

**PROGRAMMING
MANUAL**

**MACRO-6
ASSEMBLY LANGUAGE**

PDP-6

**PDP-6 PROGRAMMING MANUAL
MACRO-6 ASSEMBLY LANGUAGE**

Copyright 1965 by Digital Equipment Corporation

CONTENTS

<u>Chapter</u>		<u>Page</u>
1	LANGUAGE FUNDAMENTALS	1
	Introduction	1
	Character Set	1
	The Location Counter	2
	Elements	3
	Symbols	3
	Numbers	3
	Point	4
	Text	5
	Literals	5
	Expressions	6
	Evaluation of Symbols	7
2	STATEMENTS	9
	Comment Statements	10
	Instruction Statements	10
	Primary Instruction Statements	10
	I/O Instruction Statements	11
	Extended Instruction Statements	12
	Assembly	13
	Number Codes	13
	Data Statements	13
	Assembler Control Codes	17
	Listing Control	23
3	MACROS	25
	Definition of Macros	25
	Macro Calls	26
	Additional Considerations	26
	Created Symbols	28

CONTENTS (continued)

<u>Chapter</u>		<u>Page</u>
	Concatenation	30
	Indefinite Report	30
	Nesting and Redefinition	32
4	RELOCATION AND LINKING	35
	Relocation	35
	Linking Subroutines	37
5	ERRORS	39
	The Error Flags	39
6	ASSEMBLY OUTPUT	41
	Assembly Listing	41
	Binary Program	41
	Rim Format	41
	LINK Format	42
	The Formats for the Block Types	43
7	ASSEMBLER INITIALIZATION	46
<u>Appendix</u>		
1	CODES	A1
2	SUMMARY OF ERROR FLAGS	A3
3	PROGRAMMING EXAMPLES	A4
4	CHARACTER SETS	A7

CHAPTER 1

LANGUAGE FUNDAMENTALS

INTRODUCTION

MACRO-6 programs consist of a sequence of statements, each of which may generate one or more machine instructions, generate words of data, or give special instructions to the MACRO-6 Assembler. The statements, in turn, are subdivided into fields: a label field, a code field, argument fields, and comment fields. The fields may contain one—or more—of the basic elements of the language described below. The interpretation of the basic element depends on the field in which it appears.

This chapter begins with the MACRO-6 character set. It then describes the basic elements of the language and how they may be constructed.

Character Set

The characters which are meaningful to the MACRO-6 Assembler are:

(space)	-	:	G	T
!	.	;	H	U
"	/	<	I	V
#	0	=	J	W
\$	1	>	K	X
%	2	?	L	Y
&	3	@	M	Z
'	4	A	N	[
(5	B	O]
)	6	C	P	↑
*	7	D	Q	
+	8	E	R	
,	9	F	S	

The corresponding ASCII, 6-bit ASCII, and punched-card codes are shown in Appendix 4. Two of the characters shown in the appendix do not appear above. They are back slash and reverse arrow. These two characters are ignored by the Assembler and should not be used.

Punched Paper Tape

The ASCII code is used for paper tape input. In addition to the characters shown above there are some nonprinting characters of significance, i.e., carriage return, line feed, and tabs. Tabs are equivalent to a number of spaces and are properly translated to the correct number of spaces on the output listings. Both tabs and spaces may be freely used (except for one restriction—see Code Fields) to improve the appearance of programs. Statements must be terminated by a semicolon or by a carriage return. All carriage returns must be followed by a line feed, and all line feeds must be preceded by either a carriage return or another line feed.

Punched Cards

A modified Hollerith code (Appendix 4) is used for card input. Only the first 72 columns are considered by the processor; the remaining 8 may be used for identifying information. The Assembler does not recognize a fixed-field input from the cards. That is, fields within a statement are not delimited by appearing in specified card columns. The fields must be delimited by specified characters; the delimiters being exactly the same as for punched tape. The statement itself is automatically delimited by reaching the end of the card—column 72. To skip lines, blank cards which generate no information may be inserted.

THE LOCATION COUNTER

In general, statements generate 36-bit binary words, which are placed into consecutive memory locations. The location counter is a register used by the MACRO-6 processor to keep track of the next available location in memory. It is updated after processing each statement. A statement which generates a single machine instruction would update the location counter by one; a statement which generates six data words would update it by six. The location counter may be explicitly set by the LOC or RELOC codes.

ELEMENTS

Elements represent binary integers less than 2^{35} . There are five types of elements: symbols, numbers, characters, points, and literals.

Symbols

These are strings of letters, numbers, or decimal points, the first of which must be a letter or decimal point. The characters % and \$ are regarded as letters in forming symbols, although a symbol may be any length, only the first six characters are considered, and any additional characters are ignored. Symbols which are identical in their first six characters are considered identical.

X	
A65	
NUMERIC	(equivalent to NUMERI)
X.3B	
HIGH.	
N12345	

Numbers

A number is a string of digits. If the string contains a decimal point, it is evaluated as a floating-decimal number and the digits are taken radix 10. If the string does not contain a decimal point, the digits are assigned values according to the prevailing radix. (This prevailing radix is normally regulated by the RADIX code.) If 8 were the prevailing radix, the number 17 would have the value $17_8 = 15_{10}$. If 10 were prevailing, 17 would have the value $17_{10} = 21_8$. The number 17.0 would always have the value ~~205420000000~~ since the decimal point denotes a floating-decimal number. A number must always begin with a digit or a decimal point.

