Xerox Meta-Symbol

Sigma 5-9 Computers

Language and Operations
Reference Manual

META-SYMBOL DIRECTIVES

			Page No.
[label$_1$, . . . , label$_n$]	ASECT		26
	BOUND	boundary	25
	CDISP	symbol$_1$[, . . . , symbol$_n$]	62
	CLOSE	[symbol$_1$, . . . , symbol$_n$]	44
label$_1$[, . . . , label$_n$]	CNAME[,n]	[list]	59
label$_1$[, . . . , label$_n$]	COM[,field list]	[value list]	51
[label$_1$, . . . , label$_n$]	CSECT	[expression]	26
[label$_1$, . . . , label$_n$]	DATA[,f]	['value$_1$, . . . , value$_n$]	53
	DEF	[symbol$_1$, . . . , symbol$_n$]	46
	DISP	[list]	57
[label$_1$, . . . , label$_n$]	DO	[expression]	37
[label$_1$, . . . , label$_n$]	DO1	[expression]	34
label	DSECT	[expression]	26
	ELSE		35, 37
[label$_1$, . . . , label$_n$]	END	[expression]	34
[label$_1$, . . . , label$_n$]	EQU[,s]	[list]	42
	ERROR[,level[,c]]	'cs$_1$'[, . . . ,'cs$_n$']	57
	FDISP	symbol$_1$[, . . . , symbol$_n$]	62
	FIN		35, 37
label$_1$[, . . . , label$_n$]	FNAME	[list]	59
[label$_1$, . . . , label$_n$]	GEN[,field list]	[value$_1$, . . . , value$_n$]	50
	GOTO[,k]	label$_1$[, . . . , label$_n$]	34
	LIST	[expression]	56
[label$_1$, . . . , label$_n$]	LOC [1, 2, 4, 8]	[location]	25
	LOCAL	[symbol$_1$, . . . , symbol$_n$]	43
	OPEN	[symbol$_1$, . . . , symbol$_n$]	44
[label$_1$, . . . , label$_n$]	ORG [1, 2, 4, 8]	[location]	24
	PAGE		58
	PCC	[expression]	56
	PEND	[list]	60
	PROC		60
[label$_1$, . . . , label$_n$]	PSECT	[expression]	27
	PSR	[expression]	56
	PSYS	[expression]	57
	REF[,n]	[symbol$_1$, . . . , symbol$_n$]	48
[label$_1$, . . . , label$_n$]	RES[,n]	[expression]	26
[label$_1$, . . . , label$_n$]	SET[,s]	[list]	43
	S:RELP		62
label$_1$[, . . . , label$_n$]	S:SIN,n	[expression]	53
	SOCW		55
	SPACE	[expression]	55
	SREF[,n]	[symbol$_1$, . . . , symbol$_n$]	48
	SYSTEM	name	33
[label$_1$, . . . , label$_n$]	TEXT	'cs$_1$'[, . . . ,'cs$_n$']	54
[label$_1$, . . . , label$_n$]	TEXTC	'cs$_1$'[, . . . ,'cs$_n$']	55
	TITLE	['cs']	56
[label$_1$, . . . , label$_n$]	USECT	name	27
[label$_1$, . . . , label$_n$]	WHILE	[expression]	35

XEROX

Xerox Meta-Symbol

Sigma 5-9 Computers

Language and Operations

Reference Manual

90 09 52G

October 1975

File No.: 1X23
XG48, Rev. 0
Printed in U.S.A.

REVISION

This edition of the Xerox Meta-Symbol/LN, OPS Reference Manual, Publication Number 90 09 52G, merely incorporates the 90 09 52F-1 revision package into the manual. There are no other technical changes. The manual documents the H01 version of the Meta-Symbol.

RELATED PUBLICATIONS

Title	Publication No.
Xerox Sigma 5 Computer/Reference Manual	90 09 59
Xerox Sigma 6 Computer/Reference Manual	90 17 13
Xerox Sigma 7 Computer/Reference Manual	90 09 50
Xerox Sigma 8 Computer/Reference Manual	90 17 49
Xerox Sigma 9 Computer/Reference Manual	90 17 33
Xerox Batch Processing Monitor (BPM)/BP, RT Reference Manual	90 09 54
Xerox Universal Time-Sharing System (UTS)/TS Reference Manual	90 09 07

Manual Content Codes: BP – batch processing, LN – language, OPS – operations, RBP – remote batch processing, RT – real-time, SM – system management, TS – time-sharing, UT – utilities.

CONTENTS

APPENDIXES

FIGURES

TABLES

PREFACE

Communication between the computer and the user in current high-speed systems can be improved greatly through the use of highly discriminative programming languages. Such languages must be capable of expressing even intricate problems in a brief, incisive, and readily comprehensible form.

Ideally, a programming language should be machine-independent, easily learned, and universally applicable to the problems of science, engineering, and business. Prior to the advent of the meta-assembler concept, no single programming language had the capacity and flexibility required for the efficient programming of all types of applications. Some languages were intended for the solution of mathematical problems, while others were designed for business applications. Such programming languages are said to be "problem-oriented".

The vocabulary of a symbolic programming language consists of the permissible names, literals, operators, and other symbols that may be used to express a symbolic program. The syntax of such a language consists of the set of rules governing its sentence (i.e., statement) structure. In the past, the syntax rules for a symbolic programming language for a given computer were strongly influenced by the hardware characteristics of that machine. This resulted in programming languages that were "machine-oriented" and which, consequently, had numerous restrictions and unduly complex syntax rules. Because Sigma Meta-Symbol is neither a problem-oriented nor a machine-oriented assembler, there are fewer rules to learn, and therefore the flexibility of programming is greatly enhanced.

The Xerox Sigma Meta-Symbol processor can be used both as an assembler and as a meta-assembler. Used as an assembler, it translates symbolic programs into object-language code. Used as a meta-assembler, it enables the user to design his own programming languages and to generate processors for such languages with a minimum of effort.

Note that programs written for the Sigma 9 can be assembled on the Sigma 5/6/7.

1. INTRODUCTION

PROGRAMMING FEATURES

The following list summarizes Meta-Symbol's more important features for the programmer.

- The argument field can contain both arithmetic and Boolean (logical) expressions, using constant or variable quantities.
- Full use of lists and subscripted elements is possible.
- The DO and WHILE directives allow selective generation of areas of code, with parametric constants or expressions determined at the time of the assembly.
- Command procedures allow a macro-assembler capability of generating many units of codes for a given procedure call line. Further sophistication provides completely parameterized coding, with procedures applicable to many programs.
- Function procedures return values to the reference line.
- The call line and its individual parameters can be tested both arithmetically and logically.
- Nested procedures are used, and one procedure can call another.
- Complete use of arithmetic and Boolean operators in procedures is permitted.

META-SYMBOL PASSES

Meta-Symbol is a two-pass assembler that runs under control of various Xerox monitors. In addition to the two assembly passes (referred to as Pass 1 and Pass 2), there is an encoding pass (Pass 0) preceding the first assembly pass.

PASS 0

Pass 0 reads the input program (which may be symbolic, compressed, or compressed with symbolic corrections) and produces an encoded program for the assembler to process. If requested to do so, Pass 0 will output the encoded program in compressed form.

During Pass 0 the source program is checked for syntactical errors. If such errors are found, appropriate notification is given, and the encoding operation continues. Because the function of Pass 0 is to prepare the source program for processing by the assembler, it must recognize and process those directives concerned with manipulation of symbols (SYSTEM, LOCAL, OPEN, CLOSE). Thus, it is Pass 0 that locates the designated systems in the system library and incorporates them in the encoded program.

PASS 1

After Pass 0 is finished, Pass 1 is executed. Pass 1 reads the encoded program, builds the symbol table, and allocates storage space for each statement that is to be generated.

PASS 2

Pass 2 is the final assembly phase which generates the object code. It reads the encoded program and, using the symbol table produced by Pass 1, provides the correct addresses for all symbols. During this phase, literals and forward references are defined, and references to externally defined symbols are noted to be provided by the loader[†]. Pass 2 also produces the assembly listing, the format for which is described in Chapter 6.

[†]Xerox loaders are routines that form and link programs to be executed. A loader may be part of a monitor system or may be an independent program.

2. LANGUAGE ELEMENTS AND SYNTAX

LANGUAGE ELEMENTS

Input to the assembler consists of a sequence of characters combined to form assembly language elements. These language elements (which include symbols, constants, expressions, and literals) make up the program statements that comprise a source program.

CHARACTERS

Meta-Symbol source program statements may use the characters shown in Table 1.

Table 1. Meta-Symbol Character Set

Alphabetic:	A through Z, and $, @, #, ␣ (break character - prints as "underscore"). (: is the reserved alphabetic character, as explained below).
Numeric:	0 through 9
Special Characters:	Blank
	+ Add (or positive value)
	- Subtract (or negative value)
	* Multiply, indirect addressing prefix, or comments line indicator
	/ Divide
	// Covered quotient
	. Decimal point
	, Comma
	(Left parenthesis
	) Right parenthesis
	' Constant delimiter (single quotation mark
	& Logical AND
	| Logical OR (vertical slash)
	|| Logical exclusive OR (vertical slashes)
	¬ Logical NOT or complement
	< Less than
	> Greater than
	= Equal to or introduces a literal
	<= Less than or equal to
	>= Greater than or equal to
	¬= Not equal to
	; Continuation code
	** Binary shift
	TAB Syntactically equivalent to blank.

The colon is an alphabetic character used in internal symbols of standard Xerox software. It is included in the names of monitor routines (M:READ), assembler routines (S:IFR), and library routines (L:SIN). To avoid conflict between user symbols and those employed by Xerox software, it is suggested that the colon be excluded from user symbols.

SYMBOLS

Symbols are formed from combinations of characters. Symbols provide programmers with a convenient means of identifying program elements so they can be referred to by other elements. Symbols must conform to the following rules:

1. Symbols may consist of from 1 to 63 alphanumeric characters: A-Z, $, @, #, :, ␣ , 0-9. At least one of the characters in a symbol must be alphabetic. No special characters or blanks can appear in a symbol.

2. The symbols $ and $$ are reserved by the assembler to represent the current value of the execution and load location counters, respectively (see Chapter 3).

The following are examples of valid symbols:

ARRAY

R1

INTRATE

BASE

7TEMP

#CHAR

$PAYROLL

$ (execution location counter)

The following are examples of invalid symbols:

BASE PAY	Blanks may not appear in symbols.
TWO = 2	Special characters (=) are not permitted in symbols.

CONSTANTS

A constant is a self-defining language element. Its value is inherent in the constant itself, and it is assembled as part of the statement in which it appears.

Self-defining terms are useful in specifying constant values within a program via the EQU directive (as opposed to entering them through an input device) and for use in constructs

that require a value rather than the address of the location where that value is stored. For example, the Load Immediate instruction and the BOUND directive both may use self-defining terms:

```
LI,2     57  }
                 2, 57, 8 are self-defining terms.
BOUND    8   }
```

SELF-DEFINING TERMS

Self-defining terms are considered to be absolute (non-relocatable) items since their values do not change when the program is relocated. There are three forms of self-defining terms:

1. The decimal digit string in which the constant is written as a decimal integer constant directly in the instruction:

   ```
   LW,R   HERE + 6    6 is a decimal digit string.
   ```

2. The character string constant in which a string of EBCDIC† characters is enclosed by single quotation marks, without a qualifying type prefix. A complete description of C-type general constants is given below.

3. The general constant form in which the type of constant is indicated by a code character, and the value is written as a constant string enclosed by single quotation marks:

   ```
   LW,R   HERE + X'7B3'   7B3 is a hexadecimal
                          constant representing the
                          decimal value 1971.
   ```

There are seven types of general constants:

Code	Type
C	Character string constant (redundant notation)
X	Hexadecimal constant
O	Octal constant
D	Decimal constant
FX	Fixed-point decimal constant
FS	Floating-point short constant
FL	Floating-point long constant

C: Character String Constant. A character string constant consists of a string of EBCDIC† characters enclosed by single quotation marks and preceded by the letter C:

```
C'ANY CHARACTERS'
```

Each character in a character string constant is allocated eight bits of storage.

†A table of Extended Binary-Coded Decimal Interchange Codes, as well as information concerning hexadecimal arithmetic and hexadecimal to decimal conversion, can be found in the appropriate Sigma Computer Reference Manuals.

Because single quotation marks are used as syntactical characters by the assembler, a single quotation mark in a character string must be represented by the appearance of two consecutive quotation marks. For example,

```
C'AB"C"'
```

represents the string

```
AB'C'
```

Character strings are stored four characters per word. The descriptions of TEXT and TEXTC in Chapter 4 provide positioning information pertaining to the character strings used with these directives. When used in other data-generating directives, the characters are right-justified and a null EBCDIC character(s) fills out the field.

X: Hexadecimal Constant. A hexadecimal constant consists of an unsigned hexadecimal number enclosed by single quotation marks and preceded by the letter X:

```
X'9C01F'
```

The assembler generates four bits of storage for each hexadecimal digit. Thus, an eight-bit mask would consist of two hexadecimal digits.

The hexadecimal digits and their binary equivalents are as follows:

0 - 0000	8 - 1000
1 - 0001	9 - 1001
2 - 0010	A - 1010
3 - 0011	B - 1011
4 - 0100	C - 1100
5 - 0101	D - 1101
6 - 0110	E - 1110
7 - 0111	F - 1111

O: Octal Constant. An octal constant consists of an unsigned octal number enclosed by single quotation marks and preceded by the letter O:

```
O'7314526'
```

The size of the constant in binary digits is three times the number of octal digits specified, and the constant is right-justified in its field. For example:

Constant	Binary Value	Hexadecimal Value
O'1234'	001 010 011 100	0010 1001 1100 (29C)

The octal digits and their binary equivalents are as follows:

0 - 000	4 - 100
1 - 001	5 - 101
2 - 010	6 - 110
3 - 011	7 - 111

D: Decimal Constant. A decimal constant consists of an optionally signed value of 1 through 31 decimal digits, enclosed by single quotation marks and preceded by the letter D.

D'735698721' = D'+735698721'

The constant generated by Meta-Symbol is of the binary-coded decimal form required for Sigma decimal instructions. In this form, the sign[†] occupies the last digit position, and each digit consists of four bits. For example:

Constant	Value
D' + 99'	1001 1001 1100

A decimal constant could be used in an instruction as follows:

```
LW, R     L(D'99')
```

Load (LW) as a literal (L) into register R the decimal constant (D) 99.

The value of a decimal constant is limited to that which can be contained in four words (128 bits).

FX: Fixed-Point Decimal Constant. A fixed-point decimal constant consists of the following components in the order listed, enclosed by single quotation marks and preceded by the letters FX:

1. An optional algebraic sign.
2. d, d., d.d, or .d, where d is a decimal digit string.
3. An optional exponent:

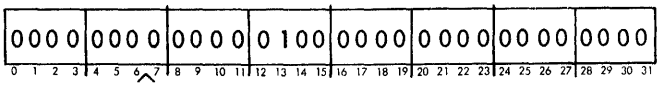
   the letter E followed optionally by an algebraic sign, followed by one or two decimal digits.
4. A binary scale specification:

   the letter B followed optionally by an algebraic sign, followed by one or two decimal digits that designate the terminal bit of the integer portion of the constant (i.e., the position of the binary point in the number). Bit position numbering begins at zero.

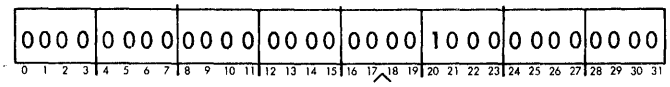
[†]A plus sign is a four-bit code of the form 1100. A minus sign is a four-bit code of the form 1101.

Parts 3 and 4 may occur in any relative order:

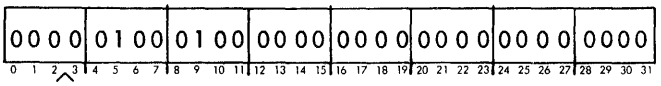
FX'.0078125B6'



FX'1.25E-1B17'



FX'13.28125B2E-2'

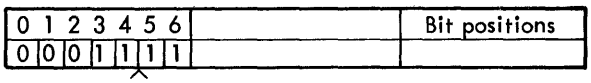


Example 1. Storing Fixed-Point Decimal Constants

Assume a halfword (16 bits) is to be used for two fields of data; the first field requires seven bits, and the second field requires nine bits.

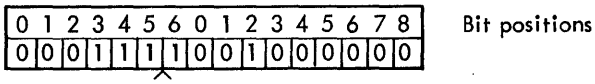
The number FX'3.75B4' is to be stored in the first field. The binary equivalent of this number is 11^11. The caret represents the position of the binary point. Since the binary point is positioned between bit positions 4 and 5, the number would be stored as

Field 1 Field 2



The number FX'.0625B-2' is to be stored in the second field. The binary equivalent of this number is ^0001. The binary point is to be located between bit positions -2 and -1 of field 2; there, the number would be stored as

Field 1 Field 2



In generating the second number, Meta-Symbol considers bit position -1 of field 2 to contain a zero, but does not actually generate a value for that bit position since it overlaps field 1. This is not an error to the assembler. However, if Meta-Symbol were requested to place a 1 in bit position -1 of field 2, an error would be detected since significant bits cannot be generated to be stored outside the field range. Thus, leading zeros may be truncated from the number in a field, but significant digits are not allowed to overlap from one field to another.

