is given the value 4, and when the symbol SET is detected during assembly the object code for the instruction ISZ␣SWITCH will be generated. This type of direct assignment cannot be used in a relocatable program. 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 PASS 2. When first encountered in PASS 1, the LAC Y statement is incomplete because Y is not yet defined. Later in PASS 1, Y is given the value 1. In PASS 2, the Assembler finds that Y = 1 in the symbol table, and forms the complete storage word.

Since MACRO-15 basic assembly operations are performed in two passes, only one-step forward references are allowed. The following is illegal:

```
LAC Y↵
Y=Z↵
Z=1↵
```

In the listing, during PASS 1, the line which contains Y = Z will be printed as a warning.

2.2.5 Undefined Symbols

If any symbols, except global symbols, remain undefined at the end of PASS 1 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.

Examples:

Flag	Location Counter	Source Statements	Generated Code	Comments
		→ .LOC ␣ 100 ↵		
u	100	→ LAC ␣ UNDEF1 ↵	200106	Undefined Symbol
	101	→ LAC ␣ TAG# ↵	200104	
	102	→ LAC ␣ TAG#1 ↵	200105	
u	103	→ LAC ␣ UNDEF2 ↵	200107	Undefined Symbol
		→ .END ↵		

2.3 NUMBERS

The initial radix (base) used in all number interpretation by the Assembler is octal (base 8). 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 the other information as a comment and flags it as a questionable line. 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 the program.

Examples:

Flag	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
Q	→ .OCT ␣ 99		Octal radix takes effect even though line is flagged
	→ 76	000076	8
N	→ 99	000143	The non-octal digit forces a decimal radix for this number only

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 (8 or 9) is encountered the string of digits will be assembled as if the decimal radix was in effect and it will be flagged as a possible error.

Example:

Flag	Coded Value	Generated Value (Octal)	Comment
	.DEC		
	3779	007303	
	.OCT		
	-5	777773	Two's complement
	3347	003347	
N	3779	007303	Possible error, decimal assumed

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

Examples:

Flag	Coded Value	Generated Value (Octal)	Comment
	-8	777770	Two's complement
	+256	000400	
N	-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

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

(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:

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
Exclamation point	!	Inclusive OR } Boolean
Back slash	\	Exclusive OR } Boolean
Comma	,	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)	Comments
A	000002	
B	000010	
C	000003	
D	000005	
X	010000	Index Register Value

The following expressions would be evaluated.

Expression	Evaluation (Octal)	Comments
A+B-C,X	010007	Index Register Usage
A/B+A*C	000006	(The remainder of A/B is lost)
B/A-2*A-1+X	010003	Index Register Usage
A & B	000000	
C+A&D	000005	
B*D/A	000024	
B*C/A*D	000074	
A,X+D,X	010007	Index Register Usage Error

In the last example the expression is evaluated as follows:

Sequence of arithmetic

a. A,X = 000002 XORed with 010000 = 010002

b. A,X+D = 010002 + 000005 = 010007

c. A,X+D,X = 010007 XORed with 010000 = 000007

Note that arithmetic produces 000007 yet the value given in the example is 010007. Regardless of how the index register is used in the address field, the index register bit will always be turned on by the Assembler. In the sequence of address arithmetic above, the line would be flagged with an X because of the illegal use of the index register symbol (X).

Using the symbol X to denote index register usage causes the following restrictions:

a.	X cannot appear in the TAG field	X ⇥ LAC ⇥ A
b.	X cannot be used in a .DSA statement	.DSA A,X
c.	X can only be used once in an expression	LAC A,X+D,X

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 is 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.

CLA*	Indirect addressing may not be specified
LAW* 17777	in non-memory reference instructions.

2.4.3 Indexed Addressing

To specify indexed addressing an X is used with an operator directly after the address. No spaces or tabs may appear before the operator. The Assembler will perform whatever operation is specified with the index register symbol, and then continue to evaluate the expression. At completion of the expression evaluation, if the index bit is not on and the location counter is pointing to page 0 of any bank, the line is flagged with a B for bank error. The standard code used to indicate indexing is:

```
LAC   A,X.
```

Example:

```
                 Object
Location         Code
                                 .ABSP
000000           210000    A     LAC →| X              /Same as LAC 0,X
000001           210005    B     DAC →| A,X+1,7-1      /
000002           210001          LAC →| B+X            /000001 010000
                                 .LOC   10000          /SET to page 1
010000           210001    C     LAC    X,D
010001           210000    D     LAC    C,X
                                 .END
```

expression evaluation where A = 000000, B = 000001, C = 010000, X = 010000

Location	Address Field	Discussion
0	X	The value of X is added to 0. Absence of an operator always implies addition.
1	A,X+1,7-1	000000 + 010000 = 010000 010000 + 000001 = 010001 010001 + 000007 = 010006 010006 - 000001 = 010005
2	B+X	000001 + 010000 = 010001
10000	X,D	010000 + 010001 = 000001 The index bit has been turned off during expression evaluation. Because the location counter (10000) is pointing to page 1, this line is not flagged and the index register bit is turned on.
10001	C,X	010000 + 010000 = 000000 Same as example at location 10000.

NOTE: + = exclusive OR

2.4.4 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 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.