Occasionally, it may be desirable to change the value of the radix for only one numeric element. This is done by the qualifier † followed by a letter. Numbers are qualified in this manner to be Decimal, Octal, Binary, or Fixed point decimal fractions irrespective of the prevailing radix. Thus:

$\uparrow D17 = 17_{10}$ $\uparrow O17 = 15_{10}$ $\uparrow B1\emptyset1\emptyset = 1\emptyset_{10} = 12_8$

These qualifiers have no further effect on the prevailing radix. Floating-decimal numbers never consider qualifiers, except F. The exponent parts of floating-decimal numbers may be further augmented by following the number by $E\pm n$; the number is then considered to be multiplied by $1\emptyset^{\pm n}$.

$1.$ $1\emptyset.\emptyset E-1$ $\emptyset.\emptyset\emptyset\emptyset1 E4$ $.\emptyset\emptyset1 E+3$	The binary representation of each is $2\emptyset14\emptyset\emptyset\emptyset\emptyset\emptyset\emptyset\emptyset$
---	--

If, in addition, the characters $\uparrow F$ are prefixed to a number, it is considered to be a fixed decimal fraction. In this case, B_n should be suffixed to the number where n is an integer and $\emptyset \leq n \leq 35$. The decimal point is then taken to be to the right of bit n . (If no bit position, B , is specified, 35 is assumed.) Any integer part of the number's truncated to fit in n bits.

$\uparrow F3.25B8 =$	$\emptyset\emptyset32\emptyset\emptyset\emptyset\emptyset\emptyset\emptyset\emptyset$
$\uparrow F.281250B12 =$	$\emptyset\emptyset\emptyset\emptyset22\emptyset\emptyset\emptyset\emptyset\emptyset\emptyset$
$\uparrow F.4498\emptyset46E+1B11$	$\emptyset\emptyset\emptyset4377\emptyset\emptyset\emptyset\emptyset\emptyset$

A number may also be logically shifted left by following it by B_r . The number is shifted left so that the right-hand (low-order) bit is in position r (decimal) of the 36-bit computer word. Thus:

$\emptyset3B35 =$	$\emptyset\emptyset\emptyset\emptyset\emptyset\emptyset\emptyset\emptyset\emptyset\emptyset3$
$\emptyset3B31 =$	$\emptyset\emptyset\emptyset\emptyset\emptyset\emptyset\emptyset\emptyset\emptyset6\emptyset$
$\emptyset3B17 =$	$\emptyset\emptyset\emptyset\emptyset\emptyset3\emptyset\emptyset\emptyset\emptyset\emptyset\emptyset$

Point

The decimal point alone has a special meaning; it represents the current value of the location counter. For example:

```
A: JRST.-1
;EQUIVALENT TO
A: JRST A-1
```

Text

If the first nonblank character of an element is a quote("), the characters following it are assembled right justified as their 7-bit ASCII representations. Only printing characters are assembled. This element is terminated by a quote or a carriage return. If more than five characters are included within the quotes, only the right-hand five are considered.

```
"AXE" is equivalent to 4066105
(This representation is useful with
immediate mode operations.)
```

Literals

Literals are referenced in the argument field of a statement and are delimited by a pair of brackets. The information within the brackets (whether it be a data word or machine instruction) is assembled and assigned a specific storage location (usually at the end of the program). The address of the generated word appears in the statement which referenced the literal. Literals may be nested to any reasonable depth.

```
ADD 2, [DEC 65], DECIMAL LITERAL
FAD 1, [8.14], FLOATING POINT
MOVE 3, [ASCII .BYTES.]
XCT[XCT[XCT[ADD 2,X]] (4)]; NESTED
```

The last example generates the following constants. For example:

```
LIT1: XCT LIT2(4)
LIT2: XCT LIT3
LIT3: ADD 2,X
```


An expression enclosed by angle bracket may be regarded as an element, allowing compound expressions to be formed:

$\langle A+B \rangle / 5$ represents 2

$C * \langle A+B * \langle D-C \rangle \rangle$ represents 54_{10}

EVALUATION OF SYMBOLS

The value represented by a symbol is assigned by one of three mechanisms: a label, a direct assignment, or a variable.

Label

If a statement begins with a symbol followed by a colon, the symbol is called a label. It is assigned the current value of the location counter.

Direct Assignment

If a symbol appears on the left-hand side of the equal sign in a direct assignment statement, it is assigned a value equal to the value represented by the expression on the right-hand side. A direct assignment statement has the form:

$$\text{SYM} = \text{EXP}, \quad \text{COMMENT}$$

where SYM is a symbol and EXP is an expression.

For example:

$$\begin{aligned} A &= B+2 \\ \text{TSIZE} &= \text{TEND}-\text{TSTART} \\ K &= 4 \end{aligned}$$

Variable

If a symbol is followed by a number sign (#), a storage cell is automatically reserved (usually at the end of the program), and the symbol represents the location of this storage cell. The number sign

may appear after any one or more occurrences of the variable; it need not appear after all occurrences, nor after the first occurrence.

This is useful for defining a symbolic temporary storage location. For example: TEMP#, which reserves a cell whose address is represented by TEMP.

If a value is assigned directly, it may be altered by another direct assignment statement. If it is defined as a label or variable, it may not be altered.

CHAPTER 2

STATEMENTS

There are four types of statements in the MACRO-6 language: comment statements, instruction statements, data statements, and assembler control statements. The type of a statement is identified by the fields present and the code contained in the code field.

The possible fields are listed below in the order in which they would appear from left to right. Each field extends from the terminator for the preceding field, or from the beginning of the statement if all preceding fields are null, to its own terminator.

Label Field

If present, this field must be terminated by a colon. This field contains a string of characters representing one, and only one, symbol. When a symbol appears in a label field, it is immediately defined to have a value equal to the current value of the location counter.

Code Field

This field is terminated by either a space or a comma. It may contain mnemonics representing either PDP-6 instructions or any of the pseudo-operation codes recognized by the Assembler.