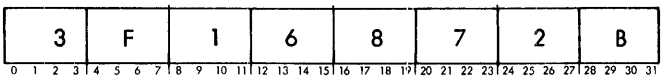
FS: Floating-Point Short Constant. A floating-point short constant† consists of the following components in order, enclosed by single quotation marks and preceded by the letter FS:

1. An optional algebraic sign.
2. d, d., d.d, or .d where d is a decimal digit string.
3. An optional exponent:

   the letter E followed optionally by an algebraic sign followed by one or two decimal digits.

Thus, a floating-point short constant could appear as

FS'5.5E-3'



The value of a floating-point short constant is limited to that which can be stored in a single word (32 bits).

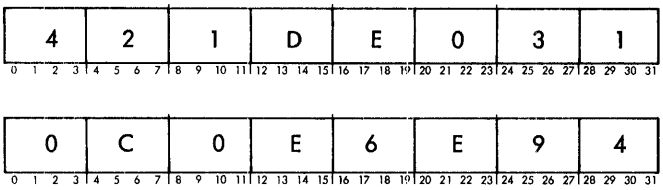
FL: Floating-Point Long Constant. A floating-point long constant† consists of the following components in order, enclosed by single quotation marks and preceded by the letters FL:

1. An optional algebraic sign.
2. d, d., d.d, or .d where d is a decimal digit string.
3. An optional exponent:

   the letter E followed optionally by an algebraic sign, followed by one or two decimal digits.

Thus, a floating-point long constant could appear as

FL'2987574839928.E-11'



The maximum size constants permitted by Meta-Symbol is as follows:

Constant Designation	Type	Maximum Size
	Decimal integer	64 bits (18 + digits)
C	Character string	504 bits (63 characters)
X	Hexadecimal number	64 bits (16 digits)
O	Octal number	64 bits (21 + digits)
D	Decimal number	128 bits (31 digits + sign)
FX	Fixed-point decimal number	32 bits
FS	Floating-point short number	32 bits
FL	Floating-point long number	64 bits

†Refer to the appropriate Xerox Sigma Computer Reference Manual for an explanation of floating-point format.

ADDRESSES

An address value is an element that is associated with a storage location in the Sigma main memory. There are two types of address values:

1. An absolute address has a value that corresponds exactly with a storage location in memory. Absolute address values will not be altered by the process of loading (linking) the program. Although absolute address values are invariant under the linking process, they are not considered as constants by Meta-Symbol. It is necessary to inform the Xerox loaders of the difference between constants and absolute addresses; for this reason, Meta-Symbol treats both absolute and relocatable addresses as a single type address.
2. A relocatable address has a value that consists of two parts, control section base and offset from this base. The base of any control section is determined by the Xerox loaders; thus, the only correspondence between a relocatable address value and an actual storage location is the offset from a base section location.

LITERALS

A literal is an expression enclosed by parentheses and preceded by the letter L:

L(-185)	decimal value -185
L(X'5DF')	hexadecimal value 5DF
L($+AB-3)	an address value

or an expression preceded by an equals sign:

= -185	decimal value -185
= X'5DF'	hexadecimal value 5DF
= $+AB-3	an address value

Literals are transformed into references to data values rather than actual values. Literals may be used in any construct that requires an address of a data value rather than the actual value. For example, the Load Word instruction

requires the address of the value to be loaded into the register, and use of a literal will satisfy that requirement:

```
LW,7        L(768)   The value 768 is stored in the
                     literal table and its address
                     is assembled as part of this
                     instruction.
```

A literal preceded by an asterisk specifies indirect addressing:

```
*=10      or      *L(10)
```

When a literal appears in a statement, Meta-Symbol produces the indicated value, stores the value in the literal table, and assembles the address of that storage location into the statement. The address is assembled as a word address, regardless of the intrinsic resolution of the literal control section. This address may be referenced, however, as a byte, halfword, or doubleword address (see "Addressing Functions" in Chapter 3). Literals may be used anywhere a storage address value is a valid argument field entry. However, literals may not be used in directives that require previously defined expressions.

During an assembly Meta-Symbol generates each literal as a 32-bit value on a word boundary in the literal table. The assembler detects duplicate values and makes only one entry for them in the table.

When Meta-Symbol encounters the END statement, it generates all literals declared in the assembly. The literals are generated at the current location (word boundary) of the currently active program section.

Any of the previously discussed types of constants except floating-point long (FL) may be written as literals:

```
L(1416)                integer literal
L(C'BYTE')             character string literal
L(X'F0F0')             hexadecimal literal
L(O'7777')             octal literal
L(D'37879')            decimal literal
L(FX'78.2E1B10')       fixed-point decimal literal
L(FS'-8.935410E-02')   floating-point short literal
```

EXPRESSIONS

An expression is an assembly language element that represents a value. It consists of a single term or a combination of terms (multitermed) separated by arithmetic operators.

The Meta-Symbol language permits general expressions of one or more terms combined by arithmetic and/or Boolean (logical) operators. Table 2 shows the operators processed by Meta-Symbol.

PARENTHESES WITHIN EXPRESSIONS

Multitermed expressions frequently require the use of parentheses to control the order of evaluation. Terms inside parentheses are reduced to a single value before being combined with the other terms in the expression. For example, in the expression

```
ALPHA*(BETA + 5)
```

the term BETA + 5 is evaluated first, and that result is multiplied by ALPHA.

Expressions may contain parenthesized terms within parenthesized terms:

```
DATA+(HRS/8-(TIME*2*(AG + FG)) + 5)
```

The innermost term (in this example, AG + FG) is evaluated first. Parenthesized terms may be nested to any depth.

Table 2. Meta-Symbol Operators

Operator	Binding Strength†	Function††
+	7	Plus (unary)
-	7	Minus (unary)
¬	7	Logical NOT or complement (unary)
**	6	Binary shift (logical)
*	5	Integer multiply
/	5	Integer divide
//	5	Covered quotient†††
+	4	Integer add
-	4	Integer subtract
<	3	Less than
>	3	Greater than
<=	3	Less than or equal to
>=	3	Greater than or equal to
=	3	Equal to
¬=	3	Not equal to
&	2	Logical AND
|	1	Logical OR
||	1	Logical exclusive OR

†See below, "Operators and Expression Evaluation".

††All operators are binary (i.e., require two operands) except the first three, specifically indicated as unary.

†††A//B is defined as (A + B - 1)/B

OPERATORS AND EXPRESSION EVALUATION

A single-termed expression, such as 36 or $ or SUM, takes on the value of the term involved. A multitermed expression, such as INDEX + 4 or ZD*(8+XYZ), is reduced to a single value as follows:

1. Each term is evaluated and replaced by its internal value.
2. Arithmetic operations are performed from left to right. Operations at the same parenthetical level

with the highest "binding strength" are performed first. For example,

A + B * C / D

is evaluated as

A + ((B * C) / D)

3. All arithmetic and logical operations in expressions are carried out in double precision (64 bits) with the following exceptions:

   a. Multiplication allows only single precision operands (32 bits) but may produce a double precision product.

   b. Division allows a single precision divisor and a double precision dividend and produces a single precision quotient.

4. Division always yields an integer result; any fractional portion is dropped.

5. Division by zero yields a zero result and is indicated by an error notification.

An expression may be preceded by an asterisk (*), which is often used to denote indirect addressing. Used as a prefix in this way, the asterisk does not affect the evaluation of the expression. However, if an asterisk precedes a subexpression, it is interpreted as a multiplication operator.

Multitermed expressions may be formed from the following operands:

1. Symbols representing absolute or relocatable addresses, which may be previously defined, forward, or external references.

2. Decimal integer constants (e.g., 12345) or symbols representing them.

3. All other general constants, namely character string (C), hexadecimal (X), octal (O), decimal (D), fixed-point (FX), floating-point short (FS), and floating-point long (FL), or symbols representing them.

The following should be noted with regard to expression evaluation:

1. To allow for greater flexibility in generating and manipulating C, D, FX, FS, and FL constants, the assembler treats them as integers when they are used arithmetically in multitermed expressions and carries the results internally as integers. Character constants (C) so used are limited to 8 bytes (64 bits), and decimal constants (D) to 15 characters + sign (64 bits).

2. All operators may be used but only the + and - operators and the comparison operators may take an address as an operand. An address operand is considered to be

   a. Any symbol that has been associated with an address in a relocatable or absolute section.

   b. Any local symbol referenced prior to its definition.

   c. Any symbol that is an external reference.

3. The sum of any two address operands is an address. The difference of any two address operands is an address, except for the case where both items are in the same control section and of the same resolution; the result then is an integer constant.

4. An address operand plus or minus a constant must use a single precision constant. Combining a negative constant with an address operand, however, will produce an error only if the negative constant cannot be represented correctly in single precision form. For example, external reference -1 is correct; external reference -9,589,934,592 is incorrect.

5. Meta-Symbol carries negatives as double precision numbers and will therefore provide for generated negative values of up to 64 bits.

LOGICAL OPERATORS

The logical NOT (¬), or complement operator, causes a one's complement of its operand:

Value	Hexadecimal Equivalent	One's Complement
3	00 . . . 0011	11 . . . 1100
10	00 . . . 1010	11 . . . 0101

The binary logical shift operator (**) determines the direction of shift from the sign of the second operand: a negative operand denotes a right shift and a positive operand denotes a left shift. For example:

5**-3

results in a logical right shift of three bit positions for the value 5, producing a result of zero.

The result of any of the comparisons produced by the comparison operators is

0 if false (or operands are of incompatible type)

1 if true

so that

Expression	Result	
3 > 4	0	3 is not greater than 4.
¬ 3=4	0	The 32-bit value ¬ 3 is equal to 11 . . . 1100 and is not equal to 4; i.e., 00 . . . 0100.

Expression	Result	
3 — =4	1	3 is not equal to 4.
—(3 =4)	11. . . 11	3 is not equal to 4, so the result of the comparison is 0 which, when complemented, becomes a 64-bit value (all one's).

The logical operators & (AND), I (OR), and II (exclusive OR) performs as follows:

AND

First operand:	0011
Second operand:	0101
Result of & operation:	0001

OR

First operand:	0011
Second operand:	0101
Result of I operation:	0111

Exclusive OR

First operand:	0011
Second operand:	0101
Result of II operation:	0110

Expressions may not contain two consecutive binary operators; however, a binary operator may be followed by a unary operator. For example, the expression

-A * -¬B / - C - 12

is evaluated as

(((-A) * (—¬B)) / (-C)) - 12

and the expression

T + U * (V + -W) - (268 / -X)

is evaluated as

(T + (U * (V + (-W)))) - (268 / (-X))

SYNTAX

Assembly language elements can be combined with computer instructions and assembler directives to form statements that comprise the source program.

STATEMENTS

A statement is the basic component of an assembly language source program; it is also called a source statement or a program statement.

Source statements are written on the standard coding form shown in Figure 1.

FIELDS

The body of the coding form is divided into four fields: label, command, argument, and comments. The coding form is also divided into 80 individual columns. Columns 1 through 72 constitute the active line; columns 73 through 80 are ignored by the assembler except for listing purposes and may be used for identification and a sequence number.

The columns on the coding form correspond to those on a standard 80-column card; one line of coding on the form can be punched into one card.

Meta-Symbol provides for free-form symbolic lines; that is, it does not require that each field in a statement begin in a specified column. The rules for writing free-form symbolic lines are:

1. The assembler interprets the fields from left to right: label, command, argument, comments.
2. A blank column terminates any field except the comments field, which is terminated at column 72 on card input or by a carriage-return or new-line character on Teletype.
3. One or more blanks at the beginning of a line specify there is no label field entry.
4. The label field entry, when present, must begin in column 1, except when the initial line of a statement contained a semi-colon in column 1. The label field may then start in any active column in the second line.
5. The command field begins with the first nonblank column following the label field or in the first nonblank column following column 1, if the label field is empty.
6. The argument field begins with the first nonblank column following the command field. An argument field is designated as a blank in either of two ways:
   a. Sixteen or more blank columns follow the command field.
   b. The end of the active line (column 72) is encountered.
7. The comments field begins in the first nonblank column following the argument field or after at least 16 blank columns following the command field, when the argument field is empty.

ENTRIES

A source statement may consist of one to four entries written on a coding sheet in the appropriate field: a label field entry, a command field entry, an argument field entry, and a comments field entry.

Xerox Data Systems

XEROX

Xerox Sigma Symbolic Coding Form

PROBLEM NOTHING WHATEVER

PROGRAMMER MVK

IDENTIFICATION 73 80

PAGE 1 OF 1

DATE 2-15-71

LABEL (1 5)	COMMAND (10 15)	ARGUMENT (20 25 30 35)	COMMENTS (37 40 45 50 55 60 65 70 72)
*			*
*** THIS	PROGRAM	PRINTS NOTHING WHATEVER ***	
*			*
		SYSTEM SIG7	
		REF M:LL	
NOTHING	WHATEVER	RES 0	
		TEXTC 'NOTHING WHATEVER'	
PARAMETER	TABLE1		EQU $
		SEN,8,24 1,0	
		PZE *0	
		DATA NOTHING WHATEVER	
START ;			
		CAL1,2 PARAMETER TABLE1	PRINT NOTHING WHATEVER
EXIT		CAL1,9 1	AND THEN STOP.
		END START	

Figure 1. Xerox Sigma Symbolic Coding Form

LABEL FIELD

A label entry is normally a list of symbols that identifies the statement in which it appears. The label enables a programmer to refer to a specific statement from other statements within the program.

The label on a procedure reference line (see Chapter 5) may contain any list of valid Meta-Symbol expressions, constants, or symbols.

Multiple labels may appear in the label field of any instruction and of any directive except DSECT, which must have one and only one label. A label for some directives is not meaningful and is ignored unless it is the target label of a GOTO search. The labels must be separated by commas. A series of labels may be continued onto following lines by writing a semicolon after any character in the label and writing the next character on another line, starting in any column after column 1.

Example 2. Label Field Entries

```
YEAR_TO_DATE, ACCUMULATED_SALARY;
   ,COMMISSIONS
```

Note that the semicolon does not replace the comma that is required to separate the entries.

The label of a value, a list, or a function procedure may have the same configuration as a command, without conflict, since Meta-Symbol is able to distinguish through context which usage is intended. For example, the mnemonic code for the Load Word command is LW. An instruction may be written with LW in the label field without conflicting with the command LW.

The name of any intrinsic function that requires parentheses (ABSVAL, BA, CS, DA, HA, L, NUM, S:IFR, S:NUMC, S:PT, S:UFV, S:UT, SCOR, TCOR, and WA) may be used as a label in either a main program or a procedure definition, if the parentheses are omitted. The intrinsic functions AF, AFA, CF, LF, and NAME may be used as labels in a main program, but within a procedure definition they are always interpreted as functions.

Example 3. Label Field Entry

LABEL (1 5)	COMMAND (10 15)	ARGUMENT (20 25 30 35)
PAY_RATE		
A(I+3,X)		
A3		
COST@		
'FIFTEEN', X'F'		

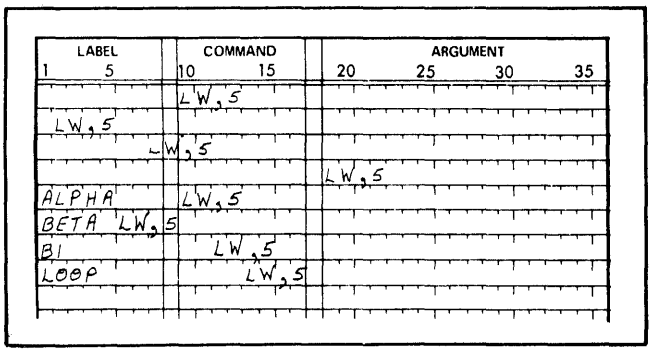
COMMAND FIELD

A command entry is required in every active line. Thus, if a statement line is entirely blank following the label

field or if the command entry is not an acceptable instruction or directive, the assembler declares the statement in error.

The command entry is a mnemonic operation code, an assembler directive, or a procedure name. Meta-Symbol directives and valid mnemonic codes for machine operations are listed in the Appendixes. Procedures are discussed in Chapter 5.

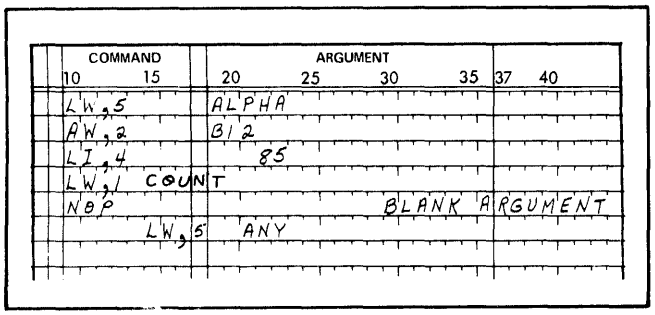
Example 4. Command Field Entry



ARGUMENT FIELD

An argument entry consists of one or more symbols, constants, literals, or expressions separated by commas. The argument entries for machine instructions usually represent such things as storage locations, constants, or intermediate values. Arguments for assembler directives provide the information needed by Meta-Symbol to perform the designated operation.