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

Duplicate literals, completely defined when scanned in the source program during PASS 1, are stored only once so that many uses of the same literal in a given program result in the allocation of only one memory location 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 following sample program illustrates how the Assembler handles literals.

Location Counter	Source Statement	Generated Code
	→| .LOC ␣ 100	
100	TAG1 →| 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 0)	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 in order 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:

TAG1;TAG2;TAG3;TAG4	A new logical line starts after each semicolon. This line is equivalent to TAG1 →| 0↵ TAG2 →| 0↵ TAG3 →| 0↵ TAG4 →| 0↵

If there was a tab or a space after the semicolon the symbol would be evaluated as an operator instead of a tag. The sequence

```
TAG1;␣TAG2;TAG3;␣TAG4
```

is evaluated as follows:

```
TAG1 →| 0↵
        TAG2↵
TAG3 →| 0↵
        TAG4↵
```

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.

When writing numbers separated by semicolons, the first number must be preceded by a tab (→|) or a space (␣). The sequence

```
TABLE ␣ 1;2;3;4;5
```

produces TAG errors because the first symbol of a tag cannot be numeric. The correct way to write the table sequence is as follows:

TABLE ␣ 1; ␣ 2; ␣ 3; ␣ 4; ␣ 5

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:

Flag	Location Counter	Statement	Storage Word Generated	Notes
M	100	A →| LAC →| B	200103	
M	101	A →| LAC →| C	200104	Error, multiple definition
D	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 ↵

The line will be flagged with a "T" for tag error. The tag will be ignored but the rest of the line will continue to be processed. The only time that an error tag is not ignored is when the error occurs after the sixth character. The statement:

TAGERROR*1 ␣ NOP

will be assembled as:

TAGERR →| NOP

and the line will be printed and flagged with a "T".

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:

⇥ or ␣ (s) (field delimiters)
↵ 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 defer bit (bit 4) of the instruction word to be set; that is, the reference will be an indirect reference. If the asterisk (*) is appended on either a non-memory reference instruction or any symbol in the address field, it will cause an error condition which will be flagged as a symbol error (S-flag). The asterisk will be ignored and the assembly process will continue.

Examples:

Assembled Value	Legal	Assembled Value	Illegal
360001	→| TAD* →| A	360001	→| LAC →| A*
220002	→| LAC* →| B	740000	→| CLA*

where A = 1 and B = 2

However, the asterisk (*) may be used anwhere 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 will cause an error message. The address field must be terminated by one of the following characters:

→| or ␣ (s) (field delimiters)
↵ 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 semocolon, or a carriage-return/line-feed sequence.

Examples:

→| JMP →| BEGIN ↵

→| TAD →| A; →| DAC →| B →| LAC

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.

In the second line of the preceding example, the address field B is delimited by a tab. The LAC after the B →| 1 is ignored and is treated as a comment; but, the line is questionable because only a comment field occurs on a line after the address field. If the LAC had been preceded by a slash (/), the line would have been correct.

When the address field is a relocatable expression, an error condition may occur. The size of the relocatable program is restricted to 4K (4096_{10}) words and cannot be loaded across pages or memory banks. Therefore, any relocatable address field whose value exceeds 7777_8 is meaningless and will be flagged in error. This does not apply if the user specifies bank addressing (refer to description of .EBREL).

When the address field is an absolute expression, an error condition will exist if the extended memory and page address bits (3, 4 and 5) do not match the corresponding bits of the address of the bank currently being assembled into and these address bits are not 0.

NOTE

In absolute mode, the page bits do not have to be equal if the .ABS or .FULL pseudo-ops are used instead of the .ABSP or .FULLP pseudo-ops.

Examples:

Location (octal)	Instruction	Comments
30000	→| LAC ␣ 30100	Will not cause error messages
30001	→| DAC ␣ 101	
30002	→| JMS ␣ 250	
30005	→| ISZ ␣ 40146	Will cause a bank (B) error message because the address is on a different page.

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 location 10_8 - 17_8 in page 0 of memory bank 0. The hardware will always ensure that <u>indirect</u> references to 10_8 - 17_8 in any page or bank will 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.	␣ (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 semicolon in order to have the second and third RTRs interpreted as being in the operation field and not in the label field.

2.6 STATEMENT EVALUATION

When MACRO-15 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.

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 in paragraph 2.6.4.

The operation value is scanned for the following special cases:

Mnemonic	Operation Field Value
LAW	760000
AAC	723000
AAS	720000
AXR	737000
AXS	725000

If the operation field is not one of the special cases, the object word value is computed as follows:

(Operation Field + (Address Field and 17777)) = Word Value

If the index register is used anywhere in the address field, the index register bit is set to one in the word value. Extensive error checking is then performed on the address field value. The following are the rules used to ensure correct results:

a. If index register usage is specified, the result of XORing bit 5 of the location counter and bit 5 of the address field value must be non-zero.

Example:

Flag	Location	Word Value		.ABSP	/Page Addressing
	00000	210001		LAC A,X	/Page 0
	00001	740000	A	NOP	
	10000			.LOC 10000	/Page 1
	10000	210001		LAC B,X	
B	10001	210001	B	LAC A,X	
				.END	

The result of statement evaluation has produced the following results:

A,X = 10001 A = 00001

B,X = 00001 B = 10001

Note that when index register usage is specified, the index register bit may or may not be on. For B,X above, the index register bit was turned off. The Assembler turns this bit on when the word is evaluated, not at statement evaluation time.

