Book 2

Assembling the Source Program

[redacted]

MACRO-10 Assembler

CONTENTS

		Page
	CHAPTER 1 INTRODUCTION	
1.1	MACRO-10 Language - Statements	1-1
1.1.1	Labels	1-2
1.1.2	Operators	1-2
1.1.3	Operands	1-2
1.1.4	Comments	1-3
1.2	Symbols	1-3
1.2.1	Symbolic Addresses	1-3
1.2.2	Symbolic Operators	1-4
1.2.3	Symbolic Operands	1-4
1.2.4	The Symbol Table	1-4
1.2.4.1	Direct Assignment Statements	1-5
1.2.5	Deleted Symbols	1-5
1.3	Numbers	1-6
1.3.1	Binary Shifting	1-7
1.3.2	Left Arrow Shifting	1-8
1.3.3	Floating-Point Decimal Numbers	1-8
1.3.4	Fixed-Point Decimal Numbers	1-8
1.3.5	Arithmetic and Logical Operations	1-9
1.3.6	Evaluating Expressions	1-10
1.3.7	Numeric Terms	1-10
1.4	Address Assignments	1-11
1.4.1	Setting and Referencing the Location Counter	1-11
1.4.2	Indirect Addressing	1-12
1.4.3	Indexing	1-12
1.4.4	Literals	1-12
1.4.4.1	Multiline Literals	1-13
1.5	Instruction Formats	1-13
1.5.1	Primary Instruction Format	1-14
1.5.2	Input/Output Instruction Format	1-15
1.6	Communication With Monitors	1-16
1.7	Operating Procedures	1-16

CONTENTS (Cont)

		Page
	CHAPTER 2 MACRO-10 ASSEMBLER STATEMENTS - PSEUDO-OPS	
2.1	Address Mode: Relocatable or Absolute	2-1
2.1.1	Relocation Before Execution - PHASE and DEPHASE Statements	2-3
2.2	Entering Data	2-3
2.2.1	RADIX Statements	2-3
2.2.2	Entering Data Under the Prevailing Radix	2-4
2.2.3	DEC and OCT Statements	2-5
2.2.4	Changing the Local Radix for a Single Numeric Term	2-5
2.2.5	RADIX50 Statement	2-6
2.2.6	EXP Statement	2-6
2.2.7	Z Statement	2-6
2.3	Input Data Word Formatting	2-7
2.3.1	BYTE Statement	2-7
2.3.2	POINT Statement - Handling Bytes	2-8
2.3.3	IOWD Statement: Formatting I/O Transfer Words	2-9
2.3.4	XWD Statement: Entering Two Half-Words of Data	2-9
2.3.5	Text Input	2-10
2.3.5.1	ASCII, ASCIZ, and SIXBIT Statements	2-10
2.3.6	Reserving Storage	2-11
2.3.6.1	Reserving a Single Location	2-11
2.3.6.2	BLOCK Statements	2-11
2.4	Conditional Assembly	2-12
2.5	Assembler Processing Statements	2-13
2.5.1	END Statements	2-13
2.5.2	PASS2 Statements	2-14
2.5.3	LIT Statements	2-14
2.5.4	VAR Statements	2-14
2.5.5	PURGE Statements	2-14
2.5.6	Listing Control Statements	2-15
2.5.7	Assembler Control Statements	2-17
2.5.7.1	REPEAT Statements	2-17
2.5.7.2	OPDEF Statements	2-18

CONTENTS (Cont)

		Page
2.5.7.3	SYN Statements	2-19
2.5.7.4	Permanent Symbols	2-19
2.5.7.5	Extended Instruction Statements	2-19
2.5.8	Linking Subroutines	2-20
2.5.8.1	EXTERN Statements	2-20
2.5.8.2	INTERN Statements	2-21
2.5.8.3	ENTRY Statements	2-21
2.5.9	HISEG Statements	2-22
	CHAPTER 3 MACROS	
3.1	Definition of Macros	3-1
3.2	Macro Calls	3-2
3.3	Macro Format	3-2
3.4	Created Symbols	3-3
3.5	Concatenation	3-5
3.6	Indefinite Repeat	3-5
3.7	Nesting and Redefinition	3-7
3.7.1	ASCII Interpretation	3-8
	CHAPTER 4 ERROR DETECTION	
4.1	Teletype Error Messages	4-4
	CHAPTER 5 RELOCATION	
	CHAPTER 6 ASSEMBLY OUTPUT	
6.1	Assembly Listing	6-1
6.2	Binary Program Output	6-1
6.2.1	Relocatable Binary Programs - LINK Format	6-2
6.2.1.1	LINK Formats for the Block Types	6-3
6.2.2	Absolute Binary Programs	6-4
6.2.2.1	RIM10B Format	6-4
6.2.2.2	RIM10 Format	6-5
6.2.2.3	RIM Format	6-6
6.2.2.4	END Statements	6-6

CONTENTS (Cont)

Page

CHAPTER 7
PROGRAMMING EXAMPLES

APPENDIX A
OP CODES, PSEUDO-OPS, AND MONITOR I/O COMMANDS

APPENDIX B
SUMMARY OF PSEUDO-OPS

APPENDIX C
SUMMARY OF CHARACTER INTERPRETATIONS

APPENDIX D
ASSEMBLER EVALUATION OF STATEMENTS AND EXPRESSIONS

APPENDIX E
TEXT CODES

APPENDIX F
RADIX 50 REPRESENTATION

APPENDIX G
SUMMARY OF RULES FOR DEFINING AND CALLING MACROS

APPENDIX H
OPERATING INSTRUCTIONS

ILLUSTRATIONS

6-1	General RIM10B Format	6-7
6-2	RIM10B Loader	6-8
7-1	Sample Program, CLOG	7-2
7-2	Example of Nested Macro	7-3
7-3	Two Byte Unpacking Subroutines	7-3
7-4	IRPC Example	7-4

TABLES

4-1	Error Codes	4-1

CHAPTER 1
INTRODUCTION

MACRO-10 is the symbolic assembly program for the PDP-10, and operates in a minimum of 5K of 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 Teletype 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 and unique 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 unique programming job.

1.1 MACRO-10 LANGUAGE - STATEMENTS

MACRO-10 programs are usually prepared on a Teletype, 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 element.

Statements are written in the general form:

label: operator operand, operand; comments (carriage return)

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

1.1.1 Labels

A label is the symbolic name, created by the source programmer to identify the statement. If present, the label is written first in a statement, and is terminated by a colon (:).

1.1.2 Operators

An operator may be one of the 366 mnemonic machine instruction codes (see PDP-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 also create pseudo-ops to extend the power of the assembly language.