Argument Fields

The function of these fields is determined by the code field. They may describe data, machine addresses, accumulators, assembler control parameters, index registers, etc. They may be delimited by commas, parentheses, or angle-brackets, depending on their function in the statement.

If a statement is ended with fewer fields than are normally required, the unspecified fields are considered null. If a statement has more fields than are required, the superfluous fields are taken to be comments. The information between the semicolon, if present, and the end of a card or carriage return is also taken as a comment.

The field completely determines the interpretation to be given to its contents. For example, if the characters ADD appeared in a label field, they would be interpreted as a statement label and would receive a value equal to the statements location in memory. If the same characters appeared in a code field, they would be interpreted as the mnemonic for a PDP-6 instruction and would receive the value 270B8.

COMMENT STATEMENTS

A statement with an empty or blank code field is considered to be a comment statement. The presence of the empty field is indicated by the presence of the field's delimiter, i.e., a semicolon. For example:

; THIS IS A COMMENT

INSTRUCTION STATEMENTS

Instruction statements may have any or all of the possible fields. They must have a nonempty code field. There are three types of instruction statements: primary instruction statements, extended instruction statements, and I/O instruction statements.

Primary Instruction Statements

The primary instruction statements must have in their code field one of the PDP-6 instruction mnemonics, (including the appropriate mode suffix) except for the eight I/O instructions. (For the complete list of mnemonics and mode controls, see F-65.) There must be no space between the instruction mnemonic and the mode control since this space would attempt to terminate the code field.

If the field following the code field is terminated by a comma, it is an accumulator field; otherwise, it is the operand address field. If there is an accumulator field, the next field is the address field. If a field is enclosed in parentheses, it is an index register field. The character @ appearing in the address field denotes indirect addressing. The contents of these argument fields may be any desired expression.

The accumulator field may be left out and the code field delimited by a comma or space. In this case, the accumulator is considered to be accumulator \emptyset . If indexing is not used, the index field may also be left out and the address field delimited by a comma, carriage return, or semicolon; otherwise, it is delimited by the opening parenthesis of the index field. For example:

```
SUM: ADD 2, TABLE(X3)
      ADD AC2, Y
      JRST .-3;
      JMP (4)
```

I/O Instruction Statements

The I/O instruction statements are exactly like the primary instruction statements with the following exceptions:

1. The code field must contain one of the I/O instruction mnemonics for the PDP-6 (see F-65).
2. The accumulator field is replaced by a device field. The device field may contain either a device number or a device mnemonic (see F-65). For example:

```
READ: DATA1 PTR, @NUM(4)
      CONO 203; ENABLE PC ON CH 3
```

Extended Instruction Statements

For programming convenience, some extended operation codes are provided in the MACRO-6 Assembler. Primarily, these are to replace those PDP-6 instructions where the combination of instruction mnemonic and accumulator field are both used to denote a single instruction. For example:

JRST 4,

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

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

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

Extended Mnemonics	Equivalent PDP-6 Mnemonics	Meaning
JEN	JRST 12,	Jump and enable the PI system
HALT	JRST 4,	Halt
JRSTF	JRST 2,	Jump and reset flags
JOV	JFCL 8,	Jump on overflow and clear
JCRYØ	JFCL 4,	Jump on CRYØ and clear
JCRY1	JFCL 2,	Jump on CRY1 and clear
JCRY	JFCL 6,	Jump on CRYØ or CRY1 and clear
JPC	JFCL 1,	Jump on PC change flag and clear
RSW	DATAI Ø,	Read the console switches

Assembly

Instructions are assembled in the following manner. Each instruction code represents a 36-bit number. If it is a primary instruction code, the low-order 4 bits of the value of the accumulator expression are IORed into positions 9-12. The low-order 9 bits of the value of the device expression of an I/O instruction are IORed into positions 3-11. The low-order 18 bits of the value of the address expression are IORed into the right half of the instruction. If the indirect address symbol @ appears in the address field, a bit is placed into position 13. Finally, if the index register field exists, the lower four bits are IORed into positions 14-17.

Numeric Codes

Numeric codes are considered to indicate primary instructions. The remainder of the statement is assembled as a normal instruction. If the numeric code is preceded by a minus sign, the 2's complement of the number is taken. The minus sign is ignored for other codes. Character elements are considered to be numeric. For example:

Ø,A; THE VALUE OF A IS IN THE RT HALF
27ØB8 2,X; EQUIVALENT TO ADD 2,X
↑DØ5; ØØØØØØØØØØ1Ø1 IS GENERATED
-1; 777777777777 IS GENERATED

DATA STATEMENTS

Several codes are used to indicate various data formats. These codes describe the type of data to be generated. A label on a data-generating statement refers to the first word assembled.

DEC (decimal data)—Set the radix to 1Ø for this statement only and generate a word for each expression following the code. Expressions are separated by commas. For example:

DEC 1Ø, 4.5, 3.1416, 6.Ø3E-26, 3;
;5 WORDS GENERATED

RADIX50

Set the radix to 50 for this statement only. RADIX50 requires two arguments; the symbol and a code digit. For example:

```
RADIX50 SYM,3
```

OCT

(octal data)—Similar to the DEC code, but the radix is temporarily set to 8. For example:

```
OCT -3, 2, 777, 4.1;THE 4TH ITEM IS FLOATING PT.
```

EXP

(expressions)—The radix is unchanged. Each expression following the code generates one 36-bit data word. For example:

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

XWD

(transfer word)—Two expressions follow this code which generates one data word. The low-order 18 bits of the value of the first are placed into the left-half word, and the low-order 18 bits of the value of the second expression are placed into the right half. For example:

```
ATOB: XWD A,B; POINTER WORD FOR BLOCK TRANSFER
```