At location 10001, the result of XORing bit 5 of A,X and bit 5 of the location counter is 0. This signals the Assembler that the address reference (A) is in a different page.

b. If index register usage is not specified and the program is not assembled in bank mode*, the result of XORing bits of the location counter and the address field value must be 0, otherwise the line is flagged with a B for bank error.

Example:

Flag	Location	Object Word		.ABSP
B	00000 10500 10500	210500 740000	 A	LAC A .LOC 10500 NOP .END

c. The bank bits (3,4) of the address field value in a relocatable program must never be on. The bank bits are always lost when the address field value and the operation are combined to form the object word value.

Example:

Flag	Location	Object Word Value		
B	00000 R 17777 R 17777 R 20000 R	200000 R 740000 A 740000 A	C A	LAC A /Bank bit lost .LOC C+17777 NOP NOP .END

d. If the bank bits of an absolute program are not zero, they must equal the bank bits of the location counter.

Example:

Line	Flag	Location	Object Word Value	
1				. ABSP
2		20000		.LOC 20000
3		20000	200001	LAC 1
4		20001	200001	LAC 20001
5	B	20002	210001	LAC 30001
6	B	20003	217777	LAC 17777
7				.END

*See pseudo-ops .ABS, .ABSP, .FULL, .FULLP, .EBREL, .DBREL

The address value for lines 3 and 4 are identical. The bank bits of ine 5 do not match those of the location counter, and indexed addressing was not specified, therefore, the line is flagged.

2.6.3 Word Evaluation of the Special Cases

a. LAW - The operation field value and the address field value are combined as follows:

(Operation Value + (Address Field Value and 17777)) = Word Value

A validity check is then performed on the address field value as follows:

(Address Field Value and 760000) = Validity Bits

If the validity bits are not equal to 777000 or 0, the line is flagged with an E to signal erroneous results.

b. AAC, AAS, AXR, AXS - The operation field value and the address field value are combined as follows.

(Operation Value + (Address Field Value and 000777)) = Word Value

The validity check:

(Address Field Value and 777000) = Validity Bits

If If the validity bits are not equal to 777000 or 0, the line is flagged with an E to signal erroneous The address field value for this type of instruction cannot be relocated. The line is flagged with an R if the address field value is relocatable.

Example:

Line	Flag	Location	Object Word Value				
1		0	777777		LAW	17777	/17777
2		1	777777		LAW	-1	/777777
3	E	2	777777		LAW	677777	/677777
4		3	760000	A	LAW		/0
5		4	720776		AAS	-2	/777776
6	E	5	720000		AAS	-2000	/776000

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.

2.6.4 Assembler Priority List

Label Field	Operation Field	Address Field
Current Value of Location Counter	1. Pseudo-op	1. User symbol table (including direct assignments)
	2. User macro in user symbol table	2. Undefined
	3. System macro table	
	4. Direct assignment in user symbol table	
	5. Permanent symbol table	
	6. User symbol table	
	7. 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 PASS 1; 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 user macro names, but may not be redefined as user symbols by direct assignment or by use as statement labels.

The user may use machine instruction codes and MACRO-15 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-15 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 will take the first six characters of the symbol in the address field as the new name of the program to appear in the header on the listing device. The listing device will be advanced to the top of form after which the line will be printed as a comment. The name will appear as the program name until the next .TITLE pseudo-op. The .TITLE pseudo-op has no effect on the binary or listing file name.

→| .TITLE␣NAME OF PROGRAM	/(NAME)	Name on listing delimited by space;
→| .TITLE →| TEST1	/(TEST1)	Name on listing

3.2 OBJECT PROGRAM OUTPUT (.ABS, .ABSP, .FULL, .FULLP, .DBREL, .EBREL)

The normal object code produced by MACRO-15 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.

a. The .ABS, .ABSP, .FULL, and .FULLP pseudo-ops, specifying the type of output, must appear before any object code generating statements (excluding .TITLE and COMMENTS), otherwise the line will be flagged and ignored. Once one of these four pseudo-ops is specified, the user is not allowed to change output modes.

b. Any options provided for in the address field of the .ABS and .ABSP are useful only if the output device is paper tape.

3.2.1 .ABSP, .ABS

Label Field	Operation Field	Address Field
Not used	.ABSP	NLD or ␣
Not used	.ABS	NLD or ␣

Both of the absolute pseudo-ops cause 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.

The .ABSP pseudo-op causes all memory referencing instructions whose addresses are in a different page to be flagged as bank errors. A DBA instruction is executed by the absolute loader before control is given to the user program. Addresses which have bit 5 on will signal the processor to use the index register to compute effective addresses.

The .ABS pseudo-op does not flag memory referencing instructions whose addresses are in a different page. An EE7 instruction is executed, and control is given to the user in bank addressing mode. All indexing instructions (see Appendix B) are disabled and executed as I/O transfer instructions and complete bank addressing of 8K is allowed. The processor will interpret bit 5 of all memory referencing instructions as the high order address bit. A listing of the Absolute Binary Loader is given in Appendix F.

3.2.2 .FULL, .FULLP Pseudo-ops

Label Field	Operation Field	Address Field	(Only useful if output is paper tape)
Not used	.FULL	Not used	
Not used	.FULLP	Not used	

The .FULL and .FULLP pseudo-ops cause full binary mode output to be produced. The program is assembled as uncheckedsummed 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 output as the last word.