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.

1.1.3 Operands

Operands are usually the symbolic addresses of the data to be accessed when an instruction is executed, or the input data or arguments of 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.

1.1.4 Comments

The programmer may add notes to a statement following a semicolon. Such comments do not normally affect assembly processing or program execution, but are useful in the program listing for later analysis or debugging. The use of the following special characters should be avoided in comments: < > [ ].

1.2 SYMBOLS

The programmer may create symbols to use as statement labels, as operators, and as operands. A symbol contains from one to six characters from the following set:

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

The first character in a symbol must not be a digit. 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.

1.2.1 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 symbol can be defined as a label only once. If a programmer attempts to define a symbol as a label again, the second or successive attempt is ignored and an error is indicated. The assembler recognizes only the first definition. These are legal symbolic addresses:

```
ADDR
.TOTAL
$SUM:
ABC: DEF:              (Both Labels are legal)
```

The following are illegal:

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

1.2.2 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:

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

1.2.3 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 arguments 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 specifed by the value $>17_8$, the four least significant bits are used.

1.2.4 The Symbol Table

The assembler processor symbols in source program statements by referencing its symbol table, which contains all defined symbols, along with the binary value assigned to each symbol.

Initially, the symbol table contains the mnemonic op codes of the machine instructions, the Monitor I/O command mnemonics, and the assembler pseudo-op codes, as listed in Appendix A. As the source program is processed, symbols defined in the source program, as well as new symbols defined by MACRO-10 for use by this program, are added to the symbol table.

1.2.4.1 Direct Assignment Statements - The programmer 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. 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.

1.2.5 Deleted Symbols

Sometimes a programmer may want to define a symbol in MACRO but not want to have that symbol typed out by DDT. In such a case, the programmer should define that symbol with a double equal sign:

```
FLAG== 200
```

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= 200
```

and then typing

```
FLAG$K
```

to DDT

If a symbol is defined with == and declared internal, then the == will be ignored.

1.3 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

$$2 \leq \text{radix} \leq 10$$

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).

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). Thus,

↑D10	is stored as	1010
↑O10	is stored as	1000
↑B10	is stored as	0010

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
↑L0	44	0000000000....0000000000
		41_8 zero bits
↑L5	41	0000000000....0000000101
↑L-1	0	1111111111....1111111111

1.3.1 Binary Shifting

A number may be logically shifted left or right by following it with the letter B, followed by a number, n, representing the bit position in which the right-hand bit of the number should be placed. Bmay be any bit position 0 -35 decimal; if B 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:

1B0	400000 000000
1B17	000001 000000
1B35	000000 000001
-1B35	777777 777777 (see explanation below)
-1B53	000000 777777
-1B70	000000 000001

Note that the following expressions are equivalent:

10B32 ≡ ↑O10B32 ≡ 10B42 - 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 1.3.6.

1.3.2 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.

1.3.3 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,

```
17.0)  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)
```

1.3.4 Fixed-Point Decimal Numbers

As shown in Section 1.3, †D followed by a numeric term, is used to enter decimal intergers.

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     ↑F123.45517
346314 631462     ↑F123.458-1

777777 777604     -↑F123.45
777604 431463     -↑F123.45517
431463 146316     -↑F123.458-1
```

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

1.3.5 Arithmetic and Logical Operations

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

Operator	Meaning
+	Add
-	Subtract
*	Multiply
/	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 arithmetical and logical operators are called expressions.

1.3.6 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+5-29>*<A-41-X>>+1
```

1.3.7 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 particularly 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
or ↑D<5*2+13>
or ↑D<TEN*2+THREE>
```

but 23_{10} may not be written;

```
↑D100-77
```

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) and is followed by another term (the bit position of the assumed point). The following are legal:

```
↑F167B17
↑B10011B8
 56685
<MARK + SIGN>B<PT-XXV>
```

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

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

1.4.1 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
```

*The terms Monitor command (as used here) and programmed operator are synonymous.

1.4.2 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, DEC-10-HGAA-D (page 1-7).

1.4.3 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 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.

1.4.4 Literals

In a MACRO-10 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 the bracketed data in its literal table, assigns an address to the first word of the data and inserts that address in the machine instruction.

A literal may be any term, symbol, expression or statement, but it must generate data. Statements which do not generate data, i.e., some pseudo-ops, such as RADIX, and direct assignment statements, may not be written as literals. Literals may be nested, up to 18 levels.

Here is a simple example. Byte instructions must reference a byte pointer word, like this:

```
    LDB     4,BP
BP: POINT   10,A+3,14
```

(POINT is a speudo-op which sets up a byte pointer word.) A literal can be used to insert the POINT statement directly. (The use of literals is also shown in Chapter 7, Figure 7-3.)

```
LDB 4,[POINT 10,A+3,14]
```

1.4.4.1 Multiline Literals - MACRO optionally allows multiline literals. The following is legal:

```
GETCHR: SOSG   IBUF+2                          ;ANY CHARS LEFT?
        PUSHJ P,[IN    N,                      ;NO, READ SOME IN
                 POPJ  P,                      ;NO UNUSUAL CONDITIONS
                 STATZ N,740000                ;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,                              ;TRA 1,4
```

Two new pseudo-operations have been added to control whether or not this feature is available. Use of these pseudo-ops is required since

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

is legal in MACRO-10, even though the closing right bracket (]) of the literal has been omitted. In normal mode, MACRO allows such an unterminated literal. However, the pseudo-op

```
MLON
```

causes 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 no longer ends a literal, but rather the programmer must insert a right bracket. The pseudo-op

```
MLOFF
```

places MACRO back into the (initial) 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.

1.5 INSTRUCTION FORMATS

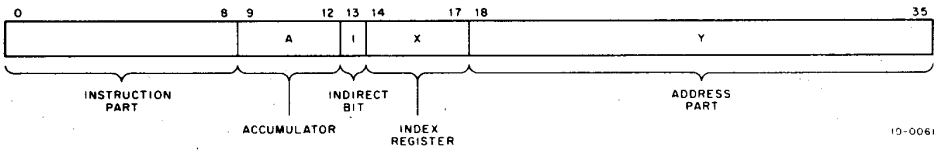
There are two types of machine instruction word formats: primary and input/output.

The 366 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.

1.5.1 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.

The 16 accumulators are specified by writing 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/)
```

