DEC-6-0-TP-MAC-LM-FP-ACT01

PROGRAMMING MANUAL

MACRO-6 ASSEMBLY LANGUAGE

PDP-6

DIGITAL EQUIPMENT CORPORATION • MAYNARD, MASSACHUSETTS

PDP-6 PROGRAMMING MANUAL
MACRO-6 ASSEMBLY LANGUAGE

DIGITAL EQUIPMENT CORPORATION • MAYNARD, MASSACHUSETTS

CONTENTS

CONTENTS (continued)

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 1Ø. 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 1Ø were prevailing, 17 would have the value $17_{10}=21_8$. The number 17.Ø would always have the value 2Ø542ØØØØØØØ 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 t 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:

```
↑D17 = 17₁₀
↑O17 = 15₁₀
↑B1Ø1Ø = 1Ø₁₀ = 12₈
```

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±n; the number is then considered to be multiplied by 1Ø±n.

```
1.
1Ø.ØE-1
Ø.ØØØ1E4      } The binary representation of each is 2Ø14ØØØØØØØØ
.ØØ1E+3
```

If, in addition, the characters ↑F are prefixed to a number, it is considered to be a fixed decimal fraction. In this case, Bn should be suffixed to the number where n is an integer and $Ø \leq n \leq 35$. 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.

```
↑F3.25B8 =              ØØ34ØØØØØØØ
↑F.281250B12 =          ØØØØ3ØØØØØØ
↑F.4498Ø46E+1B11        ØØØ4377ØØØØ
```

A number may also be logically shifted left by following it by Br. 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:

```
Ø3B35 = ØØØØØØØØØØ3
Ø3B31 = ØØØØØØØØ6Ø
Ø3B17 = ØØØØØ3ØØØØØ
```

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 4Ø661Ø5
(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
```

EXPRESSIONS

Expressions are strings of elements separated by arithmetic or Boolean operators. Expressions represent numeric values less than 2^{35} in magnitude. The value of an expression is calculated by substituting the numeric values for each of the elements and performing the specified operations. The allowable operations are:

Operator	Meaning
*	multiply
/	divide
+	add
-	substract
&	and
!	inclusive or

When combining elements, the Boolean operations (AND, IOR) are performed first, from left to right. Then the multiplications are performed from left to right; and, finally, the additions and subtractions are performed. Division always truncates the fraction part. All arithmetic is performed modulo $2^{35} = 34, 359, 738, 368$.

For example, suppose the element:

A represents the value 2_{10}

B represents the value 8_{10}

C represents the value 3_{10}

D represents the value 5_{10}, the expression:

A/B + A*C represents 6_{10}

B/A - 2*A-1 represents $-1_{10} = 777777777777_8$

A&B represents Ø

B+D1Ø+C represents 21_{10}

1 + A&C represents 3_{10}

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

<A+B> /5 represents 2

C* <A+B* <D-C ≫ 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:

```
SYM = EXP,      COMMENT
```

where SYM is a symbol and EXP is an expression. For example:

```
A = B+2
TSIZE = TEND-TSTART
K = 4
```

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 delimitor, 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 Ø. 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:  DATAI PTR, @NUM(4)
       CONO 2Ø3;  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:

```
ØA;  THE VALUE OF A IS IN THE RT HALF
27ØB8 2,X;  EQUIVALENT TO ADD 2,X
tDØ5;  ØØØØØØØØØ1Ø1 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.ØE-26, 3;
;5 WORDS GENERATED
```

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

```
EXP X, 4, tD65, 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 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 44_8. 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 41146Ø5Ø4142, 456451430Ø051;
        OCT 564657625541, 645157560ØØØ;
```

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
```

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 1Ø, no matter what value the prevailing radix may have. For example:

```
RADIX 1Ø
AX:  BYTE (6) 1Ø, 4, 9, 1, 1, 3, 6
Q:   BYTE (15) 12, 3, 9,
STR: BYTE (6) 1Ø, 4, 9 (12) "AB"
,EQUIVALENT TO
AX:   OCT 12Ø411Ø1Ø1Ø3, Ø6ØØØØØØØØØØ;
Q:    OCT ØØØ4ØØØØØ3ØØ, ØØØ11ØØØØØØØ;
STR:  OCT 12Ø411ØØ4142;
```

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 repititions 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*SQ+1-2*.*SQ+2*. -2*SQ>
;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 Ø.
IFG	Assemble if expression is positive.
IFGE	Assemble if expression is positive or Ø.
IFL	Assemble if expression is negative.
IFLE	Assemble if expression is negative or Ø.
IFN	Assemble if expression is nonzero.