Z

(zero word)—One word containing zeros is generated. For example:

```
TEMP: Z; TEMPORARY STORAGE
```

IOWD

(I/O transfer word)—used in the BLKI and BLKO instructions. Two expressions separated by a comma follow this code, which generates one data word. The left half of the assembled word contains the 2's complement of the value of the first expression, and the right half contains the value of the second expression minus one. For example:

```
INAREA: IOWD 6, †D265;  
ASSEMBLES AS 777772000377
```

POINT

(byte pointer word)—The first expression indicates the byte size, the second indicates the address, and the third indicates the position of the right-hand bit of the byte position. The indirect character @ and index expression in parentheses may appear in conjunction with the address part. The local radix for the position and size expressions is always 10, regardless of the prevailing radix. For example:

```
STRING: POINT 6,@N(4),5;  
;POINTS TO THE LH CHAR
```

If the position expression is left blank, the position part will assemble as 44₈. On incrementing, the pointer will point to the left-hand byte.

SIXBIT

(alphabetic information)—This code is used to generate characters in 6-bit ASCII code, pack them into 6-character words and place the words in sequential registers. The first nonblank character following the code is the delimiter. Information is assembled

from the second character until the first character is repeated. Only the printing characters of the ASCII code are assembled, except line feeds which are assembled as 74 (\). The characters are left justified. For example:

```
NUMBER2 SIXBIT "2"  
ALPHA: SIXBIT /ALPHABETIC INFORMATION/  
,EQUIVALENT TO  
ALPHA: OCT 411460504142, 456451430051;  
OCT 564657625541, 645157560000;
```

ASCII (alphabetic information)—This code is similar to SIXBIT, but it packs words with the low 7 bits of the full ASCII representation. The entire ASCII character set may be assembled under this mode, including the reverse slash (\) and back arrow (←). For example:

```
ASCII . )  
.;A CARRIAGE RETURN AND LINE FEED
```

ASCIZ Same as ASCII except that a word of 0 is assembled at the end of the data if the number of ASCII characters is a multiple of five.

BLOCK (block of storage reserved)—The expression following the code indicates the number of cells to be reserved. The location counter is incremented by the value of the expression. The expression may be an absolute value or a mixed arithmetic. For example:

```
MATRIX: BLOCK N*M
```

BYTE

(byte strings)—The first expression following this code is enclosed in parentheses and is the byte size. Subsequent expressions, separated by commas, are evaluated, truncated to the byte size, packed and assembled into sequential memory locations. If a byte cannot fit into a word, it is assembled as the first byte of the next word. The byte size may be altered in the middle of a word or a string by inserting a byte size expression in parentheses. The local radix for the size portion is always considered to be 10, no matter what value the prevailing radix may have. For example:

```
RADIX 10
AX: BYTE (6) 10, 4, 9, 1, 1, 3, 6
Q:  BYTE (15) 12, 3, 9,
STR: BYTE (6) 10, 4, 9 (12) "AB"
, EQUIVALENT TO
AX:  OCT 120411010103, 0600000000000;
Q:    OCT 000140000300, 000110000000;
STR:  OCT 120411030200;
```

ASSEMBLER CONTROL CODES

These statements do not generate data or instructions, but control the operation of the assembler.

REPEAT

This code causes a character string to be processed repeatedly. The code is followed by an expression whose value indicates the number of repetitions desired. This is followed by the string to be repeated enclosed by angle-brackets. The expression for the number of repetitions in a REPEAT statement must be followed by a comma. For example:

```

ADDX: REPEAT 3, <ADD 6, X(4)
      ADDI 4, 1>
, EQUIVALENT TO
ADDX: ADD 6, X(4)
      ADDI 4, 1
      ADD 6, X(4)
      ADDI 4, 1
      ADD 6, X(4)
      ADDI 4, 1
.....
SQ: REPEAT N, <
    EXP <.-SQ+1>*<.-SQ+1>>
;A TABLE OF SQUARES

```

The label of a repeat is placed on the first statement generated. REPEATs may be nested to any reasonable degree. For example:

```

REPEAT N+1, <MOVE 6, A(K)
REPEAT N, <ADD 6, (3)
          ADDI 3, L>
          MOVEM 6, A(K) >

```

IFn

(conditional assembly)—An IFn code is followed by an expression, and a string of coding enclosed in angle-brackets. The expression for a conditional assembly must be followed by a comma. If the expression fulfills the condition indicated by n, the string is processed; if not, it is ignored. The IFn codes are:

- IFE Assemble if expression is \emptyset .
- IFG Assemble if expression is positive.
- IFGE Assemble if expression is positive or \emptyset .
- IFL Assemble if expression is negative.
- IFLE Assemble if expression is negative or \emptyset .
- IFN Assemble if expression is nonzero.

IF1 Assemble if PASS 1 (no expression).

IF2 Assemble if PASS 2 (no expression).
For example:

```
IFE X-Y, <ADD Z, X;>  
;ASSEMBLED ONLY IF X=Y
```

IFIDN

(conditional assembly on character strings)—This is followed by three sets of angle-brackets. If the character strings enclosed by the first two sets of angle-brackets are identical, the coding within the third set is processed. For example:

```
IFIDN <+> <+>, <FAD 3,X>  
;FAD 3,X; WILL BE PROCESSED
```

IFDIF

(conditional assembly on character strings)—This is the converse of IFIDN and is similar in format. The coding within the third set of angle-brackets is processed if the two character strings differ.

