DEC-10-AMZC-D

MACRO-10 ASSEMBLER
PROGRAMMER'S REFERENCE MANUAL

DIGITAL EQUIPMENT CORPORATION • MAYNARD, MASSACHUSETTS

1st Edition April 1967
2nd Printing October 1967
3rd Edition (Rev) August 1968
4th Edition (Rev) June 1969
5th Edition (Rev) October 1969
6th Edition (Rev) August 1970
7th Edition (Rev) April 1972

Copyright © 1967, 1968, 1969, 1970, 1971, 1972 by
Digital Equipment Corporation

The material in this manual is for information purposes and is subject to change without notice.

The following are trademarks of Digital Equipment Corporation, Maynard, Massachusetts:

DEC	PDP
FLIP CHIP	FOCAL
DIGITAL	COMPUTER LAB

CONTENTS

CHAPTER 1	INTRODUCTION	205
1.1	MACRO-10 LANGUAGE - STATEMENTS	206
1.2	INSTRUCTION WORD FORMATS	206
1.2.1	Primary Instruction Format	207
1.2.2	Input/Output Instruction Format	208
1.3	COMMUNICATION WITH MONITORS	209
1.4	OPERATING PROCEDURES	209
1.5	MACRO STATEMENTS	209
1.5.1	Symbols	209
1.5.2	Labels	210
1.5.3	Symbolic Addresses	210
1.5.4	Operators	211
1.5.5	Symbolic Operators	211
1.5.6	Operands	212
1.5.7	Symbolic Operands	212
1.5.8	Comments	213
1.6	STATEMENT PROCESSING	213
1.6.1	Order of Statement Evaluation	214
1.6.2	Order of Expression Evaluation	214
1.7	USER-DEFINED SYMBOLS	215
1.7.1	Direct Assignment Statements	215
1.7.2	Local and Global Symbols	216
1.7.3	Deleted Symbols	217
1.8	NUMBERS	218
1.8.1	Arithmetic and Logical Operations	219
1.8.2	Evaluating Expressions	219
1.8.3	Numeric Terms	220
1.8.4	Binary Shifting	221
1.8.5	Left Arrow Shifting	222
1.8.6	Floating Point Decimal Numbers	222
1.8.7	Fixed Point Decimal Numbers	222
1.9	ADDRESS ASSIGNMENTS	223
1.9.1	Setting and Referencing the Location Counter	224
1.9.2	Indirect Addressing	224
1.9.3	Indexing	224
1.10	LITERALS	225

CHAPTER 2 MACRO-10 ASSEMBLER STATEMENTS - PSEUDO-OPS 227

2.1 ADDRESS MODE: RELOCATABLE OR ABSOLUTE 227

2.1.1 Relocation Before Execution - PHASE and DEPHASE Statements 229

2.2 NAMING PROGRAMS 230

2.2.1 Program Subtitles 231

2.3 PROGRAM ORIGIN 231

2.3.1 HISEG Statements - The HISEG Pseudo-Op Statement 232

2.3.2 TWOSEG Statements 232

2.4 ENTERING DATA 233

2.4.1 RADIX Statements 233

2.4.2 Entering Data Under the Prevailing Radix 234

2.4.3 DEC and OCT Statements 234

2.4.4 Changing the Local Radix for a Single Numeric Term 235

2.4.5 RADIX 50 Statement 236

2.4.6 EXP Statement 236

2.4.7 Z Statement 236

2.5 INPUT DATA WORD FORMATTING 236

2.5.1 BYTE Statement 236

2.5.2 POINT Statement - Handling Bytes 237

2.5.3 IOWD Statement: Formatting I/O Transfer Words 239

2.5.4 XWD Statement: Entering Two Half-Words of Data 239

2.5.5 Text Input 240

2.5.5.1 ASCII, ASCIZ, and SIXBIT Statement 240

2.5.6 Reserving Storage 241

2.5.6.1 Reserving a Single Location 242

2.5.7 VAR Statements 243

2.5.8 BLOCK Statements 243

2.5.9 END Statements 243

2.5.10 LIT Statements 244

2.5.11 Multi-Program Assembly 244

2.5.12 PASS2 Statements 245

2.5.13 PURGE Statements 245

2.5.14 XPUNGE Statements 245

2.5.15 Linking Subroutines 246

2.5.15.1 EXTERN Statements 246

2.5.15.2 INTERN Statements 247

2.5.15.3 ENTRY Statements 247

2.6	SUPPRESSION OF SYMBOLS	248
2.6.1	SUPPRESS SYMBOL Statement	248
2.6.2	ASUPPRESS Statement	248
2.6.3	Listing Control Statements	249
2.7	CONDITIONAL ASSEMBLY	252
2.8	ASSEMBLER CONTROL STATEMENTS	253
2.8.1	REPEAT Statements	253
2.8.2	OPDEF Statements	254
2.8.3	SYN Statements	255
2.8.4	Extended Instruction Statements	256
2.9	MULTI-FILE ASSEMBLY	257
2.9.1	UNIVERSAL Name	257
2.9.2	SEARCH Name	258
CHAPTER 3	MACROS	259
3.1	DEFINITION OF MACROS	259
3.2	MACRO CALLS	260
3.3	MACRO FORMAT	261
3.4	CREATED SYMBOLS	262
3.5	CONCATENATION	263
3.6	DEFAULT ARGUMENTS	264
3.7	INDEFINITE REPEAT	265
3.8	NESTING AND REDEFINITION	266
3.8.1	ASCII Interpretation	268
CHAPTER 4	ERROR DETECTION	269
4.1	SINGLE-LETTER ERROR CODES	269
4.2	ERROR MESSAGES	275
4.2.1	LOOKUP Errors	277
4.2.2	MACRO I/O Error Messages	278
CHAPTER 5	RELOCATION	279
CHAPTER 6	ASSEMBLY OUTPUT	283
6.1	ASSEMBLY LISTING	283
6.2	BINARY PROGRAM OUTPUT	284
6.2.1	Relocatable Binary Programs - LINK Format	284

6.2.1.1 LINK Formats for the Block Types 285
6.2.2 Absolute Binary Programs 288
6.2.2.1 RIM10B Format 288
6.2.2.2 RIM10 Format 289
6.2.2.3 RIM Format 290
6.2.2.4 END Statements 290

CHAPTER 7 PROGRAMMING EXAMPLES 293

APPENDIX A OP CODES, PSEUDO-OPS, AND MONITOR I/O COMMANDS 307
A.1 ASSEMBLER PSEUDO-OPS AND MONITOR CO COMMANDS 307
A.2 MACHINE MNEMONICS AND OCTAL CODES 309

APPENDIX B SUMMARY OF PSEUDO-OPS 311
B.1 PSEUDO-OPS 311
B.1.1 Conditional Assembly Statements 313

APPENDIX C SUMMARY OF CHARACTER INTERPRETATIONS 315

APPENDIX D STORAGE ALLOCATION 319

APPENDIX E TEXT CODES 323

APPENDIX F RADIX 50 REPRESENTATION 325

APPENDIX G SUMMARY OF RULES FOR DEFINING AND CALLING MACROS 327
G.1 ASSEMBLER INTERPRETATION 327
G.2 CHARACTER HANDLING 327
G.2.1 Blanks 327
G.2.2 Brackets 327
G.2.3 Parentheses 328
G.2.4 Commas 328
G.2.5 Semicolons 328
G.2.6 Carriage Return 328
G.2.7 Back-Slash 328
G.3 CONCATENATION 328

APPENDIX H	OPERATING INSTRUCTIONS	331
H.1	REQUIREMENTS	331
H.2	INITIALIZATION	331
H.3	COMMANDS	332
H.3.1	General Command Format	332
H.3.2	Disk File Command Format	332
H.4	SWITCHES	334

Chapter 1 Introduction

MACRO-10 is the symbolic assembly program for the PDP-10, and operates in a minimum of 7K pure plus 1K impure core memory in all PDP-10 systems. MACRO-10 is a two-pass assembler. It is completely device independent, allowing the user to select standard peripheral devices for input and output files. For example, a terminal can be used for input of the symbolic source program, DECtape for output of the assembled binary object program, and a line printer can be used to output the program listing.

This assembler performs many useful functions, making machine language programming easier, faster, and more efficient. Basically, the assembler processes the PDP-10 programmer's source program statements by translating mnemonic operation codes to the binary codes needed in machine instructions, relating symbols to numeric values, assigning relocatable or absolute core addresses for program instructions and data, and preparing an output listing of the program which includes notification of any errors detected during the assembly process.

MACRO-10 also contains powerful macro capabilities which allow the programmer to create new language elements, thus expanding and

adapting the assembler to perform specialized functions for each programming job.

1.1 MACRO-10 LANGUAGE - STATEMENTS

MACRO-10 programs are usually prepared on a terminal, with the aid of a text editing program, as a sequence of statements. Each statement is normally written on a single line and terminated by a carriage return-line feed sequence. MACRO-10 statements are virtually format free; that is, elements of a statement are not placed in numbered columns with rigidly controlled spacing between elements, as in punched-card oriented assemblers.

There are four types of elements in a MACRO-10 statement which are separated by specific characters. These elements are identified by the order of appearance in the statement, and by the separating, or delimiting, character which follows or precedes the elements.

Statements are written in the general form:

```
label: operator    operand,operand;comments(carriage return-line feed)
```

The assembler converts statements written in the foregoing form and translates them into machine instruction words. The formats used by the machine instructions are described in the following paragraphs.

1.2 INSTRUCTION WORD FORMATS

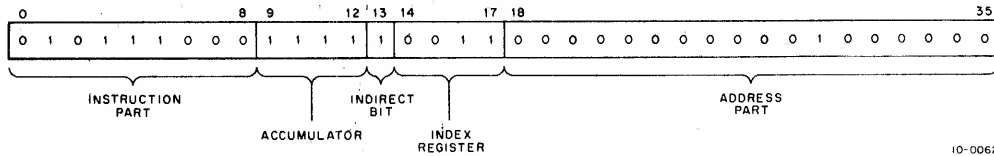
There are two types of machine instruction word formats: primary and input/output.

The PDP-10 machine instructions are fully described in the PDP-10 System Reference Manual and listed alphabetically in Appendix A of this manual. Monitor I/O commands, or programmed operators have the same formats. (See monitor manuals.)

The primary instruction statements may have two operands: (1) an accumulator address and (2) a memory address. A memory address may be modified by indexing and indirect addressing.

1.2.1 Primary Instruction Format

After processing primary instruction statements, the assembler produces machine instructions in the general 36-bit word format shown below:



In general, the mnemonic operation code, or operator, in the symbolic statement is translated to its binary equivalent and placed in bits 0-8 of the machine instruction. The address operand is evaluated and placed in the address part (bits 18-35 of the machine instruction. The assembler assigns sequential binary addresses to each statement as it is processed by means of the location counter. Labels are given the current value of the location counter and are stored in the assembler's symbol table, where the corresponding binary addresses can be found if another instruction uses the same symbol as an address reference.

Any one of 16 possible accumulators may be specified in an instruction by identifying them symbolically or numerically as operands in the statement followed by a comma. The indirect address bit is set to 1 when the character @ prefixes a memory reference. Indexing is specified by writing the index register used in parentheses immediately following the memory reference. (All PDP-10 accumulators, except accumulator 0, may be used as index registers.) Actually, expressions enclosed in parentheses (in the index register position) are evaluated as 36-bit quantities; their halves are exchanged, and then each half is added into the corresponding half of the binary word being assembled. For example, the statements

```
MOVSI AC,(1.Ø)          ;MOVE 1.Ø TO AC
MOVSI AC,(SIXBIT /DSK/)
```

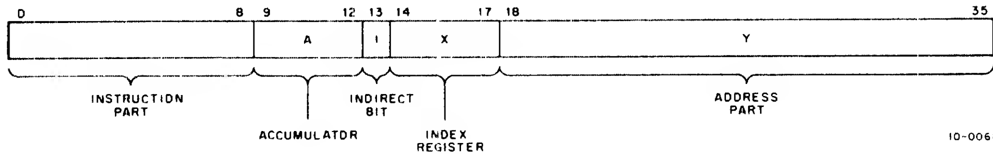
are equivalent to

```
MOVSI AC,2Ø14ØØ         ;MOVE 1.Ø TO AC
MOVSI AC,446353
```

To illustrate this general view of assembler processing, here is a typical symbolic instruction. Assume that AC17, TEMP and XR are defined symbols, with values of 17, 100, and 3, respectively.

```
LABEL:  ADD AC17,@TEMP(XR)    ;STATEMENT EXAMPLE)
```

This is processed by the assembler and stored as a binary machine instruction like this:



The mnemonic instruction code, ADD, has been translated to its octal equivalent, 270, and stored in bits 0-8. The first operand specifies accumulator 17_8. The effective memory address will be found at execution time by adding the contents of index register 3 to the value of TEMP, then taking this value as the address of the word whose address points to the word to be added to AC17.

A comment following a semicolon does not affect the program in any way, but it is printed in the output listing.

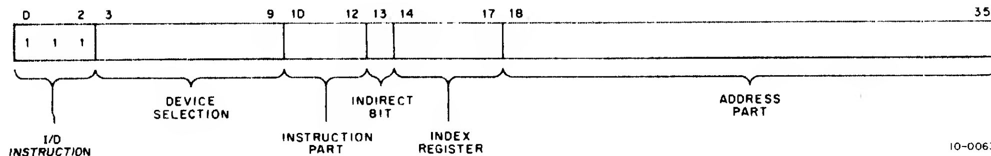
1.2.2 Input/Output Instruction Format

There are eight PDP-10 I/O statements; in each statement the first operand is either a peripheral device number or a device mnemonic (see PDP-10 System Reference Manual for complete list). The second operand is a memory address. For example,

```
READ:  DATAI PTR,@NUM(4))
```

requests that data be read in from a paper-tape reader, to be stored at the indirect, indexed, address given.

The format for I/O instruction words is shown below:



1.3 COMMUNICATION WITH MONITORS

Programs assembled with MACRO-10 which operate under executive control of a monitor must use monitor facilities for device independent I/O services. This is done by means of programmed operators (operation codes 040 through 077) such as CALL. INIT, LOOKUP, IN, OUT, and CLOSE.

Additional monitor commands are available to allow the user program to exercise control over central processor trapping, to modify its memory allocation, and other services, which are described in the monitor programmer's manuals.

Monitor commands are listed in Appendix A.

1.4 OPERATING PROCEDURES

Commands for loading and executing MACRO-10 are contained in Appendix H.

1.5 MACRO STATEMENTS

As previously stated (paragraph 1.1) macro statements consist of a label, an operator, an operand and optional comments.

The assembler interprets and processes these statements, generating one or more binary instructions or data words, or performaing an assembly process. A statement must contain at least one of these elements and may contain all four types. Some statements are written with only one operand; but others may have many. To continue a statement on the following line, the control (CTRL) left arrow (←), echoed as ←↓, is used before the carriage return-line feed sequence (← ↓ or ⟩). Examples of program statements are given in Chapter 7, Figures 7-1 and 7-3.

Statement labels, operators and operands may be represented either numerically or symbolically. The assembler interprets all symbols and replaces them with a numeric (binary) value.

1.5.1 Symbols

The programmer may create symbols to use as statement labels, as operators and as operands. A symbol may consist of any

combination of from one to six characters of the following set:

The 26 letters, A-Z
Ten digits, 0-9
Three special characters: $ (Dollar Sign)
% (Percent)
. (Period)

The foregoing character set is the Radix-50 character set.

Any statement character which is not in the Radix-50 set is treated as a symbol delimiter when encountered by the assembler.

If the first characters of a symbol are numeric, the symbol is treated as through the numeric characters were not present. If the first character is a period, it must not be followed by a digit. Spaces must not be embedded in symbols. A symbol may actually have more than six characters, but only the first six are meaningful to MACRO-10.

MACRO-10 accepts programs written using both upper and lower case letters and symbols (e.g., programs written using the Teletype Model 37). Lower case letters are treated as upper case in symbols; additional special characters, and lower case letters elsewhere, are taken without change.

1.5.2 Labels

A label is the symbolic name created by the source programmer to identify a statement. If present, the label is written as the first item in a statement and is terminated by a colon (:). (Refer to paragraph 1.5.1 for a description of how symbolic names are formed.)

1.5.3 Symbolic Addresses

A symbol used as a label to specify a symbolic address must appear first in the statement and must be immediately followed by a colon (:). When used in this way, a symbol is said to be defined. A defined-symbol can reference an instruction or data word at any point in the program.

A label can be defined with only one value; if a programmer attempts to redefine a label with a different value, the second value is

ignored and an error is indicated (see Chapter 4 for error messages). The following are legal labels:

```
$SUM:
ABC: DEF:          (Both labels are legal)
FOO
```

The following are illegal:

```
7ABC:              (First character must not be a digit.)
LAB :              (Colon must immediately follow label.)
```

If too many characters are used in a label, only the first six characters given are used. For example the label ABCDEFGH: is recognized by the assembler as being ABCDEF:.

Labels are used for programmer reference as addresses for jump instructions, for loops and for debugging.

1.5.4 Operators

An operator may be one of the mnemonic machine instruction codes (see DECsystem-10 System Reference Manual), a command to Monitor, or a pseudo-operation code which directs assembly processing. These assembly pseudo-op codes are described in this manual, and listed with all other assembler defined operators in Appendix A.

Programmers may extend the power of the assembler by creating their own pseudo-operators (see OPDEF pseudo-op).

An operator may be a macro name, which calls a user-defined macro instruction. Like pseudo-ops, macros direct assembly processing; but, because of their unique power to handle repetitions and to extend and adapt the assembly language, macros are considered separately (see Chapter 3). Operators are terminated with a space or tab.

1.5.5 Symbolic Operators

Symbols used as operators must be predefined by the assembler or by the programmer. If a statement has no label, the operator may appear first in the statement, and must be terminated by a space, tab, or carriage return. The following are examples of legal operators:

MOV (A mnemonic machine instruction operator.)
LOC (An assembler pseudo-op.)
ZIP (Legal only if defined by the user.)

1.5.6 Operands

Operands are usually the symbolic addresses of the data to be accessed when an instruction is executed, or the input data or arguments or a pseudo-op or macro instruction. In each case, the interpretation of operands in a statement depends on the statement operator. Operands are separated by commas, and terminated by a semicolon (;) or by a carriage return-line feed.

In the mnemonic machine instruction and UUO call set, if an operand is followed by a comma (spaces in the line are ignored) then the operand is identified as an accumulator (see instruction format description in paragraph 1.2.1). If an operand is not followed by a comma, then it is viewed as an address (either indexed or indirect if negative).

1.5.7 Symbolic Operands

Symbols used as operands must have a value defined by the user. These may be symbolic references to previously defined labels where the argument to be used by this instruction are to be found, or the values of symbolic operands may be constants or character strings. If the first operand references an accumulator, it must be followed by a comma.

```
TOTAL:  ADD AC1,TAG)
```

The first operand, AC1, specifies an accumulator register, determined by the value given to the symbol AC1 by the user. The second operand references a memory location, whose name or symbolic address is TAG. If the user has equated AC1 to 17, and the assembler has assigned TAG to the binary address, 000537, then the assembler inserts 17 in the accumulator field (bits 9-12) and 000537 in the address field (bits 18-35) of this instruction. If an accumulator is not specified, but the operator requires one, accumulator 0 is assumed by default. If an accumulator is specifies by the value $>17_8$, the four least significant bits are used.

1.5.8 Comments

The programmer may add notes to a statement following a semicolon. Such comments do not affect assembly processing or program execution, but are useful in the program listing for later analysis or debugging. The use of angle brackets (<>) should be avoided in comments because they may affect the assembly.

Each line of a program may contain a comment which explains the purpose of the line and any special action it causes. A line may also consist of only a comment; this is usually done at the beginning of each routine or major program section to explain the major flow of control, entry and exit points and any other pertinent information.

1.6 STATEMENT PROCESSING

The assembler has several symbol tables and corresponding search routines. The symbol tables arranged in the order in which they are searched are:

1. Macro Table - This symbol table contains macros, user-defined operator definitions (op-defs) and synonyms (refer to the description of the SYN pseudo-op, paragraph 2.8.3). The macro table is initially empty; it grows as the user defines items.

2. Op-Code Table - This symbol table contains all of the operator-codes (op-codes), the UUO calls and the assembler pseudo-operators (pseudo-ops). Lists of the foregoing items are given in Appendices A and B. The op-code table is generated by the assembler and is of fixed length; it cannot be changed except by reassembling MACRO.

3. User Symbol Table - This symbol table contains all user-defined symbols other than those which are placed in the Macro Table. This table is initially empty; it grows as the user defines items.

4. Mnemonic Table - This table contains the mnemonics for the CALLI, MTAPE and TTCALL UUO's. The mnemonic table is searched only if all other measures fail. Any symbol found in this table is put into the macro table as an op-def as though the user had defined it. Examples of the mnemonics contains by this table are

   a) RESET as defined by the CALLI Ø,Ø
   b) EXIT as defined in CALLI Ø,12
   c) OUTSRT as defined in TTCALL 3,Ø

Internally, the macro table and the user symbol table occupy the same space; however, the entries of each table are easily distinguishable so no confusion takes place.

1.6.1 Order of Statement Evaluation

The following table shows the order in which the assembler searches each statement field:

Label Field	Operator Field	Operand Field
1. Symbol suffixed by colon. If colon is not found, no label is present.	1. Number	1. Number
	2. Macro/OPDEF	2. Symbol
	3. Machine operator	3. Macro/OPDEF
	4. Assembler operator	4. Machine operator
	5. Symbol	5. Assembler operator
	6. CALLI mnemonic	

A single symbol could be used as a label, an operator, or an operand, depending on where it is used.

The assembler first checks the operator field for a number, and if found, assumes that no operator is present. Likewise, if a symbol is not a macro, OPDEF, machine operator or assembler operator, the assembler will search the symbol table. If a defined symbol is found, no operator is present.

If a defined operator appears in an operand field, it must generate at least one word of data. Statements that do not generate data may not be used as part of operand expressions. If a statement used in an operand expressions generates more than one word of data, only the first word generated is meaningful.

1.6.2 Order of Expression Evaluation

Expressions are evaluated in the following order:

- (Unary operator)
↑D, ↑O, ↑B, ↑F, ↑L
B Shift, ← Shift
Logical operators
Multiply/Divide
Add/Subtract

At each level, operations are performed left to right.

1.7 USER-DEFINED SYMBOLS

User-defined symbols are of two types: labels and assignments. Labels are generated by entering a symbol followed immediately by a colon (e.g., TAG:). Symbols used as labels cannot be redefined with a different value once they have been defined. The value of a label is the value of the location counter at the time that the label is defined.