IF1 Assemble if PASS 1 (no expression).

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

```
IF    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 1Ø,
;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 2ØØ
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 Ø;
```

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:

```
        MOVE [XWD LOOPX,LOOP]
        BLT LOOP+5
         :
LOOP:   PHASE 11
LOOP:   Z
        MOVN A(X)
        FMP MPYR
        FADM A(Y)
        SOJGE X, .-3
        JRST @LOOP
DEPHASE
```

This example is the central loop in a matrix inversion. Before executing it, the routine will be moved into fast accumulator memory locations 11-16. The symbol LOOP represents 11 and the point in the SOJGE instruction represents 15. The routine is, however, loaded into the normal sequential registers. A phased area is terminated by a DEPHASE, LOC or RELOC code. The DEPHASE code has no effect on the next sequential loading location, but restores the location counter to this value.

PASS2 — This code causes the location assignment phase, PASS1 to be suspended and PASS2 to commence.

END — This statement must be the last statement in a program or subroutine. If it is a program, the following expression is the location of the first instruction to be executed.

NOSYM — The assembler will normally output a symbol table or list of the symbols used and their definitions. The code NOSYM will suppress this.

LIT — This code will cause literals that have been previously defined to be assembled starting at the current location.

If n literals have been defined, the next free cell will be at (.+n). This statement will have no effect on literals which are defined after it. LIT may not be used more than eight times.

VAR — This code will cause the symbols which have been defined by following them with # in previous statements to be assembled as block statements. This has no effect on subsequent symbol definitions of the same type. This, and the previous pseudo-op, LIT, are useful in controlling storage allocation. If these codes do not appear, all variables and literals are placed at the end of the program.

RIM — In paper tape assemblies, this code will cause binary output to be punched in RIM format.

OPDEF — (define an operation mnemonic)—This is followed by a symbol and a pair of brackets containing a statement that will generate one word of data. The symbol then becomes a mnemonic for the operation code represented by the 36-bit data word. For example:

```
OPDEF PUSHP [PUSH PP,Ø]
OPDEF PUNCH [DATAO PIP,]
```

These OPDEFs may then be treated as ordinary op codes. For example:

```
PUSHP X
PUNCH Y
```

SYN — (define synonyms)—This code is followed by two symbols or macros. The first must have been previously defined, and the second is made equivalent to the first. If the first is a symbol, the second becomes a symbol with the same value; if the first is a macro, the second becomes a macro which acts identically; if the first is a machine-op, control code, or data generating code, the second will be interpreted in the same manner. For example:

```
SYN K,X
SYN FAD,ADD
SYN END,XEND
```

If the first item is identical to both a symbol and a code, the second item (which is the synonym) is made synonymous with the symbol in preference to the code.

Listing Control

Several codes are used to control the final listing.

LIST — Causes the assembler to begin listing the assembled program in both octal and source text.

XLIST — Causes the assembler to stop listing the assembled program.

LALL — Causes the assembler to list all text that is processed: macro expansions, list control codes, repeats, etc.

XALL — Causes the assembler to stop listing all text.

TITLE — The comments field is written at the top of each printed page.

SUBTTL	The comments field is written as a second line at the top of each printed page.
PAGE	The listing begins a new page. (A form feed on the input tape also has the same effect.)

These list control codes are never printed in the final listings, except under LALL.

CHAPTER 3

MACROS

When writing a program, it often happens that certain coding sequences are used several times with just 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. A single statement referring to the macro by name, along with a list of real arguments, will generate the correct sequence.

DEFINITION OF MACROS

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

The character sequence, which constitutes the body of the macro, is delimited by angle brackets. The body of the macro may consist of any proper string of coding; normally, but is not restricted to, a group of complete statements.

Example: A macro to compute the length of a vector.

```
DEFINE VMAG (A,B)       ROUTINE FOR THE LENGTH OF A VECTOR
<MOVE 0,A;              GET THE FIRST COMPONENT
 FMP 0;                 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>
```

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 within 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 definition 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:

```
VMAG VECT,LENGTH
```

The appearance of this statement in a program would generate the code sequence defined above for the macro VMAG. The character string VECT would be substituted for each occurrence in the coding of the dummy argument A, the character string LENGTH being 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.

Additional Considerations

1. 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 ADD 2,X; then the calling macro instruction would be written:

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

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

The general rule is: If an argument contains commas, semicolons, or any other argument delimiters, the argument must be enclosed in angle brackets.

2. 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.