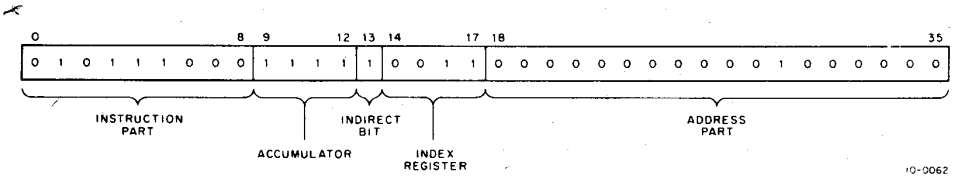
are equivalent to

```
MOVSI AC,201400                ;MOVE 1.0 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, STATEMENT EXAMPLE, follows a semicolon. Such comments do not affect the program in any way, but are printed in the output listing.

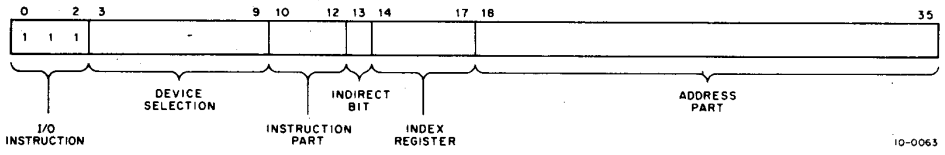
1.5.2 Input/Output Instruction Format

In the eight PDP-10 I/O statements, the first operand is either a peripheral device number or a device mnemonic (see PDP-10 User's Handbook 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:



1.6 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.

1.7 OPERATING PROCEDURES

Commands for loading and executing MACRO-10 are contained in the PDP-10 System User's Guide (DEC-10-NGCC-D).

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 1.2) and must be terminated by a character other than those listed in paragraph 1.2 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 loaded 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 0)
```

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, beginning 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 200)

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

000000'             ADD 1,X          ;RELOC 0 IS IMPLICIT.
                    :
000074'             LOC 1000         ;CHANGES TO ABSOLUTE, STARTING
001000              SUB 5,TOT        ;WITH 001000.
                    :/
001034              RELOC            ;SETS LOCATION COUNTER TO 74.
000074'             ADD 2,XAT
000075'             LOC              ;SWITCHES LOCATION COUNTER
001034              EXP A-X+7        ;BACK TO ABSOLUTE SEQUENCE.
```

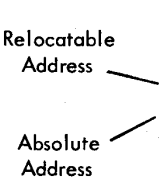
When operating in the relocatable mode, the assembler determines whether each location in the object program is relocatable or absolute, using an algorithm described in Chapter 5.

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

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

2.2 ENTERING DATA

2.2.1 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 10)
```

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.

2.2.2 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.

2.2.3 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  10,4.5,3.1416,6.03E-26,3)
```

generates five decimal words of data.

2.2.4 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/200>
```

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

```
TOTAL2:MOVE   †D10,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   †D60,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.

2.2.5 RADIX50 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.)

2.2.6 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.

2.2.7 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.

2.3 INPUT DATA WORD FORMATTING

2.3.1 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 5 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 5 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 character is enclosed in quotation marks.

```
BYTE   (7)"A")
```

Text handling pseudo-ops are discussed in Section 2.3.4. An example of the use of the BYTE statement is given in Chapter 7, Figure 7-3.

2.3.2 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 the 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 instruction 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.

An example of the use of the POINT statement is given in Chapter 7, Figure 7-3.

2.3.3 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 push-down 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.

2.3.4 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-wrod. Both are formed in 36-bit registers and the low order 18-bits are placed in the half-words. 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 trar.sfer 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.

2.3.5 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. In 7-bit ASCII, five control characters are also accepted:

Horizontal Tab
Line Feed
Vertical Tab
Form Feed
Carriage Return

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

2.3.5.1 ASCII, ASCIZ, and SIXBIT Statements - To enter one or more words of text characters, the operators ASCII, SIXBIT, and ASCIZ are used. The delimiter for the string of text characters is the first nonblank 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. The statement,

```
ASCII "AXE")
```

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 the full 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.

2.3.6 Reserving Storage

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

2.3.6.1 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.

2.3.6.2 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, 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.

2.4 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

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-X>,<...>	(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.

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.

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).

2.5 ASSEMBLER PROCESSING STATEMENTS

These statements direct the assembler to perform various kinds of processing.

2.5.1 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
```

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, literals starting at the current location.*

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

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

2.5.2 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.

The two assembly operators, LIT and VAR, are used to control assembly allocation of storage.

2.5.3 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.

2.5.4 VAR Statements

VAR↵

This statement causes symbols which have been defined by suffixing with the # sign 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.

2.5.5 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. Purged symbol table space is reused by the assembler.

If the programmer uses the same symbol for both a macro call and/or OPDEF 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.

2.5.6 Listing Control Statements

As the source program statements are processed during pass 2, the program listing is normally printed on a line printer or a Teletype, depending on the listing file device specified. A sample listing is shown in Figure 7-1.

From left to right, the standard columns contain the location counter, the instruction or data is octal (divided into two 6-digit columns for easier reading), and the symbolic instruction or data, followed by comments. Relocatable object-code addresses are suffixed by a single quote mark ('), which may occur in either the left or right half.

A line printer listing always begins 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.

Listing is suppressed within macro expansions, so that only the macro call and any succeeding lines that generate object program coding are listed.

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

TITLE NAME)	The single operand may contain up to 60 characters which will be printed on the top of each page. The first six characters of the title appear in the assembled program as the program name. If no title is given, the assembler inserts ".MAIN". The program name given in the TITLE statement is used when debugging with DDT to gain access to the program's symbol table.
SUBTTL SUBTITLE)	The single operand may contain up to 40 characters. It is printed as the second line at the top of each page. If the subtitle is changed by another SUBTTL statement, the new subtitle appears in the second line of the following page.

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.)

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.

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, list control codes, and repeats, all of which are suppressed in the standard listing.

XALL) Normally used following a LALL statement to resume standard listing with all text, macro expansions, list control codes and repeats suppressed.

NOSYM) The assembler normally prints out the symbol table at the end of the program, but the NOSYM statement suppresses the symbol table printout.

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
↑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.

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. The operand is treated as a comment and will be output on the error message media. It is not counted as an error, but if error messages are suppressed, PRINTX messages are also suppressed.
REMARK COMMENTS)	The REMARK operator is used for statements which contain only comments. Such statements may also be started with a semi-colon.

2.5.7 Assembler Control Statements

2.5.7.1 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>)
```

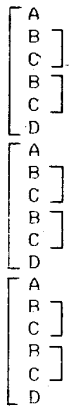
assembles as,

```
ADDX:  ADD 6,X(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,



NOTE

Brackets indicate repetition.