Assignments are used to represent, symbolically, numbers or bit patterns. Assignments ease the coding task in that only one line has to be changed (that containing the assignment) in order to change a number or bit pattern which is used throughout the program. Assignment statements may be changed at any time, the current value of an assignment is the last value given to the symbol used.

1.7.1 Direct Assignment Statements

The macro inserts new symbols with their assigned values directly into the symbol table by using a direct assignment statement of the form,

```
symbol=value)
```

where the value may be a number or expression. Note that the equal sign must immediately follow the symbol. For example,

```
ALPHA= 5)
BETA= 17)
```

A direct assignment statement may also be used to give a new symbol the same value as a previously defined symbol:

```
BETA= 17)
GAMMA= BETA)
```

The new symbol, GAMMA, is entered into the symbol table with the value 17.

The value assigned to a symbol may be changed:

```
ALPHA= 7)
```

changes the value assigned in the first example from 5 to 7.

Direct assignment statements do not generate instructions or data in the object program. These statements are used to assign values so that symbols can be conveniently used in other statements.

1.7.2 Local and Global Symbols

User-defined symbols may be used as local and global symbols in addition to beging used as label and assignment symbols.

Local symbols are symbols which are known only to the program in which they are defined. Two separately assembled macro programs may contain local symbols which have the same mnemonic but different definitions; these programs, however, may be loaded and executed without conflict since the symbols are defined as local to each program.

Global symbols are symbols which can be recognized by programs other than the one in which it is defined. The manner in which a global symbol is written or defined depends on where it is located: in the program in which it is defined or the program in which it is a reference to a symbol defined elsewhere.

Global symbols located in the program in which they are defined must be declared as available to other programs by the use of the pseudo-ops INTERN or ENTRY (see paragraphs 2.5.14.1 and 2.5.14.3) or by the use of the delimiter =: in their definition statement. For example, the symbol FLAG may be declared a global symbols by:

a. INTERN FLAG (the symbol FLAG is declared internal),
b. ENTRY FLAG (identifies the entry point of a library subroutine),
c. FLAG=: 2ØØ (FLAG is given the value 200 and is declared internal).

NOTE

The statement in item c of the foregoing examples (i.e., FLAG=: 2ØØ) is equivalent to the series

```
INTERN FLAG
FLAG= 2ØØ
```

Global symbols located in a program in which they are references to symbols defined in other programs must be declared as external symbols by the use of the EXTERN pseudo-op (see paragraph 2.5.14.1) or a ## suffix. For example, the statement

```
EXTERN FLAG
```

declares the symbol FLAG as an external reference. The statement

```
MOVE Ø,FLAG##
```

also declares the symbol FLAG as an external reference; this statement is the equivalent of the series:

```
EXTERN FLAG
MOVE Ø,FLAG
```

1.7.3 Deleted Symbols

Sometimes a programmer may want to define a symbol in MACRO but not have that symbol typed out by DDT (refer to the *DDT Programmer's Reference Manual*). In such a case, the programmer should define that symbol with a double equal sign:

```
FLAG== 2ØØ↵
```

FLAG will be assigned the value 200 and will be

a. Fully available in MACRO.

b. Available for type-in with DDT (assuming that symbols were loaded for the program containing FLAG).

c. Unavailable for type-out by DDT.

This is equivalent to defining FLAG by:

```
FLAG= 2ØØ↵
```

and then typing

```
FLAG$K          (the symbol $ represents ALT MODE)
```

to DDT.

A symbol may be defined with == and declared internal in the following manner

```
FLAG==:2ØØ↵
```

is equivalent to

```
INTERN  FLAG)
FLAG==2ØØ)
```

The programmer may also want to define a label in MACRO but have the output of the label suppressed in DDT. The following constructions may be used:

```
LABEL:!    LABEL is a suppressed local symbol.
LABEL::!   LABEL is a suppressed internal symbol.
```

1.8 NUMBERS

Numbers used in source program statements may be signed or unsigned, and are interpreted by the assembler according to the radix specified by the programmer, where

$2 \leq radix \leq 10$

The programmer may use an assembler pseudo-op, RADIX, to set the radix for the numbers which follow. If the programmer does not use a RADIX statement, the assembler assumes a radix of 8 (octal) except in the case of the POINT pseudo-op (see paragraph 2.5.2).

The radix may be changed for a single numeric term, by using the qualifier followed by a letter, D (for decimal), O (for octal), B (for binary), or F (for fixed-point decimal fractions). Note that these are not control characters. Thus,

↑D1Ø	is stored as	1Ø1Ø
↑O1Ø	is stored as	1ØØØ
↑B1Ø	is stored as	ØØ1Ø

The qualifier ↑L is used for bit position determination of a numeric value. ↑Ln generates an octal value equal to the number of 0 bits to the left of the leftmost 1, if the numeric value n were stored in a computer word.

Expression	Resultant Value	
		44_8 zero bits
↑LØ	44	0000000000. . . .0000000000

Expression	Resultant Value	41_8 zero bits
↑L5	41	0000000000. . . .0000000101
↑L-1	0	1111111111. . . .1111111111

The suffixes K, M and G may be added to numbers as a shorthand method of specifying the number of zeros which are to follow the given number. The meaning of each suffix is:

a) K, add three zeros (e.g., 5K = 5000),
b) M, add six zeros (e.g., 5M = 5000000),
c) G, add nine zeros (e.g., 5G = 5000000000).

1.8.1 Arithmetic and Logical Operations

Numbers and defined symbols may be combined using arithmetic and logical operators. The following arithmetic and logical operators may be used.

Operator	Meaning
+	Add
-	Subtract
*	Multiply
/	Integer Divide
&	AND
!	Inclusive OR

The assembler computes the 36-bit value of a series of numbers and defined symbols connected by arithmetic and logical operators, truncating from the left, if necessary. The following examples show how these arithmetic and logical operators are written in statements.

```
B=     65+X11-3)
MULI   AC1+7,RHO/31)
MOVE   A+3,BETA-5)
```

Combinations of numbers and defined symbols using arithmetic and logical operators are called expressions.

1.8.2 Evaluating Expressions

When combining elements of an expression, the assembler first performs unary operations (leading + or -), then binary shifts. The logical operations are then done from left to right, followed by multiplications

and divisions, from left to right. Division always truncates the fractional part. Finally, additions and subtractions are performed, left to right. All arithmetic operations are performed modulo 2^{35}.

For example, in the statement:

```
TAG:  TRO 3,1+A&C)
```

the first operand field is evaluated first; the comma terminating this operand indicates that this is an accumulator. In the second operand field, the logical AND is performed first, the result is added to one, and the sum is placed in the memory address field of the machine instruction.

To change the normal order of operations, angle brackets may be used to delimit expressions and indicate the order of computation. Angle brackets must always be used in pairs.

Expressions may be nested to any level, with each expression enclosed in a pair of angle brackets. The innermost expression is evaluated first, the outermost is evaluated last. The following are legal expressions:

```
A+B/5
<<C-D+B-29>*<A-41>>+1
A=<B=<C=10>>
```

1.8.3 Numeric Terms

A numeric term may be a digit, a string of digits, or an expression enclosed in angle brackets. The assembler reduces numeric terms to a single 36-bit value. This is useful when specifying operations such as local radix changes and binary shifts, which require single values.

For example, the ↑D operator changes the local radix to decimal for the numeric term that follows it. The number 23_{10} may be represented by

```
↑D23
↑D<5*2+13>
↑D<TEN*2+THREE>
```

but 23_{10} may not be written,

```
↑D1ØØ-77
```

because the ↑D operator affects only the numeric term which follows it, and in this example the second term (77) is taken under the prevailing radix, which is normally octal.

The B shift operator is preceded by a numeric term (the number to be shifted) andis followed by another term (the bit position of the assumed point). The following are legal:

```
↑F167B17
↑B1ØØ11B8
 566B5
<MARK + SIGN>B<PT-XXV>
```

A bracketed numeric term may be preceded by a + or a - sign.

1.8.4 Binary Shifting

A number may be logically shifted left or right by following it with the letter B, followed by a numeric term, n, representing the bit position in which the right-hand bit of the number should be placed. The numeric term, n, may be any (decimal) bit position, starting with zero and numbering from left to right. If n is not used, B35 is assumed; n is taken as modulo 256 decimal. Thus, the number ↑D10 is stored as 000000 000012; but ↑D10B32 is shifted left three binary positions and stored as 000000 000120; and D10B4 is shifted left 31 positions, so that its rightmost bit is in bit 4 and stored as 240000 000000.

Binary shifting is a logical operation, rather than an arithmetic one.

The following are legal binary shifts:

```
1BØ      400000 000000
1B17     000001 000000
1B35     000000 000001
-1B35    777777 777777   (see explanation below)
-1B53    000000 777777
-1B7Ø    000000 000001
```

Note that the following expressions are equivalent:

```
10B32   ↑O10B32 ≡ 10B <42-10>≡ 10B<  ↑D  <42-10>>≡ 10B<↑ D42-  ↑D10>
```

The unary operators preceding a value are interpreted first by the assembler before the binary shift. A leading plus sign has no effect, but a leading minus sign causes the assembler to shift and then to store the 2's complement.

Binary shifting may operate on numeric terms, as defined in Section 1.3.2.

1.8.5 Left Arrow Shifting

If two expressions are combined with the operator "←", i.e., <m>←<n>, the 36-bit value of expression m is shifted V bits (where V is the value of expression n) in the direction of the arrow (left) if V is positive or against the arrow if V is negative. The effective magnitude of V is that of the address of an LSH instruction.

1.8.6 Floating-Point Decimal Numbers

If a string of digits contains a decimal point, it is evaluated as a floating point decimal number, and the digits are taken radix 10. For example, the statement,

    17.Ø   is stored as 205420 000000.

Floating-point decimal numbers may also be written, as in FORTRAN, with the number followed by the letter E, followed by a signed exponent representing a power of 10. The following examples are valid:

```
NUM1:  17.2E-4↵
NUM2:  3.85E2↵
NUM3:  -567.825E33↵
```

1.8.7 Fixed-Point Decimal Numbers

As shown in Section 1.8, ↑D followed by a numeric term, is used to enter decimal integers.

Fixed-point decimal numbers (mixed numbers) are preceded by ↑F followed by a number (not a numeric term, defined below) which normally contains a decimal point. The assembler forms these fixed-point numbers in two 36-bit registers, the integer part in the first and the fractional part in the second. The value is then stored in one storage word in the object program, the integer part to the left of the assumed binary point, the fractional part to the right.

The binary shift (B) operator is used to position the assumed point. The number ↑F123.45B8 is formed in two registers:

```
000000 000173     (the integer part)
346314 631462     (the fraction part, left-justified)
```

The B operator sets the assumed point after bit 8, so the integer part is placed in bits 0-8, and the fraction part in bits 9-35 of the storage word. In this case, the integer part is truncated from the left to fit the 9-bit integer field. The fraction part is moved into the 27-bit field following the assumed point and is truncated on the right. The result is,

```
173346 314631
   ↑
(assumed point)
```

If a B shift operator does not appear in a fixed-point number, the point is assumed to follow bit 35, and the fractional part is lost.

Fixed-point numbers are assumed to be positive unless a minus sign precedes the qualifier:

```
000000 000173      ↑F123.45
000173 346314      ↑F123.45B17
346314 631462      ↑F123.45B-1

777777 777604      -↑F123.45
777604 431463      -↑F123.45B17
431463 146316      -↑F123.45B-1
```

Negative fixed-point numbers, such as the example above, are assembled as if they were positive numbers, complemented, and then logically shifted.

1.9 ADDRESS ASSIGNMENTS

As source statements are processed, the assembler assigns consecutive memory addresses to the instruction and data words of the object program. This is done by incrementing the location counter each time a memory location is assigned. A statement which generates a single object program storage word increments the location counter by one. Another statement may generate six storage words, incrementing the location counter by six.

The mnemonic instruction and monitor command[1] statements generate a single storage word. However, direct assignment statements and some assembler pseudo-ops do not generate storage words, and do not affect the location

[1]The terms monitor command (as used here) and programmed operator are synonymous.

counter. Other pseudo-ops and macros may generate many words in the object program.

1.9.1 Setting and Referencing the Location Counter

The MACRO-10 programmer may set the location counter by using the pseudo-ops, LOC and RELOC, which are described in Chapter 2. He may reference the location counter directly by using the symbol, point (.). For example, he can transfer to the second previously assigned storage word by writing:

```
JRST .-2)
```

1.9.2 Indirect Addressing

The character @ prefixing an operand causes the assembler to set bit 13 in the instruction word, indicating an indirect address. For an explanation of indirect addressing and effective address calculation, see the *PDP-10 System Reference Manual.*

1.9.3 Indexing

If indexing is used to increment the address field, the address of the index register used is entered in parentheses, as the last part of the memory reference operand. This is normally a symbolic name defined by a direct assignment statement, or an octal number in the range 1-17, specifying 1 of the 15 index registers. However, the address of the index register may be any legal expression or an expression element.

This is a symbolic, indirect, indexed, memory reference:

```
A: ADD 4,@NUM(17))
```

NOTE

The parentheses cause the value of the enclosed expression to be interpreted as a 36-bit word with its two halves interchanged, e.g., (17) is effectively 000017000000_8. The 36-bit value is added to the instruction and may modify it. This is often used to generate right half values from left half expressions; for example, the statement

```
TLO AC,(1BØ)
```

which sets the sign bit.

1.10 LITERALS

In a MACRO statement, a symbolic data reference may be replaced by a direct representation of the data enclosed in square brackets ([]). This direct representation is called a literal. The assembler stores data found within brackets in a Literal table, assigns an address to the first word of the data and inserts that address in the machine instruction.

A literal may consist of more than one statement and may generate more than one word of data. A literal must, however, generate at least one word but no more than 18 words. Literals which consist of only pseudo-ops (such as RADIX) which do not generate data or direct assignments are illegal.

Literals may be nested (i.e., bracketed data within other sets of bracketed data) up to 18 levels.

The following is a simple example of the user of literals. Byte instructions must reference by a byte pointer in this manner:

```
        LDB     4,BP
    BP: POINT   1Ø,A+3,14
```

(POINT is a pseudo-op which sets up a byte pointter word.) A literal can be used to insert the POINT statement directly. For example

```
    LDB 4,[POINT 1Ø,A+3,14]
```

Literals are often used as constants as, for example:

a) PUSH 17,[Ø] (note that Ø generates one word of zero).
b) MOVE L. [3,14]

The following is an example of a multi-line literal:

```
GETCHR: SOSG  IBUF+2                        ;ANY CHARS LEFT?
        PUSHJ P,[IN     N,                  ;NO, READ SOME IN
                 POPJ   P,                  ;NO UNUSUAL CONDITIONS
                 STATZ  N,74ØØØØ            ;CHECK FOR ERRORS
                 JRST   [MOVEI E, [SIXBIT /INPUT ERROR/]
                         JRST ERRPNT]       ;PUBLISH ERROR MESSAGE
                 JRST   ENDFIL]             ;END OF FILE HANDLER
        ILDB  AC,IBUF+1                     ;PICKUP NEXT CHAR
        POPJ  P,
```

NOTE

The closing right square bracket does not terminate the literal if placed after the semicolon.

The excessive use of literals, especially for small subroutines, is not recommended since they use up assembler space at the rate of four locations per data word generated. Literals also make debugging more difficult and may cause page faults in the KI-10 processor virtual memory allocation.

The PDP-6 version of macro (MACRO-6) only permitted literals to contain one statement but it permitted the right bracket to be dropped. Dropping the right bracket is not permitted by MACRO-10.

Two pseudo-ops MLON and MLOFF provide compatibility with old programs. Use of these pseudo-ops is required since

```
MOVE AC,[SIXBIT/TEXT/)
```

is legal in MACRO-6, even though the closing right bracket (]) of the literal has been omitted. In normal mode, MACRO does not allow such an unterminated literal. The pseudo-op

```
MLON
```

is set at the start of each assembly to cause the assembler to consider all code following a left bracket as part of a literal, until such time as the assembler processes a matching right bracket. Thus, carriage-return, line-feed does not end a literal, but rather the programmer must insert a right bracket. The pseudo-op,

```
MLOFF
```

set by the switch /O, places MACRO into the compatibility mode in which literals may occupy only a single line.

The symbol . (current location) is not changed by the use of literals. It retains the value it had before the literal was entered.

Chapter 2
MACRO-10 Assembler Statements—Pseudo-Ops

Assembler statements or pseudo-ops direct the assembler to perform certain assembler processing operations, such as converting data to binary under a selected radix, or listing selected parts of the assembled object program. In this chapter, these assembler processing operations are fully described.

NOTE

> The pseudo-op name must follow the rules for constructing a symbol (refer to Paragraph 1.5.1) and must be terminated by a character other than those listed in Paragraph 1.5.1 as valid symbolic characters. (Normally, a space or tab is used as a terminator.)

2.1 ADDRESS MODE: RELOCATABLE OR ABSOLUTE

MACRO-10 normally assembles programs with relocatable binary addresses, so that the program can be located anywhere in memory for execution as a function of what has been previously loaded. When desired, the assembler will also assign absolute location addresses, either for the entire program or for selected parts. Two pseudo-ops control the address mode: RELOC and LOC.

RELOC N↵

This statement sets the location counter to n, which may be a number or an expression, and causes the assembler to assign relocatable addresses to the instructions and data which follow. Since most relocatable programs start with the location counter set to 0; the implicit statement,

RELOC Ø↵

begins all programs, and need not be written by the programmer who wants his program assembled with relocatable addresses.

LOC N↵

This statement sets the location counter to n, a number or an expression, and causes the assembler to assign absolute addresses, begin ing with n, to the instructions and data which follow. If the entire program is to be assigned absolute locations, a LOC statement must precede all instructions and data.

If n is not specified

(LOC↵)

zero is assumed initially.

If only a part of the program is to be assembled in absolute locations, the LOC statement is inserted at the point where the assembler begins assigning absolute locations. For example, the statement,

LOC 2ØØ↵

causes the assembler to begin assigning absolute addresses, and the next machine instruction or data word is stored at location 200_8.

To change the address mode back to relocatable, an explicit RELOC statement is required. If the programmer wants the assembler to continue assigning relocatable addresses sequentially, he writes,

RELOC ↵

To switch back to the next sequential absolute assignment, he writes,

LOC ↵

Thus, the programmer is not required to insert a location counter value in either a LOC or RELOC statement, and unless he does, both the relocatable coding and the absolute coding will be assigned sequential addresses. This is shown in the following skeleton coding. The single quote mark is used here, and in MACRO-10 listings, to identify relocatable addresses.

```
Location Counter    Program
    ØØØØØØ'         ADD 1,X         ;RELOC Ø IS IMPLICIT.
                    .
                    .
                    .
    ØØØØ74'         LOC 1ØØØ        ;CHANGES TO ABSOLUTE, STARTING
    ØØ1ØØØ          SUB 5,TOT       ;WITH ØØ1ØØØ.
                    .
                    .
                    .
    ØØ1Ø34          RELOC           ;SETS LOCATION COUNTER TO 74.
    ØØØØ74'         ADD 2,XAT
    ØØØØ75'         LOC             ;SWITCHES LOCATION COUNTER
    ØØ1Ø34          EXP A-X+7       ;BACK TO ABSOLUTE SEQUENCE.
```

When operating in the relocatable mode, the assembler determines whether each location in the object program is relocatable or absolute, using an algorithm described in Chapter 5.

2.1.1 Relocation Before Execution - PHASE and DEPHASE Statements

Part of a program can be moved into other locations for execution. This feature is often used to relocate a frequently used subroutine, or iterative loop, into fast memory (accumulators $0\text{-}17_8$) just prior to execution.

To use this feature, the subroutine is assembled at sequential relocatable or absolute addresses along with the rest of the program, but the first statement before the subroutine contains the assembler operator, PHASE, followed by the address of the first location of the block into which the subroutine is to be moved prior to execution. All address assignments in the subroutine are in relation to the argument of the PHASE statement. The subroutine is terminated by a DEPHASE statement, which requires no operands, and which restores the location counter.

In the following example, which is the central loop in a matrix inversion, a block transfer instruction moves the subroutine LOOP into accumulators 11-16.

```
                                  MOVE [XWD LOOPX,LOOP]
    Relocatable                   BLT LOOP+4
      Address                     JRST LOOP
                      LOOPX:      PHASE 11
                      LOOP:       MOVN A (X)
                                  FMP MPYR
    Absolute                      FADM A (Y)
    Address                       SOJGE X, .-3
                                  JRST MAIN
                                  DEPHASE
```

The label LOOP represents accumulator 11, and the point in the SOJGE instruction represents accumulator 14.

Note that the code inside the phase to dephase program segment is loaded into the address following the previous relocatable code; all labels inside the segment, however, have the address corresponding to the phase address. Thus the phased code cannot, in general, be executed until it has been moved to the address for which it was assembled.

2.2 NAMING PROGRAMS

Normally the first statement in a program gives the name of the program using the TITLE pseudo-op. This pseudo-op has the form

TITLE NAME ↵

in which the single operand (i.e., NAME) may contain up to 60 characters.

The name given will be printed at the top of each page of the program listing. The first 6 characters of the given title will appear in the assembled program as the program name. If no title is given, the assembler inserts the name .MAIN. The program name given in the TITLE statement is used when debugging with DDT in order to gain access to the program's symbol table.

Only one TITLE pseudo-op is permitted in a program; it can appear anywhere in the program but is normally the first line on the first page. Remember that a name may be longer than 6 characters, however, only the first 6 symbol combinations (within the radix-50 set) will be used for the program name.

2.2.1 Program Subtitles

After the first page of a program listing, the first data line encountered on a page may be a subtitle. Subtitles are generated using the pseudo-op SUBTTL. This pseudo-op has the form

```
SUBTTL SUBTITLE)
```

in which the single operand (SUBTITLE) may contain up to 40 characters. A subtitle is printed as the first data line on a page and all succeeding pages until the end of the listing or until the subtitle is changed. If the current subtitle is changed by another SUBTTL statement which is the first data line on a page, the new subtitle appears on the new page and all subsequent pages. If the SUBTTL statement is not the first statement on a page, the new subtitle appears on the next page and all subsequent pages.

Subtitles can be changed as often as required; they do not generate data and they do not affect the binary procedure only the listing. They are used for informational purposes only.

2.3 PROGRAM ORIGIN

Initially all programs start with an implicit RELOC Ø which sets the mode to be relocatable and the first address to be Ø. Unless otherwise changed, the code generated will be a single-segment program.

The programmer can change the relocatable nature of the program by using a LOC statement to generate absolute code (normally used for diagnostics) or to generate high-segment code.

High-segment (or two-segment programs) have two logical address spaces; one starting at Ø and increasing, the other starting at 4ØØØØØ (128K) and increasing. Two pseudo-ops, HISEG and TWOSEG control High or two-segment program operation.