Example 5. Argument Field Entry



COMMENT FIELD

A comments entry may consist of any information the user wishes to record. It is read by the assembler and output as part of the source image on the assembly listing. Comments have no effect on the assembly.

COMMENT LINES

An entire line may be used as a comment by writing an asterisk in column 1. Any EBCDIC character may be used in comments. Extensive comments may be written by using a series of lines, each with an asterisk in column 1.

The assembler reproduces the comment lines on the assembly listing and counts comment lines in making line number assignments (see Chapter 6 for a description of output formats).

STATEMENT CONTINUATION

If a single statement requires more space than is available in columns 1 through 72, it can be continued onto one or more following lines. When a statement is to be continued on another line, the following rules apply:

1. Each line that is to be continued on another line must be terminated with a semicolon. The semicolon must not be within a character constant string. Anything in the initial line following the semicolon is treated as comments. A semicolon within comments is not treated as a continuation code.

2. Column 1 of each continuation line must be blank.

3. Comment lines may not be continued.

4. Comment lines may be placed between continuation lines.

5. Leading blanks on continuation lines are ignored by the assembler. Thus, significant blanks that must follow label or command entries must precede the semicolon indicating continuation.

Example 6. Statement Continuation

```
BEGIN   LW,3        A;          Continuation
                    +B
          .
          .
          .
NEW     TEXT        'A;B'       ; is not a contin-
          .                     uation character.
          .
          .
        LOCAL A,START,R1,;
        D,RATIO,B12,;           Continuation
        C,MAP
ANS     LW,3        ;           The blank that
                    SUM,1       terminates the
                                command field
                                precedes the
                                semicolon.
```

PROCESSING OF SYMBOLS

Symbols are used in the label field of a machine instruction to represent its location in the program. In the argument field of an instruction, a symbol identifies the location of an instruction or a data value.

The treatment of symbols appearing in the label or argument field of an assembler directive varies.

DEFINING SYMBOLS

A symbol is "defined" by its appearance in the label field of any machine language instruction and of certain directives:

ASECT, CNAME, COM, CSECT, DATA, DO, DO1, DSECT, END, EQU, FNAME, GEN, LOC, ORG, PSECT, RES, SET, S:SIN, TEXT, TEXTC, WHILE, and USECT.

For all other directives a label entry is ignored (except as a target label of a GOTO directive); that is, it is not assigned a value.

Any machine instruction can be labeled; the label is assigned the current value of the execution location counter.

The first time a symbol is encountered in the label field of an instruction, or any of the directives mentioned above, it is placed in the symbol table and assigned a value by the assembler. The values assigned to labels naming instructions, storage areas, constants, and control sections represent the addresses of the leftmost bytes of the storage fields containing the named items.

Often the programmer will want to assign values to symbols rather than having the assembler do it. This may be accomplished through the use of EQU and SET directives. A symbol used in the label field of these directives is assigned the value specified in the argument field. The symbol retains all attributes of the value to which it is equated.

Note: The use of labels is a programmer option, and as many or as few labels as desired may be used. However, since symbol definition requires assembly time and storage space, unnecessary labels should be avoided.

REDEFINING SYMBOLS

Usually, a symbol may be defined only once in a program. However, if its value is originally assigned by a SET, DO, or WHILE directive, the symbol may be redefined by a subsequent SET directive or by the processing of a DO or WHILE loop. For example:

```
SYM   SET   15     SYM is assigned the value 15.
      :
      :
SYM   DO    3      SYM is changed to zero and
                   is incremented by 1 each time
                   the DO loop is executed.
      :
      :
NOW   SET   SYM    NOW is assigned the value
                   SYM had when the DO loop
                   was completed; i.e., 3not 15.
```

SYMBOL REFERENCES

A symbol used in the argument field of a machine instruction or directive is called a symbol reference. There are three types of symbol references.

PREVIOUSLY DEFINED REFERENCES

A reference made to a symbol that has already been defined is a previously defined reference. All such references are completely processed by the assembler. Previously defined references may be used in any machine instruction or directive.

FORWARD REFERENCES

A reference made to a symbol that has not been defined is a forward reference. There are two distinct types of forward references, local forward references and nonlocal forward references. Table 3 summarizes the permissible places where each type may be used. Directives not listed either do not allow forward references (e.g., DO) or completely ignore them (e.g., PAGE, PROC).

Table 3. Legal Use of Forward References

	Command Field		Argument Field	
Command	Local	Nonlocal	Local	Nonlocal
Machine Instruction	X	X	X	X
CDISP				X
CLOSE				X
CNAME			X	X
COM				X
DATA			X	X
DEF			X	X
DISP			X	X
EQU		X	X	X
ERROR		X		X
FDISP				X
FNAME			X	X
GEN			X	X
GOTO			X	X
LIST				X
LOCAL			X	
OPEN				X
PCC				X
PEND				X
PSR				X
PSYS				X
SET		X	X	X
S:SIN			X	X
SPACE				X
TITLE				X
Procedure	X	X	X	X

There are two general restrictions on the use of forward references:

1. A forward reference may not be subscripted.

2. A subscripted symbol may not have a forward reference in the subscript list.

Meta-Symbol permits the use of forward references in multi-termed expressions.

EXTERNAL REFERENCES

A reference made to a symbol defined in a program other than the one in which it is referenced is an external reference.

A program that defines external references must declare them as external by use of the DEF directive. An external definition is output by the assembler as part of the object program, for use by the loader.

A program that uses external references must declare them as such by use of a REF or SREF directive.

A machine instruction containing an external reference is incompletely assembled. The object code generated for such references allows the external references and their associated external definitions to be linked at load time.

After a program has been assembled and stored in memory to be executed, the loader automatically searches the program library for routines whose labels satisfy any existing external references. These routines are loaded automatically, and interprogram communication is thus completed.

The permissible places in which external references may be used are identical to the legal uses for local forward references, as given in Table 3.

Meta-Symbol permits the use of external references in multi-termed expressions.

CLASSIFICATION OF SYMBOLS

Symbols may be classified as either local or nonlocal.

A local symbol is one that is defined and referenced within a restricted program region. The program region is designated by the LOCAL directive, which also declares the symbols that are to be local to the region.

A symbol not declared as local by use of the LOCAL directive is a nonlocal symbol. It may be defined and referenced in any region of a program, including local symbol regions.

The same symbol may be both nonlocal and local, in which case the nonlocal and local forms identify different program elements.

SYMBOL TABLE

The value of each defined symbol is stored in the assembler's symbol table. Each value has a value type associated with it, such as absolute address, relocatable address, integer, or external reference. Some types require additional information. For example, relocatable addresses, which are entered as offsets from the program section base, require the intrinsic resolution of the symbol (see Chapter 3 for a discussion of intrinsic resolution and the section number).

When the assembler encounters a symbol in the argument field, it refers to the symbol table to determine if the symbol has already been defined. If it has, the assembler obtains from the table the value and attributes associated with the symbol, and is able to assemble the appropriate value in the statement.

If the symbol is not in the table, it is assumed to be a forward reference. Meta-Symbol enters the symbol in the table, but does not assign it a value. When the symbol is defined later in the program, Meta-Symbol assigns it a value and designates the appropriate attributes.

LISTS

A list is an ordered set of elements. Each element occupies a unique position in the set and can, therefore, be identified by its position number. The nth element of list R is designated as R(n). An element of a list may also be another list. Any given element of a list may be numeric, symbolic, or null (i.e., nonexistent).

A list may be either linear or nonlinear. A linear list is one in which all non-null elements consist of a single numeric or symbolic expression of the first degree (i.e., having no element with a sub-subscript greater than 1). A nonlinear list has at least one compound element; that is, a non-null element with a sub-subscript greater than 1.

These definitions are explained in greater detail below.

Lists may be used in two ways: as value lists or as procedure reference lists. Value lists are discussed in this chapter; see Chapter 5 for a description of procedure reference lists.

VALUE LISTS

LINEAR VALUE LISTS

A linear value list may consist of several elements or of only a single non-null element having a specific numeric value (e.g., a signed or unsigned integer, an address, or a

floating-point number). Thus, a single value and a linear value list of one element are structurally indistinguishable.

An example of a linear value list, named R, having the four elements 5, 3, -16, and 17 is shown below.

R ≡ 5, 3, -16, 17

(The symbol ≡ means "is identical to".)

Reference Syntax. In the example given above, the four elements of list R would be referred to as:

R(1) = 5

R(2) = 3

R(3) = -16

R(4) = 17

The numbers in parentheses are the subscripts of the elements. Note that, for the above example:

R(n) = null for n > 4

A null value is not a zero value. An element having a value of zero is not considered a null element, because zero is a specific numeric value. The null elements of a value list are those that have not been assigned a value, although they do have specific subscript numbers. That is, all subscript numbers not assigned to non-null elements may be used to reference implicit null elements. For example, the list R, as defined above, consists of four elements:

R(1) = 5

R(2) = 3

R(3) = -16

R(4) = 17

and any number of implicit null elements:

R(5) = null

R(6) = null

R(n) = null for n > 4

A null value used in an arithmetic or logical operation has the same effect as a zero value. Thus, if

LIST(a) = null

then

LIST(b) + LIST(a) = LIST(b)

also

0 + LIST(a) = 0

also

LIST(a) + null = 0

Example 7. Linear Value List[†]

```
A       SET     8,6,9
```

defines list A as

A(1) = 8

A(2) = 6

A(3) = 9

A(4) = null

A(n) = null for n ≥ 4

The list could be altered by assigning additional elements to list A:

```
A(4)    SET     -65
A(5)    SET     231
```

changing list A to

A 8,6,9,-65,231

When a list contains explicit null elements (i.e., those followed by one or more non-null elements), they are counted with the non-null elements in determining the total number of elements in the list.

Examples of lists containing explicit null elements are shown below.

```
A       SET     5, 17, 10,,,14
B       SET     ,,6
```

defines lists A and B as

A = 5, 17, 10, null, null, 14 — list A contains six explicit elements.

B = null,null,6 — list B contains three explicit elements.

A trailing comma in a list specifies a trailing explicit null element. Thus, a list defined as

```
S       SET     4,3,6,,2,
```

contains six explicit elements: 4,3,6,null,2,null.

If Q is the name of an m-element value list, e is an expression having the single value n, and no list having more than 255 elements can be accommodated by the assembler, then the reference syntax will give the values shown in Table 4.

Generation. The syntax for defining a list is

name followed by directive followed by sequence

[†]Lists values are normally defined by SET or EQU directives, which are described in Chapter 4.

Table 4. Reference Syntax for Lists

Case	Syntax of Reference	Range of n	Meaning of the Reference	Value(s) of the Reference
1	Q or Q(0)	$n = 0$	Reference to all elements of list Q.	The m values of the elements of list Q.
2	Q(e)	$1 \leq n \leq m$	Reference to the nth element of list Q.	The value of the nth element of list Q.
3	Q(e)	$m < n \leq 255$ (n is an integer)	Reference to nonexistent (null) element of list Q. (No error flag.)	Null. (Numeric effect equivalent to zero.)
4	Q(e)	$n < 0$ or $n > 255$ or n is not an integer	Error. (Subscript out of range.)	The value of Q(1).

The name may be any symbol chosen by the programmer, the directive may be either EQU or SET, and the sequence is one or more elements establishing the list structure. Note that a name is mandatory.

Each element in a list-defining sequence must be either (1) the expression to be used as the next element of the list, or (2) a reference (case 1 or 2 of Table 4) to an m-element list, whose elements are to be copied as the next elements of the list being defined. This is illustrated in Example 8, where the effects of successive SET directives are to be considered cumulative.

Example 8. Defining Linear Value Lists

Example 8a

```
Q       SET     4,7 + 2
```

creates

Q ≡ 4,9

Example 8b

```
R       SET     Q(1),17,-6
```

creates

R ≡ 4,17,-6

Example 8c

```
S       SET     Q
```

creates

S ≡ 4,9

Example 8d

```
T       SET     Q,19,Q,R(3)
```

creates

T ≡ 4,9,19,4,9,-6

Example 8e

```
Q       SET     T(6),T(3),205
```

redefines

Q ≡ -6,19,205

Note: Example 8 does not result in redefinition of R, S, or T, although they were initially defined in terms of elements of Q; only Q will have new values after execution of this directive.

Example 8f

```
T       SET     T(5)
```

redefines

T ≡ 9

Note: The evaluation of T(5) is performed before redefinition of T. All elements of T that are of higher order than T(1) will be null elements after execution of this directive (i.e., T(n) ≡ null for n > 1).

Example 8g

```
S       SET     S,6
```

redefines

S ≡ 4,9,6

Example 8h

```
S       SET     1,S
```

redefines

S ≡ 1,4,9,6

Manipulation. The SET directive can be used not only to define or redefine an entire list, but also to define or redefine any single element of a linear value list. The syntax

of the directive is still name followed by directive followed by sequence, but the name is a subscripted symbol identifying some particular list element; and the sequence is only a single expression, representing either a specific numeric value or the name of a previously defined element having a single value.

In Example 9 below, the effects of successive SET directives are to be considered cumulative, but not retroactive.

Example 9. Redefining a Linear Value List

Example 9a

```
A       SET     5,6,4
A(2)    SET     17
```

redefines

A ≡ 5, 17, 4

Example 9b

```
A(3)    SET     A(3) + 6
```

redefines

A ≡ 5, 17, 10

NONLINEAR VALUE LISTS

A nonlinear value list has at least one compound element; that is, a non-null element having a sub-subscript greater than 1. A compound element in a list is identified by enclosure within parentheses. Example 10 illustrates this notation.

Example 10. Parentheses in Nonlinear Value Lists

X ≡ (4)	Redundant parentheses.
X ≡ (4, 7)	Not redundant.
X ≡ (A)	If A has previously been equated to a single value, the parentheses are redundant.
	If A has previously been equated to a list of values, the parentheses are not redundant.

In Example 11, notice the use of parentheses in specifying the level of the subelements. Z(1) consists of one subelement: (2, 3, 4), which is composed of three sub-subelements: 2, 3, 4, as compared with Z(2) which consists of three subelements: 9, 8, 11, and no sub-subelements. Meta-Symbol places no limit on the number of levels that may be specified for subelements.

Example 11. Nonlinear Value List Notation

Z ≡ ((2, 3, 4)), (9, 8, 11), 7, (6, (5, 4))

The elements of list Z are

Z(1) ≡ (2, 3, 4)

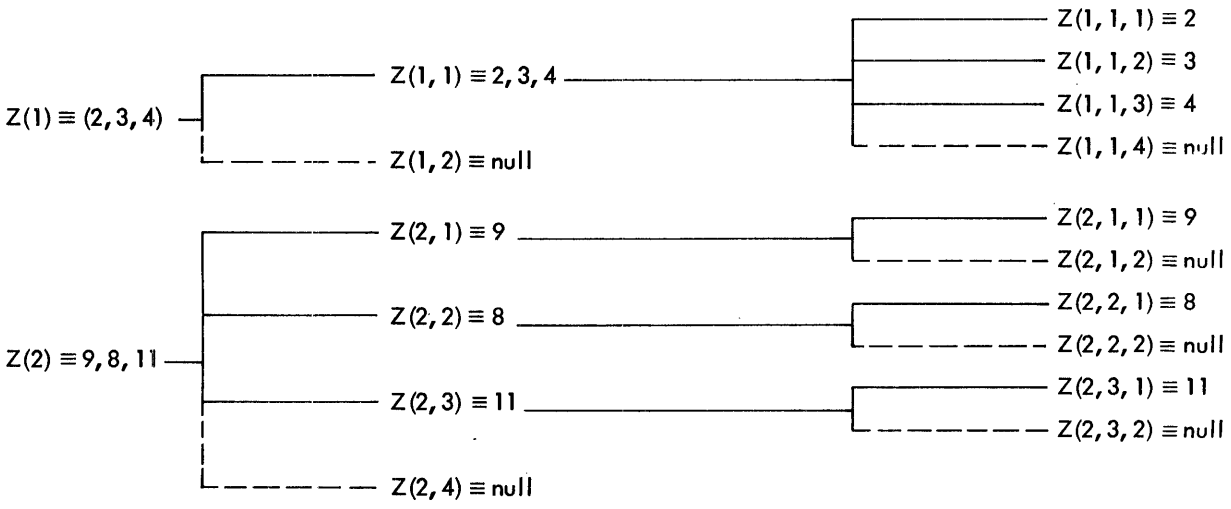
Z(2) ≡ 9, 8, 11

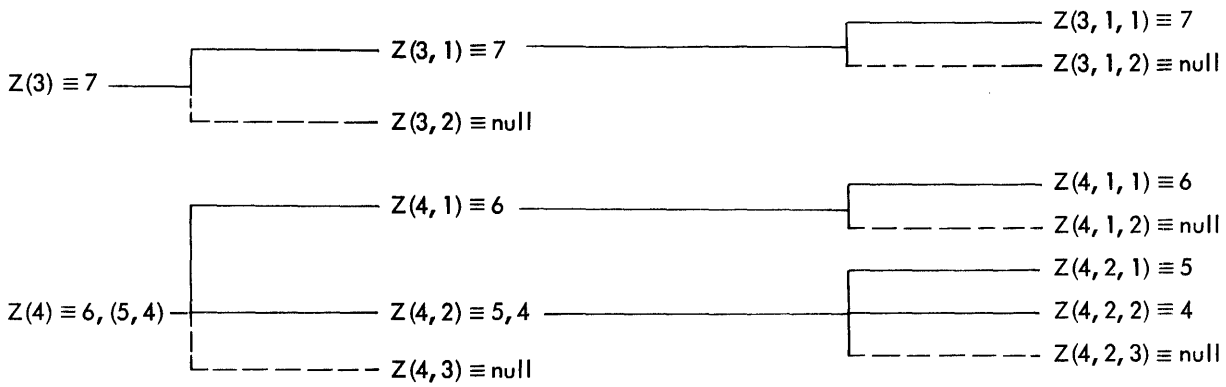
Z(3) ≡ 7

Z(4) ≡ 6, (5, 4)

Z(n) ≡ null for n > 4

Subelements of list Z are identified by means of multiple subscripts (i.e., sub-subscripts):





A number of implicit null elements could be identified as subelements. In this example implicit null elements are indicated with broken lines and only one such element is shown for each subdivision.