2.5.7.2 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 [USRUUO]
```

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

```
030140 001234   CAL1 3,1234
```

which is equivalent to,

```
030140 001234   Z  3,1234(30000)
```

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.

2.5.7.3 SYN Statements - The statement

SYN symbol, symbol

defines the second operand as synonomous 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 synonomous with the label.

The following are legal SYN statements:

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

2.5.7.4 Permanent Symbols - Redefinition of permanent symbols (e.g., device names like DIS) is permitted. Macro takes the newly defined value, but also flags the line with a "Q" warning message.

2.5.7.5 Extended Instruction Statements - For programming convenience, some extended operation codes are provided in the MACRO-10 Assembler. Primarily, these are used to replace those PDP-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. Additional, 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 reset flags
JOV	JFCL 1Ø,	Jump on overflow and clear
JCRYØ	JFCL 4,	Jump on CRY0 and clear
JCRY1	JFCL 2,	Jump on CRY1 and clear
JCRY	JFCL 6,	Jump on CRY0 or CRY1 and clear
JFOV	JFCL 1,	Jump on floating overflow
RSW	DATAI Ø	Read the console switches

2.5.8 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.

2.5.8.1 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   Q[XWD B+3,A-5]
777774  777773      Q
```

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. An internal symbol may not be set equal to an external symbol.

2.5.8.2 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.

2.5.8.3 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 except for 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.

2.5.9 HISEG Statements

```
HISEG)
```

The HISEG pseudo-op statement generates information that directs the Loader to load the current program into the high segment if the system has re-entrant (two-segment) capability. (Refer to "Block Type 3 Load Into High Segment" in paragraph 6.2.1.1 for additional information.) This pseudo-op may appear anywhere in the source program, but it is recommended that it be placed near the beginning to avoid confusion.

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 construction 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>
```

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 arguments. 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 argument, etc. For example the appearance of the statement:

```
VMAG VECT, 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.

3.3 MACRO FORMAT

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

```
DEFINE JEQ(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 ADD2,X, the calling macro instruction would be written:

```
JEQ<ADD2,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.

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

```
DATAO PTP,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 PTP,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 dymmy 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 a null string. This would override the % prefixed to the second dummy argument and would subsitute 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 and may not be used otherwise inside a macro definition. (Outside a macro definition, it is ignored except as a character in textual data.) A macro argument need not be a complete symbol. Rather, it may be a string of characters which form a complete symbol 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 operator (') may be used in nested macros. However, the assembler removes the operator when it performs concatenation in first level macros, but does not remove the operator during concatenation in the second or deeper levels.

3.6 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. 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 STOPI. When the STOPI 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<0,1,2,3,4,5,6,7>
```

generates:

```
<IRP
IFE K-0,<STOPI
CONV1 0>
IFE K-1,<STOPI
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.

3.7 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 user'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 D,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 7-2.

3.7.1 ASCII Interpretation

If the reverse slash (\ ) is used as the first character 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 characters "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, 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.

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.
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. Example:
		`EXT: EXTERN TXT,BRT,EXT`
		`EXT CANNOT BE BOTH AN EXTERNAL AND INTERNAL SYMBOL.`

L	Literal error	A literal is improper. A literal must generate 1 to 18 words. `EXP [SIXBIT //]; NO CODE GENERATED.`

TABLE 4-1 (Cont)
ERROR CODES

Error Code	Meaning	Explanation

M	Multiply-defined symbol	A symbol is defined more than once. 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).
		
		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. Examples: `↑F13.33E38` (Exceeds range) `↑D15BZ` (Number must follow B shift operator.) But `↑D15B<Z>` is legal if Z is defined. If a number contains meaningless letters or special characters, a Q error is given.
O	Operation code undefined	The operation field of this statement is undefined. It is assembled with a numeric code of 0.
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

TABLE 4-1 (Cont)
ERROR CODES

Error Code	Meaning	Explanation
p	Phase error (cont)	phase error. 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.
		Example:
		`ADD ,TOTAL SUM;`
		`SUM IS NOT NEEDED AND IS TREATED AS A COMMENT.`

R	Relocation error	LOC or RELOC are used improperly. Example: `LOCA; WHERE A IS NOT DEFINED.`
S	Symbol format error	Usually caused by inclusion of illegal special characters. Example: `SY?M: ADD 3,X;`
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.

Error messages during pass 1 consist of two lines. The most recently used label is printed on the first line, 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.

4.1 TELETYPE ERROR MESSAGES

The following error messages may be typed out on the Teletype by MACRO. Those preceded by a question mark are treated as fatal errors when running under Batch Processor (the run is terminated by BATCH.)

END OF PASS 1	Manual loading is required to start pass 2 when the input is paper tape or cards.
LOAD THE NEXT FILE	Manual loading is required if the next file is on paper tape or cards.
?COMMAND ERROR ?NO END STATEMENT ENCOUNTERED ON INPUT FILE	Error in MACRO command string.
?CANNOT ENTER FILE XXX	PASS 1 cannot be completed because the source program is not terminated by an END statement.
?CANNOT FIND FILE XXX	
?INSUFFICIENT CORE	
?.PDP, OVERFLOW, TRY/P	
?INPUT ERROR ON DEVICE DEV	
?DATA ERROR ON DEVICE DEV	
?DEV NOT AVAILABLE	
?THERE ARE N ERRORS	This is the total number of errors detected by MACRO during assembly. These are the errors marked by letter codes on the listing. Under BATCH, if there are one or more errors the run is terminated.

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.* 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.* Finally, references to literals are relocatable.*

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
```

*Except under the LOC code or a PHASE code which specifies absolute addressing.

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 ks, 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, k and 0.
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.

If any of these rules are 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 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. From left to right, the columns on a listing page contain:

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.

b. The assembled instruction and data words, shown in two columns for easier reading, the 6-digit left half-word and the 6-digit right half-word. An apostrophe following either half-word indicates that the word is relocatable.

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 subroutines 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 insure 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
.	
.	
.	
WORD N	(N 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 04:	Global (internal) definition
2ND WORD	Bits 0-35 value of symbol
CODE 10:	Local definition
2ND WORD	Bits 0-35 value of symbol
CODE 60:	Chained global requests:
2ND WORD	Bits 0-17 = 0
2ND WORD	Bits 18-35 pointer to first word of chain requiring definition (see Loader Manual)
CODE 60:	Global symbol additive request: (see Loader Manual)
2ND WORD	Bit 0 = 1
BIT 1	Subtract value before addition
BIT 2	Swap halves before addition
BIT 3	Rotate left 5 before addition
BIT 9	Replace left half with result in storage
BIT 10	Replace right half with result in storage
BIT 11	Replace index field with result in storage
BIT 12	Replace accumulator field with result in storage
BITS 18-35	Pointer to word requiring addition

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 any entry blocks (type 4). This block type transmits no additional data.