2.3.1 HISEG Statements - The HISEG Pseudo-Op Statement

This pseudo-op does not affect the assembly operations in any way except to generate information that directs the Loader to load the current program into the high segment if the program has reentrant (two-segment) capability. (Refer to Block Type 3 Load Into The High Segment, paragraph 6.2.1.1, for additional information.) This pseudo-op should appear at the beginning of the source program.

NOTE

> Whenever possible the pseudo-op TWOSEG should be used instead of HISEG. This pseudo-op provides functions which are superior to those of HISEG.

HISEG may be followed by an optional argument which represents the program high-segment origin address. This argument, when used, must be equal to or greater than 4ØØØØØ and must be a K-bound (even multiple of 2000) value. The code produced by HISEG will execute at either relocatable Ø or relocatable 4ØØØØØ depending on the loading instructions given.

HISEG must not be used if the programmer wishes to reference data in the low segment since locations in the low segment are referenced by absolute addresses only.

2.3.2 TWOSEG Statements

The TWOSEG pseudo-op generates code that directs MACRO and LOADER to assemble and load a two-segment program in one file. This pseudo-op outputs a block type 3 (refer to Paragraph 6.2.1.1) which signals the LOADER to expect two segments. An optional argument may be present

which is the first address in the high segment. If no argument is present, 400000 is assumed.

The high segment code must be preceded by

```
RELOC 4ØØØØØ
```

or greater; the low segment code by

```
RELOC Ø
```

or an argument indicating the low segment. Each RELOC pseudo-op switches the relocation.

The listing produced by the TWOSEG pseudo-op shows high segment addresses as greater than 400000 or the argument of the pseudo-op, and low segment addresses as less than 400000 or the argument of the pseudo-op. All relocatable addresses are flagged with a single quote.

2.4 ENTERING DATA

2.4.1 RADIX Statements

When the assembler encounters a numerical value in a statement, it converts the number to a binary representation reflecting the radix indicated by the programmer. The statement,

```
RADIX N↵
```

where n is a decimal number, $2 \leq n \leq 10$, sets the radix to n for all numerical values that follow, unless another RADIX statement changes the prevailing radix or a local radix change occurs (see below).

For example, if the programmer wants the assembler to interpret his numbers as decimal quantities, then the prevailing radix must be set to decimal before he uses decimal numbers.

```
RADIX 1Ø↵
```

The statement, RADIX 2, sets the prevailing radix to binary.

The implicit statement, RADIX 8, begins every program; if the programmer wants to enter octal numbers, this statement is not necessary.

2.4.2 Entering Data Under the Prevailing Radix

Data is entered under the prevailing radix by typing the data, followed by a carriage return:

```
765432234567↵
```

Data may be labeled and contain expressions:

```
LAB:  456+A+B/<C+D>↵
```

Data may also be entered by first using a direct assignment statement to place a symbol with an assigned value in the symbol table, and then using the symbol to insert the assigned value in the object program. For example, the direct assignment statements,

```
A=2↵
B=5↵
```

cause two entries in the symbol table. The following statement enters the sum of the assigned values in the object program at symbolic address REX.

```
REX:  A+B↵   REX contains 000000 000007
```

The radix can also be changed locally, that is, for a single statement or a single value, after which the prevailing radix is automatically restored, as described in Section 1.3.

2.4.3 DEC and OCT Statements

To change to a local radix for a single statement, the programmer writes:

```
DEC N,N,N,...N↵
```

where all of the numbers and expressions are to be interpreted as decimal numbers. The numbers or expressions following the operator

are separated by commas, and each will generate a word of storage.

```
OCT N,N,N,...N ↵
```

changes the local radix to octal for this statement only, and generates a word of memory for each number or expression.

The statement,

```
DEC  1Ø,4.5,3.1416,6.Ø3E-26,3 ↵
```

generates five decimal words of data.

2.4.4 Changing the Local Radix for a Single Numeric Term

To change the radix for a single number or expression, the numeric term is prefixed with ↑D, ↑O, ↑B, or ↑F, as explained in Chapter 1. If an expression is used, it must be enclosed in angle brackets,

```
↑D<A+B-C/2ØØ> ↵
```

These prefixes may generate a word, or part of an instruction word. The statement,

```
TOTAL2:MOVE  ↑D1Ø,ABZ ↵
```

causes the contents of ABZ to be moved to accumulator 12_8.

When the assembler encounters a numeric term, it forms the binary representation in a 36-bit register under the prevailing or local radix. If the quantity is a part of an instruction, it is truncated to fit in the required field.

For example, the accumulator field must have a final value in the range $0\text{-}17_8$. In the statement,

```
MOVE  ↑D6Ø,ABZ ↵
```

the assembler first interprets the accumulator address in a 36-bit register, forming the integer 000000000074: but takes only the rightmost four bits and places them in the accumulator field of the instruction, which results in the selection of accumulator 14_8.

2.4.5 RADIX 50 Statement

Another radix changing statement is available, but it is used primarily in systems programming. This is RADIX50 n,sym↲ which is used by the assembler, PDP-10 Loader, DDT, and other systems programs to pack symbolic expressions into 32 bits and add a 4-bit code field n in bits 0-3. This is explained in Appendix F of this manual. (The mnemonic SQUOZE may be used in place of RADIX50.)

2.4.6 EXP Statement

Several numbers and expressions may be entered by using the EXP statement:

```
EXP  X,4,  ↑D65,HALF,B+362-A ↲
```

which generates one word for each expression; five words were generated for the above example.

2.4.7 Z Statement

A zero word can be entered by using the operator, Z.

```
LABEL:  Z↲
```

generates a full word of all zeros at LABEL. If operands follow the Z, the assembler forms a primary machine instruction, with the operator field and other unknown fields zeroed. In the statement,

```
Z 3,↲
```

the assembler finds an accumulator field, but no address field, and generates this machine instruction: 000140 000000.

2.5 INPUT DATA WORD FORMATTING

2.5.1 BYTE Statement

To conserve memory, it is useful to store data in less than full 36-bit words. Bytes of any length, from 1 to 36 bits, may be entered by using a BYTE statement.

```
BYTE  (N) X,X,X ↲
```

The first operand (n) is the byte size in bits. It is a decimal number in the range 1-36, and must be enclosed in parentheses. The operands following are separated by commas, and are the data to be stored. If an operand is an expression, it is evaluated and, if necessary, truncated from the left to the specified byte size. Bytes are packed into words,

starting at bit 0, and the words are assigned sequential storage locations. If, during the packing of a word, a byte is too large to fit into the remaining bits, the unused bits are zeroed and the byte is stored left-justified in the next sequential location.

In the following statement, three 12-bit bytes are entered:

```
LABEL:  BYTE (12)5,177,N ↵
```

This assembles at LABEL as, 0005 0177 0316, where N=316.

The byte size may be altered by inserting a new byte size in parentheses immediately following any operand. Notice that the parentheses serve as delimiters, so commas must not be written when a new byte size is inserted. The following are legal:

```
BYTE  (6)5(14)NT(3)6,2,5 ↵
```

where 6 is entered in a 6-bit byte, NT in the following 14-bit byte, 6 in the following 3-bit byte, followed by 2 and 6 in 3-bit bytes. A BYTE statement can be used to reserve null fields of any byte size. If two consecutive delimiters are found, a null field is generated.

```
BYTE  (18),5 ↵
```

When the assembler finds two delimiters, it assembles a null byte. In this case, 000000 000005. To enter ASCII characters in a byte, the characters are enclosed in quotation marks.

```
BYTE  (7)"A" ↵
```

Text handling pseudo-ops are discussed in paragraph 2.5.5.

2.5.2 POINT Statement - Handling Bytes

Five machine instructions are available for byte manipulation. These instructions reference a byte pointer word, which is generated by the assembler from a POINT statement of the form,

```
LABEL:POINT s, address, b ↵    (s and b are decimal)
```

where the first operand s is a decimal number indicating the byte size, the second operand is the address of the memory location which contains the byte, and the third operand, b, is the bit position in the word of the right-hand bit of the byte (if b is not specified, the bit position is the nonexistent bit to the

left of bit 0). The address specified in the second operand may be indirect and indexed. If the byte size is not specified, MACRO-10 assumes 36 bits.

In the following example, an LDB (load a byte from a memory location into an accumulator) and an ILDB instructions are used, along with three assembler statements. The ILDB instruction "increments" AC to look like AB, then does a load byte; the effect of the two instructions is the same.

```
000000' 050000  000000   AA:      BYTE    (6)5
000001' 360600  000000'  AB:      POINT   6,AA,5
000002' 440600  000000'  AC:      POINT   6,AA

000003' 135140  000001'  START:   LDB     3,AB
000004' 134140  000002'           ILDB    3,AC
```

The first statement enters the quantity 5 in a 6-bit byte at symbolic address AA which is 0. The second statement is for reference by the load byte instruction. When the LDB is executed, the machine goes to AB for the byte size, its address, and bit position. In this case, it finds that the byte size is 6 bits, the byte is located in the word AA, and the right-hand bit of the byte is in bit 5. The byte is then loaded into accumulator 3, where it looks like this: 000000 000005.

The other byte manipulation mnemonic instructions reference the byte pointer word in similar ways. The deposit byte (DPB) instruction takes a byte from an accumulator and deposits it, in the position specified by the pointer word, in a memory word.

The increment byte pointer (IBP) instruction increments the bit position indicator (the third operand in the referenced POINT word) by the byte size. This is useful when loading or depositing a string of bytes, using the same byte pointer word.

The increment and load byte (ILDB) and increment and deposit byte (IDPB) instructions increment the byte pointer word by the byte size before loading or depositing.

2.5.3 IOWD Statement: Formatting I/O Transfer Words

The assembler generates I/O transfer words in a special format for use in BLKI and BLKO and all four pushdown instructions. The general statement is,

```
IOWD  N,M↵
```

where two operands, which may be numbers or expressions, follow the IOWD operator. This statement generates one data word. The left half of the assembled word contains the 2's complement of the first operand n, and the right half-word contains the value of the second operand m, minus one. For example,

```
IOWD 6,↑D256↵
```

assembles as 777772 000377.

2.5.4 XWD Statement: Entering Two Half-Words of Data

The XWD statement enters two half-words in a single storage word. It is written in the form,

```
XWD  LHW,RHW↵
```

where the first operand is a symbol or expression specifying the left half-word, and the second operand specifies the right half-word. Both are formed in 36-bit registers and the low order 18-bits are placed in the half-words. The high-order 18 bits of each operand are ignored. Three examples follow:

```
XWD  A,B↵
XWD  SUM+2,DES+5↵
XWD  START,END↵
```

XWD statements are used to set up pointer words for block transfer instructions. Block transfer pointer words contain two 18-bit addresses: the left half is the starting location of the block to be moved, and the right half is the first location of the destination. A,,B may also be used to duplicate the results of XWD A,B.

2.5.5 Text Input

The assembler translates text written in full 7-bit ASCII or 6-bit compressed ASCII. It will also format 7-bit ASCII with a null character at the end of text, if desired. These codes are listed in Appendix E.

In all three text modes, the printing symbols in the code set are translated to their binary representation.

To translate and store a single word containing as many as five 7-bit ASCII characters, right-justified, the input characters are enclosed in quotation marks.

```
"AXE")    is stored as
          0 0000000 0000000 1000001 1011000 1000101
          0   null    null     A       X       E
```

Notice that characters are right-justified, and bit 0, which is not used, is set to zero.

Up to six 6-bit ASCII characters may be translated and stored, right-justified, in a single word by enclosing the input characters in single quotation marks.

```
'TABLES' is stored as

110100 100001 100010 101100 100101 110011
  T      A      B      L      E      S
```

NOTE

The quotation marks (single or double) may only be used to assemble a single word. To input strings of text characters, the following three pseudo-ops must be used.

2.5.5.1 ASCII, ASCIZ, and SIXBIT Statement - To enter strings of text characters, the operators ASCII, SIXBIT, and ASCIZ are used. The delimiter for the string of text characters is the first non-blank character following the character that terminates the operator (refer to the note on page 2.1). The binary codes are left-justified. Unused character positions are set to zero (null). Text is terminated by repeating the initial delimiter. If the initial delimiter is a symbol constituent, the pseudo-op must be followed by a space or a tab.

The statement

```
ASCII "AXE" ↵
```

where the quotation marks are the delimiters, assembles as

```
1000001 1011000 1000101 0000000 0000000 0
   A       X       E      null    null  0
```

The operator ASCIZ (ASCII Zero) guarantees a null character at the end of text. If the number of characters is a multiple of five, another all zero word is added. For example,

```
ASCIZ/"AXE"/↵
```

assembles as,

```
0100010 1000001 1011000 1000101 0100010 0
   "       A       X       E       "
```

followed by another word of zeros.

```
0000000 0000000 0000000 0000000 0000000 0
null
```

When thefull 7-bit ASCII code set is not required, the 64-character 6-bit subset may be entered, using the SIXBIT operator. Six characters are left-justified in sequential storage words. Format of the SIXBIT statement is the same as for ASCII statements. To derive SIXBIT code:

a. Convert lower case ASCII characters to upper case characters.

b. Add 40_8 to the value of the character.

c. Truncate the result to the rightmost six bits.

2.5.6 Reserving Storage

The programmer can reserve single locations, or blocks of many locations for use during execution of his program.

2.5.6.1 Reserving a Single Location - The number sign (#), suffixing a symbol in an operand field, is used to reserve a single location. The symbol is defined, entered in the assembler's symbol table,and can be referenced elsewhere in the program without the number sign. For example,

```
LAB:  ADD 3,TEMP# ⤸
```

reserves a location called TEMP at the end of the program, which may be used to store a value entered at some other point in the program. This feature is useful for storing scalars, and other quantities which may change during execution.

The pseudo-op INTEGER may be used to reserve storage locations at the end of the program on a one-per-given name basis. For example the statement

```
INTEGER TEMP,FOO,BAR⤸
```

will reserve 3 locations identified as TEMP, FOO and BAR. The assignment of the locations to the names given is performed on an alphabetical basis by the assembler rather than on the order in which the names are given. For example, the order of the locations reserved by the foregoing INTEGER statement would be BAR, FOO then TEMP.

Multiple word locations may be reserved by the ARRAY pseudo-op. For example, the statement

```
ARRAY FOO[2*3]⤸
```

reserves a 2-word by 3-word array in memory which is identified by the name FOO.

NOTE

> If the pseudo-op TWOSEG is used, the variables reserved by an array statement must be assigned to the low segment only; thus, a VAR pseudo-op is required after a RELOC back to the low segment.

2.5.7 VAR Statements

VAR ⮐

This statement causes symbols which have been defined by suffixing with the # sign (array and integer pseudo-ops) in previous statements to be assembled as block statements. This has no effect on subsequent symbol definitions of the same type.

If the LIT and VAR statements do not appear in the program, all literals and variables are stored at the end of the program.

2.5.8 BLOCK Statements

To reserve a block of locations, the BLOCK operator is used. It is followed by a single operand, which may be a number or an expression in the current radix, indicating the number of words to be reserved. The assembler increments the location counter by the value of the operand. For example,

MATRIX: BLOCK N*M ⮐

reserves a block of N*M words starting at MATRIX for an array whose dimensions are M and N.

BLOCK is used to reserve words in a specific order; remember that data words should be stored in the low segment in two-segment programs.

2.5.9 END Statements

The END statement must be the last statement in every program. A single operand may follow the END operator to specify the address of the first instruction to be executed. Normally this operand is given only in the main program; since subprograms are called from the main program, they need not specify a starting address.

END START⮐ start is the label at the starting address

When the assembler first encounters an END statement, it terminates pass 1 and begins pass 2. The END also terminates pass 2, after which

the assembler automatically assembles all previously defined variables and literals starting at the current location.[1]

The following processing operations can be performed at any point in the program.

2.5.10 LIT Statements

LIT ↲

This statement causes literals that have been previously defined to be assembled, starting at the current location. If n literals have been defined, the next free storage location will be at location counter plus n. Literals defined after this statement are not affected.

If a LIT statement does not appear before the END statement, the literals are XLISTed (refer to paragraph 2.6.3). If the output of literals is desired, the LIT pseudo-op should appear immediately before the END statement.

NOTE

In a two-segment program LIT must be given in the high segment. The END statement must also be given in the high segment or the literals will go to the low segment.

2.5.11 Multi-Program Assembly

The pseudo-op PRGEND is used to compress many small files into one large file to save space and disk lookups. This pseudo-op has the form PRGEND↲ . PRGEND allows multiprogram assemblies, and is used for assembling library files (LIB40) in which all programs are very short. PRGEND takes the place of all but the last END statement. The output is a binary file which can be loaded in search mode. The use of PRGEND costs assembler space since the symbol tables, literal tables and titles of each of the small files (i.e., programs) involved must be saved at the end of pass 1. Also, since PRGEND is functionally an END statement, macros cannot be used over it (i.e., macros cannot generate PRGEND as part of their expansions).

---

[1]The END statement is also used to specify a transfer word in some output file formats. (See Section 6.2.2.4.)

If the LIT and VAR statements do not appear in the programs, all literals and variables are stored at the end of the program.

2.5.12 PASS2 Statements

PASS2 ⤶

This statement switches the assembler to pass 2 processing for the remaining coding. Coding preceding this statement will have been processed by pass 1 only. This is used primarily for debugging, such as testing macros defined in the pass 1 portion.

2.5.13 PURGE Statements

The PURGE statement is used to delete defined symbols. Its general form is:

PURGE symbol, symbol, symbol ⤶

where each operand is a user-created label, operator, or macro call which is to be deleted from the assembler's tables. The PURGE statement is normally used at the end of programs to conserve storage and to delete symbols for DDT. Purged symbol table space is reused by the assembler.

If the programmer uses the same symbol for both a macro call and/or OPDEF (refer to Section 2.8.2) and for a label, a PURGE statement deletes the macro call or OPDEF. A repeat of the symbol in the PURGE statement also purges the label. For example, the following statement purges both:

PURGE SOLV,SOLV ⤶

The first SOLV purges the macro call; the second SOLV purges the label.

2.5.14 XPUNGE Statements

The XPUNGE pseudo-op deletes all local symbols during pass 2; it has the form:

XPUNGE ⤶

The use of this pseudo-op reduces the size of the REL file and speeds up loading (especially of DDT). XPUNGE should be placed just prior to the END statement.

2.5.15 Linking Subroutines

Programs usually consist of subroutines which contain references to symbols in external programs. Since these subroutines may be assembled separately, the loader must be able to identify "global" symbols. For a given subroutine, a global symbol is either a symbol defined internally and available for reference by other subroutines, or a symbol used internally but defined in another subroutine. Symbols defined within a subroutine, but available to others, are considered internal. Symbols which are externally defined are considered external.

These linkages between internal and external symbols are set up by declaring global symbols using the operators EXTERN, INTERN, or ENTRY. The double colon (::) may also be used.

2.5.15.1 EXTERN Statements - The EXTERN statement identifies symbols which are defined elsewhere. The statement,

```
EXTERN SQRT, CUBE,TYPE↵
```

declares three symbols to be external. External symbols must not be defined within the current subroutine. These external references may be used only as an address or in an expression that is to be used as an address. For example, the square root routine declared above might be called by the statement,

```
PUSHJ  P,SQRT ↵
```

External symbols may be used in the same manner as any other relocatable symbol. Examples:

```
                         EXTERN A
200300   000003*  MOVE    6,A+3
000003*  000000*  XWD     A+3,A
777777   777771   B=      A-7
                  OPDEF   U[XWD B+3,A-5]
777774*  777773*  U
```

The external symbols are flagged with asterisks. There are three restrictions for the use of external symbols:

a. Externals may not be used in LOC and RELOC statements.

b. The use of more than one external in an expression is not permitted. Thus, A+B (where A and B are both external) is illegal.

c. Globals may only be additive; therefore, the following are illegal

```
-A        EXP-A
2*A       2*A-A
```

An alternative method for generating external symbols is to use a double pound sign (##) following the symbol name. This method eliminates specifying the EXTERN statement. For example,

```
MOV Ø,JOBREL##
```

is equivalent to

```
EXTERN JOBREL
MOVE Ø,JOBREL
```

2.5.15.2 INTERN Statements - To make internal program symbols available to other programs as external symbols, the operators INTERN or ENTRY are used. These statements have no effect on the actual assembly of the program, but will make a list of symbol equivalences available to other programs at load time. The statement,

```
INTERN MATRIX ↵
```

makes the subroutine MATRIX available to other programs. An internal symbol must be defined within the program as a label, variable, or by direct assignment.

2.5.15.3 ENTRY Statements - Some subroutines have common usage, and it is convenient to place them in a library. In order to be called by other programs, these library subroutines must contain the statement,

```
ENTRY NAME ↵
```

where "name" is the symbolic name of the entry point of the library subroutine.

ENTRY is equivalent to INTERN with the following additional feature. All names in a list following ENTRY are defined as internal symbols and are placed in a list at the beginning of the library of subroutines. If the loader is in library search mode, a subroutine will be loaded if the program to be executed contains an undefined global symbol which matches a name on the library ENTRY list.

If the MATRIX subroutine mentioned before is a library subroutine, it must contain the statement,

```
        ENTRY MATRIX ⤶
```

Since library subroutines are external to programs using them, the calling program must list them in EXTERN statements.

2.6 SUPPRESSION OF SYMBOLS

When a parameter file is used in assemblies, many symbols get defined but are never used. Unused defined symbols take up space in the binary file and complicate listings of the file. Unused and unwanted symbols may be removed from symbol tables by the use of a pseudo-op, either SUPPRESS or ASUPRESS. These pseudo-ops control a suppress bit in each location of the symbol table; if a suppress bit is on, the symbol in that location is not output. The suppress bit is used in the file S.MAC so that if a bit is on and the symbol in that location is not used later, the symbol is not output in the CREF table.

2.6.1 SUPPRESS SYMBOL Statement

The SUPPRESS statement turns on the suppress bit for the specified symbols.

2.6.2 ASUPPRESS Statement

The ASUPPRESS statement turns on the suppress bit for all the symbols in the symbol table.

2.6.3 Listing Control Statements

Program listings are normally printed on a line printer or a terminal depending on the listing file device specified. Listings are printed as the source program statements are processed during pass 2. A sample listing is shown in Chapter 7.

From left to right the standard columns of a listing contain

a) the location counter,
b) the instruction or data in octal form, and
c) the symbolic instruction or data followed by comments.

Relocatable object-code addresses are suffixed by a single quotation mark (') which may occur in either the left or right half.

Data is displayed in one of several modes depending on the statement format. The possible statement formats are:

1)	Halfword	- two 18-bit bytes
2)	Instruction	- a 9-bit op-code, 4-bit accumulator code, 1-bit indirect bit, 4-bit index, and an 18-bit address segment
3)	Input/Output	- 3-bit I/O indicator, 7-bit I/O device specification, 3-bit operand, 1-bit indirect address bit, 4-bit index and an 18-bit address segment
4)	Byte pointer	- 6-bit byte position, 6-bit byte size, 1 unused bit, 1-bit indirect address bit, 4-bit index and an 18-bit address segment
5)	ASCII	- 5 Seven-bit bytes
6)	SIXBIT	- 6 six-bit bytes.