Regardless of which pseudo-op (.FULL or .FULLP) the user specifies, he must always execute a DBA instruction if he wishes to use indexing in his program because depressing I/O reset before the user program is loaded causes the PDP-15 processor to enter bank addressing mode.

The only difference between the .FULL and .FULLP pseudo-ops is that memory references across page boundaries are flagged in .FULLP mode; in .FULL mode they are not.

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.

The following two pseudo-ops enable relocation mode switching. They can be used anywhere and as often as the programmer wishes in a relocatable program. If these pseudo-ops are used in an absolute (.ABS, .ABSP, .FULL, .FULLP) program, they will be flagged (I-ignored). These pseudo-ops will be most useful for the user who has a VT15 display. The VT15 has its own processor and uses 13-bit addresses.

Mnemonic	Description
.EBREL	Enable bank mode relocation Relocatable programs are normally in PDP-15 mode (12-bit relocation). This pseudo-op will cause a data word to be output to the Linking Loader having an octal code of 31_8. This octal code will signal the Linking Loader to treat all 03 loader codes as 13-bit relocatable. The data word will be ignored by the Linking Loader. Addresses having 13-bits will not be flagged while in this mode.
.DBREL	Disable bank mode relocation A data word is output having a Linking Loader code of 32_8. This code will signal the loader to treat all 03 codes as 12-bit relocation (normal PDP-15 mode); the data word will be ignored.

NOTE

The previous mode is not saved when an .EBREL is encountered; for this reason, a .DBREL pseudo-op goes directly to PDP-15 relocation regardless of previous mode.

3.3 SETTING THE LOCATION COUNTER (.LOC)

Label Field	Operation Field	Address Field
Not used	.LOC	Predefined 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 PASS 2. 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
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.

Flag	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

Flag	Source Program	Generated Value (Octal)	Radix in Effect
	→| 275	000423	10
	→| .OCT		
	→| 76	000076	8
N	→| 85	000125	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 PASS 2. 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
Used 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-15 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↵
```

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 nonprinting 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 filled with zeros.

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 output.

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.

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).

Example:

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-15 generates for a given line of text.

Example:

→| .ASCII →| 'ABC'<015><12>'DEF

Generated Coding

Word Number	Octal	Binary			
Word 1	406050	1000001	10000010	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  →|D             /A is an internal global
D →| JMS* →| B            /These two instructions reference
  →| JMS* →| C            /External symbols indirectly
     .END
```

The .GLOBL statement may appear anywhere within the program.

Each external symbol causes an additional word to be reserved in the user program. This word will be used by the Linking Loader to store the actual address at load time.

The example above is assembled as follows:

Flag	Location	Word Value	.GLOBL B,C
	000000 R	200001 R	A LAC D
	000001 R	120003 R	D JMS* B
	000002 R	120004 R	JMS* C
		000001	.END D
	000003 R	000003 *E	
	000004 R	000004 *E	

The values for locations 3 and 4 will be put in by the Linking Loader.

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-op 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	

Source Code	Generated Object Code
→| 5	000005
	000004
	000003
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-15 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.

Pseudo-op	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 (described in Chapter 4). 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,X
                        →| DAC    →| ARG3
(Terminating Line)      →| .ENDM
```

The pseudo-op .DEFIN 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 and the dummy arguments must be legal MACRO-15 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, five characters per two 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 they may be 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, it will be flagged as undefined.

If index register usage is desirable, it should be specified in the body of the definition, not in the argument string.

```
.DEFIN     XUSE,A,B,C
LAC  A
DAC  B,X
LAC  C
.ENDM
```

If .ASCII or .SIXBT is used in the body of a macro, a slash (/) or number sign (#) 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). A dummy argument name should not inadvertently be used as part of the text string.

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

4.3 MACRO CALLS

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

a. space (␣)

b. tab (→|)

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:

```
        ⇥ MAC ⇥ REAL1,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; this label 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-15 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-15 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. Like other symbols, they are entered into the symbol table as they are required.

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:

```
                        Level 1
→| .DEFIN →| LEVEL1,A,B
→| LAC →| A
→| TAD →| B             Level 2
→| .DEFIN →| LEVEL2,C,D
→| ISZ →| C
→| DAC →| D             Level 3
→| :DEFIN →| LEVEL3,E,F
→| AND →| E
→| XOR →| F
→| .ENDM                                  LEVEL 3  .ENDM
→| DAC →| X
→| .ENDM                                  LEVEL 2  .ENDM
→| DAC →| Y
→| .ENDM                                  LEVEL 1  .ENDM
```

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 irrelevant 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, calling the level n macro 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:

```
→| .DEFIN →| MAC1,A,B,C,D,E
→| LAC →| A
→| TAD →| B
→| MAC2 →| C,D                    /EMBEDDED CALL
→| DAC →| E
```

```
→| .ENDM
→| .DEFIN →| MAC2,A,B          /DEFINITION OF EMBEDDED CALL
→| XOR →| A
→| AND →| B
→| .ENDM
```

The call

```
→| MAC1 →| TAG1,TAG2, (400, (777, TAG3
```

causes generation of

```
→| LAC →| TAG1
→| TAD →| TAG2
→| MAC2 →| (400, (777
→| XOR →| (400
→| AND →| (777
→| DAC →| 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:

```
→| .DEFIN →| MAC,A,B,C
→| LAC →| A
→| TAD →| B
→| DAC →| C
→| MAC →| A,B,C          /RECURSIVE CALL
→| .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; 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
      →| .DEFIN →| MAC,C,D
```

```
       →| LAC →| C
       →| DAC →| D
A= A + 1
       →| .IFNZR →| B-A
       →| MAC →| SAVE,TEMP            /RECURSIVE CALL
       →| .ENDC
       →| .ENDM
```

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

Example:

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

OPERATING PROCEDURES

5.1 INTRODUCTION

Detailed descriptions of the assembler calling procedure, command string format, general operating procedures and printouts are given in this chapter.

5.2 CALLING PROCEDURE

The MACRO-15 Assembler is called by typing MACRO↵ after the Monitor's $ request. When the Assembler has been loaded, it identifies itself by typing:

MACRO-15 VNN ↵

on the Teletype and waiting for command string lines from the user.

5.3 GENERAL COMMAND CHARACTERS

The following characters are frequently used in the entry and control of MACRO programs.

Character	Printout	
RUBOUT	(Echoes \)	delete single character
CTRL U	(Echoes @)	delete current line
CTRL P	(Echoes ↑P)	a. If the input file is sectioned into separate units ending with a .EOT, ready the input device with the next section and type CTRL P. b. If paper tape input, or sectional input, ready the input device with the next pass and type ↑P. c. If the Assembler is not waiting for more input, or is not waiting to start the next pass, typing ↑P will cause the Assembler to restart at PASS 1.
CTRL D	(Echoes ↑D)	If the user specifies the Teletype as the input parameter device, he can delimit the parameter code by typing control D(↑D). MACRO will respond with EOT↑P. The user should then ready the input device assigned to .DAT-11 and type ↑P. MACRO will immediately begin assembling programs.

5.4 COMMAND STRING

The command string format consists of a string of options, followed by a left arrow, followed by the program name, followed by a terminator.

OPTIONS ← FILE NAME

The format for the option string is flexible; that for the program name after the left arrow is fixed. Some examples of the command string are given below. Terminating the command string with a carriage return will cause MACRO to re-initialize itself to PASS 1 at the completion of assembly. Terminating the command string with ALT MODE will cause a return to the MONITOR at the end of assembly.

Example 1:

```
P,L,S,B←FILE
```

Example 2:

```
C A B L,S,N,P V P,P,P HELLO  G E ←FILE1
```

The option designators may be typed in any sequence so long as they appear on the same line. A file name must be typed. All characters to the left of the left arrow which do not represent valid options are ignored.

Example 3:

```
←FILE1
```

No options are required in the command string. If no options are specified, it is assumed that the programmer is assembling for errors, therefore, all assembly errors are printed on the Teletype. If a command string error occurs, the whole line must be retyped, starting with the command string options.

5.4.1 Program Name

A name can be any of the valid symbol characters and can appear in any order.

Examples:

PROPER NAME	EXTENSION
123456	789
ABCDEF	GHI
J	K
. . . . . .	. . .
. %	. . .

5.4.2 Options

As illustrated in the examples, the options may be used in any combination or not at all. If no options are desired, ← is sufficient and the sole output will be assembly error messages on the Teletype. The following table shows the action and the default of the options.

Option	Action	Default Action
B	Generate a binary file	A binary file is not generated.
L	Generate a listing file on the requested output device.	A listing file is not generated (see options N,C).
P	Before assembly begins read program parameters from DAT SLOT-10. The device assigned to DAT-10 must be non-file-oriented. The code read from DAT SLOT-10 is read only once; for this reason only direct assignments should be used.	No parameters, begin assmebly immediately after command string termination.
N	Number each source line (decimal). If this option is used, it is not necessary to type the L option.	Source lines are not numbered.
A	Print symbols at end of PASS 2 in alpha-numeric sequence	Symbols are not printed in alpha-numeric sequence.
V	Print symbols at end of PASS 2 in value sequence.	Symbols are not printed in value sequence. (If neither option V nor A is requested, symbols are not printed.)
S	Same as selecting both A and V above.	Symbols are not printed.
C	Program areas that fall between unsatisfied conditionals are not printed. It is not necessary to type the L option if this option is used.	All source lines are printed.
G	Print only the source line of a macro expansion. It is not necessary to type L option.	Generate printouts for macro expansions and expandable pseudo-ops (e.g., REPT)
X	At completion of PASS 2, PASS 3 is loaded to perform the cross-referencing operation, it is not necessary to type the L or N option if this option is used. At completion of PASS 3 the Assembler will call in PASS 1 and 2, to continue assembling programs. If the command string was terminated by an ALT MODE, control will return to the Monitor at the end of assembly.	A cross-reference is not provided and PASS 3 is not called in.

5.5 ASSEMBLY LISTINGS

If the user requests a listing via the 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.

Line Number	Error Flags	Location	Address Mode	Object Code	Address Type	Source	Statement
XXXX	XXX	XXXXX	[R] [A]	XXXXXX	[R] [A] [E]	X	X

where:

Line Number = Each source line is numbered (decimal), comments lines and generated lines are not included. Lines are not numbered unless the X, or N option is specified.

Flags = Errors encountered by the assembler

Location = Relative or absolute location assigned to the object code.

Address Mode = Indicates the type of user address.

A = absolute
R = relocatable

Object Code = The contents of the location (in octal)

Address Type = Indicates the classification of the object code.

A = absolute
R = relocatable
E = external

Variable locations, and object codes assigned for literals and external symbols are listed following the program.

5.6 SYMBOL TABLE OUTPUT

At the end of PASS 2, the symbol table may be output. If the A option is used, the table will be printed in alphanumeric sequence; if the V option is used, the symbol table will be printed in numeric value sequence; if the S option is used, the symbol table will be output in both alphanumeric and numeric sequence. The format is as follows:

Symbol	Value	Type
SYMBL1	XXXXX	E
SYMBL2	XXXXX	R
DIRECT	XXXXXX	A

The Xs represent the value assigned to the symbol. This is usually the location where the value is defined. Note that for SYMBL1 and SYMBL2 there are five Xs but that there are six Xs for the symbol DIRECT. Symbols having six octal numbers to represent their values are directed assignments.

The symbol table shows the type of symbol:

A = absolute

R = relocatable

E = external

Locations assigned to variables immediately precede the last object code producing statement in the assembled program. Locations and object codes assigned for literals and external symbols are listed immediately following the variables; if no variables are used in the program, they immediately follow the program.

5.7 RUNNING INSTRUCTIONS

When the Assembler is ready,

a. Place the source program to be assembled on the appropriate input device. (If paper tape, push the tape-feed button to clear the end-of-tape flag.)

b. Type the command string.

5.7.1 Paper Tape Input Only

The following steps are required when the source program is encountered in the paper tape reader:

a. At the end of PASS 1, MACRO types