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 one word 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.

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.

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 is 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 TIM10B 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 1000)
RIM10B )
```

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 during assembly 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.

In order to use the Read-in-Mode switch with format, the programmer must begin with the statement:

```
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, LOC n, 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 20

     CONO PTR,60
A:   CONSO PTR,10
     JRST .-1
     DATAI PTR,B
     CONSO PTR,10
     JRST .-1
B:   0
     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

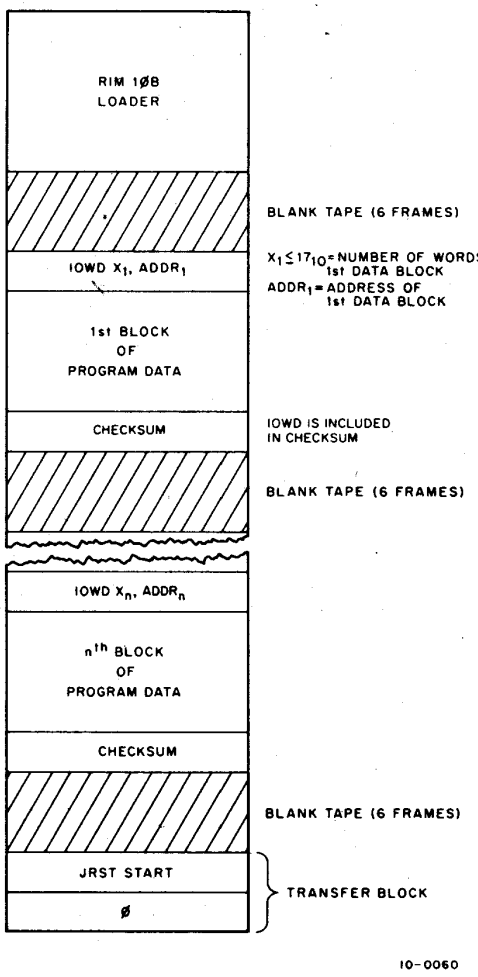


Figure 6-1 General RIM10B Format

```
                XWD -16,0
ST:             CONO PTR,60
ST1:            HRRI A,RD+1
RD:             CONSO PTR,10
                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

A MACRO-10 routine for calculating the logarithm of a complex argument is shown in Figure 7-1. The 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, shown in Figure 7-2, contains a nested macro, SBL, and uses conditional assembly statements, which cause PIXTART and PIXOPT to be generated as either internal or external symbols, depending on the value of SBLSW. In the example, both are externals.

The third example, Figure 7-3, shows two ways of writing a byte unpacking subroutine. Both UNPACK and UNPAX use literals to set up pointer words, and load the bytes in accumulators 0 and 1. The calling sequence for UNPACK actually contains the bytes to be unpacked. For UNPAX, the calling sequence contains the address of the bytes, thus, UNPAX must refer to them indirectly.

The fourth example, Figure 7-4, demonstrates the use of the IRPC (indefinite repeat character) pseudo-op. A macro call, HEX, is made with the arguments ANS, a symbol name, and F, a hexadecimal number. The processing of the macro causes the symbol, ANS, to be assigned the converted value of the hexadecimal number, F. In this example the hexadecimal "digits", listed in ascending order, are: 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E and F.

NOTE

Each complete program (see Figures 7-1 and 7-4) must have an END statement. All other statements may be used at the programmer's discretion; however, a TITLE statement is recommended for documentation and debugging purposes.

```
CLOG     MACROX,H      13:46    7-APR-67      PAGE 1

                              TITLE   CLOG
                              SUBTTL  APRIL 7,1967

                        ;COMPLEX LOGARITHM FUNCTION
                        ;THIS ROUTINE CALCULATES THE LOGARITHM OF A COMPLEX ARGUMENT
                        ; Z = X+I*Y WITH THE FOLLOWING ALGORITHM

                        ;LOG(Z) = LOG(ABSF(Z)) + I*THETA
                        ;WHERE ABSF(Z) = 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

                    000000
                    000001    A=0
                    000010    B=1
                    000011    C=10
                    000016    D=11
                              Q=16

000000  000000  000000  CLOG:   0                       ;ENTRY TO COMPLEX LOG ROUTINE
000001  201436  000000          MOVEI   C, 0(1)         ;GET ADDRESS OF COMPLEX ARGUMENT
000002  200450  000001          MOVE    D,1(C)          ;GET IMAGINARY PART OF ARGUMENT
000003  200410  000000          MOVE    C,(C)           ;GET REAL PART OF ARGUMENT
000004  266700  000000          JSA     1,CABS          ;CALCULATE MAGNITUDE OF Z
000005  000000  000010          EXP     C               ;ADDRESS OF COMPLEX ARGUMENT
000006  266700  000000          JSA     Q,ALOG          ;CALCULATE LOG(ABSF(Z))
000007  000000  000000          EXP     A               ;ADDRESS FOR LOG ROUTINE
000010  250000  000010          EXCH    A,C             ;SWAP ANSWER WITH REAL PART
000011  266700  000000          JSA     Q,ATAN2         ;CALCULATE ANGLE AS ATAN(Y/X)
000012  000000  000011          EXP     D               ;ADDRESS OF Y
000013  000000  000000          EXP     A               ;ADDRESS OF C
000014  200040  000000          MOVE    B,A             ;PUT THETA IN IMAGINARY PART
000015  200000  000010          MOVE    A,C             ;RESTORE REAL PART
000016  267716  000001          JRA     A,1(Q)          ;EXIT

                              END

THERE ARE NO ERRORS

PROGRAM BREAK IS 000017


CLOG    MACROX,H        13:46    7-APR-67      PAGE 2
        SYMBOL TABLE

A               000000
ALOG            000006' EXT
ATAN2           000011' EXT
B               000001
C               000010
CABS            000004' EXT
CLOG            000000' INT
D               000011
Q               000016

5K CORE USED
```

Figure 7-1 Sample Program, CLOG

```
        LALL
000001  PSBLSW=1
        DEFINE  SBLR (A)<IRP A,<SBL A>>
        DEFINE  SBL (A)<
                IFE SBLSW-PSBLSW,<INTERN A>
                IFN SBLSW-PSBLSW,<EXTERN A>>
000003  SBLSW=3
        SBLR <PIXSTART,PIXOPT>↑IRP
        SBL PIXSTART↑
                IFE SBLSW-PSBLSW,<INTERN PIXSTART>
                IFN SBLSW-PSBLSW,<EXTERN PIXSTART>↑
        SBL PIXOPT↑
                IFE SBLSW-PSBLSW,<INTERN PIXOPT>
                IFN SBLSW-PSBLSW,<EXTERN PIXOPT>↑
        ↑