NOTE

Refer to the DECsystem-10 System Reference Manual for a complete description of word formats.

The listing function is suppressed within macro expansion, therefore only the macro call and any succeeding lines that generate code are

listed. Line printer listings always begin at the top of a page and up to 55 lines are printed on each page. Consecutive page numbers are printed in the upper right-hand corner of each page. Each page also contains a title and a subtitle.

The standard listing operations can be augmented and modified by using the following listing control statements.

STATEMENT	DESCRIPTION
PAGE ↵	This statement causes the assembler to skip to the top of the next page. (A form feed character in the input text has the same effect and is preferred.
XLIST ↵	This statement causes the assembler to stop listing the assembled program. The listing printout actually starts at the beginning of pass 2 operations. Therefore, to suppress all program listing, XLIST must be the first statement in the program. If only a part of the program listing is to be suppressed, XLIST is inserted at any point to stop listing from that point. Literals are XLISTed if no LIT statement is seen before the END statement.
LIST ↵	Normally used following an XLIST statement to resume listing at a particular point in the program. The LIST function is implicitly contained in the END statement.
LALL ↵	This statement causes the assembler to list everything that is processed including all text, macro expansions and list control codes suppressed in the standard listing.
XALL ↵	Normally used following a LALL statement to resume standard listing.
SALL ↵	This causes suppression of all macro and repeat expansions and their text; only the input file and the binary generated will be listed. SALL can be nullified by either XALL or LALL and the /M switch can be used instead of SALL.
NOSYM ↵	The assembler normally prints out the symbol table at the end of the program, but the NOSYM statement suppresses the symbol table printout.

STATEMENT	DESCRIPTION
TAPE ↲	This pseudo-op causes the assembler to begin assembling the program contained in the next source file in the MACRO command string. For example,

```
.R MACRO
*DSK:BINAME,LPT:←TTY:,DSK:MORE
PARAM=6
TAPE
;THIS COMMENT WILL BE IGNORED
↑Z
```

would set the symbol PARAM equal to 6 and then assemble the remainder of the program from the source file DSK:MORE. Since MACRO is a 2-pass assembler, the TTY: file would probably be repeated for pass 2.

```
END OF PASS 1
PARAM=6
TAPE
↑Z
```

Note that all text after the TAPE pseudo-op is ignored.

STATEMENT	DESCRIPTION
PRINTX MESSAGE↲	This statement, when encountered, causes the single operand following the PRINTX operator to be typed out on the TTY. This statement is frequently used to print out conditional information. PRINTX statements are also used in very long assemblies to report the progress of the assembler through pass 1.
REMARK COMMENTS↲	On pass 1 the message is printed on both the list device and TTY. On pass 2 it is printed on the TTY, but only if it is not the list device.
	The REMARK operator is used for statements which contain only comments. Such statements may also be started with a semi-colon.
COMMENT↲	This pseudo-op treats the text between the first non-blank character (delimiter) and the next occurrence of the same character as a comment. If the first occurrence of the delimiter is a right (left) angle bracket, the next occurrence of the delimiter must also be a right (left) angle bracket. The text may include the carriage return, line feed sequence. For example,

```
COMMENT/THIS IS A COMMENT
THAT IS MORE THAN ONE LINE LONG
/
```

Internally, the pseudo-op functions as ASCII, but no binary is produced.

2.7 CONDITIONAL ASSEMBLY

Parts of a program may be assembled, or not assembled, on an optional basis depending on conditions defined by an assembler IF statement. The general form is,

```
IF N, <.............>
```

where the coding within angle brackets is assembled only if the first operand, N, meets the statement requirement.

The IF statement operators and their conditions are listed below:

Operator	Assemble angle-bracketed coding IF:
IFE N, <...>	N=0, or blank
IFG N, <...>	N>0
IFGE N, <...>	N=0, or N>0
IFL N, <...>	N<0
IFLE N, <...>	N=0, or N<0
IFN N, <...>	N≠0
IF1, <...>	encountered during pass 1
IF2, <...>	encountered during pass 2

In the following conditional statements, assembly depends on whether or not a symbol has been defined. The coding enclosed in angle brackets is assembled if,

IFDEF SYMBOL, <...>	this symbol is defined
IFNDEF SYMBOL, <...>	this symbol is not defined

NOTE

SYMBOL can be an op-code or pseudo-op as well as a user symbol.

The following conditional statements operate on character strings. Arguments are interpreted as 7-bit ASCII character strings, and the assembler makes a logical comparison, character-by-character to determine if the condition is met.

The coding within the third set of angle brackets is assembled if the character strings enclosed by the first two sets of angle brackets:

IFIDN <A-Z> <A-Z>,<...>	(1) are identical
IFDIF <A-Z> <A-Z>,<...>	(2) are different

These statements, IFIDN and IFDIF, are usually used in macro expansions (see Chapter 3) where one or both arguments are dummy variables.

An alternate form is to use delimiters as in ASCII. For example:

```
IFDIF/A-Z/"A-Z',<--->
```

This allows the use of > inside the character string. If the first non-blank (space or tab) character is a < character, then the < > method is used; otherwise, the character is used as a delimiter.

The last pair of conditional statements is followed by a single bracketed character string, which is either blank or not blank, and which is followed by conditional coding in brackets.

The coding enclosed in the second set of angle brackets is assembled if,

`IFB <...>,<....>`	the first operand is blank
`IFNB <...>,<.....>`	the first operand is not blank

A blank field is either an empty field or a field containing only the ASCII characters space (40_8) or tab (11_8).

Again, delimiters can be used as in

```
IFB / .... / , <.....>
```

2.8 ASSEMBLER CONTROL STATEMENTS

2.8.1 REPEAT Statements

The statement

```
REPEAT N,  <...>↵
```

causes the assembler to repeat the coding enclosed in angle brackets n times. If more than one instruction or data word is to be repeated, each is delimited by a carriage return. For example,

```
ADDX:    REPEAT 3,  <ADD 6,X(4)↵
                     ADDI 4,1>↵
```

assembles as,

```
ADDX:   ADD 6,(4)
        ADDI 4,1
        ADD 6,X(4)
        ADDI 4,1
        ADD 6,X(4)
        ADDI 4,1
```

Notice that the label of a REPEAT statement is placed on the first line of the assembled coding. REPEAT statements may be nested to any level. The following simplified example shows how a nested REPEAT statement is interpreted.

```
REPEAT 3,<A↵
REPEAT 2,<B↵
          C>↵
          D>↵
```

assembles as,

```
A
B
C
B
C
D
A
B
C
B
C
D
A
B
C
B
C
D
```

NOTE

Brackets indicate repetition.

2.8.2 OPDEF Statements

The programmer can define his own operators using an OPDEF statement, which is written in the form:

```
OPDEF SYM [STATEMENT]
```

where the first operand is defined as an operator, whose function is defined by the second operand, which is enclosed in square *brackets. The* second operand is evaluated as a statement, and the

result is stored in a 36-bit word. For example,

```
OPDEF CAL1 [Ø3ØØØØ ØØØØØØ]
```

defines CAL1 as an operator, with the value 030000 000000. CAL1 may now be used as a statement operator.

```
Ø3Ø14Ø ØØ1234  CAL1 3,1234
```

which is equivalent to,

```
Ø3Ø14Ø ØØ1234  Z  3,1234(3ØØØØ)
```

When MACRO-10 encounters a user-defined operator, it assembles a single object-program storage word in the format of a primary instruction word (see Chapter 1). The defined 36-bit value is modified by accumulator, indirect, memory address and index fields as specified by the user-defined operator.

For example,

```
OPDEF CAL [MOVE 1,@SYM(2)]↩
CAL 1,BOL(2)↩
```

The CAL statement is equivalent to:

```
MOVE 2,@SYM+BOL(4)↩
```

In this modification the accumulator fields are added, the indirect bits are logically ORed, the memory address fields are added, and the index register addresses are added.

2.8.3 SYN Statements

The statement

```
SYN symbol, symbol
```

defines the second operand as synonymous with the first operand, which must have been previously defined. Either operand may be a symbol or a macro name. If the first operand is a symbol, the second is defined as a symbol with the same value. If the first is

a macro name, the second becomes a macro name which operates identically. If the first is a machine, assembler, or user-defined operator, the second will be interpreted in the same manner. If the first operand in a SYN statement has been previously defined as both a label and as an operator, the second operand is synonymous with the label.

The following are legal SYN statements:

```
SYN K,X↵          ;IF K=5, X=5
SYN FAD,ADD↵
SYN END,XEND↵
```

2.8.4 Extended Instruction Statements

For programming convenience, some extended operation codes are provided in the MACRO-10 Assembler. Primarily, these are used to replace those DECsystem-10 instructions where the combination of instruction mnemonic and accumulator field is used to denote a single instruction. For example:

```
JRST 4
```

is equivalent to a halt instruction. In addition, they are used to replace certain commonly used I/O instruction-device number combinations.

The extended instruction statements are exactly like the primary instruction statements or I/O instruction statements, except that they may not have an accumulator field or device field.

The operator field must have one of the following extended mnemonics:

Extended Instructions	Equivalent Machine Instructions	Meaning
JEN	JRST 12,	Jump and enable the PI (priority interrupt) system
HALT	JRST 4,	Halt
JRSTF	JRST 2,	Jump and restore flags
JOV	JFCL 1Ø,	Jump on overflow and clear
JCRYØ	JFCL 4,	Jump on CRYØ and clear
JCRY1	JFCL 2.	Jump on CRY1 and clear
JCRY	JFCL 6,	Jump on CRYØ or CRY1 and clear
JFOV	JFCL 1,	Jump on floating overflow
RSW	DATAI Ø	Read the console switches

2.9 MULTI-FILE ASSEMBLY

2.9.1 UNIVERSAL Name

UNIVERSAL files may be used to generate data, however, they are normally used to generate symbols, macros and opdef's (user-defined operators). The symbols generated by UNIVERSAL files need not be declared as INTERNAL symbols since all local symbols in files of this type are made available to all programs permitted access to the file.

UNIVERSAL files used to generate data can save time by being set up for a one-pass operation since symbol definition needs to be assembled on one pass only. This one-pass operation can be accomplished in either of two ways:

1)
```
UNIVERSAL NAME
PASS 2
.
.
.
END
```

2)
```
UNIVERSAL NAME
IF 2, <END>
.
.
.
END
```

The first generates a listing; the second does not.

If the UNIVERSAL pseudo-op is seen in a program, the NAME is stored in a table and a flag is set. When the END statement is seen, the symbol table is moved to just after the pushdown stacks and buffers; therefore, the pushdown stacks and buffers cannot be increased during assembly. The first assembly should use the maximum of I/O devices to be used later. The free core pointer is moved to after the top of the moved symbol table, and pointers are stored to enable the table to be scanned.

When assembling is done from indirect files, the universal files must be recompiled by the /COMPIL switch. Otherwise if a REL file later than the source exists, the universal file will not be compiled, and the symbol table will not be available. In addition, if the universal routine is modified, all routines which use it must be recompiled by either using /COMPIL or deleting all REL files.

2.9.2 SEARCH Name

The SEARCH statement opens the specified symbol table for MACRO to scan if the required symbol is not found in the current symbol table. Multiple symbol tables may be specified by separating them with commas; they are searched in the order specified. A maximum of ten symbol tables may be specified since each name requires four words of core. This maximum may be redefined with the symbol .UNIV in MACRO.

When the SEARCH pseudo-op is seen, the specified names are compared with the UNIVERSAL table. If the specified names cannot be found, the message

CANNOT FIND UNIVERSAL name

is output. If the specified names are found, a table of searching sequence is built. This sequence is to search the universal symbol tables in the order specified whenever a symbol is not found in the current symbol table. This search is to continue until the symbol is found or all the tables have been searched. When a symbol is found in an auxiliary symbol table, it is moved into the current symbol table. This procedure saves time on future references at the expense of core.

Universal files may search other universal files as long as all names in the search list have been assembled. The table of universal names is cleared on each RUN or START, but is not cleared when MACRO responds with an asterisk.

Chapter 3
Macros

When writing a program, certain coding sequences are often used several times with only the arguments changed. If so, it is convenient if the entire sequence can be generated by a single statement. To do this, the coding sequence is defined with dummy arguments as a macro instruction. A single statement referring to the macro by name, along with a list of real arguments, generates the correct sequence.

3.1 DEFINITION OF MACROS

The first statement of a macro definition must consist of the operator DEFINE followed by the symbolic name of the macro. The name must be constructed by the rules for constructing symbols. The macro name may be followed by a string of dummy arguments enclosed in parentheses. The dummy arguments are separated by commas and may be any symbols that are convenient--single letters are sufficient. A comment may follow the dummy argument list.

The character sequence, which constitutes the body of the macro, is delimited by angle brackets. The body of the macro normally consists of a group of complete statements.

For example, this macro computes the length of a vector:

```
DEFINE VMAG (A,B)       ;ROUTINE FOR THE LENGTH OF A VECTOR
<MOVE Ø,A               ;GET THE FIRST COMPONENT
FMP Ø                   ;SQUARE IT
MOVE 1,A+1              ;GET THE SECOND COMPONENT
FMP 1,1                 ;SQUARE IT
FAD 1                   ;ADD THE SQUARE OF THE SECOND
MOVE 1,A+2              ;GET THE THIRD COMPONENT
FMP 1,1                 ;SQUARE IT
FAD 1                   ;ADD THE SQUARE OF THE THIRD
JSR FSQRT               ;USE THE FLOATING SQUARE ROOT ROUTINE
MOVEM B                 ;STORE THE LENGTH>
```

NOTE

Storing comments in a macro takes up space. If the comments start with a double semicolon (;;) the comment will not be stored; therefore, it lists in the original definition but does not list when the macro is expanded.

3.2 MACRO CALLS

A macro may be called by any statement containing the macro name followed by a list of arguments. The arguments are separated by commas and may be enclosed with parentheses. If parentheses are used (indicated by an open parenthesis following the macro name), the argument string is ended by a closed parenthesis. If there are n dummy arguments in the macro definition, all arguments beyond the first n, if any, are ignored. If parentheses are omitted, the argument string ends when all the dummy arguments of the macro definitions have been assigned, or when a carriage return or semicolon delimits an argument.

The arguments must be written in the order in which they are to be substituted for dummy arguements. That is, the first argument is substituted for each appearance of the first dummy argument; the second argument is substituted for each appearance of the second dummy arguemnt, etc. For example the appearance of the statement:

```
VMAG VEC, LENGTH
```

in a program generates the instruction sequence defined above for the macro VMAG. The character string VECT is substituted for each occurrence in the coding of the dummy argument A, and the character string LENGTH is substituted for the single occurrence of B in the coding.

Statements with a macro call may have label fields. The value of the label is the location of the first instruction generated.

CAUTION

> MACRO arguments are terminated only by COMMA, CARRIAGE RETURN, SEMICOLON or CLOSE PARENTHESIS (when the entire argument string was started with an open parenthesis). These characters may not be included in arguments unless <> are used. Specifically, spaces or tabs do not terminate arguments; they will be treated as part of the argument itself. The symbol does not terminate arguments, it just permits commas and other symbols to be used as part of an argument.

3.3 MACRO FORMAT

a. Arguments must be separated by commas. However, arguments may also contain commas. For example:

```
DEFINE JFQ(A,B,C)
<MOVE [A]
CAMN B
JRST C>
```

If the data in location B is equal to A (a literal), the program jumps to C. If A is to be the instruction ADD 2,X, the calling macro instruction would be written

```
JEQ<ADD 2,X>,B,INSTX
```

The angle brackets surrounding the argument are removed, and the proper coding is generated.

The general rule is: If an argument contains commas, semicolons, or any other argument delimiters, the argument must be enclosed in angle brackets. For every level of nesting, one set of angle brackets is removed; therefore, to pass arguments containing commas to nested macros the argument should be enclosed by one set of angle brackets for each level of nesting. The > does not terminate the argument, a comma must be used.

b. A macro need not have arguments. The instruction:

```
DATAO PIP,PUNBUF(4)
```

which causes the contents of PUNBUF, indexed by register 4, to be punched on paper tape, may be generated by the macro:

```
DEFINE PUNCH
<DATAO PIP,PUNBUF(4)>
```

The calling macro instruction could be written:

```
PUNCH
```

PUNCH calls for the DATAO instruction contained in the body of the macro.

c. The macro name, followed by a list of arguments, may appear anywhere in a statement. The string within the angle brackets of the macro definition exactly replaces the macro name and argument string. For example:

```
DEFINE L(A,B)<3*<B-A+1>>
```

gives an expression for the number of items in a table where three words are used to store each item. A is the address of the first item, and B is the address of the last item. To load an index register with the table length, the macro can be called as follows:

```
MOVEI X,L(FIRST,LAST)
```

3.4 CREATED SYMBOLS

When a macro is called, it is often convenient to generate symbols without explicitly stating them in the call, for example, symbols for labels within the macro body. If it is not necessary to refer to these labels from outside the macro, there is no reason to be concerned as to what the labels are. Nevertheless, different symbols must be used for the labels each time the macro is called. Created symbols are used for this purpose.

Each time a macro that requires a created symbol is called, a symbol is generated and inserted into the macro. These generated symbols are of the form..hijk, that is, two decimal points followed by four digits. The user is advised not to use symbols starting with two points. The first created symbol is ..0001, the next is ..0002, etc.

If a dummy symbol in a definition statement is preceded by a percent sign (%), it is considered to be a created symbol. When a macro is called, all missing arguments that are of the form %X are replaced by created symbols. However, if there are sufficient arguments in the calling list that some of the arguments are in a position to be assigned to the dummy arguments of the form %X, the percent sign is overruled and the stated argument is assigned in the normal manner.

Null arguments are not considered to be the same as missing arguments. For example, suppose a macro has been defined with the dummy string:

```
(A,%B,%C)
```

If the macro were called with the argument string:

(OPD,) or OPD,,

The second argument would be considered to have been declared as null string. This would override the % prefixed to the second dummy argument and would substitute the null string for each appearance of the second dummy argument in the statement. However, the third argument is missing. A label would be created for each occurrence of %C. For example:

```
DEFINE TYPE(A,%B)
<JSR TYPEOUT
JRST %B
SIXBIT/A/
%B:>
```

This macro types the text string substituted for A on the console Teletype. TYPEOUT is an output routine. Labeling the location following the text is appropriate since A may be text of indefinite length. A created symbol is appropriate for this label since the programmer would not normally reference this location. This macro might be called by:

```
TYPE HELLO
```

which would result in typing HELLO when the assembled macro is executed. If the call had been:

```
TYPE HELLO,BX
```

the effect would be the same. However, BX would be substituted for %B, overruling the effect of the percent sign.

3.5 CONCATENATION

The apostrophe character or single quote (') is defined as the concatenation operator. A macro argument need not be a complete symbol Rather, it may be a string of characters which form a complete symbol or expression when joined to characters already contained in the macro definition. This joining, called concatenation, is performed by the assembler when the programmer writes an apostrophe between the strings to be so joined. As an example, the macro:

```
DEFINE J(A,B,C)
<JUMP'A B,C>
```

When called, the argument A is suffixed to JUMP to form a single symbol. If the call were:

```
J (LE,3,X+1)
```

the generated code would be:

```
JUMPLE 3,X+1
```

The concatenation (') may be used in nested macros. The assembler removes one operator when it performs concatenation if it is next to (before or after) a dummy argument.

3.6 DEFAULT ARGUMENTS

Missing arguments in macros are generally replaced by nulls. For example, the macro

```
DEFINE FOO (A,B,C)>
EXP A,B,C>
```

when called by FOO(1) would generate three words of 1, Ø, and Ø.

Default arguments may be supplied to override missing arguments. When supplied, default arguments are written within angle brackets (<>) after each argument. For example, the addition of default arguments 222 and 333 to arguments B and C of the foregoing example macro would be written as

```
DEFINE FOO (A,B<222>, <<333>)
EXP A,B,C>
```

If the foregoing macro is called by FOO(1) it would generate the number 1,222,333.

The following example program illustrates the use of defined default arguments.

```
.MAIN    MACRO 47(113) 1Ø:14 28-MAR-72 PAGE 1
FOO      MAC      28-MAR-72 1Ø:13
                                           DEFINE FOO1 (A,B,C)<
                                           EXP A,B,C>
                                           DEFINE FOO2 (A<111>,B<222>,C<333

>)<

                                           EXP A,B,C>

                                           FOO1 (1)↑
         ØØØØØØ' ØØØØØØ  ØØØØØ1            EXP1,,↑
         ØØØØØ1' ØØØØØØ  ØØØØØØ
         ØØØØØ2' ØØØØØØ  ØØØØØØ
                                           FOO2 (1)↑
         ØØØØØ3' ØØØØØØ  ØØØØØ1            EXP 1,222,333↑
         ØØØØØ4' ØØØØØØ  ØØØ222
         ØØØØØ5' ØØØØØØ  ØØØ333
                                           END

NO ERRORS DETECTED

PROGRAM BREAK IS ØØØØØ6

2K CORE USED
```

3.7 INDEFINITE REPEAT

It is often convenient to be able to repeat a macro one or more times for a single call, each repetition substituting successive arguments in the call statement for specified arguments in the macro. This may be done by use of the indefinite repeat operator, IRP. The operator IRP is followed by a dummy argument, which may be enclosed in parentheses. This argument must also be contained in the DEFINE statement's list. This argument is broken into subarguments. When the macro is called, the range of the IRP is assembled once for each subargument, the successive subarguments being substituted for each appearance of the dummy argument within the range of the IRP. For example, the single argument:

```
<ALPHA,BETA,GAMMA>
```

consists of the subarguments ALPHA,BETA, and GAMMA. The macro definition:

```
DEFINE DOEACH(A),
<IRP A
<A>>
```

and the call:

```
DOEACH<ALPHA,BETA,GAMMA>
```

produce the following coding:

```
ALPHA
BETA
GAMMA
```

An opening angle bracket must follow the argument of the IRP statement to delimit the range of the IRP since the argument is one argument to the macro. A closing angle bracket must terminate the range of the IRP. IRPC is like IRP except it takes only one character at a time; each character is a complete argument. An example of a program that uses an IRPC is given in Chapter 7, Figure 7-4.