3. 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 will exactly replace 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 cells 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)
```

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 first created symbol is ..ØØØ1, the next is ..ØØØ2, etc.

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

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

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

If the macro were called with the argument string:

```
(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 if a macro can be repeated 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.

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
CONV1 A>
≫
```

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
        CONV1 Ø>
        IFE K-Ø,<STOPI
        CONV1 1>
        IFE K-2,<STOPI
        CONV1 2>
        IFE K-3,<STOPI
        CONV1 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, 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 CONV1 3, i.e., a call for the macro CONV1 (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:     HLRZ 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 @(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. Most programs begin in location 60_8; if a program is loaded into cells beginning at location 1400_8, the relocation constant K would be 1320_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

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 Ø 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 Ø.
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

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;
```

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 Ø.
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 Ø.
N	(number error)—There is a meaningless string of characters that resembles a number. It is assembled as Ø.

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 Ø.
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 Ø, 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 Ø.
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:

Ø	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 2ØØ,ØØØ OCTAL)

Block Type 2 Symbols

CONSISTS OF WORD PAIRS

1ST WORD	BITS Ø-3 CODE BITS
1ST WORD	BITS 4-35 RADIX 5Ø REPRESENTATION OF SYMBOL (See Below)
2ND WORD	DATA (VALUE OR POINTER)
CODE Ø4:	GLOBAL (INTERNAL) DEFINITION
2ND WORD	BITS Ø-35 VALUE OF SYMBOL
CODE 1Ø:	LOCAL DEFINITION
2ND WORD	BITS Ø-35 VALUE OF SYMBOL
CODE 6Ø:	CHAINED GLOBAL REQUESTS:
2ND WORD	BITS Ø-17 = Ø
2ND WORD	BITS 18-35 POINTER TO FIRST WORD OF CHAIN REQUIRING DEFINITION (See Loader Manual)
CODE 6Ø:	GLOBAL SYMBOL ADDITIVE REQUEST: (See Loader Manual)
2ND WORD	BIT Ø = 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 5Ø 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 5Ø). 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 1Ø 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 5Ø Representation

Radix 5Ø 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

$$L_6L_5L_3L_2L_1$$

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:

Code (Octal)	Characters
00	Space
01-12	0-9
13-44	A-Z
45	.
45	$
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	= ØØØ	Arithmetic Processor
PRS	= ØØ4	Priority Interrupt System
PTP	= 1ØØ	Paper Tape Punch
PTR	= 1Ø4	Paper Tape Reader
CP	= 11Ø	Card Punch
CR	= 114	Card Reader
TTY	= 12Ø	Console Teleprinter
LPT	= 124	Line Printer
DI	= 13Ø	Display
DC	= 2ØØ	Data Control
UT	= 21Ø	Micro Tape Control
UTS	= 214	Micro Tape Status
MTC	= 22Ø	Mag Tape Control
MTS	= 224	Mag Tape Status
MTM	= 23Ø	Mag Tape Status
DCSA	= 3ØØ	Data Communication System
DCSB	= 3Ø4	Data Communication System
DRUM	= 4ØØ	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

PROCESSOR CONTROL CODES

REPEAT	Repeat character string
IFn	Conditional assembly

n	Condition
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

OPDEF	Define an op mnemonic
SYM	Define a synonym

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 Ø FOR LOG(Ø) OR LOG(-Ø)
        ASHC    A,-33   ;SEPARATE FRACTION FROM EXPONENT
        ADDI    A,211ØØØ ;FLOAT THE EXPONENT AND MULTIPLY BY 2
        MOVSM   A,LS    ;NUMBER NOW IN CORRECT FLOATING FORMAT
        MOVSI   A,567377 ;SET UP -4Ø1.Ø IN A
        FADM    A,LS    ;SUBTRACT 4Ø1 FROM THE EXPONENT*2
        ASH     B,-1Ø   ;SHIFT FRACTION PART FOR FLOAT
        TLC     B,2ØØØØØ ;FLOAT THE FRACTION PART
        FAD     B,L1    ;B=B-SQRTF(2.Ø)/2.Ø
        MOVE    A,B     ;A=B
        FAD     A,L2    ;A=A+SQRTF(2.Ø)/2.Ø
        FDV     B,A     ;B=B/A
        MOVEM   B,LZ    ;STORE NEW VARIABLE IN LZ
        FMP     B,B     ;CALCULATE Z↑2
        MOVE    A,L3    ;PICK UP FIRST CONSTANT
        FMP     A,B     ;MULTIPLY BY Z↑2
        FAD     A,L4    ;ADD IN NEXT CONSTANT
        FMP     A,B     ;MULTIPLY BY Z↑2
        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    ;-Ø.7Ø71Ø6781187
L2:     2Ø1552Ø23632    ; 1.414213562374
L3:     2ØØ462532522    ; Ø.5989786496
L4:     2ØØ7542136Ø4    ; Ø.9614706323
L5:     2Ø2561251ØØ6    ; 2.8853912903
L7:     2ØØ54271Ø3ØØ    ; Ø.6931471805 6