```
END PASS 1
↑P
```

b. Replace the source tape in the reader, pushing the tape-feed button to clear the end-of-tape flag.

c. Type CTRL P to start PASS 2.

At the end of PASS 2, PASS 3 will be loaded by the Assembler to perform the cross-referencing operation. At completion, PASS 1 and 2 will be reloaded to assemble additional programs.

5.7.2 Cross-Reference Output

When a cross reference output is requested, the symbols are listed in alphabetic sequence. The first address after the symbol is the location where the symbol is defined. All subsequent locations represent the line number (decimal) where the symbol was referenced. Leading zeros are suppressed for the cross-reference symbol table. Ten locations are printed on one line and subsequent locations are continued on the next line.

Example:

```
PAGE
A        1        XXXXX  XXXXX...  ....XXXXX
         XXXXX    XXXXX
B        5000     XXXXX
SYMBOL   100      XXXXX
```

5.8 PROGRAM RELOCATION

The normal output from the MACRO-15 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, is added to it. This relocation constant, is added at load time by linking the loader; it 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), which are absolute (marking these non-relocatable words with an A) and which are external (marking these with an E). The rules that the Assembler follows to determine whether a storage word is absolute or relocatable are as follows.

a. If the address is a number (not a symbol), the address 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 PASS 1.

e. .LOCATION counter reference) .GET 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 relocation error.

If a relocatable program exceeds 4K, the following warning message will be typed at the end of PASS 2:

*WARNING*PROG>4K

5.9 ERROR CONDITIONS AND RECOVERY PROCEDURES

Printout	Recovery Procedure
IOPS 4	Device is not ready. Ready device and type CTRL R (↑R)
	IOPS 0-43 unrecoverable I/O error. Control returns to Monitor (see Monitors manual).

5.9.1 Restart Control Entries

CTRL P	Restart Assembler, if running
CTRL C	Return to Monitor

5.10 ERROR DETECTION

MACRO-15 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, or, if the lines are numbered, between the line number and the location. The following table shows the error flags and their meanings.

Flag	Meaning
A	Error in direct symbol table assignment; assignment ignored.
B	a. Memory bank error (program segment too large) b. Page error - the location of an instruction and the address it references are on different pages.
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; will also occur on undefined .END value.
I	Line ignored. a. Relocatable pseudo-op in absolute program ←relocatable b. Redundant pseudo-op c. Absolute pseudo-op in relocatable program

Flag	Meaning
L	a. Literal phasing error. Literal encountered in PASS 2 does not equal any literal encountered in PASS 1. b. Nested literals are illegal.
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.
O	Operand error, instruction cannot have an operand.
P	Phase error. PASS 1 value does not equal PASS 2 value of a symbol. PASS 1 value will be used.
Q	Questionable line. If the address field has been delimited by a space or tab and is followed by another symbol.
R	Possible relocation error.
S	Symbol error. An illegal character was encountered and ignored.
T	Tag error a. X used in tag field. b. An illegal character was encountered in tag field.
U	Undefined symbol.
W	Line overflow during macro expansion.
X	a. Illegal usage of macro name. b. Illegal use of index register.

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.
WARNING*PROG<4K		Relocatable program exceeds page boundary (it is greater than 4K in length).

CHARACTER SET

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

*Illegal as source, except in a comment or text. All other characters are illegal to MACRO-15 and are flagged and ignored.

APPENDIX B

PERMANENT SYMBOL TABLE

Operate

OPR	740000
NOP	740000
CMA	740001
CML	740002
OAS	740004
RAL	740010
RAR	740020
IAC	740030
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
SWHA	742030
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	640444
LRSS	660500
LLSS	660600
ALSS	660700
GSM	664000

IOTs

IOT	700000
IORS	700314
DBK	703304
DBR	703344
IOF	700002
ION	700042
CAF	703302
RES	707721

Memory Reference

CAL	000000
DAC	040000
JMS	100000
DZM	140000
LAC	200000
XOR	240000
ADD	300000
TAD	340000
XCT	400000

Memory Reference (Cont)

ISZ	440000
AND	400000
SAD	540000
JMP	600000

Automatic Priority Interrupt Type KF09A

SPI	705501
ISA	705504

Memory Extension Control Type KE09B

SEM	707701
EEM	707702
LEM	707704

Index Instructions Which Take an Immediate Nine-bit Operand

AAC	723000
AAS	720000
AXR	737000
AXS	725000

Index and Limit Register Instructions Which do not use Operands

CLLR	736000
PAL	722000
PAX	721000
PLA	730000
PLX	731000
PXA	724000
PXL	726000

Mode Switching

EBA	707724
DBA	707722

Index Register Value

X	10000

APPENDIX C

MACRO-15 CHARACTER INTERPRETATION

Character Name	Character Symbol	Function
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	,	An argument delimiter in MACRO definitions or an exclusive OR operator.
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
Line feed		
Form feed		
Vertical tab		

Character	Function
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-15 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 .ABSP	3.2.1 3.2.1	→| .ABS →| NLD ↵ →| .ABSP →| NLD ↵	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.
.DBREL	3.2	→| .DBREL ↵	Enable bank mode relocation.
.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.
.EBREL	3.2	→| .EBREL ↵	Disable bank mode relocation.
.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.

Pseudo-op	Section	Format	Function
.FULL .FULLP	3.2.2 3.2.2	→| .FULL ↵ →| .FULLP ↵	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.
.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-15 outputs the address of last location plus one occupied by the object program.
.TITLE	3.1	→| .TITLE␣name and/or ↵ any comments	The first six legal symbol characters are printed as header of the program listing. A space, tab, or a carriage return will delimit the name.

APPENDIX E

SUMMARY OF SYSTEM MACROS

System macros (Monitor commands) are defined in the Monitor manual, and are summarized here for the convenience of the PDP-15 programmers.

System macros are predefined to MACRO-15. 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 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

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

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

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-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-file

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

d = 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

*May be used with any non-file-structured mass storage device.

To <u>delete</u> a file