It is sometimes desirable to stop processing an indefinite repeat depending on conditions given by the assembler. This is done by the operator STOP1. When the STOP1 is encountered, the macro processor finishes expanding the range of the IRP for the present argument and terminates the repeat action. An example:

```
DEFINE CONVERT (A)
<IRP A<IFE K-A,<STOPI
CONV1 A>>
```

Assume that the value of K is 3: then the call:

```
CONVERT Ø,1,2,3,4,5,6,7
```

```
<IRP
IFE K-Ø,<STOPI
CONV1 Ø>
IFE K-1,<STOP1
CONV1 1>
IFE K-2,<STOPI
CONV1 2>
IFE K-3,<STOPI
CONV1 3>
```

The assembly condition is not met for the first three arguments of the macro. Therefore, the STOPI code is not encountered until the fourth argument, which is the number 3. When the condition is met, the STOPI code is processed which prevents further scanning of the arguments. However, the action continues for the current argument and generates CONV1 3, i.e., a call for the macro CONV1 (defined elsewhere) with an argument of 3.

3.8 NESTING AND REDEFINITION

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, i.e., those defined directly to the macro processor, may be called first level macros. Macros

defined within first level macros may be called second level macros; macros defined within second level macros may be called third level macros; etc.

At the beginning of processing, first level macros are known to the macro processor and may be called in the normal manner. However, second and higher level macros are not yet defined. When a first level macro containing second and higher level macros is called, all its second level macros become defined to the processor. Thereafter, the level of definition is irrelevant, and macros may be called in the normal manner. Of course, if these second level macros contain third level macros, the third level macros are not defined until the second level macros containing them have been called.

When a macro of level n contains a macro of level n+1, calling the macro results in generating the body of the macro into the user's program in the normal manner until the DEFINE statement is encountered. The level n+1 macro is then defined to the macro processor; it does not appear in the ueer's program. When the definition is complete, the macro processor resumes generating the macro body into the user's program until, or unless, the entire macro has been generated.

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

The first example of a macro calculation of the length of a vector may be rewritten to illustrate both nesting and redefinition.

```
DEFINE VMAG (A,B,%C)
<DEFINE VMAG (D,E)
<JSP SJ,VL
EXP C,E>
VMAG (A,B)
      JRST %C
VL:   HRRZ 2, (SJ)
      MOVE (2)
      FMP Ø
      MOVE 1,1(2)
      FMP 1,1
      FAD 1
      MOVE 1,2(2)
      FMP 1,1
      FAD 1
      JSR FSQRT
      MOVEM @1 (SJ)
      JRST 2(SJ)
%C:>
```

The procedure to find the length of a vector has been written as a closed subroutine. It need only appear once in a user's program. From then on it can be called as a subroutine by the JSP instruction.

The first time the macro VMAG is called, the subroutine calling sequence is generated followed immediately by the subroutine itself. Before generating the subroutine, the macro processor encounters a DEFINE statement containing the name VMAG. This new macro is defined and takes the place of the original macro VMAG. Henceforth, when VMAG is called, only the calling sequence is generated. However, the original definition of VMAG is not removed until after the expansion is complete.

Another example of a nested macro is given in Chapter 7, Figure 7-4.

3.8.1 ASCII Interpretation

If the reverse slash (\) is used as the first character of an argument in a macro call, the value of the following symbol is converted to a 7-bit ASCII character in the current radix. If the call is

```
MAC \A
```

and if A=500 (in the current radix), this generates the three ASCII character "500".

Chapter 4
Error Detection

MACRO-10 makes many error checks as it processes source language statements. If an apparent error is detected, the assembler prints a single letter code in the left-hand margin of the program listing (and on the TTY, unless the listing is on the TTY), on the same line as the statement in question.

The programmer should examine each error indication to determine whether or not correction is required. At the end of the listing, the assembler prints a total of errors found; this is printed even if no listing is requested.

Each error code indicates a general class of errors. These errors, however, are all caused by illegal usage of the MACRO-10 language, as described in the preceding three chapters of this manual.

4.1 SINGLE-LETTER ERROR CODES

Table 4-1 lists the single-letter error codes output by the assembler.

TABLE 4-1
Error Codes

Error Code	Meaning	Explanation
A	Argument error in pseudo-op	This is a broad class of errors which may be caused by an improper argument in a pseudo-op. The following represent the majority of the conditions which would cause an A code error. a. Symbol used is improperly formed. For example AB?CD would result in an A code since the character ? is not in the Radix 50 character set. b. IFIDN comparison string is too large. c. OPDEF of macro is SYN,. d. OPDEF, no code generated. e. Invalid SIXBIT character in SIXBIT/TEST Tab/ f. Byte size too big in byte (>4D36). g. Radix 50 code not absolute, that is Radix 50 FOO,BAR where FOO is not Ø-74 absolute. h. End of line on IFx SYM reached before an < character is seen. i. Assignment made in an address field (e.g., MOVE A=1Ø). j. Assignment of a label (e.g., TAG: TAG=1). k. Missing symbol in SYN SYM1,. l. Unknown symbol in SYN,. m. Missing right parenthesis ()) in index (e.g., MOVE 1,(2...). n. Missing left parenthesis in BYTE statement (e.g., BYTE 3 1, 1, 1). o. No comma after repeat count (e.g., REPEAT 3 <). p. IRP not in a macro.

TABLE 4-1 (Cont)

Error Code	Meaning	Explanation
		q. Argument for IRP is not a dummy symbol; for example DEFINE FOO (A) < IRP(B), <>> r. IRP argument is a created symbol. s. STOP1 not in IRP.
D	Multiply-defined symbolic reference error	This statement contains a tag which refers to a multiply-defined symbol. It is assembled with the first value defined.
E	External symbol error	Improper usage of an external symbol. The following represent the majority of the conditions which will cause an E code error. a. Attempting to use the same symbol as both an external and an internal symbol. For example, the statement EXT: EXTERN TXT,BRT,EXT attempts to use EXT as both an external and an internal symbol. b. Using an external symbol for an AC or index. c. Using an external symbol for IFx. d. Using an external symbol in a LOC, RELOC, PHASE, HISEG or TWOSEG pseudo-op. e. Using an external symbol in the left half of IOWD. f. Using an external symbol in an ARRAY size statement. g. Using an external symbol in a REPEAT count.
L	Literal error	A literal is improper. A literal must generate 1 to 18 words. EXP [SIXBIT //];NO CODE GENERATED
M	Multiply-defined symbol	A symbol is defined more than cnce. The symbol retains its first definition, and the error message M is typed out during pass 1. If this type of error occurs during pass 2, it is a phase error (see below).

TABLE 4-1 (Cont)

Error Code	Meaning	Explanation
		If a symbol is first defined as a #-sign suffixed tag, and later as a label, it retains the label definition. Examples: A: ADD 3,X; A: MOVE ,C; M ERROR A: ADD 3,X#; X: MOVE ,C; X IS ASSIGNED THE CURRENT VALUE OF THE LOCATION COUNTER Multiple appearances of the TITLE pseudo-op (which generates both a title line and program name) are flagged as "M" (Multiple definition) errors.
N	Number error	A number is improperly entered. The following represent the majority of the conditions which would cause an N-type error. a. The number exceeds the permitted range (e.g., ↑F13.33E38). b. A number does not follow a B shift operator (e.g., ↑D15BZ). c. The number exceeds the current radix (e.g., if radix is 8 the single character 9 is acceptable but the number 19 is not acceptable). d. The binary shift given does not represent an absolute numeric. For example, 4B<sym> is illegal if sym is relocatable. e. The character given after an up arrow (↑) is not B, O, F, L or D. f. The expression given after E was not a signed (±) number.
O	Operation code undefined	The operation field of this statement is undefined. It is assembled with a numeric code of Ø.
P	Phase error	A symbol is assigned a value as a label during pass 2 different from that which it received during pass 1. In general, the assembler should generate the same number of program locations in pass 1 and pass 2, and any discrepancy causes a phase error.

TABLE 4-1 (Cont)

Error Code	Meaning	Explanation
		For example, if an assembly conditional, IF1, generates three instructions, a phase error results unless another conditional, such as IF2, generates three program locations during pass 2.
Q	Questionable	This is a broad class of possible errors in which the assembler finds ambiguous language. Q-errors may or may not generate correct code; the assembler will attempt to do what the programmer intended. The following represent the majority of the conditions which would cause a Q-type error. a. More than 5 ASCII characters are detected by the assembler before a closing " symbol is detected (e.g., "ABCDEFG" or "ABC). When more than 5 characters are detected, only the first 5 are stored. b. More than 6 SIXBIT characters are detected by the assembler before a closing " symbol is detected. As in item a, only the first 6 characters are stored when more than 6 are detected. c. A given number is too big; in such cases, the high-order bits of the number are lost. d. E in a number is followed by something other than a signed (±) numeric (e.g., 1.ØEX). e. An illegal control character is detected in a line. ASCII characters Ø-4Ø are not permitted except for HT, LF, VT, EF, CR and ESC. f. A comma is detected in a statement after all of the required fields have been filled (e.g., MOVE 1,2,) g. Relocatable code is generated by the assembler before either the pseudo-op HISEG or TWOSEG is found by the assembler.

TABLE 4-1 (Cont)

Error Code	Meaning	Explanation
		h. An instruction address pointer is detected by the assembler which does not have either all Ø's or all 1's in the left half of its word location.
R	Relocation error	A LOC or RELOC pseudo-op is used improperly. All of the following conditions will cause an R-type error. a. An expression or assignment is made in which relocation is not Ø or 1 (e.g., A+B, A*Z, 1/B, or X=3*B where a and B are relocatable). b. A BLOCK statement is written with a relocatable size (e.g., BLOCK: A where A is relocatable). c. A relocatable variable is used to specify an accumulator (e.g., MOVE A,1 where A is relocatable).
U	Undefined symbol	A symbol is undefined.
V	Value previously undefined	A symbol used to control the assembler is undefined prior to the point at which it is first used. Causes error message in pass 1. For example, BLOCK:A where A is undefined.
X	Macro definition error	An error occurred in defining or calling a macro.

Error messages printed during pass 1 consist of two parts. The page and sequence number, if used, plus the most recently used label is printed on the first line. This material is then followed by +n, where n is the (decimal) number of lines of coding between the labeled statement and the statement containing an error. The second line of the error message is a copy of the erroneous line of coding, with a letter code in the left-hand margin to indicate the type of error. If more than one type of error occurs on the same line, more than one letter is printed; but if the same type of error occurs more than once in the same line, a single letter code is printed.

During pass 2, as the listing is printed out, lines containing errors are marked by letter codes, and a total of errors found is printed at the end of the listing.

4.2 ERROR MESSAGES

The following error messages may be typed out on the user's terminal. Any error message preceded by a question mark (?) is treated as a fatal error when running under the BATCH processor (the run is terminated by BATCH).

Message	Meaning
END OF PASS 1	This message indicates that manual loading is required to start pass 2. This message is issued when the input is paper tape, cards or keyboard.
LOAD THE NEXT FILE	This message indicates that manual loading is required when the files to be input are on paper tape, cards or being input from the terminal.
?COMMAND ERROR	This message indicates that an error was found in the last command string input.
?INSUFFICIENT CORE	Not enough core is available.
?.PDL OVERFLOW,TRY/P	This message indicates that the pushdown list is too small. The use of a /P switch increases the size of the pushdown list by 80 locations. As many /P switches may be used as desired.
?DEV NOT AVAILABLE	The specified device cannot be initialized because another user is using it.
?N ERRORS DETECTED ?1 ERROR DETECTED NO ERRORS DETECTED	These three statements indicate the number of errors detected by MACRO during assembly (errors marked by letter codes on the listing. Under BATCH if any error occurs, the run is terminated.

?NO END STATEMENT ENCOUNTERED ON INPUT FILE

This message is followed by one of the following:

IN LITERAL
IN DEFINE
IN TEXT
IN CONDITIONAL OR REPEAT
IN CONDITIONAL
IN MACRO CALL

and

ON PAGE xxx AT yyy

where xxx = a page number and yyy = a sequence number or TAG+offset.

NOTE

The foregoing type of message usually indicates some error other than a missing END statement. For example:

```
ASCIZ/TEXT
    .
    .
    .
END
```

where TEXT has not been closed or

```
JRST [statements
    .
    .
    .
END
```

where the literal has not been closed.

?PRGEND ERROR

This error message indicates that the macro failed to restore the symbol table for one of the programs.

?TOO MANY UNIVERSALS

This error message indicates that too many universal programs have been assembled. The number of universal programs permitted is a Macro parameter; to prevent this error from reoccurring, the user must reassemble macro with a new parameter which will permit the desired assembly.

?CANNOT FIND UNIVERSAL xxx

This message indicates that a search has been made for UNIVERSAL program xxx but it was not found (i.e., it was not assembled). To clear this error the program xxx must be assembled.

xxx UNASSIGNED DEFINED AS IF EXTERNAL

This message indicates that an undefined symbol was found and that it has been treated as if it was an external symbol.

PROGRAM BREAK IS xxx

Where xxx is the length of the low segment.

HI-SEG BREAK IS xxx

Where xxx is the length of the relocated high segment.

ABSOLUTE BREAK IS xxx	Where xxx is the highest absolute address seen over 14Ø.
xK CORE USED	Message indicates the size of the low segment used to assemble the source program.

?UNIVERSAL PROGRAM(S) MUST HAVE SAME OUTPUT SPECIFICATIONS AS OTHER FILES

This error message indicates that a universal program was found which did not have either a binary or a listing device specified but all of the following files had such specifications. For example the sequence

```
*,←UNIV
*rel,List←file
```

is illegal. The legal sequence would be

```
*rel, LIST←UNIV
*REL,LIST←FILE
```

?ERROR WHILE EXPANDING xxx

This error message indicates that the assembler experienced an internal error while expanding the macro identified as xxx. Errors of this type are extremely rare; if it occurs the user should rewrite the macro involved.

4.2.1 LOOKUP Errors

The following error messages can occur during a monitor LOOKUP, RENAME or ENTER request on disk. The form of the error messages is:

? filename.ext then one of the following

(Ø) FILE WAS NOT FOUND or (Ø) ILLEGAL FILE NAME (used for enter errors only)

(1) NO DIRECTORY FOR PROJECT-PROGRAMMER NUMBER

(2) PROTECTION FAILURE

(3) FILE WAS BEING MODIFIED

(4) RENAME FILE NAME ALREADY EXISTS

(5) ILLEGAL SEQUENCE OF UUOS

(6) BAD UFD OR BAD RIB

(7) NOT A SAV FILE

(1Ø) NOT ENOUGH CORE

(11) DEVICE NOT AVAILABLE

(12) NO SUCH DEVICE

(13) NOT TWO RELOC REG. CAPABILITY

(14) NO ROOM OR QUOTA EXCEEDED

(15) WRITE LOCK ERROR

(16) NOT ENOUGH MONITOR TABLE SPACE
(17) PARTIAL ALLOCATION ONLY
(2Ø) BLOCK NOT FREE ON ALLOCATION
(21) CAN'T SUPERSEDE (ENTER) AN EXISTING DIRECTORY
(22) CAN'T DELETE (RENAME) A NON-EMPTY DIRECTORY
(23) SFD NOT FOUND
(24) SEARCH LIST EMPTY
(25) SFD NESTED TOO DEEPLY
(26) NO-CREATE ON FOR SPECIFIED SFD PATH

If the error code (V) is greater than 26_8, the error message:

?(V) LOOKUP,ENTER, OR RENAME ERROR

is printed.

4.2.2 MACRO I/O Error Messages

The following error messages are generated for error conditions found during input or output operations with peripheral devices. The messages are self-explanatory.

```
?OUTPUT WRITE-LOCK ERROR DEVICE xxx
?OUTPUT DATA ERROR DEVICE xxx
?OUTPUT CHECKSUM OR PARITY ERROR DEVICE xxx
?OUTPUT QUOTA EXCEEDED ON DEVICE xxx
?OUTPUT BLOCK TOO LARGE DEVICE xxx
?MONITOR DETECTED SOFTWARE INPUT ERROR DEVICE xxx
?INPUT DATA ERROR DEVICE xxx
?INPUT CHECKSUM OR PARITY ERROR DEVICE xxx
?INPUT BLOCK TOO LARGE DEVICE xxx
```

Chapter 5
Relocation

The MACRO-10 assembler will create a relocatable object program. This program may be loaded into any part of memory as a function of what has been previously loaded. To accomplish this, the address field of some instructions must have a relocation constant added to it. This relocation constant, added at load time by the PDP-10 Loader, equals the difference between the memory location an instruction is actually loaded into and the location it is assembled into. If a program is loaded into cells beginning at location 1400_8, the relocation constant k would be 1400_8.

Not all instructions must be modified by the relocation constant. Consider the two instructions:

```
MOVEI 2,.-3
MOVEI 2,1
```

The first is used in address manipulation and must be modified; the second probably should not. To accomplish the relocation, the actual expression forming an address is evaluated and marked for modification by the Linking Loader. Integer elements are absolute and not modified. Point elements (.) are relocatable and are always

modified.[1] Symbolic elements may be either absolute or relocatable. If a symbol is defined by a direct assignment statement, it may be relocatable or absolute depending on the expression following the equal sign (=). If a symbol is defined as a macro, it is replaced by the string and the string itself is evaluated. If it is defined as a label or a variable (#), it is relocatable.[1] Finally, references to literals are relocatable.[1]

To evaluate the relocatability of an expression, consider what happens at load time. A constant, k, must be added to each relocatable element and the expression evaluated. Consider the expression:

$$X - A+2*B-3*C + D$$

where A,B,C, and D are relocatable. Assume k is the relocation constant. Adding this to each relocatable term we get:

$$X_R = (A+K)+2*(B+K)-3*(C+K)+(D+K)$$

This expression may be rearranged to separate the k's, yielding:

$$X_R = A+2*B-3*C+D+K$$

This expression is suitable for relocation since it involves the addition of a single k. In general, if the expression can be rearranged to result in the addition of

Ø*K	The expression is legal and fixed.
1*K	The expression is legal and relocatable.
N*K	Where n is any positive or negative integer other than 0 or 1, the expression is illegal.

Finally, if the expression involves k to any power other than 1, the expression is illegal. This leads to the following conventions:

a. Only two values of relocatability for a complete expression are allowed (e.g., nK where n = Ø or +1).
b. An element may not be divided by a relocatable element.
c. Two relocatable elements may not be multiplied together.
d. Relocatable elements may not be combined by the Boolean operators.

---

[1]Except under the LOC code or PHASE code which specifies absolute addressing.

If any of these rules is broken, the expression is illegal and the assembled code is flagged.

If A, C, and B are relocatable symbols, then:

```
A+B-C     is relocatable
A-C       is fixed
A+2       is relocatable
2*A-B     is relocatable
2&A-B     is illegal
```

A storage word may be relocatable in the left half as well as in the right half. For example:

```
XWD A,B
```

Chapter 6
Assembly Output

There are two MACRO-10 outputs, a binary program and a program listing. The listing is controlled by the listing control pseudo-ops, which were described in Chapter 2.

6.1 ASSEMBLY LISTING

All MACRO-10 programs begin with an implicit LIST statement.

Each page begins with a TITLE line; this line contains the program's name, the assembler version, the time of assembly, the date of assembly and a page number. The page number is incremented by a Form-Feed or PAGE pseudo-op.

If the code listed requires more than one page, the basic page number given on the title line does not change but a subpage number is added and incremented for each additional page (e.g., 6-1, 6-2, 6-3, etc.).

The second line printed on each page is the SUBTITLE line. This line contains the program filename and extensions, creation time, creation date and any given subtitle.

From left to right, the columns on a listing page contains:

a. The 6-digit address of each storage word in the binary program. These are normally sequential location counter assignments. In the case of a block statement, only the address of the first word allocated is listed. An apostrophe following the address indicates that the address is relocatable.

b. The assembled instructions and data words shown in one of several forms for easier reading (see paragraph 2.6.3).

c. The source program statement, as written by the programmer, followed by comments, if any.

If an error is detected during assembly of a statement, an error code is printed on that statement's line, near the left edge of the page. If multiple errors of the same type occur in a particular statement, the error code is printed only once; but if several errors, each of a different type, occur in a statement, an error code is printed for each error. The total number of errors is printed at the end of the listing.

The program break is also printed at the end of the listing. This is the highest relocatable location assembled, plus one. This is the first location available for the next program or for patching.

6.2 BINARY PROGRAM OUTPUT

The assembler produces binary program output in four formats. The choice depends on whether the program is relocatable or absolute, and on the loading procedure to be used to load the program for execution.

6.2.1 Relocatable Binary Programs - LINK Format

Most binary programs are output in LINK format. Like the RELOC statement, the LINK format output is implicit and is automatically produced for all relocatable MACRO-10 programs unless another format (RIM, RIM10, RIM10B) is explicitly requested. The LINK format is the only format that may be used with the Linking Loader.

The Linking Loader loads subprograms into memory, properly relocating each one and adjusting addresses to compensate for the relocation.

It also links external and internal symbols to provide communication between independently assembled subprograms. Finally, the Linking Loader loads required subroutines while in Library Search Mode.

Data for the Linking Loader is formatted in blocks. All blocks have an identical format. The first word of a LINK block consists of two halves. The left half is a code for the block type, and the right half is a count of the number of data words in the block. The data words are grouped in sub-blocks of 18 items. Each 18-word sub-block is preceded by a relocation word. This relocation word consists of 18 2-bit bytes. Each byte corresponds to one word in the sub-block, and contains relocation information regarding that word.

If the byte value is:

0	no relocation occurs
1	the right half is relocated
2	the left half is relocated
3	both halves are relocated

These relocation words are not included in the count; they always appear before each sub-block of 18 words or less to ensure proper relocation.

All relocatable programs may be stored in LINK format, including programs on paper tape, DECtape, magnetic tape, punched cards, and disks. This format is totally independent of logical divisions in the input medium. It is also independent of the block type.

6.2.1.1 LINK Formats for the Block Types - Block Type 1 Relocatable or Absolute Programs and Data

WORD 1	The location of the first data word in the block
WORD 2	A contiguous block of program or data words (18 or less)
.	
.	
.	
WORD N	(N, from 1 to 18, must be less than 2000,000 octal)

Block Type 2 Symbols

Consists of word pairs

1ST WORD	Bits 0-3 code bits
1ST WORD	Bits 4-35 radix 50 representation of symbol (see below)
2ND WORD	Data (value or pointer)
CODE Ø4:	Global (internal) definition
2ND WORD	Bits 0-35 value of symbol
CODE 1Ø:	Local definition
2ND WORD	Bits 0-35 value of symbol
CODE 6Ø:	Chained global requests:
2ND WORD	Bits 0-17=0
2ND WORD	Bits 18-35 pointer to first word of chain requiring defintion (refer to the LOADER manual)
CODE 6Ø:	Global symbol additive request: (refer to the LOADER manual)

Block Type 3 Load Into High Segment

When block type 3 is present in a relocatable binary program, the Loader loads the program into the high segment if the system has re-entrant (two-segment) capability. When used, block type 3 appears immediately after the name block (type 6).