LS:     Ø
LZ:     Ø

        A=17
        B=Ø
        P=1

        ENTRY   LOG

        END
```

FLOATING POINT SQUARE ROOT FUNCTION

```
;ARGUMENT IS WRITTEN IN THE FORM X=F*2**2B
;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=Ø
        ASHC    A,-33       ;PUT EXPONENT IN A, FRACTION IN B
        SUBI    A,2Ø1       ;SUBTRACT 2Ø1 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,-1Ø       ;PUT FRACTION IN PROPER POSITION
        FSC     B,177(A)    ;PUT EXPONENT OF FRAC TO EITHER Ø OR 1
        MOVEM   B,ST        ;SAVE IT. 1/4<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(Ø)/X(1)
        FAD     B,A         ;X(1)+X(Ø)/X(1)
        FSC     B,-1        ;1/2(X(1)+X(Ø)/X(1))
        MOVE    A,ST        ;SECOND ITERATION NEWTON's METHOD
        FDV     A,B         ;X(Ø)/X2)
        FADR    A,B         ;X(2)+X(Ø)/X(2)
SQ1:    FSC     A,Ø         ;SCALE ANSWER FOR NEWTON AND EXPONENT
SQ2:    POPJ    P,          ;EXIT

S1:     Ø.8125              ;CONSTANT, USED IF 1/4<FRAC<1/2
        Ø.578125            ;CONSTANT, USED IF 1/2<FRAC<1
S2:     Ø.3Ø2734            ;CONSTANT, USED IF 1/4 <FRAC<1/2
        Ø.421875            ;CONSTANT, USED IF 1/2<FRAC<1

ST:     Ø

        A=17
        P=1
        B=Ø

        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(Ø)+A(1)*2+A(2)*4+..., WHERE A(I)=Ø 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,2Ø14ØØ    ;PUT 1.Ø IN ACC. T
        JUMPGE  B,FEXP2     ;CHECK SIGN OF EXPONENT
        MOVMS   B           ;NEGATE EXPONENT - SET TO POSITIVE
        PUSHJ   P,FEXP2     ;DO CALCULATION
        MOVSI   T,2Ø14ØØ    ;GET 1.Ø 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=Ø
        P=1
        A=17
        B=16

        ENTRY   EXP.2

        END
```

APPENDIX 4

CHARACTER SETS

	ASCII	6 bit ASCII	Punched Card
(space)	240	00	b
!	241	01	12-7-8
"	242	02	0-5-8
#	243	03	0-6-8
$	244	04	11-3-8
%	245	05	0-7-8
&	246	06	11-7-8
'	247	07	6-8
(	250	10	0-4-8
)	251	11	12-4-8
*	252	12	11-4-8
+	253	13	12
,	254	14	0-3-8
-	255	15	11
.	256	16	12-3-8
/	257	17	0-1
ø	260	20	ø
1	261	21	1
2	262	22	2
3	263	23	3
4	264	24	4
5	265	25	5
6	266	26	6
7	267	27	7
8	270	30	8
9	271	31	9
:	272	32	11-0
;	273	33	0-2-8
<	274	34	12-6-8
=	275	35	3-8
>	276	36	11-6-8
?	277	37	12-0

	ASCII	6 bit ASCII	Punched Card
@	300	40	4-8
A	301	41	12-1
B	302	42	12-2
C	303	43	12-3
D	304	44	12-4
E	305	45	12-5
F	306	46	12-6
G	307	47	12-7
H	310	50	12-8
I	311	51	12-9
J	312	52	11-1
K	313	53	11-2
L	314	54	11-3
M	315	55	11-4
N	316	56	11-5
O	317	57	11-6
P	320	60	11-7
Q	321	61	11-8
R	322	62	11-9
S	323	63	0-2
T	324	64	0-3
U	325	65	0-4
V	326	66	0-5
W	327	67	0-6
X	330	70	0-7
Y	331	71	0-8
Z	332	72	0-9
[	333	73	11-5-8
\	334	74	7-8
]	335	75	12-5-8
↑	336	76	5-8

5244 PRINTED IN U.S.A. 10-2/65