RADIX

The prevailing value of the radix is controlled by this code. It is followed by a decimal number between 2 and 10 which then becomes the prevailing radix. The radix may be changed at any point on the assembly; it is initially considered to be 8. For example:

```
RADIX 10,  
;SET PREVAILING RADIX DECIMAL
```

LOC

This code changes the location counter to a value equivalent to the expression which follows. The block

of coding following a LOC is assembled into the absolute locations, and any labels defined are considered absolute. For example:

```
ADD AC2,X
LOC 200
ADD AC3,@Q2
LOC .+3; SKIP 3 LOCATIONS
ADD AC1,AC2
```

RELOC

This is similar to LOC in that it explicitly sets the location counter. The block of code which follows is relocatable and all labels within the block are relocatable. The implicit statement begins all programs. For example:

```
RELOC 0;
```

PHASE

A portion of code may be moved into other registers before it is executed. PHASE gives the location counter a value different from the location into which the assembled code is to be loaded. The code is actually loaded into continuing sequential locations, but all labels within a phased area are in relation to the PHASE. Point elements (.) also relate to the PHASE. PHASE is followed by an expression indicating the first address of the subroutine when it is to be executed. For example:

If the macro were called with the argument string:

```
(ARG,) or ARG,,
```

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 substitute the null string for each appearance of the second dummy argument in the code. However, the third argument is missing. A label would be created for each occurrence of %C. For example:

```
DEFINE TYPE (A,%B)  
<JSR TYPEOUT  
  JUMP %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 probably not be interested in knowing the label.

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.

Concatenation

The character single quote (') is defined to be 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 will form a complete symbol when joined to characters already contained in the macro definition. This joining, called concatenation, is indicated by the appearance of an apostrophe appearing 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
```

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 code, IRP. The code 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>
```

produces 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.

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

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

NOTE: Avoid using IRP or IRPC inside a repeat or conditional.

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

```
CONVERT (Ø,1,2,3,4,5,6,7)
```

will generate: _____

```
<IRP
IFE K-Ø,<STOPI
CONVI Ø>
IFE K-Ø,<STOPI
CONVI 1>
IFE K-2,<STOPI
CONVI 2>
IFE K-3,<STOPI
CONVI 3>
STOPI
CONVI 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, i.e., 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 CONVI 3, i.e., a call for the macro CONVI (defined elsewhere) with an argument of 3.

Nesting and Redefinition

Macros may be nested; that is, macros may be defined within other macros. For ease of discussions, 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; space 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. Henceforth, their level of definition is irrelevant and they may be called in the normal manner. Of course, if these second level macros contain third level macros, the third level macros are still 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, 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 0
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.

CHAPTER 4

RELOCATION AND LINKING

RELOCATION

The MACRO-6 assembler will create a relocatable 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 is added at load time by the Linking Loader and is equal to 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 probably used in address manipulation and must be modified; the second probably should not. To properly accomplish the relocation, the actual expression forming an address is considered and modification is decided. Integer elements are fixed and not modified. Point elements (.) are relocatable and are always modified.* Symbolic elements may be fixed or relocatable according to the means used in their definition. If a symbol is defined by direct assignment statement, it may be relocatable or fixed 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 must be considered. 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.

*Except under the LOC code which specified absolute addressing.

Consider the expression:

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

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

$$X_r = (A+k) + 2*(B+k) - 3*(C+k) + D+k$$

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

$$X_r = A + 2*B - 3*C + D + k$$

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

- $\emptyset * k$ 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 \emptyset 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:

1. Only two values of relocatability for a complete expression are allowed, k and \emptyset .
2. An element may not be divided by a relocatable element.
3. Two relocatable elements may not be multiplied together.
4. Relocatable elements may not be combined by Boolean expressions.

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 erroneous

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

XWD A, B

LINKING SUBROUTINES

Programs usually consist of subroutines which must be linked. This is relatively easy if all subroutines are assembled together; they can be linked by JSR SUBR instructions. If independently assembled, relocatable subroutines are used, linking must be considered since the symbol tables from the assembly are inaccessible to the loader.

To accomplish this linking, selected symbols are made available to the Linking Loader by the codes EXTERN, INTERN, and ENTRY.

The EXTERN code identifies certain symbols as external to the program. The condensed object program contains the information that values for certain symbols must be derived from other programs at load time. An expression containing a reference to an external symbol must consist of only the single external symbol. The statement

```
EXTERN SQRT, CUBE, TYPE;
```

identifies the symbols SQRT, CUBE and TYPE as external symbols. Symbols defined as external must not be defined as labels, variables, macros, or assignments.

An external reference may not occur within a literal, and may only appear as the address part of a machine command.

For example, if a square root is required, it would be called by

```
PUSHJ 1, SQRT:
```

Elsewhere in the program would be the statement

```
EXTERN SQRT;
```

NOTE: A transfer vector scheme is not used.

To make internal program symbols available to other subroutines as external symbols, the code INTERN or ENTRY is used. This code has no effect on the actual assembly of the subroutine, but will make a list of symbol equivalences available to other programs at loading time. The statement

```
INTERN SIN, COS, SIND, COSD;
```

might appear in a sin-cos routine where SIN, COS, SIND and COSD are entry points to the subroutine to calculate, respectively, sines and cosines of angles in radians and degrees. Internal symbols must be defined within the subroutine as assignments, labels, or variables.

In the square root subroutine would be the statement

```
INTERN SQRT;
```

Some subroutines have common usage, and it is convenient to place them in a library. To load these subroutines, the code ENTRY is used. 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 program. If the loader is in library search mode, a program will be loaded only if an undefined global symbol, i.e., any symbol made accessible to other programs, matches an internal symbol in the ENTRY list. If the SQRT routine mentioned above were a library program, the statement

```
ENTRY SQRT;
```

would also appear in the SQRT program.