```
→| .DLETE␣a,d
```

where a = .DAT slot number in octal

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

To <u>rename</u> 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.

To <u>determine</u> 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 <u>read</u> 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 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

```
CLOAD       PAGE    1

                                 /COPYRIGHT 1969, DIGITAL EQUIPMENT CORP., MAYNARD, MASS.
                                 /
                                 /PDP-15/10 HARDWARE READIN LOADERS
                                 /
                                 /DEFINING %LOW PRODUCES THE LOW SPEED VERSION
                                 /OTHERWISE, THE HIGH SPEED VERSION IS PRODUCED.
                                 /
                                 /LOW SPEED READER VERSION:
                                 /HARDWARE READIN TO 7700 (17700 IF 8K). WHEN IT HALTS;
                                 /PLACE BINARY PROGRAM TAPE IN LOW SPEED READER
                                 /AND PRESS START, WITH BANK/PAGE MODE SWITCH IN PAGE POSITION.
                                 /
                                 /HIGH SPEED READER VERSION:
                                 /HARDWARE READIN TO 7720 (17720 IF 8K). WHEN IT HALTS;
                                 /PLACE BINARY PROGRAM TAPE IN HIGH SPEED READER
                                 /AND PRESS START, WITH BANK/PAGE MODE SWITCH IN PAGE POSITION.
                                 /
                                 /LOADER HALTS:
                                 /
                                 /AC=777777 - PROGRAM LOADED.
                                 /AC=NONZERO - CHECKSUM ERROR ON LAST BLOCK LOADED.
                                 /        REPOSITION TAPE AT BLANK FRAME PRIOR TO
                                 /        BEGINNING OF LAST BLOCK AND PRESS START
                                 /        TO REREAD.
                                 /        TO IGNORE ERROR, PRESS CONTINUE.
                                 /
                 703302          CAF=703302
                 700101          RSF=700101
                 700144          RSB=700144
                 700112          RRB=700112
                 700301          KSF=700301
                 700312          KRB=700312
                 700322          KRS=700322
                                         .FULL
                 700101          SKPFLG=RSF
                 700144          RDSLCT=RSB
                 700112          RDBFR=RRB
        17720                            .LOC 17720
                                         .IFDEF %LOW
                                 SKPFLG=KSF
                                 RDSLCT=KRS
                                 RDBFR=KRB
                                         .LOC 17700
                                         .ENDC
        17720    703302                  CAF                  /CLEAR FLAGS
        17721    157756          LDNXBK  DZM LDCKSM           /CHECKSUMMING LOCATION
        17722    117746                  JMS LDREAD           /GET A WORD
        17723    057757                  DAC LDSTAD
        17724    741100                  SPA                  /BLOCK HEADING=LOADING ADDRESS
        17725    617742                  JMP LDXFR            /START BLOCK
        17726    117746                  JMS LDREAD
        17727    057760                  DAC LDWDCT           /WORD COUNT (2'S COMPLEMENT)
        17730    117746                  JMS LDREAD
        17731    117746          LDNXWD  JMS LDREAD
        17732    077757                  DAC* LDSTAD          /LOAD DATA INTO
```