        ;GENERATES PIXSTART AND PIXOPT AS EXTERNALS
```

Figure 7-2 Example of Nested Macro

```
        A=1
        B=3
        C=10

        JSP 17,UNPACK
        BYTE (3)A(15)B(18)C
             .
             .
             .
UNPACK: LDB 0,[POINT 3,0(17),2]          ;PICK UP A
        LDB 1,[POINT 15,0(17),17]        ;PICK UP B
        HRRZ 2,0(17)                     ;PICK UP C
        JRST 1(17)                       ;RETURN
             .
             .
             .
        JSP 17,UNPAX
        EXP [BYTE (3)A(15)B(18)C]
             .
             .
             .
UNPAX:  LDB 0,[POINT 3,@0(17),2]         ;PICK UP A
        LDB 1,[POINT 15,@0(17), 17]      ;PICK UP B
        HRRZ 2,@0(17)                    ;PICK UP C
        JRST 1(17)
```

Figure 7-3 Two Byte Unpacking Subroutines

```
.MAIN  MACRO.V34 15:23 24-MAR-69 PAGE 1

                        LALL

                        DEFINE HEX (N,X)<
                        N=0
                        IRPC X,<IFGE "X"-"A",<N=N*↑D16+"X"-"A"+↑D10>
                                IFLE "X"-"9",<N=N*↑D16+"X"-"0">>>

                        HEX ANS,F↑
                000000  ANS=0
                        IRPC
                000017  IFGE "F"-"A",<ANS=ANS*↑D16+"F"-"A"+↑D10>
                                IFLE "F"-"9",<ANS=ANS*↑D16+"F"-"0">
                        ↑
                        HEX ANS,10↑
                000000  ANS=0
                        IRPC
                        IFGE "1"-"A",<ANS=ANS*↑D16+"1"-"A"+↑D10>
                000001          IFLE "1"-"9",<ANS=ANS*↑D16+"1"-"0">
                        IFGE "0"-"A",<ANS=ANS*↑D16+"0"-"A"+↑D10>
                000020          IFLE "0"-"9",<ANS=ANS*↑D16+"0"-"0">
                        ↑
                        HEX ANS,9ABCDEF↑
                000000  ANS=0
                        IRPC
                        IFGE "9"-"A",<ANS=ANS*↑D16+"9"-"A"+↑D10>
                000011          IFLE "9"-"9",<ANS=ANS*↑D16+"9"-"0">
                000232  IFGE "A"-"A",<ANS=ANS*↑D16+"A"-"A"+↑D10>
                                IFLE "A"-"9",<ANS=ANS*↑D16+"A"-"0">
                004653  IFGE "B"-"A",<ANS=ANS*↑D16+"B"-"A"+↑D10>
                                IFLE "B"-"9",<ANS=ANS*↑D16+"B"-"0">
                115274  IFGE "C"-"A",<ANS=ANS*↑D16+"C"-"A"+↑D10>
                                IFLE "C"-"9",<ANS=ANS*↑D16"C"-"0">
        000002  325715  IFGE "D"-"A",<ANS=ANS*↑D16+"D"-"A"+↑D10>
                                IFLE "D"-"9",<ANS=ANS*↑D16+"D"-"0">
        000046  536336  IFGE "E"-"A",<ANS=ANS*↑D16+"E"-"A"+↑D10>
                                IFLE "E"-"9",<ANS=ANS*↑D16+"E"-"0">
        001152  746757  IFGE "F"-"A",<ANS=ANS*↑D16+"F"-"A"+↑D10>
                                IFLE "F"-"9",<ANS=ANS*↑D16+"F"-"0">
                        ↑