CHAPTER 5

ERRORS

There are two classes of errors—errors in language usage and program errors. MACRO-6 will examine the statements for errors in language usage, and print appropriate messages. These errors are caused by meaningless or inconsistent construction in the source language. When a listing is prepared during the assembly, each MACRO-6 statement that contains errors will be flagged by one or several letters in the margin. At the end of the listing will be a summary of the errors; this summary will be printed even if a listing is not prepared. Program errors which properly use the MACRO-6 language will be correctly translated into errors in the binary program.

THE ERROR FLAGS

- M (multiply defined symbol)—A symbol is defined more than once, either as a label or variable. The symbol retains its original definition.
- S (symbol error)—There is a meaningless character string that resembles a symbol or macro. It is assembled as though the value were \emptyset .
- P (phase error)—A symbol is assigned a value as a label during PASS 2 different from that which it was assigned in PASS 1.
- O (undefined code)—The code indicating the statement type is not defined in the code table. It is assembled as a numeric code of \emptyset .
- N (number error)—There is a meaningless string of characters that resembles a number. It is assembled as \emptyset .

- A (argument error)—An argument of a control code has a peculiar value.
- L (literal)—There is an error within a literal.
- F (macro definition error)—A format error exists in a DEFINE statement.
- U (undefined symbol)—A symbol or macro is undefined. It is given a value of \emptyset .
- V (value previously undefined)—A symbol used to control the processor is undefined prior to the point at which it is first used.
- R (relocation error)—An expression has a relocation constant other than 1 or \emptyset , contains division by a relocatable number, contains the product of two relocatable numbers, or involves relocatable numbers in Boolean operations. The relocation constant is set to \emptyset .
- D (multiply defined symbol reference)—The statement contains a reference to a multiply defined symbol. It is assembled with the first value defined.
- E (external)—Improper usage of external symbols.

On PASS 1, an error printout consists of two lines. The first has the most recently used tag followed by + n where n is the (decimal) number of lines of coding between the tag and the error.

The second line, and the only line in PASS 2, is a copy of the erroneous line of code with a letter(s) indicating the error type(s) in the left-hand margin.

CHAPTER 6

ASSEMBLY OUTPUT

ASSEMBLY LISTING

There are two types of assembly output—the assembly listing and the binary program. The assembly listing consists of a printout of the source program. On the same line with each source statement are three numeric fields—the location of the assembled code, the left half word, and the right half word. Above each line containing an error is an appropriate message. This listing is controlled by the List Control Codes except that error messages are always printed. All assemblies begin with an implicit LIST. Apostrophes on the assembly listing indicate relocatability. The program break is printed at the end of the assembly—this is the highest relocatable location assembled plus one.

BINARY PROGRAM

The binary program may assume two forms: RIM and LINK. The RIM (read-in mode) format is always punched into paper tape and is usually used for loaders and computer hardware maintenance programs. RIM programs may be completely loaded by the loader resident in the shadow memory located behind the accumulator memory.

Rim Format

Programs in RIM mode consist of two word pairs. The first word is an instruction:

DATAI PTR, A,

The second word of the pair is the word of instruction or data to be loaded into memory location A.

The last word of a RIM tape is a single instruction:

HALT, START;

where START is the first location of the program.

LINK Format

LINK format is the normal binary output mode. Programs in this format are acceptable to the Linking Loader and are usually relocatable. The Linking Loader will load subprograms into memory, properly relocating each one and adjusting addresses to compensate for the relocation. It will also link external and internal symbols to provide communication between independently assembled subprograms. Finally, the Linking Loader will load subroutines in library search mode.

LINK format data is 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 programs (except those in paper tape RIM format) are stored in this format, including programs on paper tape, DECtape, standard magnetic tape, punched cards, drums and discs. This format is totally independent of logical divisions in the input medium (40-word check summed paper tape blocks, 128-word blocks on DECtape and drums, 23-word check summed punched cards, etc.). It is also independent of the block type.

The 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 200,000 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:	
2ND WORD	GLOBAL (INTERNAL) DEFINITION BITS 0-35 VALUE OF SYMBOL
CODE 10:	
2ND WORD	LOCAL DEFINITION BITS 0-35 VALUE OF SYMBOL
CODE 60:	
2ND WORD	CHAINED GLOBAL REQUESTS: BITS 0-17 = 0
2ND WORD	BITS 18-35 POINTER TO FIRST WORD OF CHAIN REQUIRING DEFINITION (See Loader Manual)
CODE 60:	
2ND WORD	GLOBAL SYMBOL ADDITIVE REQUEST: (See Loader Manual) 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 LH WITH RESULT IN STORAGE
BIT 10	REPLACE RH WITH RESULT IN STORAGE
BIT 11	REPLACE INDEX FIELD WITH RESULT IN STORAGE
BIT 12	REPLACE AC FIELD WITH RESULT IN STORAGE
BITS 18-35	POINTER TO WORD REQUIRING ADDITION

Block Type 4 Entry Block

THIS BLOCK CONTAINS A LIST OF RADIX 50 SYMBOLS, EACH OF WHICH MAY CONTAIN A ZERO OR ONE 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 LAST BLOCK OF THIS TYPE ENCOUNTERED BY THE LOADER IS USED UNLESS THE CONTROL CHARACTER (A) HAS BEEN TYPED. 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.