Redundant parentheses frequently occur in lists. For example, the list

A ≡ ((((4 + 7) * (3 + 2)),6))

can be simplified as follows:

A ≡ ((((11) * (5)),6))

A ≡ (((55),6))

The pair of parentheses enclosing 55 is redundant, since (55) and 55 are identical. However, the remaining two sets of parentheses are not redundant since they specify the level of the subelements. The use of redundant parentheses in lists is permitted in Meta-Symbol.

Reference Syntax. The reference syntax used with nonlinear value lists is the same as that used with linear value lists, except that multiple subscripts are used to indicate the subelement.

In addition to allowing the use of redundant parentheses, the list-manipulation syntax allows lists to be defined in terms of elements of other lists or even in terms of elements of the list itself. For example, if list M is defined as

M ≡ -6, (4, 7), 3

then another list could be defined as

N(2) SET M(2) making N(2) ≡ 4, 7

or an entire list could be defined as

P SET M making P ≡ -6, (4, 7), 3

Furthermore, elements within a list can be redefined in terms of list elements:

M	SET	-6, (4, 7), 9	making M ≡ -6, (4, 7), 9
M(1)	SET	M(2, 1)	making M ≡ 4, (4, 7), 9
M(2,2)	SET	M(3)	making M ≡ 4, (4, 9), 9
M(3)	SET	M(3)	making M ≡ 4, (4, 9), 9
M(3)	SET	9	making M ≡ 4, (4, 9), 9

Notice that the last two declarations result in no change in value for element M(3).

Assume that list R is defined as equal to element A(a) of list A, that list S is defined as element R(b) of list R, and that list T is defined as element S(c) of list S. List T will then be equal to element A(a, b, c) of list A. That is, if

R SET A(a)

and

S SET R(b)

and

T SET S(c)

then

T ≡ A(a, b, c)

Example 12. Defining Nonlinear Value Lists

Assume list A is defined as

A ≡ 4, ((2, 6), 4, 1), 17

then the following definitions could be made

R	SET	A(2)	making R ≡ (2, 6), 4, 1
S	SET	R(1)	making S ≡ 2, 6
T	SET	S(2)	making T ≡ 6

The same definition for T could be achieved by writing

T SET A(2, 1, 2) making T ≡ 6

Generation. The definition syntax for nonlinear value lists is the same as that for linear lists, and either EQU or SET directives may be used. In Example 13 the effects of successive SET directives are to be considered cumulative, but not retroactive. Assume that all lists are initially undefined.

Manipulation. The SET directive may be used to define or redefine any single element or subelement of a nonlinear value list. The name used with the directive is a subscripted symbol identifying some particular element or subelement, and the sequence may consist of one or more expressions.

Example 13. Defining Nonlinear Value Lists

Example 13a	A	SET	(5,6),7	defines A ≡ (5,6),7 thus A(1) ≡ 5,6 A(2) ≡ 7 A(3) ≡ null
Example 13b	B	SET	1 + 2 * 3, 17, A(3, 1)	defines B ≡ 7, 17 thus B(1) ≡ 7 B(2) ≡ 17 B(3) ≡ null
Example 13c	C	SET	A, (A), A(1), B(2)	defines C ≡ (5,6),7,((5,6),7),5,6,17 thus C(1) ≡ 5,6 C(2) ≡ 7 C(3) ≡ (5,6),7 C(4) ≡ 5 C(5) ≡ 6 C(6) ≡ 17

Notice that the parentheses enclosing the second element in the definition of C are not redundant. They specify that the entire list A is to be one element of list C.

Example 13d	D	SET	A, B	defines D ≡ (5,6),7,7,17 thus D(1) ≡ 5,6 D(2) ≡ 7 D(3) ≡ 7 D(4) ≡ 17
Example 13e	B	SET	A, (B)	redefines B ≡ (5,6),7,(7,17) thus B(1) ≡ 5,6 B(2) ≡ 7 B(3) ≡ 7,17

In Example 13e, the original elements of list B are used to redefine an element of the list. This is possible because the assembler evaluates the items on the righthand side of the directive SET before equating them with the symbol(s) on the lefthand side.

In Example 14 the effects of successive SET directives are to be considered cumulative, but not retroactive. Assume all lists are initially undefined.

NUMBER OF ELEMENTS IN A LIST

The number of explicit elements (i.e., non-null elements plus explicit null elements) in a list can be determined through the use of the intrinsic function NUM. The syntax for this function is

NUM(name)

The name specified may be that of a list, of an element, or of a subelement of a list. In Example 15 the number of explicit elements is determined for list S and also for each of its elements and subelements.

If a list is defined as equal to some given element of another list, the new list will have the same number of explicit elements as the original list. That is, if

Q SET P(a)

then

NUM(Q) = NUM(P(a))

Example 17 illustrates this point.

Example 14. Manipulating Nonlinear Value Lists

Example 14a	A(1)	SET	1,2,3	defines A ≡ (1,2,3)		
				thus A(1) ≡ 1,2,3	A(1,1) ≡ 1	A(1,1,1) ≡ 1
				A(2) ≡ null	A(1,2) ≡ 2	A(1,1,2) ≡ null
					A(1,3) ≡ 3	A(1,2,1) ≡ 2
					A(2,1) ≡ null	A(1,2,2) ≡ null
						A(1,3,1) ≡ 3
						A(1,3,2) ≡ null
Example 14b	A(1,1,2)	SET	4	defines a previously null element: A(1,1,2) ≡ 4		
				making list A ≡ ((1,4),2,3)		
				thus A(1) ≡ (1,4),2,3	A(1,1) ≡ 1,4	
				A(2) ≡ null	A(1,2) ≡ 2	
					A(1,3) ≡ 3	
					A(2,1) ≡ null	
Example 14c	B(1,2)	SET	A(1,1),(A(1,2),A(1,3))			
				defines B ≡ (null,(1,4,(2,3)))		
				thus B(1) ≡ null, (1,4,(2,3))		B(1,1) ≡ null
				B(2) ≡ null		B(1,2) ≡ 1,4,(2,3)
Example 14d	C(1)	SET	A(1,2),(A(1,1,1))	defines C ≡ (2, 1)		
				thus C(1) ≡ 2,1	C(1,1) ≡ 2	
				C(2) ≡ null	C(1,2) ≡ 1	

Notice that the parentheses around A(1,1,1) are redundant in this example.

Example 14e	B(1,1)	SET	C(1,2)	defines a previously null subelement: B(1,1) ≡ 1		
				thus B ≡ (1,(1,4,(2,3)))		
				B(1) ≡ 1,(1,4,(2,3))		B(1,1) ≡ 1
				B(2) ≡ null		B(1,2) ≡ 1,4,(2,3)

Example 15. NUM Function

S ≡ A,(B,((C,D)))
NUM(S) = 2

S(1) ≡ A
NUM(S(1)) = 1

S(1,1) ≡ A
NUM(S(1,1)) = 1

S(1,2) ≡ null
NUM(S(1,2)) = 0

S(2) ≡ B, ((C, D))
NUM(S(2)) = 2

S(2,1) ≡ B
NUM(S(2,1)) = 1

S(2,1,1) ≡ B
NUM(S(2,1,1)) = 1

S(2,1,2) ≡ null
NUM(S(2,1,2)) = 0

S(2,2) ≡ (C,D)
NUM(S(2,2)) = 1

S(2,2,1) ≡ C,D
NUM(S(2,2,1)) = 2

S(2,2,1,1) ≡ C
NUM(S(2,2,1,1)) = 1

S(2,2,1,2) ≡ D
NUM(S(2,2,1,2)) = 1

S(2,2,1,3) ≡ null
NUM(S(2,2,1,3)) = 0

S(2,2,2) ≡ null
NUM(S(2,2,2)) = 0

S(2,3) ≡ null
NUM(S(2,3)) = 0

S(3) ≡ null
NUM(S(3)) = 0

Example 16. NUM Function

Assume list Z is defined as

Z	SET	3,,,4,,,	List Z consists of seven elements: 3, null, null, 4, null, null, null. (Note that the last null element is specified by the final comma in the list.)

thus, NUM(Z) = 7

If

Z(4)	SET	Z(2)	That is, the fourth element of Z is redefined as a null element.
NUM(Z) = 7			List Z would still consist of seven elements: 3, null, null, null, null, null, null.

Note that NUM(Z(2)) = 0

Example 17. NUM Function

Assume list A is defined as

A≡4, ((2,6),4,1),17			making A(2) ≡ (2,6,),4,1

If the following definitions are made:

R	SET	A(2)	making R ≡ (2,6),4,1
S	SET	R(1)	making S ≡ 2,6
T	SET	S(2)	making T ≡ 6

Then the following statements are true:

NUM(A(2)) = 3

NUM(R) = NUM(A(2)) = 3

NUM(S) = NUM(R(1)) = NUM(A(2,1)) = 2

NUM(T) = NUM(S(2)) = NUM(R(1,2))
= NUM(A(2,1,2)) = 1

3. ADDRESSING

Sigma computer addressing techniques require a register designation and an argument address that may specify indexing and/or indirect addressing. The programmer may write addresses in symbolic form, and the assembler will convert them to the proper equivalents.

RELATIVE ADDRESSING

Relative addressing is the technique of addressing instructions and storage areas by designating their locations in relation to other locations. This is accomplished by using symbolic rather than numeric designations for addresses. An instruction may be given a symbolic label such as LOOP, and the programmer can refer to that instruction anywhere in his program by using the symbol LOOP in the argument field of another instruction. To reference the instruction following LOOP, he can write LOOP+1; similarly, to reference the instruction preceding LOOP, he can write LOOP-1.

An address may be given as relative to the location of the current instruction even though the instruction being referenced is not labeled. The execution location counter, described later in this chapter, always indicates the location of the current instruction and may be referenced by the symbol $. Thus, the construct $+8 specifies an address eight units greater than the current address, and the construct $-4 specifies an address four units less than the current address.

ADDRESSING FUNCTIONS

Intrinsic functions are functions built into the assembler. Certain of these functions concerned with address resolution are discussed here. Literals were discussed in Chapter 2, and other intrinsic functions are explained in Chapter 5.

Intrinsic functions, including those concerned with address resolution, may or may not require arguments. When an argument is required for an intrinsic function, it is always enclosed in parentheses.

A symbol whose value is an address has an intrinsic address resolution assigned at the time the symbol is defined. Usually this intrinsic resolution is the resolution currently applicable to the execution location counter. The addressing functions BA, HA, WA, and DA (explained later) allow the programmer to specify explicitly a different intrinsic address resolution than the one currently in effect.

Certain address resolution functions are applied unconditionally to an address field after it is evaluated. The choice of functions depends on the instruction involved. For instructions that require values rather than addresses (e.g., LI, MI, DATA), no final addressing function is applied. For instructions that require word addresses (e.g., LW, STW, LB, STB, LH, LD), word address resolution is applied. Thus, the assembler evaluates LW,3 ADDREXP as if it were LW,3 WA(ADDREXP). Similarly, instructions that require byte addressing (e.g., MBS) cause a final byte addressing resolution to be applied to the address field.

More information on address resolution is given after the explanation of intrinsic addressing functions, which follows.

$, $$ Location Counters

The symbols $ (current value of execution location counter) and $$ (current value of load location counter) indicate that the current value of the appropriate location counter is to be generated for the field in which the symbol appears.

The current address resolution of the counter is also applied to the generated field. This resolution may be changed by the use of an addressing function.

Example 18. $, $$ Functions

```
          :
          :
A         EQU     $       Equates A to the current value
          :                 of the execution location
          :                 counter.
Z         EQU     $$      Equates Z to the current value
          :                 of the load location counter.
          :
TEST      BCS,3   $+2     Branches to the location speci-
                            fied by the current execution
                            location counter +2 if the
                            condition code and value 3
          :                 compare 1's anyplace.
          :
```

BA Byte Address

The byte address function has the format

BA(address expression)

where BA identifies the function, and address expression is the symbol or expression that is to have byte address resolution when assembled. If address expression is a constant, the value returned is the constant itself.

Example 19. BA Function

```
          :
          :
Z         LI,3    BA(L(48))  The value 48 is stored in the
                               literal table and its location
                               is assembled into this argu-
                               ment field as a byte address.
```

```
        :
        :
AA      LI,5    BA($)       The current execution loca-
                            tion counter address is evalu-
                            ated as a byte address for this
        :                   statement.
        :
```

HA Halfword Address

The halfword address function has the format

HA(address expression)

where HA identifies the function, and address expression is the symbol or expression that is to have halfword address resolution. If address expression is a constant, the value returned is the constant itself.

Example 20. HA Function

```
        :
        :
Z       CSECT               Declares control section Z.
                            Both location counters are
                            initialized to zero. Z is im-
                            plicitly defined as a word
                            resolution address.
Q       EQU   HA(Z+4)       Equates Q to a halfword ad-
        :                   dress of Z+4 (words).
        :
```

WA Word Address

The word address function has the format

WA(address expression)

where WA identifies the function, and address expression is the symbol or expression that is to have word address resolution when assembled. If address expression is a constant, the value returned is the constant itself.

Example 21. WA Function

```
        :
        :
A       ASECT               Declares absolute section A
                            and sets its location counters
                            to zero.
        LW,3    Z1          Assembles instruction to be
                            stored in location 0.
B       LW,4    Z2          Assigns the symbol B the
                            value 1, with word address
        :                   resolution.
        :
C       EQU     BA(B)       Equates C to the value of B
        :                   with byte address resolution.
        :
F       EQU     WA(C)       Equates F to the value of C,
                            with word address resolution.
        :
        :
```

DA Doubleword Address

The doubleword address function has the format

DA(address expression)

where DA identifies the function, and address expression is the symbol or expression that is to have doubleword address resolution when assembled. If address expression is a constant, the value returned is the constant itself.

Example 22. DA Function

```
        :
        :
LI,5    DA(L(ALPHA))    The symbol ALPHA is stored
                        in the literal table and its
                        location is assembled into this
                        statement as a doubleword
        :               address.
        :
```

ABSVAL Absolute Value

This function converts a relocatable address into an absolute value (viz., address expression minus relocation bias). It has the format

ABSVAL(address expression)

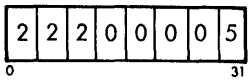
where ABSVAL identifies the function, and address expression is any valid expression containing only addresses and integers combined by addition or subtraction (no external or local forward references).

The absolute value of an address is evaluated according to the resolution; thus, the absolute value of a relocatable address, evaluated with word resolution, would result in a 17-bit address (the two bits specifying byte and halfword boundaries would be ignored). The absolute value of an external reference, a blank field, a null field, an integer, a character string, etc., is the same configuration as the item itself; e.g., ABSVAL('AXY') is the value 'AXY'.

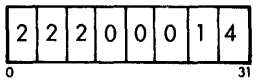
Example 23. ABSVAL Function

```
        :
        :
Q       CSECT   0          Declares control section
                           Q and sets location
                           counters to zero.
R       EQU     $+5        Equates R to the current value
                           of the execution location
                           counter plus 5 (i.e., to the
                           value 5 evaluated with
                           word resolution).
        :
        :
```

```
:
:
LI,2        ABSVAL(R)    Loads register 2 with
                         ABSVAL(R), which
                         is the value 5.
```




```
:
:
LI,2        ABSVAL(BA(R))
```



ADDRESS RESOLUTION

To the assembler an address represents an offset from the beginning of the program section in which it is defined.

Consequently, the assembler maintains in its symbol table not only the offset value, but an indicator that specifies whether the offset value represents bytes, words, halfwords, or doublewords. This indicator is called the "address resolution".

Address resolution is determined at the time a symbolic address is defined, in one of two ways:

1. Explicitly, by specifying an addressing function.

2. Implicitly, by using the address resolution of the execution location counter. (The resolution of the execution location counter is set by the ORG or LOC directives. If neither is specified, the address resolution is word.)