                        END

NO ERRORS DETECTED

PROGRAM BREAK IS 000000
```

Figure 7-4 IRPC Example

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. These programmed operators, or user utilized operation codes are called UUO's in the list.

The notes are used to specify which pseudo-ops generate data, and which do not. Pseudo-ops which 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 also given in the notes. These may be useful in checking listings.

ASSEMBLER PSEUDO-OPS AND MONITOR COMMANDS

ASCII, pseudo-op, generates data
ASCIZ, pseudo-op, generates data
BLOCK, pseudo-op, no data generated
BYTE, pseudo-op, generates data
CALL, Ø4Ø, Monitor UUO
CALLI, Ø47, Monitor UUO
CLOSE, Ø7Ø, Monitor UUO
DATA., Ø2Ø, FORTRAN UUO
DEC, pseudo-op, generates data
DEC., Ø33, FORTRAN UUO
DEFINE, pseudo-op, no data generated
DEPHASE, pseudo-op, no data generated
ENC., Ø34, FORTRAN UUO
END, pseudo-op, no data generated
ENTER, Ø77, Monitor UUO
ENTRY, pseudo-op, no data generated
EXP, pseudo-op, generates data
EXTERN, pseudo-op, no data generated
FIN., Ø21, FORTRAN UUO
GETSTS, Ø62, 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, Ø56, Monitor UUO
IN., Ø16, FORTRAN UUO
INBUF, Ø64, Monitor UUO
INF., Ø26, FORTRAN UUO
INIT, Ø41, Monitor UUO
INPUT, Ø66, Monitor UUO
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, no data generated
LOC, pseudo-op, no data generated
LOOKUP, Ø76, Monitor UUO
MLOFF, pseudo-op, no data generated
MLON, pseudo-op, no data generated
MTAPE, Ø72, Monitor UUO
MTOP., Ø24, FORTRAN UUO
NLI., Ø31, FORTRAN UUO
NLO., Ø32, FORTRAN UUO
NOSYM, pseudo-op, no data generated
OCT, pseudo-op, generates data
OPDEF, pseudo-op, no data generated
OPEN, Ø5Ø, Monitor UUO
OUT, Ø57, Monitor UUO
OUT., Ø17, FORTRAN UUO
OUTBUF, Ø65, Monitor UUO
OUTF., Ø27, FORTRAN UUO
OUTPUT, Ø67, 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
RADIX5Ø, pseudo-op, generates data
RELEAS, Ø71, Monitor UUO
RELOC, pseudo-op, no data generated
REMARK, pseudo-op, no data generated
RENAME, Ø55, Monitor UUO
REPEAT, pseudo-op, no data generated
RERED., Ø3Ø, FORTRAN UUO
RESET., Ø15, FORTRAN UUO
RIM, pseudo-op, no data generated
RIM1Ø, pseudo-op, no data generated
RIM1ØB, pseudo-op, no data generated
RTB., Ø22, FORTRAN UUO
SETSTS, Ø6Ø, Monitor UUO
SIXBIT, pseudo-op, generates data
SLIST., Ø25, FORTRAN UUO
SQUOZE, same as RADIX5Ø
STATO, Ø61, Monitor UUO
STATUS, Ø62, Monitor UUO
STATZ, Ø63, Monitor UUO
STOPI, pseudo-op, no data generated
SUBTTL, pseudo-op, no data generated
SYN, pseudo-op, no data generated
TAPE, pseudo-op, no data generated
TITLE, pseudo-op, no data generated
TTCALL, Ø51, Monitor UUO
UGETF, Ø73, Monitor UUO
UJEN, 1ØØ, Monitor UUO
USETI, Ø74, Monitor UUO
USETO, Ø75, Monitor UUO
VAR, pseudo-op, generates data
WTB., Ø23, FORTRAN UUO
XALL, pseudo-op, no data generated
XLIST, pseudo-op, no data generated
XWD, pseudo-op, generates data
Z, pseudo-op, generates data

MACHINE MNEMONICS AND OCTAL CODES

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

APPENDIX B
SUMMARY OF PSEUDO-OPS

ASCII	Seven-bit ASCII text.
ASCIZ	Seven-bit ASCII test, with null character guaranteed at end.
BLOCK	Reserves block of storage cells.
BYTE	Input bytes of length 1-36 bits.
DEC	Input decimal numbers.
DEFINE	defines macro
DEPHASE	Terminates PHASE relocation mode.
END	Last statement of the program.
ENTRY	Enters subroutine library.
EXP	Input expressions.
EXTERN	Identifies external symbols.

Conditional Assembly Statements

	Assemble if:
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

Format	Operator	Page	Type	Notes
PRI	ANDCAB	413		
	ANDCAI	411		
	ANDCAM	412		
	ANDCB	440		
	ANDCBB	443		
	ANDCBI	441		
	ANDCBM	442		
	ANDCM	420		
	ANDCMB	423		
	ANDCMI	421		
	ANDCMM	422		
PRI	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		

VAR	Assemble variables suffixed with#
XALL	Stop expanded listing
XLIST	Stop listing
XWD	Input two 18-bit half words
Z	Input zero word

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
; ;	Semi-colon. Precedes all comments.	;THIS IS A COMMENT
. .	Point. Has current value of the location counter.	JRST .+5 JUMP FORWARD FIVE LOCATIONS
, ,	Comma. General operand or argument delimiter	DEC 10,5,6 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	
	Multiplication	
	Division	Arithmetic Operators
	Add (+A outputs the value of A)	
	Subtract	
1st character of text string	In ASCII, ASCIZ and SIXBIT test 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. Use to 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+500/C>
	In conditional assembly statements contain a single argument, and the conditional coding.	IF1, <MOVE AC0, TAX>
	In REPEAT statements, contain coding to be repeated.	REPEAT 3, <SUB 17, TAG>
	In macors, enclose the macro definition	DEFINE PUNCH <DATAO PTP, PUNBUF (4)>
[ ]	Square brackets. Delimits literals.	ADD 5,[MOVEI 3,TAX]
	In OPDEF statement, contain new operator.	OPDEF CAL [MOVE]
=	Equal sign, direct assignment	SYM=6 SYM-A+B*D
"..."	Quotation marks enclose 7-bit ASCII text, from one to five characters.	"ABCDE"
#	Number sign. Defines a symbol used as a tag. Variable.	ADD 3,TAG#
'	Apostrophe or single quote. Catenation character, used only within macro definitions.	DEFINE MAC (A,B,C); <JUMP'A B, C>
\	Reverse slash. If used as the first character in a macro call, the value of the following symbol is converted to an ASCII symbol in the current radix.	MAC \ A IF A=500, THIS GENERATES THREE 7-BIT ASCII CHARACTERS, ASCII/500/
	Control left arrow. Line continuation.	
←	Left arrow. N←M shift n left (or right) n times.	100←3=1000 100←3=10

APPENDIX D

ASSEMBLER EVALUATION OF
STATEMENTS AND EXPRESSIONS

Order of Statement Evaluation:

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

Label field	Operator field	Operand fields
1. Symbol suffixed by colon. If colon not found no label is present.	1. Number 2. Macro/OPDEF 3. Machine operator 4. Assembler operator 5. Symbol	1. Number 2. Symbol 3. Macro/OPDEF 4. Machine Operator 5. Assembler Operator

Notice that a single symbol could be used as a label, an operator, or an operand, depending upon where it is used.

The assembler checks the operator field for a number, first, 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 which do not generate data may not be used as part of operand expressions. If a statement used in an operand expression generates more than one word of data, only the first word generated is meaningful.

Order of Expression Evaluation:

- (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.

APPENDIX E

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 sixbit symbols into 32 bits. Let each character of a symbol be subscripted in descending order from left to right; that is, let the symbols be of the form

$$L_6L_5L_4L_3L_2L_1$$

If C_n denotes the 6-bit code for L_n, the radix 50_8 representation is generated by the following:

$$(((((C_6*50)+C_5)*50+C_4)*50+C_3)*50+C_2)*50+C_1$$

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	%

APPENDIX G

SUMMARY OF RULES FOR DEFINING AND CALLING MACROS

Assembler Interpretation:

MACRO-10 assembles macros by direct and immediate character substitutions. Whenever 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.

Character handling:

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

b. 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 it and its matching right bracket. The matching brackets are removed from the string but the scan continues until a standard delimiter is found.

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

d. 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 substitution is processed.

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

f. Carriage return: A carriage return, except when pre-empted by angle brackets (see b above) will terminate the scan similar to the semicolon. This can be circumvented, if desired, by the control left arrow key described elsewnere.

g. 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 which produce binary code are listed. The pseudo-op LALL will cause all lines to be listed. Substituted arguments are bracketed by t's by the assembler.

Concatenation:

The rules for concatenation are as follows:

a. Within the outer level of angle bracket nesting one apostrophe is removed from each string of apostrophes. Thus, if a single apostrophe is encountered, it is removed; if a pair are encountered, one is removed and one left, etc.

b. Within nested brackets, all single apostrophes are passed on to the macro processor.

Outside of macro definitions, single apostrophes are ignored except when in text strings. Therefore, MO"VEI is the equivalent of MOVEI. In any event, apostrophes will appear on the listing.

APPENDIX H
OPERATING INSTRUCTIONS

Requirements

Monitor

Minimum Core:	6K
Additional Core:	Automatically requests additional core assignments from the time-sharing monitor as needed
Equipment:	One input device (source program input); 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 be written on either DECtape, magnetic tape, or disk.

Initialization

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

General Command Format

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.*
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.

*If /C switch is given, but no list-dev: is specified, DSK:CREF.TMP is assumed.

Disk File Command Format

DSK:filename.ext [proj,prog]

[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 standard protection* is assigned to any disk file specified as output.

NOTE

If object coding output is not desired (as in the case where a program is being scanned for source language errors), objprog-dev: is omitted. If an assembly listing is not desired, list-dev: is omitted.

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.
THERE ARE 2 ERRORS) PROGRAM BREAK IS 002537) 5K CORE USED)	Number of source errors. Size of object program. Core used by assembler.
*↑C)	Return to the Monitor.

.R MACRO) *MTA3:,MTA2:←MTA1:,,) THERE ARE NO ERRORS) PROGRAM BREAK IS 003552) 6K 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) THERE ARE NO ERRORS) PROGRAM BREAK IS 001027) 6K CORE USED)	Assemble the source files named FILE1, FILE2, and FILE5 from DTA1; produce no object coding; write the listing on the line printer.

*Standard protection (055) designates that the owner is permitted to read or write, or change the protection of, the file while others are permitted only to read the file.