Radix 50 Representation

Radix 50 representation is used to condense 6 character 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

$$L_6 L_5 L_3 L_2 L_1$$

If C_n denotes the six bit code for L_n , the radix 50 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:

<u>Code (Octal)</u>	<u>Characters</u>
00	Space
01-12	0-9
13-44	A-Z
45	.
46	\$
47	%

CHAPTER 7

ASSEMBLER INITIALIZATION

At the beginning of each assembly, the assembler is initialized to certain states affected by control codes. The initial states are:

1. Radix is set to 8.
2. The location counter is set to 0 and relocatable assembly will occur.
3. There will be a normal listing.
4. There will be LINK binary output with a symbol table.
5. Phase mode is off.
6. The title and subtitle are blanked.
7. Only device mnemonics are placed in the symbol table. They are:

CPA	=	000	Arithmetic Processor
PRS	=	004	Priority Interrupt System
PTP	=	100	Paper Tape Punch
PTR	=	104	Paper Tape Reader
CP	=	110	Card Punch
CR	=	114	Card Reader
TTY	=	120	Console Teleprinter
LPT	=	124	Line Printer
DI	=	130	Display
DC	=	200	Data Control
UT	=	210	Micro Tape Control
UTS	=	214	Micro Tape Status
MTC	=	220	Mag Tape Control
MTS	=	224	Mag Tape Status
MTM	=	230	Mag Tape Status
DCSA	=	300	Data Communication System
DCSB	=	304	Data Communication System
DRUM	=	400	Drum System

8. No macros or opdefs exist.

APPENDIX 1

CODES

DATA GENERATING CODES

DEC	Decimal numbers
OCT	Octal numbers
EXP	Expressions
XWD	Block transfer word
IOWD	Input/output transfer word
POINT	Pointer word
SIXBIT	ASCII (6-bit) character strings
BYTE	Variable length bytes
BLOCK	Block of storage reserved
ASCII	ASCII (7-bit) character strings
ASCIZ	Same as ASCII, but assembles a word of zeros if the number of ASCII characters assembled is a multiple of five.

PROCESSOR CONTROL CODES

REPEAT	Repeat character string
IFn	Conditional assembly

<u>n</u>	<u>Condition</u>
E	zero
G	positive
GE	zero or positive
L	negative
LE	zero or negative
N	non zero
B	blank
1	pass 1
2	pass 2

PHASE	Enter phase mode
DEPHASE	Leave phase mode
RIM	Assemble RIM tapes
IFIDN	Conditional assembly on character strings
IFDIF	Conditional assembly on character strings
RADIX	Radix control
LOC	Set location counter
PASS2	Terminate PASS 1
NOSYM	Suppress symbol table output
LIT	Assemble literals
VAR	Assemble variables
EXTERN	List of external symbols
INTERN	List of internal symbols
IRP	Indefinite repeat
PURGE	Purge symbols
TAPE	End of a physical tape
END	Last line

LIST CONTROL

LIST	List
XLIST	Stop listing
LALL	Expanded listing
XALL	Stop expanded listing
TITLE	Title
SUBTTL	Subtitle
PAGE	Skip to top of next page

APPENDIX 2

SUMMARY OF ERR FLAGS

A	Argument of control op
D	Reference to multiply defined symbol
E	Illegal use of an external
F	Macro definition
L	Usage of literal
M	Multiply defined symbol
N	Number
O	Undefined operation code
P	Phase discrepancy
R	Relocation
U	Undefined symbol
V	Value previously undefined
X	Macro definition error

APPENDIX 3

PROGRAMMING EXAMPLES

FLOATING POINT LOG (BASE E) SUBROUTINE

```

LOG:  MOVMS      A           ;GET ABSF(X)
      JUMPLE    A,L         ;RETURN 0 FOR LOG(0) OR LOG(-0)
      ASHC     A,-33        ;SEPARATE FRACTION FROM EXPONENT
      ADDI     A,211000     ;FLOAT THE EXPONENT AND MULTIPLY BY 2
      MOVSM    A,LS        ;NUMBER NOW IN CORRECT FLOATING FORMAT
      MOVSI    A,567377    ;SET UP -401.0 IN A
      FADM     A,LS        ;SUBTRACT 401 FROM THE EXPONENT*2
      ASH      B,-10       ;SHIFT FRACTION PART FOR FLOAT
      TLC      B,2000000   ;FLOAT THE FRACTION PART
      FAD      B,L1        ;B=B-SQRTF(2.0)/2.0
      MOVE     A,B         ;A=B
      FAD      A,L2        ;A=A+SQRTF(2.0)/2.0
      FDV      B,A         ;B=B/A
      MOVEM    B,LZ        ;STORE NEW VARIABLE IN LZ
      FMP      B,B         ;CALCULATE Z12
      MOVE     A,L3        ;PICK UP FIRST CONSTANT
      FMP      A,B         ;MULTIPLY BY Z12
      FAD      A,L4        ;ADD IN NEXT CONSTANT
      FMP      A,B         ;MULTIPLY BY Z12
      FAD      A,L5        ;ADD IN LAST CONSTANT
      FMP      A,LZ        ;MULTIPLY BY Z
      FAD      A,LS        ;ADD IN EXPONENT TO FROM LOG BASE 2
      FMP      A,L7        ;MULTIPLY BY LOG(2), BASE E
L:    POPJ     P,          ;EXIT

L1:   577225754146        ;-.707106781187
L2:   201552023632        ; 1.414213562374
L3:   200462532522        ; 0.5989786496
L4:   200754213604        ; 0.9614706323
L5:   202561251006        ; 2.8853912903
L7:   200542710300        ; 0.69314718056

LS:   0
LZ:   0

      A=i7
      B=0
      P=1

      ENTRY    LOG
      END

```

FLOATING POINT SQUARE ROOT FUNCTION

;ARGUMENT IS WRITTEN IN THE FORM $X=F*2^{**2}B$
 ;SQRT(X) IS THEN $SQRT(F)*2^{**}B$
 ;SQRT(F) IS CALCULATED BY A LINEAR APPROXIMATION
 ;SQRT(F) IS CALCULATED BY A LINEAR APPROXIMATION
 ;THE NATURE OF WHICH DEPENDS ON WHETHER $1/4 < F < 1/2$
 ;OR $1/2 < F < 1$, FOLLOWED BY 2 ITERATIONS OF NEWTON'S METHOD.