The resolution of a symbolic address affects the arithmetic performed on it. If A is the address of the leftmost byte of the fifth word, defined with word resolution, then the expression A + 1 has the value 6 (5 words + 1 word). If A is defined with byte resolution, then the same expression has the value 21 (20 bytes + 1 byte). See Example 24.

Forward and external references with addends are considered to be of word resolution when used without a resolution function in a generative statement or in an expression. Thus, a forward or external reference of the form

reference + 2

is implicitly

WA(reference +2)

Example 24. Address Resolution

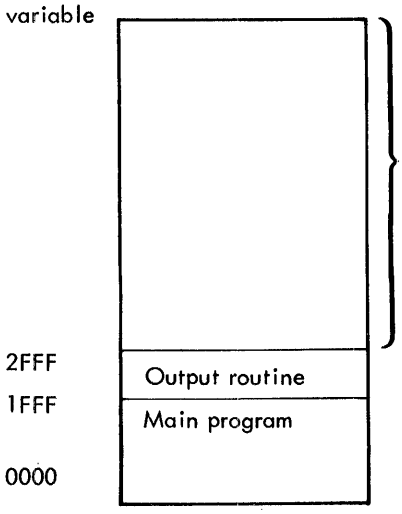
Location	Generated Code				
			CSECT		
00000			ORG	0	Sets value of location counters to zero with word resolution.
00000	FFFB	A	GEN,16	-5	Defines A as 0 with word resolution.
00000 2	0004	B	GEN,16	4	Defines B as 0 with word resolution.
00001	0000		GEN,16	BA(A)	Generates 0 with byte resolution.
00001 2	0002		GEN,16	BA(B)	Generates 2 with byte resolution.
00002	0001		GEN,16	HA(B)	Generates 1 with halfword resolution.
00002 2			ORG,1	$	Sets value of location counters to 10 with byte resolution.
00002 2	FFFF	F	GEN,16	-1	Defines F as 10 with byte resolution.
00003	000A		GEN,16	F	Generates 10 with byte resolution.
00003 2	000B		GEN,16	F+1	Generates 11 with byte resolution.
00004	0002		GEN,16	WA(F)	Generates 2 with word resolution.
00004 2	0002		GEN,16	WA(F+1)	Generates 2 with word resolution.
00005	0008		GEN,16	BA(WA(F+1))	Generates 8 with byte resolution.
00005 2	0003		GEN,16	WA(F)+1	Generates 3 with word resolution.
00006	000C		GEN,16	BA(WA(F)+1)	Generates 12 with byte resolution.
00006 2	000D		GEN,16	BA(WA(F)+1)+1	Generates 13 with byte resolution.

LOCATION COUNTERS

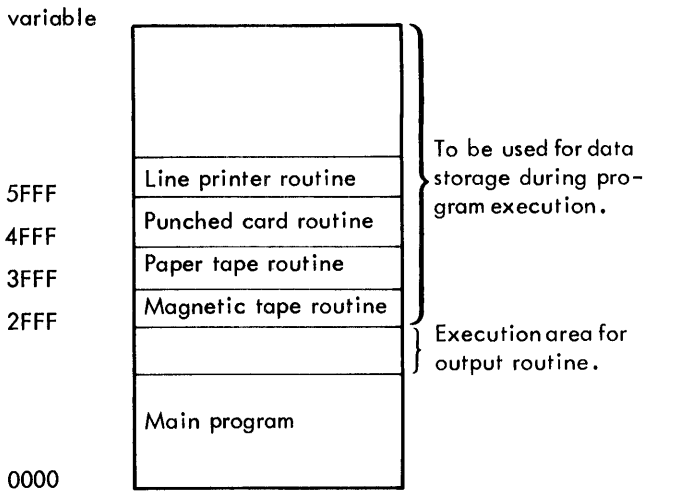
A location counter is a memory cell the assembler uses to record the storage location it assigned last and, thus, what location it should assign next. Each program has two location counters associated with it during assembly: the load location counter (referenced symbolically as $$) and the execution location counter (referenced symbolically as $). The load location counter contains a location value relative to the origin of the source program. The execution location counter contains a location value relative to the source program's execution base.

Essentially, the load location counter provides information to the loader that enables it to load a program or subprogram into a desired area of memory. The execution location counter, on the other hand, is used by the assembler to derive the addresses for the instructions being assembled. To express it another way, the execution location counter is used in computing the locations and addresses within the program, and the load location counter is used in computing the storage locations where the program will be loaded prior to execution.

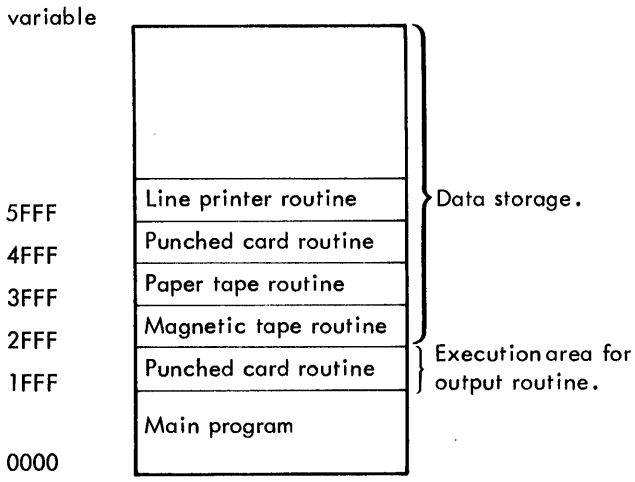
In the "normal" case both counters are stepped together as each instruction is assembled, and both contain the same location value. However, the ORG and LOC directives make it possible to set the two counters to different initial values to handle a variety of programming situations. The load location counter is a facility that enables systems programmers to assemble a program that must be executed in a certain area of core memory, load it into a different area of core, and then, when the program is to be executed, move it to the proper area of memory without altering any addresses. For example, assume that a program provides a choice of four different output routines: one each for paper tape, magnetic tape, punched cards, or line printer. In order to execute properly, the program must be stored in core as follows:



Each of the four output routines would be assembled with the same initial execution location counter value of 1FFF but different load location counter values. At run time this would enable all the routines to be loaded as follows:

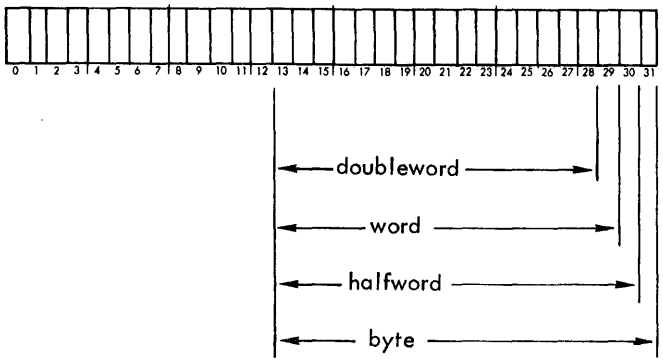


When the main program has determined which output routine is to be used, during program execution, it moves the routine to the execution area. No address modification to the routine is required since all routines were originally assembled to be executed in that area. If the punched card output routine were selected, storage would appear as:

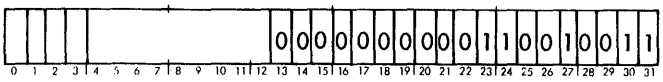


The user should not assume from this example that the execution location counter must be controlled in the manner indicated in order for a program to be relocated. By properly controlling the loader and furnishing it with a "relocation bias", any Meta-Symbol program, unless the programmer specifies otherwise, can be relocated into a memory area different than the one for which it was assembled. Most relocatable programs are assembled relative to location zero. To assemble a program relative to some other location, the programmer should use an ORG directive to designate the program origin. This directive sets both location counters to the same value. More information on program sectioning and relocatability is given at the end of this chapter.

Each location counter is a 19-bit value that the assembler uses to construct byte, halfword, word, and doubleword addresses:



Thus, if a location counter contained the value



it could be evaluated as follows:

Resolution	Hexadecimal Value
Byte	193
Halfword	C9
Word	64
Doubleword	32

The address resolution option of the ORG and LOC directives allows the programmer to specify the intrinsic resolution of the location counters. Word resolution is used as the intrinsic resolution if no specification is given. Address functions, as previously explained, are provided to override this resolution.

SETTING THE LOCATION COUNTERS

At the beginning of an assembly, Meta-Symbol automatically sets the value of both location counters to zero. The user can reset the location values for these counters during an assembly with the ORG and LOC directives. The ORG directive sets the value of both location counters. The LOC directive sets the value of only the execution location counter.

ORG Set Program Origin

The ORG directive sets both location counters to the location specified. This directive has the form

label	command	argument
[label$_1$,...,label$_n$]	ORG[,n]	[location]

where

label$_i$ are any valid symbols. Use of a label is optional. When present, it is defined as the value "location" and is associated with the first byte of storage following the ORG directive.

n is an evaluatable, integer-valued expression whose value is 1, 2, 4, or 8, specifying the address resolution for both counters as byte, halfword, word, or doubleword, respectively. If n is omitted, word resolution is assumed.

location is an evaluatable expression that results in an address or an integer. If location is an address, all attributes of location are substituted for $ and $$, and the intrinsic resolution of $ and $$ are then set to n. If location is an integer, $ and $$ remain in the current control section, but their value is set to "location" units at "n" resolution (see Example 25). If location is omitted, integer 0 is assumed.

The address resolution option of ORG may be used to change the intrinsic resolution specification to byte, halfword, or doubleword resolution. Thereafter, whenever intrinsic resolution is applicable, it will be that designated by the most recently encountered ORG directive. For example, whenever $ or $$ is encountered, the values they represent are expressed according to the currently applicable intrinsic resolution.

Example 25. ORG Directive

```
        :
AA      ORG,2   8       Sets the location counters to 8 halfwords (i.e., 4 words) and assigns that location,
                        with halfword intrinsic resolution, to the label AA.
        LW,2    INDEX   This instruction is assembled to be loaded into the location defined as AA. Thus, the
                        effect is the same as if the ORG directive had not been labeled and the label AA had
                        been written with the LW instruction.
        :
```

Example 26. ORG Directive

```
        :
Z       CSECT                   Designates section Z and sets the location counters to zero.

        ORG     Z + 4           Sets the location counters to Z + 4 with word resolution.

        :
A       LW,4    ANY             Assembles ANY with word resolution, and defines A with word resolution.

        :
        MBS,0   B               Forces a byte address. The type of address required by the command overrides the
                                intrinsic resolution of the symbol.

        LI,4    BA(ANY)         Assembles the symbol ANY as a byte address.

        :
```

LOC Set Program Execution

The LOC directive sets the execution location counter ($) to the location specified. It has the form

label	command	argument
[label$_1$, . . . , label$_n$]	LOC [,n]	[location]

where

label$_i$ are any valid symbols. Use of a label is optional. When present, it is defined as the value of location and is associated with the first byte of storage following the LOC directive.

n is an evaluatable, integer-valued expression whose value is 1, 2, 4, or 8, specifying the address resolution for the execution location counter as byte, halfword, word, or doubleword, respectively. If n is omitted, word resolution is assumed.

location is an evaluatable expression that results in an address or an integer. If location is an address, all attributes of location are substituted for $, and the intrinsic resolution of $ is then set to n. If location is an integer, $ remains in the current control section, but its value is set to "location" units at "n" resolution (see Example 25). If location is omitted, integer 0 is assumed.

Except that it sets only the execution location counter, the LOC directive is the same as ORG.

Example 27. LOC Directive

```
        :
PDQ     ASECT
        ORG     100     Sets the execution location
                        counter and load location
                        counter to 100.

        LOC     1000    Sets the execution location
                        counter to 1000. The load
                        location counter remains at
                        100.
```

Subsequent instructions will be assembled so that the object program can be loaded anywhere in core relative to the original of the program. For example, a relocation bias of 500 will cause the loader to load the program at 600 (500 + 100). However, the program will execute properly only after it has been moved to location 1000.

BOUND Advance Location Counters to Boundary

The BOUND directive advances both location counters, if necessary, so that the execution location counter is a byte multiple of the boundary designated. The form of this directive is

label	command	argument
	BOUND	boundary

where boundary may be any evaluatable expression resulting in a positive integer value that is a power of 2.

Halfword addresses are multiples of two bytes, full-word addresses are multiples of four bytes, and doubleword addresses are multiples of eight bytes.

When the BOUND directive is processed, the execution location counter is advanced to a byte multiple of the boundary designated and then the load location counter is advanced the same number of bytes. When the BOUND directive results in the location counters being advanced, zeros are generated in the byte positions skipped. Since BOUND may generate data, it should never be used in declaring a blank common section for linkage with FORTRAN programs (F4:COM DSECT).

Example 28. BOUND Directive

```
      BOUND 8        Sets the execution location
                     counter to the next higher
                     multiple of 8 if it is not al-
                     ready at such a value.

For instance, the value of the execution location counter for the current section might be 3 words (12 bytes). This directive would advance the counter to 4 (16 bytes), which would allow word and doubleword, as well as byte and halfword, addressing.
```

RES Reserve an Area

The RES directive enables the user to reserve an area of core memory.

label	command	argument
[label$_1$, . . . , label$_n$]	RES[,n]	[expression]

where

label$_i$ are any valid symbols. Use of a label is optional. When present, the label is defined as the current value of the execution location counter and identifies the first byte of the reserved area.

n is an evaluatable, integer-valued expression designating the size in bytes of the units to be reserved. The value of n must be non-negative. Use of n is optional; if omitted, its value is assumed to be four bytes.

expression is an evaluatable, integer-valued expression designating the number of units to be reserved. Its value may be positive or negative. If expression is omitted, zero is assumed.

When Meta-Symbol encounters an RES directive, it modifies both location counters by the specified number of units.

Example 29. RES Directive

```
      :
      ORG    100      Sets location counters to 100.
A     RES,4  10       Defines symbol A as location 100 and advances the location counters by 40 bytes (10 words)
                      changing them to 110.
      LW,4   VALUE    Assigns this instruction the current value of the location counters; i.e., 110.
      :
```

PROGRAM SECTIONS

An object program may be divided into program sections, which are groups of statements that usually have a logical association. For example, a programmer may specify one program section for the main program, one for data, and one for subroutines.

PROGRAM SECTION DIRECTIVES

A program section is declared by use of one of the program section directives given below. These directives also declare whether a section is absolute or relocatable. The list gives only a brief definition of these directives; their use will be made clear by successive statements and examples in this chapter.

ASECT specifies that generative statements[t] will be assembled to be loaded into absolute locations. The location counters are set to absolute zero.

CSECT declares a new control section (relocatable). Generative statements will be assembled to be loaded into this relocatable section. The location counters are set to relocatable zero.

DSECT declares a new, dummy control section (relocatable). Generative statements will be assembled to be loaded into this relocatable section. The location counters are set to relocatable zero.

[t] Generative statements are those that produce object code in the assembled program.

PSECT declares a new control section (relocatable) which will begin on a multiple of 512 (200_{16}) words. Generative statements will be assembled to be loaded into this relocatable section. The location counters are set to relocatable zero. PSECT differs from CSECT only in that the loader will align a PSECT section on a page (512-word) boundary.

USECT designates which previously declared section Meta-Symbol is to use in assembling generative statements.

The program section directives have the following form:

label	command	argument
[label$_1$, ..., label$_n$]	ASECT	
[label$_1$, ..., label$_n$]	CSECT	[expression]
label	DSECT	[expression]
[label$_1$, ..., label$_n$]	PSECT	[expression]
[label$_1$, ..., label$_n$]	USECT	name

where

label$_i$ is any valid symbol. The labels are assigned the value of the execution location counter immediately after the directive has been processed. For ASECT, the value of the label becomes absolute zero. For CSECT, DSECT, and PSECT, the label value becomes relocatable zero in the appropriate program section. The label on a USECT directive is defined as the value of the execution location counter in the current control section. The label on ASECT, CSECT, PSECT, and USECT may be externalized by appearing in a DEF directive so that the label can be referred to by other programs. For DSECT, label is implicitly an external definition, because dummy sections are typically used in order that they can be referred to by other programs.

expression is an evaluatable, integer-valued expression whose value must be from 0 to 3. This value, applicable only to CSECT, DSECT, and PSECT, designates the type of memory protection to be applied to these sections. In the following list, "read" means a program can obtain information from the protected section; "write" means a program can store information into a protected section; and "access" means the computer can execute instructions stored in the protected section.

Value	Memory Protection Feature
0	read, write, and access permitted
1	read and access permitted
2	read only permitted
3	no access, read, or write permitted

The use of expression is optional. When it is omitted, the assembler assumes the value 0 for the entry. It may not contain an external reference.

name is the label defined in a previously declared section.

ABSOLUTE SECTION

Although ASECT may be used any number of times, the assembler produces only one combined absolute section, using the successive specifications of the ASECT directives.