```
*,←DSK:FILE1.MAC[14,12])
[THERE ARE NO ERRORS)
 PROGRAM BREAK IS 000544)
 5K 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 Teletype.

```
*↑C)
```

Return to the Monitor.

```
.R MACRO
*MTA1:,TTY:←TTY:)
```

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

```
      JMP    R)
R:    AOS    G)
G:    JFCL)
      END )
```

Enter the source statements

```
↑Z)
```

Terminate input.

```
END OF PASS 1)
```

Re-enter Teletype input.

```
      JMP    R )
```

Re-enter the first statement.

```
[.MAIN    MACRO          10:14   20-DEC-67   PAGE1)
[0        000000 000000  000001'     JMP     R)
R:        AOS            G
          000001 350000  000002'  R: AOS     G)
 G:        JFCL)
          000002 255000  000000   G: JFCL)
          END )
                                        END)
```

Page heading.

First assembled.

Re-enter second.

Second assembled.

Re-enter third.

Third assembled.

Re-enter fourth.

Fourth assembled.

```
[THERE IS 1 ERROR)
PROGRAM BREAK IS 000003)
.MAIN   MACRO          10:14    20-DEC-67   PAGE 2)

        SYMBOL TABLE)
G       000002')
R       000001')

5K CORE USED)
```

Typeout of symbol table.

```
*↑C)
```

Return to the Monitor.

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.

Table 3-1
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.TMP is assigned as the filename; if no extension is given, .TMP is assigned; if no list-dev: is specified, DSK: is assumed.
E	List macro expansions (same function as LALL pseudo-op).
L	Reinstate listing (used after list suppression by XLIST pseudo-op or S switch).
N	Suppress error printouts on the Teletype.
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}).
Q	Suppress Q (questionable) error indications on the listing; Q messages indicate assumptions made during pass 1.
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.TMP.

```
END OF PASS 1)
```

The paper tape must be refed by the operator for pass 2.

```
[THERE ARE 3 ERRORS)
 PROGRAM BREAK IS 000401)
 5K CORE USED)
```

End-of-assembly messages.

```
*DTA2:ASSEMB.ONE/Z,LPT:
     ←MTA4:/W,)

[THERE ARE NO ERRORS)
 PROGRAM BREAK IS 005231)
 6K CORE USED)
```

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.

```
*MTA1:/W,LPT:←MTA3:
     /W,(AA),(BB)  )

[THERE IS 1 ERROR)
 PROGRAM BREAK IS 000655)
 5K CORE USED)
```

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.

```
*↑C)
.
```

Return to the Monitor.

Diagnostic Messages

Table 3-2
Macro-10 Diagnostic Messages

Message	Meaning
?CANNOT ENTER FILE filename.ext	DTA or DSK directory is full; file cannot be entered.
?CANNOT FIND filename.ext	The file cannot be found on the device specified.
?COMMAND ERROR	The last command string is in error.
?DATA ERROR ON DEVICE dev:	Output error has occurred on the device.

Table 3-2 (Cont)
Macro-10 Diagnostic Messages

Message	Meaning
END OF PASS1	This message is issued prior to pass 2 whenever the input source file is on a medium which must be manually re-entered by the operator (PTR:, CDR:, TTY:). When this message appears, the operator must re-feed the tape or cards or retype the entries.
?IMPROPER INPUT DATA	The input data is not in the proper format.
?INPUT ERROR ON DEVICE dev:	Data cannot be read.
?INSUFFICIENT CORE	An insufficient amount of core is available for assembly.
nK CORE USED	Amount of core used for this assembly.
LOAD THE NEXT FILE	Manual loading is required for the next card or paper tape file.
?NO END STATEMENT ENCOUNTERED ON INPUT FILE	The END statement is missing at the end of the source program file.
?dev: NOT AVAILABLE	The device is assigned to another user or does not exist.
?PDP OVERFLOW, TRY/P	A pushdown list overflow has occurred.
PROGRAM BREAK IS nnnnn	The highest relative location occupied by the object program produced.
?THERE ARE n ERRORS THERE ARE NO ERRORS ?THERE IS 1 ERROR	Number of source language errors found.

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, 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.

Table 3-3
Macro-10 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.
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. Example: EXT: EXTERN TXT, BRT, EXT EXT cannot be both an external and internal symbol.
L	Literal error	A literal is improper. A literal must generate 1 to 18 words. Example: EXP [SIXBIT //]; no code generated.
M	Multiply-defined symbol	A symbol is defined more than once. 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). 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: ADD3,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. Examples: ↑F13.33E38 (Exceeds range) ↑D15BZ (Number must follow B shift operator.) But, ↑D15B<Z> is illegal if Z is defined. If a number contains meaningless letters or special characters, a Q error is given.
O	Operation code undefined	The operation field of this statement is undefined. It is assembled with a numeric code of 0.

Table 3-3 (Cont)
Macro-10 Error Codes

Error Code	Meaning	Explanation
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. 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. Example: ADD ,TOTAL SUM; SUM is not needed and is treated as a comment.
R	Relocation error	LOC or RELOC are used improperly. Example: LOCA; where A is not defined.
S	Symbol format error	Usually caused by inclusion of illegal special characters. Example: SY?M: ADD 3,X;
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.

Monitor Commands

Assembly of Macro source program files can be performed by use of the COMPILE, LOAD, EXECUTE, and DEBUG commands. See Table 9-1, Time-Sharing Monitor Commands, in Chapter 9 of this manual for details.