SQRT:	MOVMS	A	;GET ABSOLUTE VALUE OF ARG
	JUMPLE	A, SQ2	;EXIT IF $X=0$
	ASHC	A, -33	;PUT EXPONENT IN A, FRACTION IN B
	SUBI	A, 201	;SUBTRACT 201 FROM EXPONENT
	ROT	A, -1	;CUT EXPONENT IN HALF, SAVE ODD BIT
	HRRM	A, SQ1	;SAVE FOR FUTURE SCALING OF ANSWER
	LSH	A, -43	;GET BIT SAVED BY PREVIOUS INSTRUCTION
	ASH	B, -10	;PUT FRACTION IN PROPER POSITION
	FSC	B, 177(A)	;PUT EXPONENT OF FRAC TO EITHER 0 OR 1
	MOVEM	B, ST	;SAVE IT. $1/4 < \text{FRAC} < 1$
	FMP	B, S1(A)	;LINEAR FIRST APPROX, DEPENDING ON
	FAD	B, S2(A)	;WHETHER $1/4 < F < 1/2$ OR $1/2 < F < 1$
	MOVE	A, ST	;START NEWTON'S METHOD WITH FRAC
	FDV	A, B	;CALCULATE $X(0)/X(1)$
	FAD	B, A	; $X(1)+X(0)/X(1)$
	FSC	B, -1	; $1/2(X(1)+X(0)/X(1))$
	MOVE	A, ST	;SECOND ITERATION NEWTON'S METHOD
	FDV	A, B	; $X(0)/X(2)$
	FADR	A, B	; $X(2)+X(0)/X(2)$
SQ1:	FSC	A, 0	;SCALE ANSWER FOR NEWTON AND EXPONENT
SQ2:	POPJ	P,	;EXIT

S1:	0.8125	;CONSTANT, USED IF $1/4 < \text{FRAC} < 1/2$
	0.578125	;CONSTANT, USED IF $1/2 < \text{FRAC} < 1$
S2:	0.302734	;CONSTANT, USED IF $1/4 < \text{FRAC} < 1/2$
	0.421875	;CONSTANT, USED IF $1/2 < \text{FRAC} < 1$

ST: 0
 A=17
 P=1
 B=0

ENTRY SQRT

END

FLOATING POINT NUMBER TO A FIXED POINT POWER

;ROUTINE CALCULATES A**B, A FLOATING POINT
;B IS OF THE FOLLOWING FORM:
;B=A(0)+A(1)*2+A(2)*4+..., WHERE A(I)=0 OR 1.
;ANSWER MULTIPLIED BY A**I IF A(I)=1
;THEN B IS SHIFTED TO GET NEXT BIT.

```
EXP.2:   JUMPE      A,FEXP4      ;ZERO BASE, RETURN
         MOVSI     T,201400      ;PUT 1.0 IN ACC. T
         JUMPGE    B,FEXP2      ;CHECK SIGN OF EXPONENT
         MOVMS     B              ;NEGATE EXPONENT - SET TO POSITIVE
         PUSHJ    P,FEXP2      ;DO CALCULATION
         MOVSI     T,201400      ;GET 1.0 IN T
         FDVM      T,A          ;FORM 1/A**B
         POPJ     P,            ;EXIT

FEXP1:   FMP       A,A          ;FORM A**N, FLOATING POINT
         LSH      B,-1         ;SHIFT EXPONENT FOR NEXT BIT
FEXP2:   TRZE      B,1         ;IS BIT A ZERO?
         FMP      T,A          ;NO, MULTIPLY ANSWER BY A**N
         JUMPN    B,FEXP1      ;UPDATE A**N UNLESS ALL THROUGH
         MOVE     A,T          ;PICK UP RESULT FROM T
FEXP4:   POPJ     P,            ;EXIT

T=0
P=1
A=17
B=16

ENTRY   EXP.2

END
```

APPENDIX 4

CHARACTER SETS

	ASCII	6 bit ASCII	Punched Card		ASCII	6 bit ASCII	Punched Card
(space)	240	00	b	@	300	40	4-8
!	241	01	12-7-8	A	301	41	12-1
"	242	02	0-5-8	B	302	42	12-2
#	243	03	0-6-8	C	303	43	12-3
\$	244	04	11-3-8	D	304	44	12-4
%	245	05	0-7-8	E	305	45	12-5
&	246	06	11-7-8	F	306	46	12-6
'	247	07	6-8	G	307	47	12-7
(250	10	0-4-8	H	310	50	12-8
)	251	11	12-4-8	I	311	51	12-9
*	252	12	11-4-8	J	312	52	11-1
+	253	13	12	K	313	53	11-2
,	254	14	0-3-8	L	314	54	11-3
-	255	15	11	M	315	55	11-4
.	256	16	12-3-8	N	316	56	11-5
/	257	17	0-1	O	317	57	11-6
∅	260	20	∅	P	320	60	11-7
1	261	21	1	Q	321	61	11-8
2	262	22	2	R	322	62	11-9
3	263	23	3	S	323	63	0-2
4	264	24	4	T	324	64	0-3
5	265	25	5	U	325	65	0-4
6	266	26	6	V	326	66	0-5
7	267	27	7	W	327	67	0-6
8	270	30	8	X	330	70	0-7
9	271	31	9	Y	331	71	0-8
:	272	32	11-0	Z	332	72	0-9
;	273	33	0-2-8	[333	73	11-5-8
<	274	34	12-6-8	\	334	74	7-8
=	275	35	3-8]	335	75	12-5-8
>	276	36	11-6-8	↑	336	76	5-8
?	277	37	12-0				

digital
EQUIPMENT
CORPORATION
MAYNARD, MASSACHUSETTS