RELOCATABLE CONTROL SECTIONS

A single assembly may contain from one to 127 relocatable control sections, which Meta-Symbol numbers sequentially. At the beginning of each assembly Meta-Symbol sets both the execution and load location counters to relocatable zero, with word address resolution, in relocatable control section 1. Control section 1 is opened by generating values in, or referencing or manipulating the initial location counters, or by declaring the first CSECT, DSECT, or PSECT directive.

The execution of a CSECT, DSECT, or PSECT directive always opens a new section. Therefore, if control section 1 has been opened by generating values in, or referencing or manipulating the initial location counters, the first CSECT, DSECT, or PSECT opens control section 2. For example, these three program segments

```
DATA 5          DEF     SORT        ORG 500
CSECT   HERE    EQU     $           CSECT
  .               .                   .
  .     and       CSECT       and       .
END               .                   .
                  .                 END
                END
```

each produce two relocatable control sections, one implicit (control section 1), and one explicit (control section 2); whereas

```
VALUE   EQU 5           INPUT   CNAME
        REF     OUTPUT          PROC
        CSECT           and       .
          .                       .
          .                     PEND
        END
                                CSECT
                                  .
                                  .
                                END
```

each contains only one relocatable section (control section 1). The statements preceding the CSECT do not open control section 1 because they do not generate values in, or reference or manipulate the initial location counters.

SAVING AND RESETTING THE LOCATION COUNTERS

Since there is only one pair of location counters, Meta-Symbol does the following when a new section is declared (ASECT, CSECT, DSECT, or PSECT) (see Example 30):

1. Saves the current value of the execution location counter ($) in the SAVED $ TABLE.
2. Compares the value of the load location counter ($$) with the value previously saved for the section in the SAVED MAXIMUM $$ TABLE, if assembling a relocatable control section, and saves the higher value.

The control section to which the saved values are associated is determined from the location counters. The counters have the format:

Execution Location Counter

RS	CS#	ADDR	VALUE

Load Location Counter

RS	CS#	ADDR	VALUE

where

RS specifies the resolution (BA, HA, WA, DA).

CS# specifies the control section number and the type of section (0 = absolute, X'1' – X'7F' = relocatable).

ADDR specifies that the value is an address.

VALUE is the value of the counter for the section.

After Meta-Symbol has saved the value of the execution location with the value in the SAVED MAX. $$ TABLE, it resets both location counters to zero in the new control section.

RETURNING TO A PREVIOUS SECTION

A programmer may write a group of statements for one section, declare a second section containing various statements, and then write additional statements to be assembled as part of the first section. This capability is provided by the following:

1. The SAVED $ TABLE, which contains the most recent value of the execution location counter for each section.
2. The symbol table entry, which specifies a control section number for symbols defined as addresses. The entry has the same format as the location counters.

RS	CS#	ADDR	VALUE

where

RS specifies the resolution (BA, HA, WA, DA).

CS# indicates the control section in which the label is defined (0 = absolute, X'1' – X'7F' = relocatable).

Example 30. Program Sectioning

Current Location Counters				Program			SAVED $			SAVED MAX. $$	
$	Section	$$	Section				ABS	CS1	CS2	CS1	CS2
0	ABS	0	ABS	NUMBERS	ASECT		0				
300		300			ORG	300					
:		:			:						
350		350									
0	CS1	0	CS1	RANDOM	CSECT		350				
:		:			:						
100		100									
0	CS2	0	CS2	DUMMY	DSECT			100		100	
:		:			:						
200		200			END						200

The ASECT directive sets both location counters to absolute zero; the ORG statement resets the counters to 300. Subsequent generative statements will be assembled to be loaded into absolute locations. When CSECT is encountered, Meta-Symbol saves the value of the execution location counter in the SAVED $ TABLE. The value of the load location counter is not saved. Meta-Symbol then resets the counters to relocatable zero in control section 1 and assembles generative statements to be loaded as part of this section. The DSECT directive declares a new relocatable section. Meta-Symbol saves the counters for control section 1 in the appropriate tables, resets the counters to relocatable zero in control section 2, and assembles generative statements to be loaded in this section. The END directive causes Meta-Symbol to save the value of the load location counter for control section 2. The values in the SAVED MAX. $$ TABLE are used by the loader in allocating memory. Note that the use of ORG (and LOC) when it changes the current section also causes the current value of the execution location counter to be saved. Additionally, ORG compares the current value of the load location counter with the value in the SAVED MAX. $$ TABLE and saves the higher value.

ADDR specifies that the value is an address.

VALUE is the assigned symbol value.

3. The USECT directive (see Examples 31 through 34), which specifies a previously declared section that Meta-Symbol is to use in assembling generative statements.

There is only one absolute section and although ASECT may be used any number of times, the SAVED $ value of the absolute section is always that of the last designated ASECT.

Example 31. USECT Directive

Current Location Counters				Program			SAVED $			SAVED MAX. $$	
$	Section	$$	Section				ABS	CS1	CS2	CS1	CS2
0	CS1	0	CS1		PSECT			0			
10		10		TRAP	.						
100		100		LAST	.						
0	CS2	0	CS2		DSECT			100		100	
.		.			.						
.		.			.						
200		200			.						
100	CS1	100	CS1		USECT	TRAP			200		200
					.						
					.						
					END						

When USECT TRAP is encountered, Meta-Symbol determines the control section from one symbol table entry for TRAP,

WA	1	ADDR	10

checks the SAVED $ TABLE FOR CS1, and copies this saved value (100) into both location counters.

Example 32. USECT Directive

Current Location Counters				Program			SAVED $			SAVED MAX. $$	
$	Section	$$	Section				ABS	CS1	CS2	CS1	CS2
0	ABS	0	ABS		ASECT		0				
500		500			ORG	500					
					.						
					.						
520		520		TABLE	DATA	6					
600		600			.						
					.						
0	CS1	0	CS1		CSECT		600				
100		100			.						
					.						
0	ABS	0	ABS		ASECT		0	100		100	
700		700			ORG	700					
					.						
					.						
800		800			.						
0	CS2	0	CS2		CSECT		800				
200		200			.						
					.						
800	ABS	800	ABS		USECT	TABLE			200		200

When USECT TABLE is encountered, Meta-Symbol determines the control section from the symbol table entry for TABLE,

WA	0	ADDR	520

checks the SAVED $ TABLE for the absolute section, and copies this saved value (800) into both location counters.

Example 33. Program Sectioning

Current Location Counters				Program			SAVED $			SAVED MAX. $$	
$	Section	$$	Section				ABS	CS1	CS2	CS1	CS2
0	CS1	0	CS1		CSECT			0			
1000	CS1	0	CS1	FILE	LOC	1000					
1100	CS1	100	CS1	LAST	:						
0	CS2	0	CS2		CSECT			1100		100	
200	CS2	200	CS2		:						
1100	CS1	1100	CS1		USECT	FILE			200		200
1200	CS1	1200	CS1		:						
0	ABS	0	ABS		ASECT			1200		1200	
					:						

The LOC directive advances only the execution location counter. When USECT FILE is encountered, Meta-Symbol sets both counters to the value of the saved execution location counter for CS1 (1100). The ASECT directive causes Meta-Symbol to save the value of the execution location counter for CS1 and to replace the SAVED MAX. $$ value (100) with 1200.

Example 34. Program Sectioning

Current Location Counters				Program			SAVED $			SAVED MAX. $$	
$	Section	$$	Section				ABS	CS1	CS2	CS1	CS2
0	ABS	0	ABS	CALL	ASECT		0				
100	ABS	100	ABS		ORG	100					
					:						
200		200		MAIN	LW,4	6					
0	CS1	0	CS1		CSECT		200				
					:						
50		50		HERE	EQU	$					
100		100			:						
0	CS2	0	CS2		CSECT			100		100	
FF	CS2	FF	CS2		:						
50	CS1	100	CS2		LOC	HERE			100		
300	CS1	350	CS2		:						
200	ABS	200	ABS		USECT	MAIN		300			350
400		400			:						
300	CS1	300	CS1		USECT	HERE	400				
500		500			:						
400	ABS	400	ABS		USECT	CALL		500		500	

The statement HERE EQU $ defines HERE as the current value of the execution location counter (50). When the LOC HERE statement in CS2 is encountered, Meta-Symbol sets the value of the execution location counter to 50 in CS1. Subsequent statements will be assembled to be executed as part of CS1 but will be loaded as part of CS2. The USECT MAIN statement saves the value of the execution location counter for CS1 and the value of the load location counter for CS2. The USECT HERE statement causes the counters to be set to the saved value of the execution location counter for CS(300).

DUMMY SECTIONS

In any load module, dummy sections of the same name must be the same size and have the same memory protection. (The restriction on the size of dummy sections of the same name is only enforced by certain Xerox loaders; otherwise, the largest is used.) Dummy sections provide a means by which more than one subroutine may load the same section. For example, assume that three subroutines contain the same dummy constant section:

```
SUBR 1          SUBR 2          SUBR 3
   :               :               :
CONST  DSECT    CONST  DSECT    CONST  DSECT
   :               :               :
   :               :               :
   END             END             END
```

Even though more than one of the subroutines may be required in one load module, the loader will load the dummy section only once, and any of the subroutines may reference the data.

PROGRAM SECTIONS AND LITERALS

When Meta-Symbol encounters the END statement, it generates all literals declared in the assembly. The literals are generated at the current location (word boundary) of the currently active program section (see Example 35).

Example 35. Program Sections and Literals

Example 35a:

```
AREA    CSECT
          :   }     Declares literals.
BAY     CSECT
          :   }     Declares literals.
        END         Generates literals as part of section BAY.
```

Example 35B:

```
GATE    CSECT
          :   }     Declares literals.
        ASECT
        ORG     100
        END         Generates literals beginning in absolute location 100.
```

Example 35c:

```
REAL    CSECT
          :   }     Declares literals.
LAST    RES
LOOP    CSECT   0
          :   }     Declares literals.
        USECT   REAL
        END         Generates literals as part of section REAL immediately following the location
                    assigned to LAST.
```

Example 35d:

```
NOW     DSECT
          :   }     Declares literals.
HERE    RES     25
          :   }     Declares literals.
        ORG     HERE
        END         Generates literals as part of section NOW, beginning at location HERE.
```

4. DIRECTIVES

A directive is a command to the assembler that can be combined with other language elements to form statements. Directive statements, like instruction statements, have four fields: label, command, argument, and comments.

An entry in the label field is required for the directives: CNAME, COM, FNAME, and S:SIN. The label field entries identify the generated command or procedure. The location counters are not altered by these directives.

Optional labels for the EQU and SET directives are defined as the value of the evaluated argument field, which may be a single value or a list of values.

Optional labels for the directives ORG and LOC are defined as the value to which the execution location counter is set by the directive.

If any of the directives DATA, GEN, RES, TEXT, or TEXTC are labeled, the label(s) is defined as the current value of the execution location counter, and identifies the first byte of the area generated. These directives alter the location counters according to the contents of the argument field.

Labels for the directives ASECT, CSECT, DSECT, PSECT, USECT, and DO1 identify the first word of the area affected by the directive. A label field is required for DSECT.

A label for the END directive identifies the location immediately following the last literal generated in the literal table. This is explained further under the END directive in this chapter.

A label(s) on the following directives will be ignored unless it is the target label of a GOTO search: BOUND, CDISP, CLOSE, DEF, DISP, ELSE, ERROR, FDISP, FIN, GOTO, LIST, LOCAL, OPEN, PAGE, PCC, PEND, PROC, PSR, PSYS, REF, S:RELP, SOCW, SPACE, SREF, SYSTEM, TITLE.

Labels for the DO and WHILE directives are handled in a special manner explained later.

The command field entry is the directive itself. If this field consists of more than one subfield, the directive must be in the first subfield, followed by the other required entries.

Argument field entries vary and are defined in the individual discussion of each directive.

A comments field entry is optional.

The END, LOCAL, OPEN, and CLOSE directives are the only directives unconditionally executed. They are processed even if they appear within the range of a GOTO search or an inactive DO-loop.

The Meta-Symbol language includes these directives:

Assembly Control

ASECT[t]	LOC[t]	DO1
CSECT[t]	BOUND[t]	DO
DSECT[t]	RES[t]	WHILE
PSECT[t]	SYSTEM	ELSE
USECT[t]	END	FIN
ORG[t]	GOTO	

Symbol Manipulation

EQU	OPEN	REF
SET	CLOSE	SREF
LOCAL	DEF	

Data Generation

GEN	TEXT	SOCW
COM	TEXTC	
DATA	S:SIN	

Listing Control

PAGE	PCC	DISP
SPACE	PSR	CDISP[tt]
TITLE	ERROR	FDISP[tt]
LIST	PSYS	

Procedure Control (These directives are described in Chapter 5.)

CNAME	PROC	S:RELP
FNAME	PEND	

[t] Discussed in Chapter 3.

[tt] Discussed in Chapter 5.

In the format diagrams for the various directives that follow, brackets indicate optional items.

ASSEMBLY CONTROL

SYSTEM Include System File

SYSTEM directs the assembler to retrieve the indicated file from the system storage medium, and to include it in the program being assembled. The SYSTEM directive has the form

label	command	argument
	SYSTEM	name

where name is either an actual file name (less than 32 characters), or one of the special instruction set names discussed below. When an actual file name is specified, Meta-Symbol reads the file from the appropriate account (see the AC option, Chapter 7) and inserts it at that point in the source program. The file is considered to be terminated when an END directive (discussed below) is encountered.

Any number of SYSTEM directives may be included in a program. System files may contain additional SYSTEM directives, allowing a structured hierarchy of library source files. Meta-Symbol does not protect against circular or repetitive calls for the same system.

Definitions of the Sigma machine instructions are contained in the system file, SIG7FDP. This file is invoked, not by name, but by one of the mnemonics for a particular instruction subset, as listed below. When a valid subset of SIG7FDP is specified, Meta-Symbol assigns an identifying value to the intrinsic symbol, S:IVAL, which is available to the SIG7FDP file, as well as to the main program. It then processes the file as described above.

The valid instruction set mnemonics, their meaning, and the corresponding values of S:IVAL are as shown in Table 5.

Table 5. Valid Instruction Set Mnemonics

Name	Instruction Set	S:IVAL
SIG9	Basic Sigma 9.	X'1E'
SIG9P	Sigma 9 with Privileged Instructions.	X'1F'
SIG8	Basic Sigma 8	X'1C'
SIG8P	Sigma 8 with Privileged Instructions.	X'1D'
SIG7	Basic Sigma 7.	X'08'
SIG7F	Sigma 7 with Floating-Point Option.	X'0C'
SIG7D	Sigma 7 with Decimal Option.	X'0A'

Table 5. Valid Instruction Set Mnemonics (cont.)

Name	Instruction Set.	S:IVAL
SIG7P	Sigma 7 with Privileged Instructions.	X'09'
SIG7FD	Sigma 7 with Floating-Point and Decimal Option.	X'0E'
SIG7FP	Sigma 7 with Floating-Point Option and Privileged Instructions.	X'0D'
SIG7DP	Sigma 7 with Decimal Option and Privileged Instructions.	X'0B'
SIG7FDP	Sigma 7 with Floating-Point, Decimal Option, and Privileged Instructions.	X'0F'
SIG6	Basic Sigma 6.	X'0A'
SIG6F	Sigma 6 with Floating-Point Option.	X'0E'
SIG6P	Sigma 6 with Privileged Instructions.	X'0B'
SIG6FP	Sigma 6 with Floating-Point Option and Privileged Instructions.	X'0F'
SIG5	Basic Sigma 5.	X'00'
SIG5F	Sigma 5 with Floating-Point Option.	X'04'
SIG5P	Sigma 5 with Privileged Instructions.	X'01'
SIG5FP	Sigma 5 with Floating-Point Option and Privileged Instructions.	X'05'

Example 36. SYSTEM Directive

Assume a square root subroutine, identified as SQRT, is on the system storage media and that it is to be assembled as part of the object program. The program uses the basic instruction set. These directives would appear in the source program:

```
        SYSTEM      SIG7
          .
          .
          .
        SYSTEM      SQRT
          .
          .
          .
```

END End Assembly

The END directive terminates the assembly of a system called by the SYSTEM directive as well as the assembly of the main program. It has the form

label	command	argument
[label$_1$,...,label$_n$]	END	[expression]

where

label$_i$ are one or more valid symbols. When present, the label or labels are assigned (i.e., associated with) the location immediately following the last location in the literal table.

expression is an optional expression that designates a location to be transferred to after the program has been loaded. "expression" may be external.

As explained under "Program Sections and Literals" at the end of Chapter 3, Meta-Symbol generates all literals declared in the assembly as soon as it encounters the END statement. The literals are generated in the location immediately following the currently active program section (see Example 35). If the END directive is labeled, the label or labels are associated with the first location immediately following the literal table. Thus, in Example 35c, a label on the END statement would be associated with the same location identified as LOOP, the first location in control section 2.

END is processed even if it appears within the range of a GOTO search or an inactive DO-loop.