The first word is

```
XWD 3,,2
```

The second word is the relocation word

```
2ØØØØØ,,Ø
```

The third word is

```
XWD HISEG BREAK,,TWOSEG ORIGIN
```

where twoseg origin is 400000 by default.

With the TWOSEG pseudo-op, the left half of the third word is negative. On a two-segment machine,this is ignored except to set a LOADER flag. On a one-segment machine, the difference is assumed to be the maximum length of the high segment. A one-pass assembler does not know this legth at the start of pass 1, therefore

```
XWD 400000,,400000
```

is used to signal two segments to a two-segment machine.

On a one-segment machine, this instruction gives the error message

```
TWO SEGMENTS ILLEGAL
```

since the LOADER does not know how much space to reserve for the high segment.

Block Type 4 Entry Block

This block contains a list of Radix 50 symbols, each of which may contain a 0 or 1 in the high-order code bit. Each represents a series of logical AND conditions. If all the globals in any series are requested, the following program is loaded. Otherwise, all input is ignored until the next end block. This block must be the first block in a program.

Block Type 5 End Block

This is the last block in a program. It contains two words, the first of which is the program break, that is, the location of the first free register above the program. (Note: This word is relocatable.) It is the relocation constant for the following program loaded. The second word is the highest absolute location seen (if greater than 140). In a two-segment program, the two words are:

1) the high segment break followed by
2) the low segment break.

Block Type 6 Name Block

The first word of this block is the program name RADIX 50). It must appear before any type 2 blocks. The second word, if it appears, defines the length of common. The left half of the second word is used to describe the compiler type that produced the binary file, 0 in the case of MACRO.

Block Type 7 Starting Address

The first word of this block is the starting address of the program. The starting address for a relocatable program may be relocated by means of the relocation bits.

Block Type 10 Internal Request

Each data word is one request. The left half if the pointer to the program. The right half is the value. Either quantity may be relocatable.

6.2.2 Absolute Binary Programs

Three output formats are available for absolute (non-relocatable) binary programs. These are requested by the RIM, RIM10, and RIM10B statements.

6.2.2.1 RIM10B Format - If a program is assembled into absolute locations (not relocatable), a RIM10B statement following the LOC statement at the beginning of the source program causes the assembler to write out the object program in RIM10B format. This format is designed for use with the PDP-10 hardware read-in feature.

The program is punched out during pass 2, starting at the location specified in the LOC statement. If the first two statements in the program are:

```
LOC 1ØØØ)
RIM1ØB )
```

the assembler assembles the program with absolute addresses starting at 1000, and punches out the program in RIM10B format, also starting at location 1000. The programmer may reset the location counter duringassembly of his program, but only one RIM10B statement is needed to punch out the entire program.

In RIM10B format (see Figures 6-1 and 6-2), the assembler punches out the RIM10B Loader (Figure 6-2), followed by the program in 17-word (or less) data blocks, each block separated by blank tape. The assembler inserts an I/O transfer word (IOWD) preceding each data block, and also inserts a 36-bit checksum following each data

block as shown in Figure 6-1. The word count in the IOWD includes only the data words in the block, and the checksum is the simple 36-bit added checksum of the IOWD and the data words.

Data blocks may contain less than 17 words. If the assembler assigns a non-consecutive location, the current data block is terminated, and an IOWD containing the next location is inserted, starting a new data block.

The transfer block consists of two words. The first word of the transfer block is an instruction obtained from the END statement (see Section 6.2.2.4) and is executed when the transfer block is read. The second is a dummy word to stop the reader.

6.2.2.2 RIM10 Format - Binary programs in RIM10 format are absolute, unblocked, and not checksummed. When the RIM10 statement follows a LOC statement in a program, the assembler punches out each storage word in the object program, starting at the absolute address specified in the LOC statement.

RIM10 writes an arbitrary "paper tape". If it is in the format below, it can be read in by the PDP-10 Read-In-Mode hardware.

```
IOWD N,FIRST)
```

where n is the length of the program including the transfer instruction at the end, and FIRST is the first memory location to be occupied. The last location must be a transfer instruction to begin the program, such as:

```
JRST 4,GO)
```

For example, if a program with RIM10 output has its first location at START and its last location at FINISH, the programmer may write

```
IOWD FINISH-START+1,START)
```

NOTE

In cases where the location counter is increased but no binary output occurs (such as with BLOCK, LOCn, and LIT pseudo-ops), MACRO inserts a zero word into the binary output file for each location skipped by the location counter.

6.2.2.3 RIM Format - This format, which is primarily used in PDP-6 systems, consists of a series of paired words. The first word of each pair is a paper-tape read instruction giving the core memory address of the second word. The second word is the data word.

```
DATAI PTR,LOC
DATA WORD
```

The last pair of words is a transfer block. The first word is an instruction obtained from the END statement (see Section 6.2.2.4) and is executed when the transfer block is read. The second word is a dummy word to stop the reader.

The loader that reads this format is:

```
    LOC 2Ø

    CONO PTR,6Ø
A:  CONSO PTR,1Ø
    JRST .-1
    DATAI PTR,B
    CONSO PTR,1Ø
    JRST .-1
B:  Ø
    JRST A
```

This loader is normally toggled into memory and started at location 20.

6.2.2.4 END Statements - When the programmer wants output in either RIM or RIM10B format, he may insert an instruction or starting address as the first word in the two-word transfer block by writing the instruction or address as an argument to the END statement. The second word of the transfer block is zero. In RIM10 assemblies, this argument is ignored.

If bits 0 through 8 of the instruction are zero, MACRO will insert the instruction JRST 4,0, causing a halt when executed. The END statements

```
END SA)     OR   END JRST SA)
```

will start automatically at address SA.

Some other examples:

	1st Transfer Block Word
END@XCT 1234	XCT@1234
END Z4,SA	JRST 4,SA
END	JRST 4,Ø

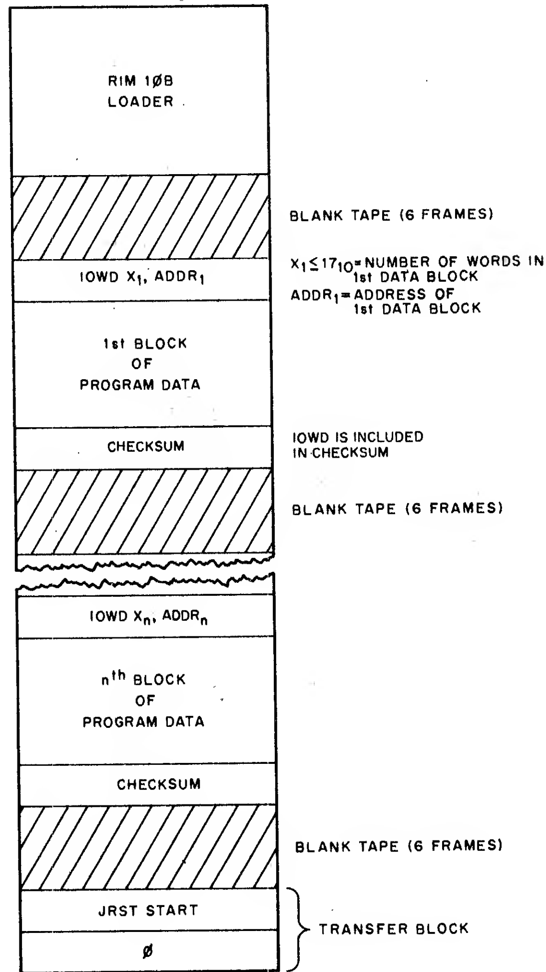


Figure 6-1 General RIM10B Format

```
                XWD -16,Ø
ST:             CONO PTR,6Ø
ST1:            HRRI A,RD+1
RD:             CONSO PTR,1Ø
                JRST
                DATAI PTR, @TBL1-RD+1(A)
                XCT         TBL1-RD+1(A)
                XCT         TBL2-RD+1(A)
A:              SOJA A,
TBL1:           CAME CKSM,ADR
                ADD CKSM,1 (ADR)
                SKIPL CKSM,ADR
TBL2:           JRST 4,ST
                AOBJN ADR,RD
ADR:            JRST     ST1
CKSM=ADR+1
```

Figure 6-2 RIM10B Loader

Chapter 7
Programming Examples

This chapter contains four examples of macro programs. The first example (Figure 7-1) presents a MACRO-10 routine for calculating the logarithm of a complex argument. This routine begins with an ENTRY statement identifying this library routine as CLOG (Complex Logarithm Function) and uses three external routines, ALOG, ATAN2 and CABS.

The second example (Figure 7-2) is the universal parameter file DEF40.MAC which is used to produce the KA-10 version of LIB40. It contains conditional assembly switches to select either a PDP-6, KA10 or KI10 mode. It defines the accumulator conventions and macros which simulate the KI10 hardware operations on the KA10 processor.

Example 3 (Figure 7-3) uses DEF40 (via the SEARCH pseudo-op) for its accumulators and the macros for DMOVE, DMOVEM and FLADD. The macro FLADD is expanded twice to show the effect of LALL on lines which generate text but no binary. The effect of SALL is also shown.

Example 4 (Figure 7-4) shows nested macros which use IRPC. The desired operation is to take an ASCII text string and store the

characters four per word, left-justified, with the character count stored in the first nine bits of the first word.

The TEXT macro counts the string characters and invokes the CODE macro to store the characters four per word.

The CODE macro invokes a SHIFT macro which left-justifies the last word if it is not already left-justified. The first part of the example shows the normal listing, then SALL is set to show what code the macros are generating.

```
CLOG     MACRO 47(113) 13:54  4-APR-72 PAGE 1
CLOG     MAC        4-APR-72 13:53        APRIL 3, 1972

                                              TITLE CLOG
                                              SUBTTL APRIL 3, 1972

                                      COMMENT ;COMPLEX LOGARITHMIC FUNCTION
                                      THIS ROUTINE CALCULATES THE LOGARITHM OF A COMPLEX ARGUMENT
                                      Ƶ = X+1*Y WITH THE FOLLOWING ALGORITHM

                                      LOG8Ƶ = LOG 8ABSF (Ƶ) + I*THETA
                                      WHERE ABSF 8Ƶ) = SQRT (X↑2 + Y↑2)
                                      AND THETA IS THE COMPLEX ANGLE ATAN(Y/X)

                                      THE ROUTINE IS CALLED IN THE FOLLOWING MANNER:
                                              JSA     Q,CLOG
                                              EXP     ARG
                                      THE REAL PART OF THE ANSWER IS RETURNED IN ACCUMULATOR A
                                      AND THE IMAGINARY PART IS RETURNED IN ACCUMULATOR B;

                                              ENTRY   CLOG
                                              EXTERN  ALOG,ATAN2,CABS

                    ØØØØØØ                    A=Ø
                    ØØØØØ1                    B=1
                    ØØØØ1Ø                    C=1Ø
                    ØØØØ11                    D=11
                    ØØØØ16                    Q=16

ØØØØØØ' ØØØØØØ  ØØØØØØ                CLOG:   Ø                       ;ENTRY TO COMPLEX LOG ROUTINE
ØØØØØ1' 2Ø1 1Ø 1 16 ØØØØØØ                    MOVEI   C,@(Q)          ;GET ADDRESS OF COMPLEX ARGUMENT
ØØØØØ2' 2ØØ 11 Ø 1Ø ØØØØØ1                    MOVE    D,1(C)          ;GET REAL PART OF ARGUMENT
ØØØØØ3' 2ØØ 1Ø Ø 1Ø ØØØØØØ                    MOVE    C,(C)           ;GET REAL PART OF ARGUMENT
ØØØØØ4' 266 Ø1 Ø ØØ ØØØØØØ*                   JSA     1,CABS          ;CALCULATE MAGNITUDE OF Ƶ
ØØØØØ5' ØØØØØØ  ØØØØØØ                        EXP     C               ;ADDRESS OF COMPLEX ARGUMENT
ØØØØØ6' 266 16 Ø ØØ ØØØØØØ*                   JSA     Q,ALOG          ;CALCULATE LOG(ABSF (Ƶ))
ØØØØØ7' ØØØØØØ  ØØØØØØ                        EXP     A               ;ADDRESS FOR LOG ROUTINE
ØØØØ1Ø' 25Ø ØØ Ø ØØ ØØØØ1Ø                    EXCH    A,C             ;SWAP ANSWER WITH REAL PART
ØØØØ11' 266 16 Ø ØØ ØØØØØØ*                   JSA     Q,ATAN2         ;CALCULATE ANGLE AS ATAN(Y/X)
ØØØØ12' ØØØØØØ  ØØØØ11                        EXP     D               ;ADDRESS OF Y
ØØØØ13' ØØØØØØ  ØØØØØØ                        EXP     A               ;ADDRESS OF C
ØØØØ14' 2ØØ Ø1 Ø ØØ ØØØØØØ                    MOVE    B,A             ;PUT THETA IN IMAGINARY PART
ØØØØ15' 2ØØ ØØ Ø ØØ ØØØØ1Ø                    MOVE    A,C             ;RESTORE REAL PART
ØØØØ16' 267 16 Ø 16 ØØØØØ1                    JRA     Q,1(Q)          ;EXIT
                                              END

NO ERRORS DETECTED

PROGRAM BREAK IS ØØØØ17

2K CORE USED
```

Figure 7-1   MACRO Program CLOG

```
CLOG        MACRO 47(113) 13:54   4-APR-72 PAGE 2
CLOG        MAC       4-APR-72 13:53          SYMBOL TABLE

A                     ØØØØØØ
ALOG                  ØØØØØ6' EXT
ATAN2                 ØØØØ11' EXT
B                     ØØØØØ1
C                     ØØØØ1Ø
CABS                  ØØØØØ4' EXT
CLOG                  ØØØØØØ' ENT
D                     ØØØØ11
Q                     ØØØØ16
```

```
UNIVERSAL DEF4Ø PARAMETER FILE FOR FORTRAN IV LIBRARY
SUBTTL  V32(343)        23-NOV-71       /TWE

IFNDEF PDP6.<IFNDEF KA1Ø,<IFNDEF KI1Ø,<KA1Ø==1>>>
IFNDEF PDP6,<PDP6==Ø>                ;CONDITIONAL ASSEMBLY PARAMETERS
IFNDEF KA1Ø,<KA1Ø==Ø>
IFNDEF KI1Ø,<KI1Ø==Ø>
IFN <PDP6!KA1Ø!KI1Ø-PDP6-KA1Ø-KI1Ø>,
        <PRINTX MACHINE PARAMETERS DEFINED WRONG>


;ACCUMULATOR ASSIGNMENTS
A=Ø
B=1
C=2
D=3
E=4
F=5
G=6
H=7

Q=16     ;FOR JSA AND ARG ADDRESS FOR PUSHJ
P=17     ;PUSH DOWN POINTER

IFE KA1Ø,<
DEFINE DOUBLE (A,B)<
        A
        B>
>

IFN KA1Ø,<
DEFINE DOUBLE (A,B)<
ƵƵ1,==A& 777ØØØ,,Ø>
IFL ƵƵ1.,<ƵƵ1.==-ƵƵ1.-<1ØØØ,,Ø>>
ƵƵ1,==ƵƵ1.-<Ø33ØØØ,,Ø>
IFE B,<ƵƵ1.==Ø>
ƵƵ2,==ƵƵ1.+<<B+2ØØ>←-B>&<ØØØ777,,777777>
IFL ƵƵ1.,<ƵƵ2.==Ø>
        A
        ƵƵ2
SUPPRESS ƵƵ1,,ƵƵ2.>
DEFINE DMOVE(AC,M)<
        IFL <Ƶ M>-<@>,<
        MOVE AC,M
        MOVE AC+1,1+M>

        IFGE <Ƶ M><@>,<
        MOVEI AC+1,M
        MOVE AC,(AC+1)
        MOVE AC+1,1(AC+1)>
>

DEFINE  DMOVN(AC,M)<
        DMOVE AC,M
        DFN AC,AC+1>

DEFINE  DMOVEM(AC,M)<
        MOVEM AC,M
        MOVEM AC+1,1+M

>
```

Figure 7-2   Universal Parameter File DEF40.MAC

```
DEFINE FLMUL (AC,M,%OV)<
        MOVEM AC,AC+2
        FMPR AC+2,1+M
        JFCL (2)
        FMPR AC+1,M
        JFCL (2)
        UFA AC+1,AC+2
        JFCL
        FMPL AC,M
        JOV %OV
        UFA AC+1,AC+2
        FADL AC,AC+2
%OV:>

DEFINE FLDIV(AC,M,%OV)<
        FDVL AC,M
        JOV %OV
        MOVN AC+2,AC
        FMPR AC+2,1+M
        JFCL (2)
        UFA AC+1,AC+2
        FDVR AC+2,M
        JFCL
        FADL AC,AC+2
%OV:>

DEFINE FLADD(AC,M,%OV)<
        UFA AC+1,1+M
        FADL AC,M
        JOV %OV
        UFA AC+1,AC+2
        FADL AC,AC+2
%OV:>

>       ;END OF KA1Ø CONDITIONAL
IFN KI1Ø,<
OPDEF   FLADD   [DFAD]
OPDEF   FLMUL   [DFMP]
OPDEF   FLDIV   [DFDV]

DEFINE DFN (A,B)<       DMOVN A,A
IFN  <<A+1>&17-<B>>,<PRINTX "DMOVN A,A" CAN'T REPLACE "DFN A,B">
>
>       ;END OF KI1Ø CONDITIONAL

END
```

```
TEST SOME MACROS          MACRO 47(113) 13:48  5-APR-72 PAGE 1
TEST     MAC       5-APR-72 13:4Ø            %1 5-APR-72
                                                     TITLE TEST SOME MACROS
                                                     SUBTTL  %1 5-APR-72

                                                     SEARCH  DEF4Ø

          ØØØØØØ'                          START:
                                                     DMOVE   A,(Q)
          ØØØØØØ' 2ØØ ØØ Ø 16 ØØØØØØ                 MOVE A,(Q)                ↑;SIMPLE DOUBLE MOVE
          ØØØØØ1' 2ØØ ØØ Ø 16 ØØØØØ1                 MOVE A+1,1+(Q)

                                                     DMOVE   A, (Q)             ;THIS ONE INDEXED
          ØØØØØ2' 2Ø1 Ø1 1 16 ØØØØØØ                 MOVEI A+1, (Q)
          ØØØØØ3' 2ØØ ØØ Ø Ø1 ØØØØØØ                 MOVE A,(A+1)
          ØØØØØ4' 2ØØ Ø1 Ø Ø1 ØØØØØ1                 MOVE A+1,1(A+1)

                                                     DMOVEM  A,E                ;STORE TO MEMORY
          ØØØØØ5' 2Ø2 ØØ Ø ØØ ØØØØØ4                 MOVEM A,E
          ØØØØØ6' 2Ø2 Ø1 Ø ØØ ØØØØØ5                 MOVEM A+1,1+#

                                                     FLADD   A,E
          ØØØØØ7' 13Ø Ø1 Ø ØØ ØØØØØ5                 UFA A+1,1+E
          ØØØØ1Ø' 141 ØØ Ø ØØ ØØØØØ4                 FADL    A,E
          ØØØØ11' 255 1Ø Ø ØØ ØØØØ14'                JOV ,.ØØØ1
          ØØØØ12' 13Ø Ø1 Ø ØØ ØØØØØ2                 UFA A+1,A+2
          ØØØØ13' 141 ØØ Ø ØØ ØØØØØ2                 FADL A,A+2

                                                     LALL                       ;LIST EVERYTHING
                                                     FLADD   A,#
          ØØØØ14' 13Ø Ø1 Ø ØØ ØØØØØ5                 UFA A+1,1+E
          ØØØØ15' 141 ØØ Ø ØØ ØØØØØ4                 FADL A,E
          ØØØØ16' 255 1Ø Ø ØØ ØØØØ21'                JOV ,.ØØØ2
          ØØØØ17' 13Ø Ø1 Ø ØØ ØØØØØ2                 UFA A+1,A+2
          ØØØØ2Ø' 141 ØØ Ø ØØ ØØØØØ2                 FADL A,A+2
          ØØØØ21'                          ..ØØØ2:   ↑

                                                     SALL                       ;CALL AND BINARY ONLY
          ØØØØ21' 2ØØ ØØ Ø ØØ ØØØØØ4                 DMOVE   A,E

                         ØØØØØØ'           END       START

NO ERRORS DETECTED

PROGRAM BREAK IS ØØØØ23

2K CORE USED
```

Figure 7-3 Test Some Macros

```
TEST SOME MACROS        MACRO 47(113) 13:48  5-APR-72 PAGE 2
TEST    MAC      5-APR-72 13:4Ø          SYMBOL TABLE

A               ØØØØØØ
E               ØØØØØ4
Q               ØØØØ16
START           ØØØØØØ'
..ØØØ1          ØØØØ14'
..ØØØ2          ØØØØ21'
```

```
STORE TEXT CHARACTER BY CHARACTER        MACRO 47(113) 14:45 5-APR-72 PAGE 1
TEXT    MAC       5-APR-72 14:44         %1 5-APR-72

                                                 TITLE STORE TEXT CHARACTER BY CHARACTER
                                                 SUBTTL  %1 5-APR-72

                                         DEFINE TEXT (C)<
                                         N==Ø                   ;;INITIAL CONDITION
                                         IRPC C,<N==N+1>        ;;COUNT CHARACTERS
                                         CODE (N,C)             ;;CALL MACRO TO STORE TEXT
                                         >                      ;END OF TEXT MACRO

                                         DEFINE CODE (N,C)<
                                         ƵƵ==N                  ;;INITIAL CONDITION
                                         IRPC C,<
                                         IFN ƵƵ&777B8,<
                                                 EXP ƵƵ         ;;WORD FULL
                                                 ƵƵ==Ø>         ;;START AGAIN WITH Ø
                                         ƵƵ==ƵƵ←9+"C">          ;;END OF IRPC
                                         IFN ƵƵ,<DEFINE SHIFT   ;;REMAINDER LEFT
                                          IFE ƵƵ&777B8,>        ;;IF NOT LEFT JUSTIFIED
                                         ƵƵ==ƵƵ←9               ;;MOVE LEFT 9 BITS
                                         SHIFT>                 ;;RECURSE
                                         >                      ;;END OF DEFINE
                                         SHIFT                  ;;START MOVING LEFT
                                         EXP ƵƵ>                ;;END OF IFN ƵƵ
                                         >                      ;END OF CODE MACRO

                                         TEXT (ABCDEFGHIJKL)↑
      ØØØØØØ' Ø141Ø1  1Ø21Ø3                     EXP ƵƵ
      ØØØØØ1' 1Ø41Ø5  1Ø61Ø7                     EXP ƵƵ
      ØØØØØ2' 11Ø111  112113                     EXP ƵƵ
      ØØØØØ3' 114ØØØ  ØØØØØØ             EXPƵƵ>

                                                 LALL
                                         TEXT (ABCDEFGHIJKL)↑
                      ØØØØØØ             N==Ø
                                         IRPC
                      ØØØØØ1             N==N+1

                      ØØØØØ2             N==N+1

                      ØØØØØ3             N==N+1
```