```
     17733     457757          ISZ LDSTAD          /MEMORY
     17734     457760          ISZ LDWDCT          /FINISHED LOADING
     17735     617731          JMP LDNXWD          /NO
     17736     357756          TAD LDCKSM          /ADD INTO CHECKSUM
     17737     740200          SZA
     17740     740040          HLT                 /CHECKSUM ERROR
     17741     617721          JMP LDNXBK
     17742     057760  LDXFR   DAC LDWDCT
     17743     457760          ISZ LDWDCT
     17744     637757          JMP* LDSTAD         /EXECUTE START ADDRESS
     17745     750041          CLC!HLT             /MANUALLY START USER PROGRAM
     17746     000000  LDREAD  0
                               .IFDEF %LOW
                               LAW -3
                               DAC LDCTR
                               DZM LDTMP
                               .ENDC
     17747     357756          TAD LDCKSM
     17750     057756          DAC LDCKSM
     17751     700144  LDRDA   RDSLCT
     17752     700101          SKPFLG
     17753     617752          JMP .-1             /WAIT FOR READER
     17754     700112          RDRFR               /READ BUFFER
                               .IFDEF %LOW
                               TAD LDMSK           /BINARY FRAME
                               SPA!CLL             /YES
                               JMP LDRDA           /NO
                               TAD LDTMP
                               ISZ LDCTR
                               SKP!RTL
                               .ENDC
     17755     637746          JMP* LDREAD
                               .IFDEF %LOW
                               RTL                 /ACCUMULATE 3 FRAMES
                               RTL                 /INTO 1 BINARY WORD
                               DAC LDTMP
                               JMP LDRDA
                       LDCTR   0                   /PACK COUNTER
                       LDTMP   0                   /BINARY WORD
                       LDMSK   777600              /BINARY FRAME MASK
                               .ENDC
     17756     000000  LDCKSM  0                   /CHECKSUM
     17757     000000  LDSTAD  0                   /LOADING/STARTING ADDRESS
     17760     000000  LDWDCT  0                   /WORD COUNT
               000000          .END
                                 NO ERROR LINES
```

APPENDIX G
SYMBOL TABLE SIZES

The following symbol table sizes are for 8K systems with the full complement of skip IOTs 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)

MACRO-15 INDEX

HOW TO OBTAIN SOFTWARE INFORMATION

Announcements for new and revised software, as well as programming notes, software problems, and documentation corrections are published by Software Information Service in the following newsletters.

Digital Software News for the PDP-8 Family
Digital Software News for the PDP-9/15 Family
PDP-6/PDP-10 Software Bulletin

These newsletters contain information applicable to software available from Digital's Program Library.

Please complete the card below to place your name on the newsletter mailing list.

Questions or problems concerning DEC Software should be reported to the Software Specialist at your nearest DEC regional or district sales office. In cases where no Software Specialist is available, please send a Software Trouble Report form with details of the problem to:

Software Information Service
Digital Equipment Corporation
146 Main Street, Bldg. 3-5
Maynard, Massachusetts 01754

These forms, which are available without charge from the Program Library, should be fully filled out and accompanied by teletype output as well as listings or tapes of the user program to facilitate a complete investigation. An answer will be sent to the individual and appropriate topics of general interest will be printed in the newsletter.

New and revised software and manuals, Software Trouble Report forms, and cumulative Software Manual Updates are available from the Program Library. When ordering, include the document number and a brief description of the program or manual requested. Revisions of programs and documents will be announced in the newsletters and a price list will be included twice yearly. Direct all inquiries and requests to:

Program Library
Digital Equipment Corporation
146 Main Street, Bldg. 3-5
Maynard, Massachusetts 01754

Digital Equipment Computer Users Society (DECUS) maintains a user Library and publishes a catalog of programs as well as the DECUSCOPE magazine for its members and non-members who request it. For further information please write to:

DECUS
Digital Equipment Corporation
146 Main Street
Maynard, Massachusetts 01754

Send Digital's software newsletters to:

Name ______________________________
Company Name ______________________
Address ___________________________
____
(zip code)

My computer is a

PDP-8/I ☐ PDP-8/L ☐
LINC-8 ☐ PDP-12 ☐
PDP-9 ☐ PDP-15 ☐
PDP-10 ☐ OTHER ☐ Please specify ______________

My system serial number is ______________ (if known)

MACRO-15 ASSEMBLER
PROGRAMMERS REFERENCE MANUAL
DEC-15-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.

Please comment on this manual's completeness, accuracy, organization, usability, and readability.

Did you find errors in this manual?

How can this manual be improved?

DEC also strives to keep its customers informed of current DEC software and publications. Thus, the following periodically distributed publications are available upon request. Please check the appropriate boxes for a current issue of the publication(s) desired.

- ☐ Software Manual Update, a quarterly collection of revisions to current software manuals.
- ☐ User's Bookshelf, a bibliography of current software manuals.
- ☐ Program Library Price List, a list of currently available software programs and manuals.

Please describe your position.

Name Organization

Street Department

City State Zip or Country