Example 37. END Directive

```
            SYSTEM      SIG7
              .
              .
CONTROL     CSECT
              .
              .
START       LW,5        TEST
              .
              .
            END         START
```

DO1 Iteration Control

The DO1 directive defines the beginning of a single statement assembly iteration loop. It has the form

label	command	argument
[label$_1$,...,label$_n$]	DO1	[expression]

where

label$_i$ are one or more valid symbols. Use of labels is optional. When present, they are defined as the current value of the execution location counter and identify the first byte generated as a result of DO1 iteration.

expression is an evaluatable, integer-valued expression that represents the number of times the line immediately following is to be assembled. There is no limit to the number of times the line may be assembled. If the expression is omitted or negative, zero is assumed.

If the expression in the DO1 directive is not evaluatable, Meta-Symbol produces an error notification, and processes the DO1 directive as if the expression had been evaluated as zero.

Example 38. DO1 Directive

The statements

```
    .
    .
    DO1       3
    AW,4      C
    .
    .
```

at assembly time would generate in-line machine code equivalent to the following lines:

```
    .
    .
    AW,4      C
    AW,4      C
    AW,4      C
    .
    .
```

The line following the DO1 directive should not be continued. If it is desired to skip or to iteratively assemble a statement containing continuation lines, a DO/FIN group should be used in place of DO1.

It is not possible to skip or repeat an END directive with a DO1; an attempt to do so causes an error diagnostic.

A LOCAL directive is unconditionally executed; it will not be skipped by DO1.

If the iteration count of a DO1 is greater than one, the next line may not contain another DO1 directive, nor a SYSTEM directive. Such a case causes an error diagnostic, and the initial DO1 directive is ignored.

GOTO Conditional Branch

The GOTO directive enables the user to conditionally alter the sequence in which statements are assembled. The GOTO directive has the form

label	command	argument
	GOTO[,k]	label$_1$[,...,label$_n$]

where

k is an evaluatable, integer-valued expression. If k is omitted, 1 is assumed.

label$_i$ are unsubscripted forward references.

A GOTO statement is processed at the time it is encountered during the assembly. Meta-Symbol evaluates the expression k and resumes assembly at the line that contains a label corresponding to the kth label in the GOTO argument field. The labels must refer to lines that follow the GOTO directive. If the value of k does not lie between 1 and n, Meta-Symbol resumes assembly at the line immediately following the GOTO directive. An error notification is given if the value of k is greater than n.

The target label of a GOTO search may be embedded in a list of labels; it will be recognized and will terminate the skip. A label will not be recognized if it is subscripted. A GOTO to a local symbol must find its target before a PEND, END, or LOCAL directive is encountered; if not, an error notification is given. Within a procedure, label$_j$ may be passed from the procedure reference line into the GOTO argument field, but the target label must physically appear within the procedure definition; it may not be passed from the reference line.

While Meta-Symbol is searching for the statement whose label corresponds to the kth label in the GOTO list, it operates in a skipping mode during which it ignores all machine-language instructions and all directives except END, LOCAL, OPEN, and CLOSE.

Skipped statements are produced on the assembly listing in symbolic form, preceded by an *S*.

When Meta-Symbol encounters the first of a logical pair of directives[t] while in the skipping mode, it suspends its search for the label until the other member of the pair is encountered. Then it continues the search. Thus, while in skipping mode, Meta-Symbol does not recognize labels that are within procedure definitions or iteration loops. It is not possible, therefore, to write a GOTO directive that might branch into a procedure definition, a DO/FIN loop, or a WHILE/FIN loop[tt]. Furthermore, it is not permissible to write a GOTO directive that might branch out of a procedure definition. If such a case occurred, Meta-Symbol would encounter a PEND directive before its search was satisfied, would produce an error notification, and would terminate the search for the label.

Example 39. GOTO Directive

```
      :
A     SET       3
      :
      :
      GOTO,A    H,K,M     Begins search for label M.
      :
      :
H     DO        5         Suppresses search for label M.
      :
      :
M     EQU       5+8       This M is not recognized
      :                   because it is within an
      :                   iteration loop.
      :
      FIN                 Terminates suppression and
                          continues search.
      :
      :
M     LW,A      BETA      Search is completed when
                          label M is found.
```

[t]Certain directives must occur in pairs: PROC/PEND, DO/FIN, and WHILE/FIN.

[tt]It is legal, however, to terminate a DO or WHILE loop by branching past the associated FIN.

WHILE/ELSE/FIN Iteration Control

The WHILE directive defines the beginning of an iteration loop; ELSE and FIN define the end of the loop. These directives have the form

label	command	argument
[label$_1$, ..., label$_n$]	WHILE	[expression]
	ELSE	
	FIN	

where

label$_j$ are one or more valid symbols. Use of labels is optional. When present, they are initially assigned the value zero and incremented by one each successive time through the loop.

expression is an evaluatable, integer-valued expression that controls processing of the WHILE loop. If the expression is greater than zero, the WHILE loop will be processed; otherwise, it will not. If expression is omitted, zero is assumed.

Figure 2 illustrates the logical flow of a WHILE/ELSE/FIN loop.

The assembler processes each WHILE as follows:

1. Establishes an internal counter and defines its value as zero.
2. If one or more labels are present for the WHILE directive, sets their value to zero.
3. If the WHILE is within a procedure, replaces any occurrence of LF, CF, AF, AFA, or NAME references in the expression with their current value. The resulting expression is then saved.
4. Evaluates the saved expression.
5. If the value of the saved expression is less than or equal to zero and this is the first time the expression has been evaluated, discontinues assembly until an ELSE or FIN directive is encountered.
   a. If an ELSE directive is encountered, assembles statements following it until a FIN directive is encountered.
   b. When the FIN directive is encountered, terminates control of the WHILE loop and resumes assembly at the next statement.

   If the value of the saved expression is less than or equal to zero and this is not the first time the expression has been evaluated (i.e., for the second and subsequent times through the WHILE loop), terminates control of the WHILE loop and resumes assembly at the statement following the FIN directive.

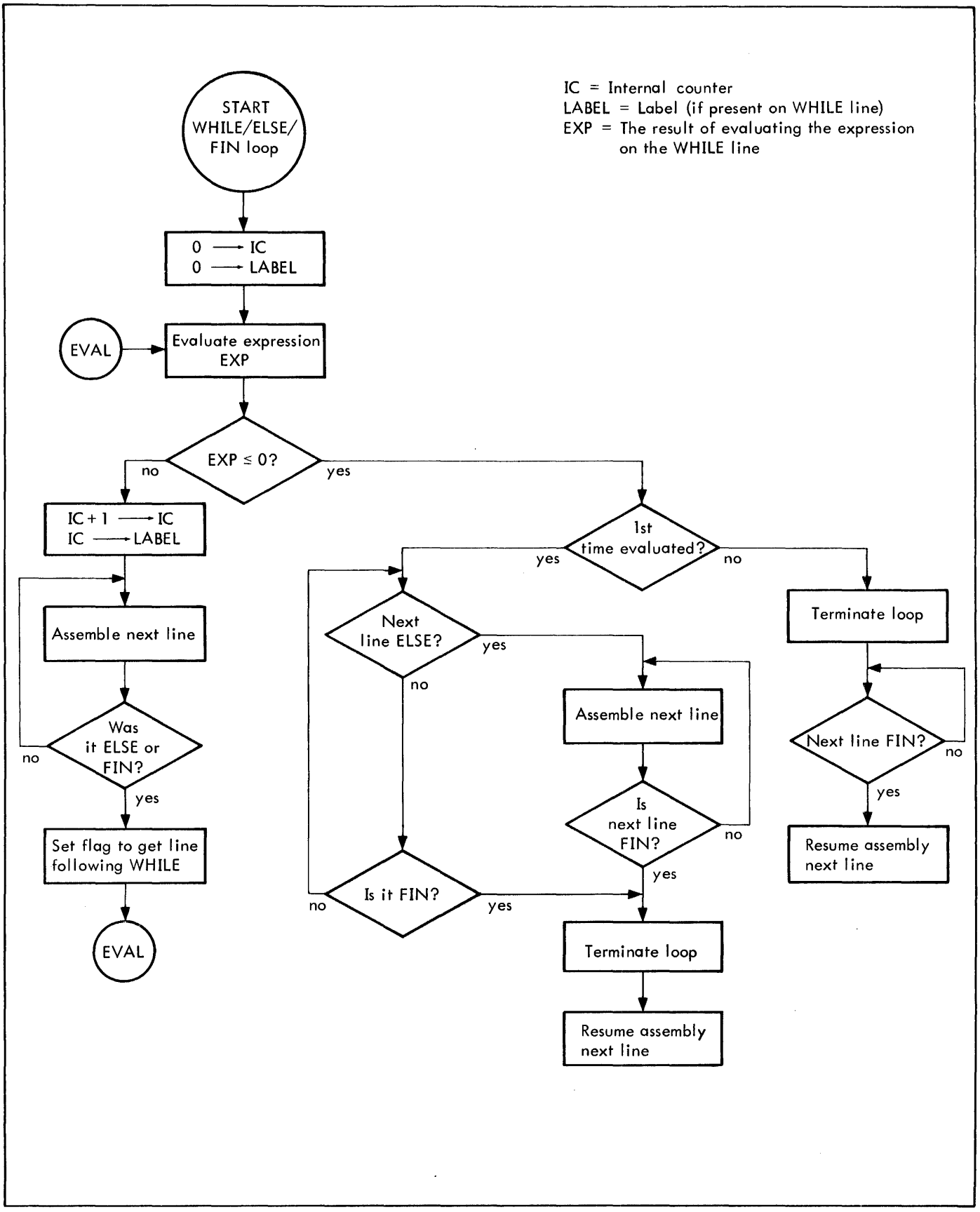


Figure 2. Flowchart of WHILE/ELSE/FIN Loop

6. If the value of the saved expression is greater than zero, increments the value of the internal counter by 1, sets the value of the label or labels (if present) to the new value of the counter, and continues assembly until an ELSE or FIN directive is encountered, then resumes the assembly at step 4. See Example 40.

If the expression is not evaluatable, Meta-Symbol sets the internal counter to the value zero, produces an error notification, and processes the WHILE directive as if the expression had been evaluated as zero.

The WHILE label is redefinable, and its value may be changed via a SET directive during the processing of the WHILE loop. Notice, however, that prior to each pass through the loop, the value of the label or labels is set to the value of the internal counter. Any symbols in the WHILE expression that are redefinable via SET may also be changed within the loop. Since the expression is reevaluated prior to each execution of the loop, such usage must be employed carefully.

WHILE directives may be nested within WHILE- and DO-loops. See Example 42.

Meta-Symbol assemblies involve various "levels". The main program is arbitrarily defined as level 0. A procedure invoked at level 0 is executed at level 1. If a procedure is invoked at level 1, it is executed at level 2; etc. A WHILE loop must be completely contained on a single program level (see Example 45 under DO directive).

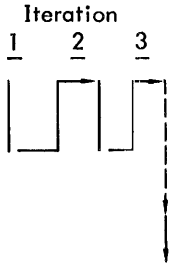
Example 40. WHILE/ELSE/FIN Directives

```
        :
B   SET     0
        :
I   WHILE   B>5     Expression is 0, so skips to ELSE.
        :
    ELSE            Resumes assembly following ELSE
                    and continues, ignores FIN, and
                    leaves control of WHILE.
        :
    FIN
        :
```

Example 41. WHILE/ELSE/FIN Directives

In this example, the dashed vertical line indicates statements are skipped; solid vertical lines indicate statements are assembled. The numbers above the vertical lines specify which iteration of the WHILE-loop is in process.

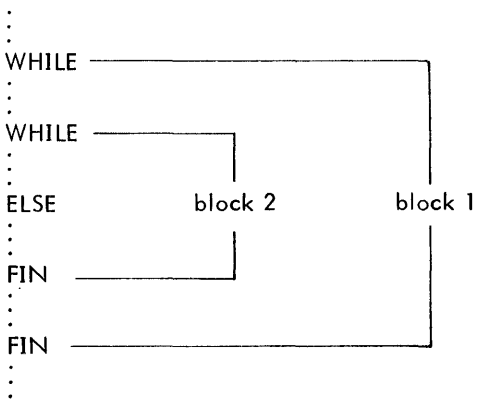
```
        :
A   WHILE   A<2
        :
    ELSE
        :
    FIN
        :
```



When WHILE is first encountered, A is set to zero. Since A is less than 2, the expression is true and has the value 1; therefore, the internal counter is incremented by 1, its new value (1) is assigned to A, and the loop is executed as far as the ELSE directive. Then control is returned to the WHILE directive where the expression is reevaluated. The current value of A is 1, so the expression is true and has the value of 1. The internal counter is again incremented by 1 and its new value (2) is assigned to A. The loop is assembled as far as the ELSE directive, and then control is returned to the WHILE directive where the expression is reevaluated. Since the current value of A is 2, the expression is no longer true, so statements are skipped until the FIN directive is encountered. Then assembly continues.

Example 42. WHILE/ELSE/FIN Directives

An iteration block within an iteration block:



Example 43. WHILE/FIN Directives

```
        :
I   WHILE       I<2
    GEN,32      I
    FIN
        :
```

This sequence generates the values 1 and 2.

DO/ELSE/FIN Iteration Control

The DO directive defines the beginning of an iteration loop; ELSE and FIN define the end of an iteration loop. These directives have the form

label	command	argument
[label$_1$,...,label$_n$]	DO	[expression]
	ELSE	
	FIN	

where

label$_i$ are valid symbols. Use of one or more labels is optional. When present, each is initially

assigned the value zero and incremented by one each successive time through the loop.

expression is an evaluatable, integer-valued expression that represents the count of the number of times the DO-loop is to be processed. If expression is omitted or is less than zero, zero is assumed.

Figure 3 illustrates the logical flow of a DO/ELSE/FIN loop.

The assembler processes each DO-loop as follows:

1. Establishes an internal counter and defines its value as zero.
2. If one or more labels are present on the DO line, sets their value to zero.
3. Evaluates the expression that represents the count.
4. If the count is less than or equal to zero, discontinues assembly until an ELSE or FIN directive is encountered.
   a. If an ELSE directive is encountered, assembles statements following it until a FIN directive is encountered.
   b. When the FIN directive is encountered, terminates control of the DO-loop and resumes assembly at the next statement.
5. If the count is greater than zero, processes the DO-loop as follows:
   a. Increments the internal counter by 1.
   b. If one or more labels are present on the DO line, sets them to the new value of the internal counter.
   c. Assembles all lines encountered up to the first ELSE or FIN directive.
   d. Repeats steps 5a through 5c until the loop has been processed the number of times specified by the count.
   e. Terminates control of the DO-loop and resumes assembly at the statement following the FIN.

In summary, there are two forms of iterative loops as shown below.

```
Form 1.   DO or WHILE  ┐
          :            ├  block 1
          :            ┘
          ELSE         ┐
          :            ├  block 2
          :            ┘
          FIN

Form 2.   DO or WHILE  ┐
          :            ├  block 1
          :            ┘
          FIN
```

If the expression in a DO directive is evaluated as a positive, nonzero value n, then in either form block 1 is repeated n times and assembly is resumed following the FIN.

If the expression in a WHILE directive is initially evaluated as a positive, nonzero value n, then in either form block 1 is assembled and the expression in the WHILE directive is reevaluated. This process continues until the evaluation of the expression in the WHILE directive no longer provides a positive, nonzero value, at which time control of the WHILE loop is terminated, and assembly resumes following the FIN.

If the expression in the DO directive is evaluated as a negative or zero value, then in

Form 1: block 1 is skipped, block 2 is assembled once, and assembly is resumed following the FIN.

Form 2: block 1 is skipped, and assembly is resumed following the FIN.

If the expression in the DO directive is not evaluatable, Meta-Symbol sets the label or labels (if present) to the value zero, produces an error notification, and processes the DO directive as if the expression had been evaluated as zero.

An iteration block may contain other iteration blocks but they must not overlap. See Example 42 for the WHILE directive.

The label or labels for the DO directive are redefinable and their value may be changed by SET directives during the processing of the DO-loop. Any symbols in the DO directive expression that are redefinable may also be changed within the loop. However, unlike the WHILE directive, the count for the DO-loop is determined only once and changing the value of any expression symbol within the loop has no effect on how many times the loop will be executed.

The processing of DO directives involves program levels in the same manner as WHILE directives. Both a DO-loop and a WHILE-loop must be completed on the same program level on which they originate. That is, if a DO or WHILE directive occurs in the main program, the ELSE and/or FIN for that directive must also be in the main program. Similarly, if a DO or WHILE directive occurs within a procedure definition, the ELSE and/or FIN for that directive must also be within the definition.

Example 44. WHILE/DO/FIN Directives

```
          :
          :
I         WHILE     I<3
J         DO        I
          GEN,32    I * J
          FIN                 end DO
          FIN                 end WHILE
          :
          :
```

This sequence of code generates the values 1, 2, 4, 3, 6, and 9:

1. The internal counter n is set to zero, and this zero value of n is assigned to I.
2. I (which is 0) is less than 3, so the WHILE-loop is executed.
3. n is incremented by 1, and its new value is assigned to I, making I = 1.
4. The DO directive is encountered:
   a. The internal counter m is set to zero, and this zero value of m is assigned to J.

b. I has the value 1 (from step 3 above) so the DO-loop is executed one time.

c. m is incremented by 1, and its new value is assigned to J, making J = 1.

d. The GEN directive produces the value I * J = 1 * 1 = 1 as a 32-bit value.

e. FIN terminates the DO--loop.