Figure 7-4   Store Text Character by Character

```
ØØØØØ4          N==N+1
ØØØØØ5          N==N+1
ØØØØØ6          N==N+1
ØØØØØ7          N==N+1
ØØØØ1Ø          N==N+1
ØØØØ11          N==N+1
```

```
STORE TEXT CHARACTER BY CHARACTER          MACRO 47(113) 14:45  5-APR-72 PAGE 1-1
TEXT    MAC      5-APR-72 14:44             %1 5-APR-72

                       ØØØØ12               N==N+1

                       ØØØØ13               N==N+1

                       ØØØØ14               N==N+1


                                            CODE (N,ABCDEFGHIJKL)↑
                       ØØØØ14               ƵƵ==N
                                            IRPC

                                            IFN ƵƵ&777B8,<
                                                     EXP ƵƵ
                                                     ƵƵ==Ø>
                       Ø141Ø1               ƵƵ==ƵƵ←9+"A"

                                            IFN ƵƵ&777B8,<
                                                     EXP ƵƵ
                                                     ƵƵ==Ø>
              ØØØØ14   1Ø11Ø2               ƵƵ==ƵƵ←9+"B"

                                            IFN ƵƵ&777B8,<
                                                     EXP ƵƵ
                                                     ƵƵ==Ø>
              Ø141Ø1   1Ø21Ø3               ƵƵ==ƵƵ←9+"C"

                                            IFN ƵƵ&777B8,<
     ØØØØØ4'  Ø141Ø1   1Ø21Ø3                        EXP ƵƵ
                       ØØØØØØ                        ƵƵ==Ø>
                       ØØØ1Ø4               ƵƵ==ƵƵ←9+"D"

                                            IFN ƵƵ&777B8,<
                                                     EXP ƵƵ
                                                     ƵƵ==Ø>
                       1Ø41Ø5               ƵƵ==ƵƵ←9+"E"
```

```
                            IFN ƵƵ&777B8,<
                                 EXP ƵƵ
                                 ƵƵ==Ø>
ØØØ1Ø4  1Ø51Ø6              ƵƵ==ƵƵ←9+"F"

                            IFN ƵƵ&777B8,<
                                 EXP ƵƵ
                                 ƵƵ==Ø>
1Ø41Ø5  1Ø61Ø7              ƵƵ==ƵƵ←9+"G"
```

```
STORE TEXT CHARACTER BY CHARACTER        MACRO 47(113) 14:45 5-APR-72 PAGE 1-2
TEXT    MAC      5-APR-72 14:44          %1 5-APR-72

                                         IFN ƵƵ&777B8,<
        ØØØØØ5' 1Ø41Ø5  1Ø61Ø7                   EXP ƵƵ
                        ØØØØØØ                   ƵƵ==Ø>
                        ØØØ11Ø           ƵƵ==ƵƵ←9+"H"

                                         IFN ƵƵ&777B8,<
                                                 EXP ƵƵ
                                                 ƵƵ==Ø>
                        11Ø111           ƵƵ==ƵƵ←9+"I"

                                         IFN ƵƵ&777B8,<
                                                 EXP ƵƵ
                                                 ƵƵ==Ø>
                ØØØ11Ø  111112           ƵƵ==ƵƵ←9+"J"

                                         IFN ƵƵ&777B8,<
                                                 EXP ƵƵ
                                                 ƵƵ==Ø>
                110111  112113           ƵƵ==ƵƵ←9+"K"

                                         IFN ƵƵ&777B8,<
        ØØØØØ6' 11Ø111  112113                   EXP ƵƵ
                        ØØØØØØ                   ƵƵ==Ø>
                        ØØØ114           ƵƵ==ƵƵ←9+"L"

                                         IFN ƵƵ, DEFINE SHIFT
                                         <IFE ƵƵ&777B8,<
                                         ƵƵ==ƵƵ←9
                                         SHIFT>
                                         >
                                         SHIFT    ↑IFE ƵƵ&777B8,<
                        114ØØØ           ƵƵ==ƵƵ←9
                                         SHIFT    ↑IFE ƵƵ&777B8,<
                ØØØ114  ØØØØØØ           ƵƵ==ƵƵ←9
                                         SHIFT    ↑IFE ƵƵ&777B8,<
                114ØØØ  ØØØØØØ           ƵƵ==ƵƵ←9
                                         SHIFT    ↑IFE ƵƵ&777B8,<
                                         ƵƵ==ƵƵ←9
                                         SHIFT>
```

```
                                        ↑
                                        ↑
                                        ↑
                                        ↑
      ØØØØØ7' 114ØØØ  ØØØØØØ            EXP ZZ
                                        ↑
                                        ↑
                                        END




STORE TEXT CHARACTER BY CHARACTER        MACRO 47(113) 14:45 5-APR-72 PAGE 1-3
TEXT    MAC       5-APR-72 14:44         %I 5-APR-72

NO ERRORS DETECTED

PROGRAM BREAK IS ØØØØ1Ø

2K CORE USED
```

Appendix A
Op Codes, Pseudo-Ops, and Monitor I/O Commands

This appendix contains a complete list of assembler defined operators including machine instruction mnemonic codes, assembler pseudo-ops, monitor programmed operators, and FORTRAN programmed operators. A programmed operator, or unimplemented user operation code is called a UUO.

A.1 ASSEMBLER PSEUDO-OPS AND MONITOR COMMANDS

The notes specify which pseudo-ops generate data, and which do not. Pseudo-ops that generate data may be used within literals, and in address operand fields.

The initial values given by MACRO-10 to I/O instructions and FORTRAN UUO's for which the octal op code is not shown, are given in the notes and are useful in checking listings.

ARRAY, pseudo-op, generates data
ARG, 320, no-op (same as JUMP)
ASCII, pseudo-op, generates data
ASCIZ, pseudo-op, generates data
ASUPPRESS, pseudo-op, no data generated
BLOCK, pseudo-op, no data generated
BYTE, pseudo-op, generates data
CALL, 040, monitor UUO
CALLI, 047, monitor UUO
CLOSE, 070, monitor UUO
COMMENT, no data generated
DATA, 020, FORTRAN UUO
DEC, pseudo-op, generates data
DEC., 033, FORTRAN UUO
DEFINE, pseudo-op, no data generated
DEPHASE, pseudo-op, no data generated

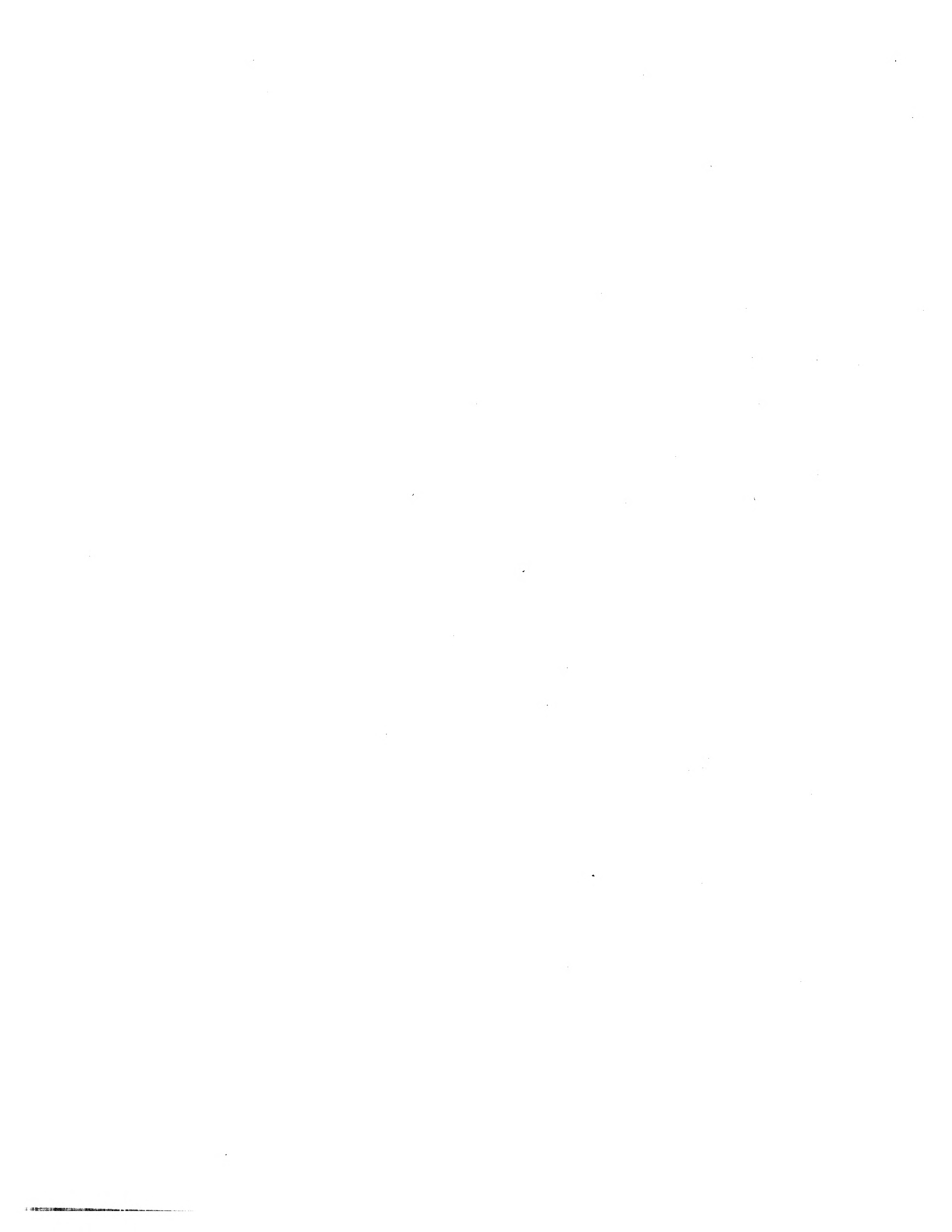
ENC., 034, FORTRAN UUO
END, pseudo-op , no data generated
ENTER, 077, monitor UUO
ENTRY, pseudo-op, no data generated
EXP, pseudo-op, generates data
EXTERN, pseudo-op, no data generated
FIN., 021, FORTRAN UUO
GETSTS, 062, monitor UUO
HISEG, pseudo-op, no data generated
IF1, conditional pseudo-op
IF2, conditional pseudo-op
IFB, conditional pseudo-op
IFDEF, conditional pseudo-op
IFDIF, conditional pseudo-op
IFE, conditional pseudo-op
IFG, conditional pseudo-op
IFGE, conditional pseudo-op
IFIDN, conditional pseudo-op
IFL, conditional pseudo-op
IFLE, conditional pseudo-op
IFN, conditional pseudo-op
IFNB, conditional pseudo-op
IFNDEF, conditional pseudo-op
IN, 056, monitor UUO
IN., 016, FORTRAN UUO
INBUF, 064, monitor UUO
IN., 026, FORTRAN UUO
INIT, 041, monitor UUO
INPUT, 066, monitor UUO
INTEGER, pseudo-op, generates data
INTERN, pseudo-op, no data generated
IOWD, pseudo-op, generates data
IRP, pseudo-op, no data generated
IRPC, pseudo-op, no data generated
LALL, pseudo-op, no data generated
LIST, pseudo-op, no data generated
LIT, pseudo-op, generates data
LOC, pseudo-op, no data generated
LOOKUP, 076, monitor UUO
MLOFF, pseudo-op, no data generated
MLON, pseudo-op, no data generated
MTAPE, 072, monitor UUO
MTOP., 024, FORTRAN UUO
NLI., 031, FORTRAN UUO
NLO., 032, FORTRAN UUO
NOSYM, pseudo-op, no data generated
OCT, pseudo-op, generates data
OPDEF, pseudo-op, no data generated
OPEN, 050, monitor UUO
OUT, 057, monitor UUO
OUT., 017, FORTRAN UUO
OUTBUF, 065, monitor UUO
OUTF., 027, FORTRAN UUO
OUTPUT, 067, monitor UUO
PAGE, pseudo-op, no data generated
PASS2, pseudo-op, no data generated
PHASE, pseudo-op, no data generated
POINT, pseudo-op, generates data
PRINTX, pseudo-op, no data generated
PURGE, pseudo-op, no data generated
RADIX, pseudo-op, no data generated
RADIX50, pseudo-op, generates data
RELEAS, 071, monitor UUO
RELOC, pseudo-op, no data generated
REMARK, pseudo-op, no data generated
RENAME, 055, monitor UUO
REPEAT, pseudo-op, no data generated
RERED., 030, FORTRAN UUO
RESET., 015, FORTRAN UUO
RIM, pseudo-op, no data generated
RIM10, pseudo-op, no data generated
RIM10B, pseudo-op, no data generated
RTB., 022, FORTRAN UUO
SEARCH, pseudo-op, no data generated
SETSTS, 060, monitor UUO
SIXBIT, pseudo-op, generates data
SLIST., 025, FORTRAN UUO
SQUOZE, same as RADIX50
STATO, 061, monitor UUO
STATUS, 062, monitor UUO
STATZ, 063, monitor UUO
STOPI, pseudo-op, no data generated
SUBTTL, pseudo-op, no data generated
SUPPRESS, pseudo-op, no data generated
SYN, pseudo-op, no data generated
TAPE, pseudo-op, no data generated
TITLE, pseudo-op, no data generated
TTCALL, 051, monitor UUO
TWOSEG, pseudo-op, no data generated
UGETF, 073, monitor UUO
UJEN, 100, monitor UUO
UNIVERSAL, pseudo-op, no data generated
USETI, 074, monitor UUO
USETO, 075, monitor UUO
VAR, pseudo-op, generates data
WTB., 023, FORTRAN UUO
XALL, pseudo-op, no data generated
XLIST, pseudo-op, no data generated
XWD, pseudo-op, generates data
Z, pseudo-op, generates data

.CREF, pseudo-op, no data generated
.XCREF, pseudo-op, no data generated
.HWFRMT, pseudo-op, no data generated
.MFRMT, pseudo-op, no data generated

A.2 MACHINE MNEMONICS AND OCTAL CODES

The following are machine mnemonics and corresponding octal codes:

Mnemonic	Code
ADD	270
ADDB	273
ADDI	271
ADDM	272
AND	404
ANDB	407
ANDCA	410
ANDCAB	413
ANDCAI	411
ANDCAM	412
ANDCB	440
ANDCBB	443
ANDCBI	441
ANDCBM	442
ANDCM	420
ANDCMB	423
ANDCMI	421
ANDCMM	422
ANDI	405
ANDM	406
AOBJN	253
AOBJP	252
AOJ	340
AOJA	344
AOJE	342
AOJG	347
AOJGE	345
AOJL	341
AOJLE	343
AOJN	346
AOS	350
AOSA	354
AOSE	352
AOSG	357
AOSGE	355
AOSL	351
AOSLE	353
AOSN	356
ASH	240
ASHC	244
BLKI	7-00
BLKO	7-10
BLT	251
CAI	300
CAIA	304
CAIE	302
CAIG	307
CAIGE	305
CAIL	301
CAILE	303
CAIN	306
CAM	310
CAMA	314
CAME	312
CAMG	317
CAMGE	315
CAML	311
CAMLE	313
CAMN	316
CLEAR	400
CLEARB	403
CLEARI	401
CLEARM	402
CONI	7-24
CONO	7-20
CONSO	7-34
CONSZ	7-30
DATAI	7-04
DATAO	7-14
DFN	131
DIV	234
DIVB	237
DIVI	235
DIVM	236
DPB	137
EQV	444
EQVB	447
EQVI	445
EQVM	446
EXCH	250
FAD	140
FADB	143
FADL	141
FADM	142
FADR	144
FADRB	147
FADRI	145
FADRM	146
FDV	170
FDVB	173
FDVL	171
FDVM	172
FDVR	174
FDVRB	177
FDVRI	175
FDVRM	176
FMP	160
FMPB	163
FMPL	161
FMPM	162
FMPR	164
FMPRB	167
FMPRI	165
FMPRM	166
FSB	150
FSBB	153
FSBL	151
FSBM	152
FSBR	154
FSBRB	157
FSBRI	155
FSBRM	156
FSC	132
HALT	254-4,
HLL	500
HLLE	530
HLLEI	531
HLLEM	532
HLLES	533
HLLI	501
HLLM	502
HLLO	520
HLLOI	521
HLLOM	522
HLLOS	523
HLLS	503
HLLZ	510
HLLZI	511
HLLZM	512
HLLZS	513
HLR	544
HLRE	574
HLREI	575
HLREM	576
HLRES	577
HLRI	545
HLRM	546
HLRO	564
HLROI	565
HLROM	566
HLROS	567
HLRS	547
HLRZ	554
HLRZI	555
HLRZM	556
HLRZS	557
HRL	504
HRLE	534
HRLEI	535
HRLEM	536
HRLES	537
HRLI	505
HRLM	506
HRLO	524
HRLOI	525
HRLOM	526
HRLOS	527
HRLS	507
HRLZ	514
HRLZI	515
HRLZM	516
HRLZS	517
HRR	540
HRRE	570
HRREI	571
HRREM	572
HRRES	573
HRRI	541
HRRM	542
HRRO	560
HRROI	561
HRROM	562
HRROS	563
HRRS	543
HRRZ	550
HRRZI	551
HRRZM	552
HRRZS	553
IBP	133
IDIV	230
IDIVB	233
IDIVI	231
IDIVM	232
IDPB	136
ILDB	134
IMUL	220
IMULB	223
IMULI	221
IMULM	222
IOR	434
IORB	437
IORI	435
IORM	436
JCRY	255-6,
JCRY0	255-4,
JCRY1	255-2,
JEN	254-12,
JFCL	255
JFFO	243
JFOV	255-1,
JOV	255-10,
JRA	267
JRST	254
JRSTF	254-2,
JSA	266
JSP	265
JSR	264
JUMP	320
JUMPA	324
JUMPE	322
JUMPG	327
JUMPGE	325
JUMPL	321
JUMPLE	323
JUMPN	326
LDB	135
LSH	242
LSHC	246
MOVE	200
MOVEI	201
MOVEM	202
MOVES	203
MOVM	214
MOVMI	215
MOVMM	216
MOVMS	217
MOVN	210
MOVNI	211
MOVNM	212
MOVNS	213
MOVS	204
MOVSI	205
MOVSM	206
MOVSS	207
MUL	224
MULB	227
MULI	225
MULM	226
OR	434
ORB	437
ORCA	454
ORCAB	457
ORCAI	455
ORCAM	456
ORCB	470
ORCBB	473
ORCBI	471
ORCBM	472
ORCM	464
ORCMB	467
ORCMI	465
ORCMM	466
ORI	435
ORM	436
POP	262
POPJ	263
PUSH	261
PUSHJ	260
ROT	241
ROTC	245
RSW	7-04
SETA	424
SETAB	427
SETAI	425
SETAM	426
SETCA	450
SETCAB	453
SETCAI	451
SETCAM	452
SETCM	460
SETCMB	463
SETCMI	461
SETCMM	462
SETM	414
SETMB	417
SETMI	415
SETMM	416
SETO	474
SETOB	477
SETOI	475
SETOM	476
SETZ	400
SETZB	403
SETZI	401
SETZM	402
SKIP	330
SKIPA	334
SKIPE	332
SKIPG	337
SKIPGE	335
SKIPL	331
SKIPLE	333
SKIPN	336
SOJ	360
SOJA	364
SOJE	362
SOJG	367
SOJGE	365
SOJL	361
SOJLE	363
SOJN	366
SOS	370
SOSA	374
SOSE	372
SOSG	377
SOSGE	375
SOSL	371
SOSLE	373
SOSN	376
SUB	274
SUBB	277
SUBI	275
SUBM	276
TDC	650
TDCA	654
TDCE	652
TDCN	656
TDN	610
TDNA	614
TDNE	612
TDNN	616
TDO	670
TDOA	674
TDOE	672
TDON	676
TDZ	630
TDZA	634
TDZE	632
TDZN	636
TLC	641
TLCA	645
TLCE	643
TLCN	647
TLN	601
TLNA	605
TLNE	603
TLNN	607
TLO	661
TLOA	665
TLOE	663
TLON	667
TLZ	621
TLZA	625
TLZE	623
TLZN	627
TRC	640
TRCA	644
TRCE	642
TRCN	646
TRN	600
TRNA	604
TRNE	602
TRNN	606
TRO	660
TROA	664
TROE	662
TRON	666
TRZ	620
TRZA	624
TRZE	622
TRZN	626
TSC	651
TSCA	655
TSCE	653
TSCN	657
TSN	611
TSNA	615
TSNE	613
TSNN	617
TSO	671
TSOA	675
TSOE	673
TSON	677
TSZ	631
TSZA	635
TSZE	633
TSZN	637
UFA	130
XCT	256
XOR	430
XORB	433
XORI	431
XORM	432



Appendix B
Summary of Pseudo-Ops

B.1 PSEUDO-OPS

A list of pseudo-ops and their functions follows:

ARRAY	Reserve multiple words of storage.
ASCII	Seven-bit ASCII test
ASCIZ	Seven-bit ASCII test, with null character guaranteed at end
ASUPPRESS	Turns on suppress bit for all symbols
BLOCK	Reserves block of storage cells
BYTE	Input bytes of length 1-36 bits
COMMENT	No binary produced; same as seven-bit ASCII
DEC	Input decimal numbers
DEFINE	Defines macro
DEPHASE	Terminates PHASE relocation mode
END	Last statement of the program
ENTRY	Entry point for subroutine library
EXP	Input expressions
EXTERN	Identifies external symbols

HISEG	Load into high segment
INTEGER	Reserve one word of storage per argument
INTERN	Define internal symbols
IOWD	Set up I/O transfer word
IRP	Indefinite repeat of macro arguments
IRPC	Indefinite repeat of one character
LALL	List all; expanded listing of macros
LIST	List in normal mode
LIT	Assemble literals
LOC	Assign absolute addresses
MLOFF	Turn off multiline literal feature
MLON	Turn on multiline literal feature
NOSYM	Suppress symbol table listing
OCT	Input octal numbers
OPDEF	Defines user-created operator; generates only one word
PAGE	Start a new listing page
PASS2	Terminates pass 1, remaining statement are processed pass 2 only
PHASE	Following coding relocated at execution time
POINT	Sets up byte pointer word
PRGEND	Allows multiprogram assemblies, end one such program
PRINTX	Output on terminal or listing device the rest of the line
PURGE	Remove symbol from table
RADIX	Sets prevailing radix to 2-10
RADIX5Ø	Compresses 36-bit words, primarily for system use
RELOC	Implied first statement; assigns relocatable addresses
REMARK	Comments only statement
REPEAT	Repeat n times
RIM	Prepare output in RIM paper-tape format
RIM1Ø	Absolute, unblocked, output format; no checksums
RIM1ØB	Absolute, blocked, checksummed output format

SALL	Suppress listing of macros; lists only call and binary generated
SEARCH	Opens symbol tables of universal program
SIXBIT	Input text in compressed 6-bit ASCII
SQUOZE	Same as RADIX 50 above
STOPI	Stop indefinite repeat of macro arguments
SUBTTL	Subtitle on listing
SUPPRESS	Turns on suppress bit for specified symbols
SYN	Make synonymous
TAPE	Stop processing the current file
TITLE	Title on listing and to DDT
TWOSEG	Assembles and loads two segment programs
UNIVERSAL	Makes symbol table available to other programs
VAR	Assemble variables suffixed with # or ARRAY or INTEGER
XALL	Stop expanded listing, resume normal list mode
XLIST	Stop listing
XPURGE	Purges local symbols on pass 2
XWD	Input two 18-bit half words
Z	Input zero word
.CREF	Resume outout pf CREF information
.XCREF	Stop output of CREF information
.HWFRMT	List binary in half word format (old)
.MFRMT	List binary in multi-format (new)

B.1.1 Conditional Assembly Statements

These conditional assembly statements in the first column are assembled if the conditions in the second column exist.

IF1	Encountered during pass 1
IF2	Encountered during pass 2
IFB	Blank
IFDEF	Defined
IFDIF	Different
IFE	Zero

IFG	Positive
IFGE	Zero, or positive
IFIDN	Identical
IFL	Negative
IFLE	Zero, or negative
IFN	Non-zero
IFNB	Not blank
IFNDEF	Not defined

Appendix C
Summary of Character Interpretations

The characters listed below have special meaning in the contexts indicated. These interpretations do not apply when these characters appear in text strings, or in comments.

Character	Meaning	Example
:	Colon. Immediately follows all labels.	LABEL: Z
;	Semicolon. Precedes all comments.	;THIS IS A COMMENT
.	Point. Has current value of the location counter or indicates floating point number.	JRST .+5 JUMP FORWARD FIVE LOCATIONS 1.Ø
,	Comma. General operand or argument delimiter.	DEC 1Ø,5,6 EXP A+B,C-D
	Accumulator field delimiter.	MOVEI 1,TAG
	References accumulator 0. The comma is optional.	MOVEI ,TAG
	Delimits macro arguments.	MACRO (A,B,C)
!	Inclusive OR } Logical Operators	
&	AND } Logical Operators	

Character	Meaning	Example
*	Multiplication	Arithmetic Operators
/	Division	Arithmetic Operators
+	Add (+A outputs the value of A)	Arithmetic Operators
-	Subtract	Arithmetic Operators
1st character of text string	In ASCII, ASCIZ and SIXBIT comment text strings, the first non-blank character is the delimiter.	ASCII/STRING/;
B	Follows number to be shifted and precedes binary shift count.	7B2
E	Exponent. Precedes decimal exponent in floating-point numbers.	F22.1E5 EXPONENT IS 5.
( )	Parentheses. Enclose index fields.	ADD AC1,X (7) MOVEI A,(SIXBIT/ABC/)
	Enclose the byte size in BYTE statements.	BYTE (6) 8, 8, 7
	Enclose the dummy argument string in macro DEFINE statements.	DEFINE MAC(A,B,C)
< >	Angle brackets. In an expression, enclose a numeric quantity.	<A-B+5ØØ/C>
	In conditional assembly statements, contain a single argument, and the conditional coding.	IF1, MOVE ACØ, TAX
	In REPEAT statements, contain coding to be repeated.	REPEAT 3, <SUB 17, TAG>
	In macros, enclose the macro definition.	DEFINE PUNCH DATAO PTP, PUNBUF (4)
[ ]	Square brackets. Delimit literals.	ADD 5,[MOVEI 3,TAX]
	In OPDEF statement, contain new operator; in ARRAY the size.	OPDEF CAL [MOVE] ARRAY FOO[212]
=	Equal sign. Direct assignment.	SYM=6 SYM-A+B*D
==	Equal sign. Direct assignment but no output to DDT.	SYM==6
=:	Equal sign and colon. Direct assignment but automatically made internal.	FLAG=:2ØØ
:!	Colon and exclamation point. Direct assignment of label, no output to DDT, and automatically made internal.	LABEL:!

Character	Meaning	Example
==:	Equal sign and colon. Direct assignment, no output to DDT, and automatically made internal.	LOOP==:32
::!	Double colon and exclamation point. Direct assignment of label, no output to DDT, and automatically made internal.	NAME::!
"..."	Quotation marks enclose 7-bit ASCII text, right justified, from one to five characters.	"ABCDE"
'...'	Single quotation marks enclose 6-bit ASCII text, right justified, from one to six characters.	'TABLES'
#	Number sign, Defines a symbol used as a tag. Variable.	ADD 3,TAG#
##	Alternate method of generating external symbols.	MOVE Ø,JOBREL##
'	Apostrophe or single quote. Concatenation character, used within macro definitions or SIXBIT data.	DEFINE MAC (A,B,C); <JUMP'A B, C> 'SIXBIT'
\	Reverse slash. If used as the first character of an argument in a macro call, the value of the following symbol is converted to an ASCII symbol in the current radix.	MAC \ A IF A=5ØØ, THIS GENERATES THREE 7-BIT ASCII CHARACTERS, ASCII/5ØØ/
↑←	Control left arrow. Line continuation.	
←	Left arrow. N M shift N left (or right) M bit positions,	1ØØ←3=1ØØØ 1ØØ←←3=1Ø
@	Indicates indirect addressing. Causes the indirect bit in an instruction to be set.	MOV AC,@ADDR

Appendix D
Storage Allocation

MACRO allocates storage in two directions:

1) the symbol table (user symbols and macro names) grows downward from top of the low segment (.JBREL)

2) Macros, literals, etc., grow upward from free space (.JBFF).

All entries in the symbol table are two words long. The first word is the symbol name in SIXBIT. The second word is flags in left half and either value or pointer in right half.

Most symbols have a value less than 18 bits and so can be represented by just the two words in the symbol table. Symbols with a 36-bit value (e.g., -1) have the value stored in a 1 word in free storage and a pointer to this value stored in the symbol table.

External symbols have two words in free storage, the first is the value (i.e., the last reference in a chain of references to the symbol). The second is the sixbit name of the symbol. This is so that additive global fixups can be output.

Opdefs tend to have 36-bit values and are stored like other 36-bit value symbols.

Macro names are stored in the symbol table, the value is a pointer to the stored text string.

The text string is stored in four (assembly parameter) word blocks which have the general form

```
1)  link to next block, [Ø if last] ,, 2 characters
2)                     5 characters
3)                     5 characters
4)                     5 characters
```

However, the first such block is special

```
1)  link to next block  ,,  link to last block
2)  pointer to default arg; ,, <number or args expected>+9+reference
3)                     5 characters                          count
4)                     5 characters
```

The number of args expected is the number of arguments in the define statement.

The reference count is incremented when the macro is called and decremented when exiting from the macro. When this count goes to zero the macro is removed from free space.

The actual arguments to a macro are stored in the same linked block, but are not in the symbol table. Repeats (2 or more times) are also stored the same way. The text blocks are removed when the macro exits or the repeat exits since the reference count has gone to zero.

The addresses of the actual argument blocks are stored in a pushdown stack in order of generation.

Default arguments are stored the same way except the list is in free core. The pointer to this default arg list is stored in the left half of the second word of the first block of the macro definition.

The text body is stored as is, except that dummy arguments are replaced by special symbols.

The ASCII character RUBOUT (177) is used to signal a special character text.

These characters are

```
ØØ1       ;end of macro
ØØ2       ;end of dummy symbol
ØØ3       ;end of Repeat
ØØ4       ;end of IRP or IRPC
```

If the character is 4<ch<77 it is illegal.
If the character is <100 then it is a dummy symbol, the value of the character is ANDed with 37 to get the dummy symbol number and the corresponding pointer retrieved from the stack of pointers.

If th- symbol was not specified (i.e., no pointer) then if the 40 bit is on this is to be a created symbol and one is created, otherwise the argument is ignored.

Verbose macros can eat up a lot of storage space.

Literals are stored in four words/block per word generated (three words if old format used).

Words are

```
-3:       form word
-2:       relocation bits
-1:       code
 Ø:       pointer to next
```

The pointer points to the Ø word of the next block. The code is the generated code. Relocation is either the relocation bits Ø or 1 per half word or external pointers if externs used.

Form word is the word used for listing, this word is not checked when comparing literals so that different forms that produce the same code are classed as equal.

Long literals are both slow and take up extra storage, they should be written as subroutines or inline.

Single quotes can also be used to indicate SIXBIT words, however, one pair of single quotes is removed by the assembler if the pair encloses a dummy argument. For example, in the macro

```
DEFINE    SXBT (A)<
MOVSI     1,"A"
MOVSI     2,"B"
>
```

B is not a dummy argument so it can be enclosed in single quotes. A, however, is a dummy argument and must be enclosed in double quotes since one pair of quotes (the inner pair) will be removed by the assembler.

Appendix E
Text Codes

This appendix contains a summary of MACRO-10 text codes.

SIXBIT	Character	ASCII 7-Bit*	SIXBIT	Character	ASCII 7-Bit*	Character	ASCII 7-Bit*
00	Space	040	40	@	100	`	140
01	!	041	41	A	101	a	141
02	"	042	42	B	102	b	142
03	#	043	43	C	103	c	143
04	$	044	44	D	104	d	144
05	%	045	45	E	105	e	145
06	&	046	46	F	106	f	146
07	'	047	47	G	107	g	147
10	(	050	50	H	110	h	150
11	)	051	51	I	111	i	151
12	*	052	52	J	112	j	152
13	+	053	53	K	113	k	153
14	,	054	54	L	114	l	154
15	-	055	55	M	115	m	155
16	.	056	56	N	116	n	156
17	/	057	57	O	117	o	157
20	0	060	60	P	120	p	160
21	1	061	61	Q	121	q	161
22	2	062	62	R	122	r	162
23	3	063	63	S	123	s	163
24	4	064	64	T	124	t	164
25	5	065	65	U	125	u	165
26	6	066	66	V	126	v	166
27	7	067	67	W	127	w	167
30	8	070	70	X	130	x	170
31	9	071	71	Y	131	y	171
32	:	072	72	Z	132	z	172
33	;	073	73	[	133	{	173
34	<	074	74	\	134	|	174
35	=	075	75	]	135	}	175
36	>	076	76	↑	136	~	176
37	?	077	77	←	137	Delete	177

*MACRO-10 also accepts five of the 32 control codes in 7-bit ASCII:

Horizontal Tab	011	Vertical Tab	013	Carriage Return	015
Line Feed	012	Form Feed	014		

Appendix F
Radix 50 Representation

Radix 50_8 representation is used to condense 6-character symbols into 32 bits. Each character of a symbol is subscripted in descending order from left to right; i.e., the symbols are of the form

$$L_6 L_4 L_5 L_3 L_2 L_1$$

If C_n denotes the octal code for L_n, the radix 50_8 representation is generated by the following

$$(((((C_6*5Ø)+C_5)*5Ø+C_4)*5Ø*C_3)*5Ø+C_2)*5Ø+C_1$$

where all numbers are octal.

The code numbers corresponding to the characters are:

Code (Octal)	Characters
00	Null character
01-12	0-9
13-44	A-Z
45	.
46	$
47	%

The top four bits are taken from the four leftmost bits of a 6-bit octal number (i.e., Ø4-74).

Appendix G
Summary of Rules for Defining and Calling Macros

G.1 ASSEMBLER INTERPRETATION

MACRO-10 assembles macros by direct and immediate character substitutions. When a macro call is encountered, in any field, the character substitution is made, the characters are processed, and the assembler continues its scan with the character following the delimiter of the last argument, except when it is delimited by a semicolon. Macros can appear any number of times on a line.

G.2 CHARACTER HANDLING

G.2.1 Blanks

A macro symbol is delimited by one blank or one tab; the character following the delimiter is the start of the argument string even if it is also a blank or tab. Other than the first delimiter, blanks and tabs are treated as standard characters in the argument string.

G.2.2 Brackets

Angle brackets are only significant in the argument fields if the first character of any field is a left angle bracket. In this case,

no terminator or parenthesis tests are made between the left angle bracket and its matching right bracket. The matching brackets are removed from the string but the scan continues until a standard delimiter is found.

G.2.3 Parentheses

Parentheses serve only to terminate an argument scan. They are significant only when the first character following the blank or tab delimiter is a left parenthesis. In this case, the left parenthesis is removed and, if it matching right parenthesis is encountered prior to the normal termination of theargument scan, it is removed and the scan discontinued.

G.2.4 Commas

When a comma is encountered in an argument scan, it acts as the delimiter of the current argument. If it delimits the last argument, the character following it will be the first scanned after the subsitution is processed.

G.2.5 Semicolons

When a semicolon is encountered in an argument scan, the scan is discontinued. If an argument has not been satisfied, the remainder is considered null. It is saved, however, and will be the first character scanned after the substitution is made, normally acting as a comment flag.

G.2.6 Carriage Return

A carriage return, except when pre-empted by angle brackets (see Section G.2.2), will terminate the scan similar to the semicolon. This can be circumvented, if desired, by the control left arrow key described elsewhere.

G.2.7 Back-Slash

If the first character of any argument is a back-slash, it must be directly followed by a numeric term. The value of the numeric term is broken down into a string of ASCII digits of the current radix, just the reverse of a fixed-point number computation. The value is

considered to be a 36-bit positive number having a value of 0 to 777777 777777. Leading zeros are suppressed except in the case of 0, in which case the result is one ASCII 0. The ASCII string is substituted and the scan continued in the normal manner (no implied terminator).

The default listing mode is XALL, in which case the initial macro call and all lines within its range that produce binary code are listed. The pseudo-op LALL will cause all lines to be listed. Substituted arguments are bracketed by ↑'s by the assembler.

G.3 CONCATENATION

The rule for concatenation is as follows:

For each string of apostrophes, one is removed if and only if it is next to (either before or after) a dummy argument to that macro.

Appendix H
Operating Instructions

H.1 REQUIREMENTS

The following are MACRO-10 operating requirements:

Minimum Core	7K pure plus 1K impure
Additional Core	Automatically requests additional core assignments from the timesharing monitor as needed.
Equipment	One input device (source program input); up to two output devices (machine language program output and listing output). If the listing output is to be used as input to the Cross Reference (CREF) program, it must not be TTY, DIS or LPT.

H.2 INITIALIZATION

The following are commands and corresponding indications:

.R MACRO)	Loads the MACRO-10 Assembler into core.
*	The Assembler is ready to receive a command.

H.3 COMMANDS

H.3.1 General Command Format

MACRO-10 general commands are as follows:

```
objprog-dev:filename.ext,list-dev:filename.ext source-dev:filename.ext,......source-n)
```

objprog-dev: The device on which the object program is to be written.

MTAn:	(magnetic tape)
DTAn:	(DECtape)
PTP:	(paper-tape punch)
DSK:	(disk)

list-dev: The device on which the assembly listing is to be written.

MTAn:	(magnetic tape)	Must be one of these if input to CREF[1]
DTAn:	(DECtape)	
DSK:	(disk)	
LPT:	(line printer)	
TTY:	(Teletype)	
PTP:	(paper-tape punch)	

source-dev: The device(s) from which the source-program input to assembly is to be read.

MTAn:	(magnetic tape)
CDR:	(card reader)
DTAn:	(DECtape)
DSK:	(disk)
PTR:	(paper-tape reader)
TTY:	(Teletype)

If more than one file is to be assembled from a magnetic tape, card reader, or paper tape reader, dev: is followed by a comma for each file beyond the first.

Input via the Teletype is terminated by typing CTRL Z (↑Z) to enter pass 1; the entries must be retyped at the beginning of pass 2.

filename.ext (DSK: and DTAn: only) The filename and filename extension of the object program file, the listing file, and the source file(s).

The object program and listing devices are separated from the source device by the left arrow symbol.

H.3.2 Disk File Command Format

MACRO-10 disk file commands are as follows:

```
DSK:filename.ext [proj,prog]
```

[1] If /C switch is given, but no list-dev: is specified, DSK:CREF.CRF is assumed.

[proj,prog] Project-programmer number assigned to the disk area to be searched for the source file(s) if other than the user's project-programmer number.

The installation standard protection is assigned to any disk file specified as output.

NOTE

If object coding output is not desired (e.g., a program is being scanned for source language errors), objprog-dev: is omitted. If an assembly listing is not desired, list-dev: is omitted. If device is not specified, DSK is assumed.

Examples:

```
.R MACRO)
*DTA3:OBJPRG,LPT: CDR:)
```

Assemble one source program file from the card reader; write the object code on DTA3 and call the file OBJPRG; write the assembly listing on the line printer.

```
END OF PASS 1)
```

The source program cards must be manually re-fed for pass 2.

```
?2 ERRORS DETECTED)
PROGRAM BREAK IS ØØ2537)
2K CORE USED)
```

Number of source errors; size of object program; core used by assembler.

```
*↑C)
```

Return to the monitor.

```
.R MACRO)
*MTA3:,MTA2: MTA1:,,)
NO ERRORS DETECTED)
PROGRAM BREAK IS ØØ3552)
2K CORE USED)
```

Assemble the next three source files located at the present position of MTA1; write the object program on MTA3; write the listing on MTA2 for later printing.

```
*,LPT: DTA1:FILE1,FILE2,FILE5)
NO ERRORS DETECTED)
PROGRAM BREAK IS ØØ1Ø27)
2K CORE USED)
```

Assemble the source files named FILE1, FILE2, and FILE5 from DTA1; produce no object coding; write the listing on the line printer.

```
*,←DSK:FILE1.MAC[14,12])
NO ERRORS DETECTED)
PROGRAM BREAK IS ØØØ544
2K CORE USED)
```

Scan the source program called FILE1.MAC, located in area 14, 12 on the disk, for source language errors; produce no object coding or assembly listing; print all error diagnostics on the terminal.

```
*↑C)
```

Return to the monitor.

```
.R MACRO

*MTA1:,TTY: TTY:)
        JMP     R)
R:      AOS     G)
G:      JFCL)
        END)
```

Assemble a source file from the terminal; write the object code program on MTA1 and print the assembly listing on the terminal.

```
↑Z)
```

Terminate input.

```
      END OF PASS 1)
        JMP     R)
```

Reenter terminal input.
Type first statement again.

```
.MAIN      MACRO          10:14        20-DEC-67       PAGE1)   Page heading.
0          000000  000000    000001'       JMP         R)       First assembled.
R:         AOS               G                                  Reenter second.
           000001  350000    000002'     R: AOS        G)       Second assembled.
G:         JFCL)                                                Reenter third.
           000002  255000    000000      G: JFCL)               Third assembled.
           END)                                                 Reenter fourth.
                                           END)                 Fourth assembled.

?1 ERROR DETECTED)                                              Typeout of symbol
PROGRAM BREAK IS 000003)                                        table.
.MAIN      MACRO          10:14        20-DEC-67       PAGE2)

           SYMBOL TABLE)
G          000002')
R          000001')

2K CORE USED)
*↑C)                                                            Return to the monitor.
```

H.4 SWITCHES

Switches are used to specify such options as:

a. Magnetic tape control
b. Macro call expansion
c. Listing suppression
d. Pushdown list expansion
e. Cross-reference file output.

All switches are preceded by a slash (/) or enclosed in parentheses, and usually occur prior to the left arrow (see Table H-1).

Table H-1

MACRO-10 Switch Options

Switch	Meaning
A	Advance magnetic tape reel by one file.
B	Backspace magnetic tape reel by one file.
C	Produce listing file in a format acceptable as input to CREF; unless the file is named, CREF. CRF is assigned as the filename; if no extension is given, .CRF is assigned; if no list-dev: is specified, DSK: is assumed. /C must appear between the comma and the left-arrow.
E	List macro expansions (same function as LALL pseudo-op).
F	New format for output binary listing (.MFRMT pseudo-op).
G	Old format for output binary listing (.HWFRMT pseudo-op).
H	Print Help text (i.e., this list of switches and explanations).
L	Reinstate listing (used after list suppression by either the XLIST pseudo-op or S switch).
M	List only call, no binary, in macro expansion (same .SALL pseudo-op).
N	Suppress error printouts on the terminal.
O	Sets the pseudo-op MLOFF which allows literals to occupy on a single line. This means literals may be terminated with a carriage return, line feed instead of a right bracket.
P	Increase the size of the pushdown list. This switch may appear as many times as desired (pushdown list is initially set to a size of 80_{10} locations; each /P increases its size by 80_{10}). /P must appear on the left of the left arrow.
Q	Suppress Q (questionable) error indications on the listing; Q messages indicate assumptions made during pass 1. /Q must appear on the left of the left-arrow.
S	Suppress listing (same function as XLIST pseudo-op).
T	Skip to the logical end of the magnetic tape.
W	Rewind the magnetic tape.
X	Suppress all macro expansions (same function as XALL pseudo-op).
Z	Zero the DECtape directory.

NOTE

Switches A through C and T, W, X, and Z must immediately follow the device or file to which the individual switch refers.

Examples:

.R MACRO) *MTA1:,DTA3:,/C←PTR:)	Assemble one source file from the paper tape reader; write the object code on MTA1; write the assembly listing on DTA3 in cross-reference format and call the file CREF.CRF.
END OF PASS 1)	The paper tape must be re-fed by the operator for pass 2.
?3 ERRORS DETECTED) PROGRAM BREAK IS ØØØ4Ø1) 2K CORE USED)	End-of-assembly messages.
*DTA2:ASSEMB.ONE/Z,LPT: MTA4:/W,)	Rewind MTA4 and assemble the first two source files on it; write the object code on DTA2, after zeroing the directory, and call the file ASSEM.ONE; write the assembly listing on the line printer.
NO ERRORS DETECTED) PROGRAM BREAK IS ØØ5231) 3K CORE USED)	
*MTA1:/W,LPT:←MTA3: /W,(AA),(BB))	Rewind MTA1 and MTA3 and assemble files 1, 4, and 3 (in that order) from MTA3; print the assembly listing on the line printer; write the object code on MTA1.
?1 ERROR DETECTED) PROGRAM BREAK IS ØØØ655) 2K CORE USED)	
*FOO,/C FOO) NO ERRORS DETECTED) PROGRAM BREAK IS ØØØ765) 2K CORE USED)	Assemble source file FOO on DSK:; write the assembly listing on DSK in cross-reference format calling the file CREF.CRF. Write object code on DSK calling it FOO.REL.
*↑C)	Return to the monitor.
.	