5. FIN returns control to the WHILE directive.

6. I (which is 1) is less than 3, so the WHILE loop is executed again.

7. n is incremented by 1, and its new value is assigned to I, making I = 2.

8. The DO directive is encountered:

a. The internal counter m is set to zero, and this zero value of m is assigned to J.

b. I has the value 2 (from step 7 above) so the DO-loop is executed twice.

c. m is incremented by 1, and its new value is assigned to J, making J = 1.

d. The GEN directive produces I * J = 2 * 1 = 2

e. FIN terminates the first iteration and returns control to the DO directive.

f. m is incremented by 1, and its new value is assigned to J, making J = 2.

g. The GEN directive produces I * J = 2 * 2 = 4

h. FIN terminates the DO-loop.

9. FIN returns control to the WHILE directive.

10. I (which is 2) is less than 3, so the loop is executed again.

11. n is incremented by 1 and its new value is assigned to I, making I = 3.

12. The DO directive is encountered.

a. The internal counter m is set to zero, and this zero value of m is assigned to J.

b. I has the value 3 (from step 11 above) so the loop is executed three times.

c. m is incremented by 1, and its new value is assigned to J, making J = 1.

d. The GEN directive produces I * J = 3 * 1 = 3.

e. FIN terminates the first iteration and returns control to the DO directive.

f. m is incremented by 1, and its new value is assigned to J, making J = 2.

g. The GEN directive produces I * J = 3 * 2 = 6.

h. FIN terminates the second iteration and returns control to the DO directive.

i. m is incremented by 1, and its new value is assigned to J, making J = 3.

j. The GEN directive produces I * J = 3 * 3 = 9.

k. FIN terminates the DO-loop.

13. FIN returns control to the WHILE directive.

14. I (which is 3) is not less than 3, so the loop is not executed.

15. Skip the DO directive, the GEN directive, the FIN directive (that is paired with DO), and the FIN directive (that is paired with WHILE).

16. Continue the assembly process.

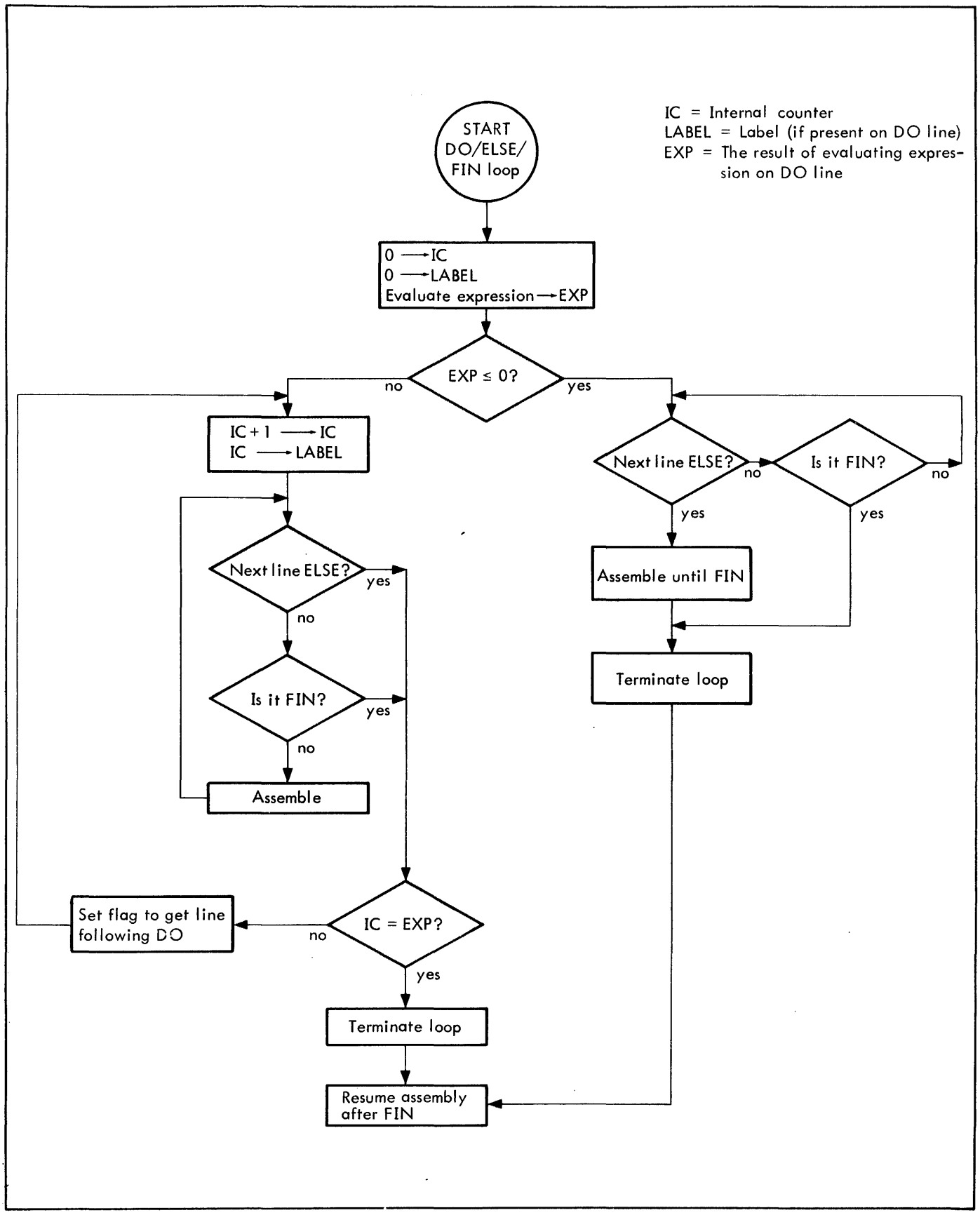
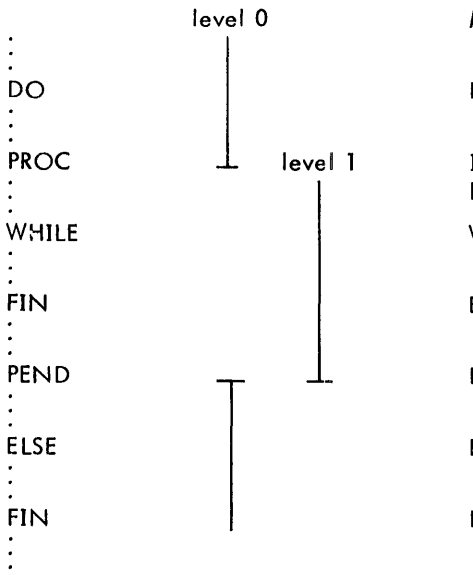


Figure 3. Flowchart of DO/ELSE/FIN Loop

Example 45. DO/ELSE/FIN Directives

Assume the main program has a DO-loop that contains a procedure definition that in turn contains a WHILE-loop. The ELSE and/or FIN for the WHILE-loop must be in the procedure definition, and those for the DO-loop must be in the main program.



Main program = level 0.

DO-loop commences at level 0.

Identifies beginning of procedure definition that will be assembled at level 1.

WHILE-loop invoked at level 1.

End WHILE-loop.

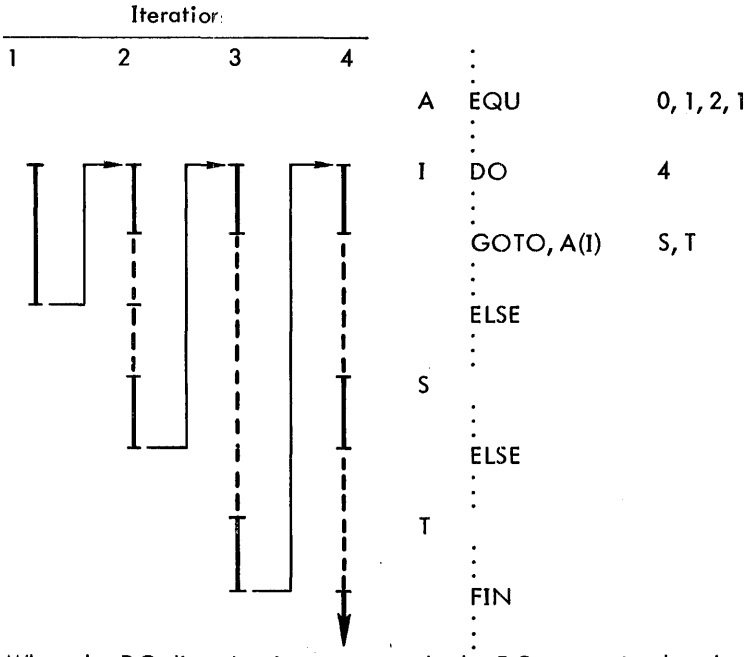
End procedure definition.

ELSE directive is part of DO/ELSE/FIN group.

End DO-loop.

Example 46. DO/ELSE/FIN Directives

In this example, the dashed vertical lines indicate statements that are skipped; solid vertical lines indicate statements that are assembled. The numbers 1, 2, 3, and 4 above the vertical lines indicate which iteration of the DO-loop is in process. This example uses a simple list; list A has four elements (0, 1, 2, 1) that can be referenced as A(1), A(2), A(3), and A(4), respectively.



When the DO directive is encountered, the DO expression has the value 4 so the loop will be executed four times. When the GOTO directive is encountered the first time through the loop, I has the value 1 so A(I) refers to the first element in the list to which A is equated. That element is the value zero. The expression for the GOTO has the value zero, so the next statement in sequence is assembled. Assembly continues in sequence until the ELSE directive is encountered, which ends the first iteration and returns control to the DO directive.

When the GOTO directive is encountered the second time through the loop, I has the value 2 so A(I) refers to the second element in list A; i.e., the value 1. Thus, the expression for GOTO has the value 1 so Meta-Symbol will skip until it finds a statement labeled S. Starting with S, Meta-Symbol assembles code until it encounters the ELSE which terminates the second iteration of the loop and returns control to the DO directive.

When the GOTO directive is encountered the third time through the loop, I has the value 3 so A(I) refers to the third element in list A; i.e., the value 2. Thus, the expression for GOTO has the value 2 so Meta-Symbol will skip until it finds a statement labeled T. Starting at T, Meta-Symbol assembles code until it encounters the FIN directive which terminates the third iteration of the loop and returns control to the DO directive.

When the GOTO directive is encountered the fourth time through the loop, I has the value 4 so A(I) refers to the fourth element in list A; i.e., the value 1. Thus, the expression for GOTO has the value 1 so Meta-Symbol will skip until it finds a statement labeled S. Starting at S, Meta-Symbol assembles code until it encounters the ELSE directive which terminates the fourth – and last – iteration of the loop. Then, Meta-Symbol skips until it encounters the FIN directive. Assembly resumes at the first statement following the FIN.

SYMBOL MANIPULATION

EQU Equate Symbols

The EQU directive enables the user to define a symbol by assigning to it the attributes of the list in the argument field. This directive has the form

label	command	argument
[$label_1$, . . . , $label_n$]	EQU[,s]	[list]

where

label$_i$ are valid symbolic names. If there are no label$_i$, the only effect of the EQU directive is to cause evaluation of the list.

s is an integer-valued expression that identifies the "type" of label. This expression is used in conjunction with the SD option (see Chapter 7) to provide explicit "type" information to a loader and, subsequently to a run-time debugging program. If s is omitted, label$_i$ are assumed to represent hexadecimal values. The legal values for s and the associated meanings are given below:

X'00'	Instruction
X'01'	Integer
X'02'	Short floating-point
X'03'	Long floating-point
X'06'	Hexadecimal (also for packed decimal)
X'07'	EBCDIC text (also for unpacked decimal)
X'09'	Integer array
X'0A'	Short floating-point array
X'0B'	Long floating-complex array
X'08'	Logical array
X'10'	Undefined symbol

list is any list. The elements in the list may contain forward and external references.

When list is an expression, label is set equivalent to the value of the expression:

VALUE EQU 2*(8-5) + 1 makes VALUE ≡ 7

ALPHA EQU XYZ - 10 makes ALPHA ≡ XYZ - 10

(The symbol ≡ means "is identical to".)

When list is a list of more than one element, label is set equivalent to all individual elements in the list. This is shown in various exapmles given in Chapters 2 and 5.

If more than one label is given, each is set equivalent to list. Thus,

A,B EQU 5 makes A ≡ 5, B ≡ 5

The value or values in list appear on the assembly listing in a special format that indicates the type of value(s) to which label has been equated. This format is explained under "Meta-Symbol Assembly Listing" in Chapter 6.

Example 47. EQU Directive

```
        .
        .
B       EQU     A       Makes B ≡ A. Because A is a
        .               forward reference, B also has the
        .               attribute of being a forward
        .               reference.
        .
        .
        GEN,32  B       Legal; the GEN directive allows
        .               the use of forward references.
        .
        .
        DO      B       Illegal; the DO directive does
        .               not permit the use of forward
        .               references, and it is processed
        .               as if B = 0.
        .
        .
        DO      A       Illegal; A is a forward reference.
        .
        .
A       EQU     5       Defines A.
        .
        .
        DO      A       Legal; A is no longer a forward
        .               reference.
        .
        .
        DO      B       Legal; B is no longer a forward
        .               reference.
        .
        .
```

SET Set a Value

The SET directive, like EQU, enables the user to define a symbol or symbols by assigning to each the attributes of the list in the argument field. SET has the form

label	command	argument
[label$_1$, . . . , label$_n$]	SET [,s]	[list]

where label, s, and list are the same as for EQU.

The SET directive differs from the EQU directive in that any symbol defined by a SET may later be redefined by means of another SET. It is an error to attempt to do this with an EQU. SET is particularly useful in writing procedures.

The value or values in list appear on the assembly listing in a special format that indicates the type of value(s) to which label has been equated. This format is explained under "Assembly Listing" in Chapter 6.

Example 48. SET Directive

```
      :
      :
A     EQU    X'FF'
      :
      :
M     SET    A          M is set to the hexadecimal
      :                 value FF.
      :
S     SET    M          Thus, S = M = X'FF'.
      :
      :
M     SET    263        Redefines symbol M.
      :
      :
S     EQU    M          Error; does not redefine
      :                 symbol S.
      :
      :
```

LOCAL Declare Local Symbols

The main program and the body of each procedure called during the assembly of the main program constitute the non-local symbol region for an assembly. Local symbol regions, in which certain symbols will be declared unique to the region, may be created within a main program or procedure by the LOCAL directive. This directive has the form

label	command	argument
	LOCAL	[symbol$_1$, . . . , symbol$_n$]

where the symbol$_i$ are declared to be local to the current region. Local symbols are syntactically the same as non-local symbols. The argument field may be blank, in which case the LOCAL directive terminates the current local symbol region without declaring any new local symbols.

A label field entry is ignored by the assembler unless it is the target label of a GOTO search.

The local symbol region begins with the first statement (other than comments or another LOCAL) following the LOCAL directive and is terminated by a subsequent use of the LOCAL directive.

Within a local symbol region a symbol declared as LOCAL may not be used as a forward reference in an arithmetic process other than addition, subtraction, or comparison. This does not limit the use of defined local symbols in other arithmetic processes.

The occurrence of the PROC directive suspends the current local symbol region until the corresponding PEND is encountered. The suspended local symbols are then reactivated. See Example 52. (PROC and PEND define the beginning and end, respectively, of a procedure definition. See Chapter 5.)

When a LOCAL directive occurs between the PROC and PEND directives, a procedure-local symbol region is created, with local symbols that may be referenced only within the specified region of the procedure being defined. When the procedure is subsequently referenced in the program, the currently active local or procedure-local symbols are suspended until the corresponding PEND is encountered. The suspended local symbols are then reactivated.

Example 49. LOCAL Directive

```
                :
                :
                LOCAL    A,B,C
                LOCAL    R,S,T,U
                LOCAL    X,Y,Z
*COMMENT
START           EQU      $
                :
                :
                LOCAL
```

The three LOCAL directives inform the assembler that the symbols A, B, C, R, S, T, U, X, Y, and Z are to be local to the region beginning with the line START. The final LOCAL directive terminates the local symbol region without declaring any new local symbols.

Example 50. LOCAL Directive

```
      :
      :
A     EQU    X'E1'
      :
      :
      LOCAL  A        New A, not the same as A above.
      :
      :
A     EQU    89       Legal, since this is the local A.
      :
      :
B     EQU    A        Defines B as the decimal
      :                value 89.
      LOCAL  Z        Terminates current local symbol
      :                region and initiates a new
      :                region.
Z     EQU    A        Z is equated to the hexadeci-
      :                mal value E1.
      :
      :
```

Example 51. LOCAL Directive

```
        .
        .
        LOCAL   B
        LW,7    B*3     Illegal because B is a local
                        forward reference and mul-
        .               tiplication is requested.
        .
B       EQU     9       Defines symbol B.
        .
        LW,9    B*3     Legal.
        .
        AW,9    A/2     Legal because A is not a
        .               local symbol.
        .
A       EQU     X'F3A'  Defines symbol A.
        .
```

Example 52. LOCAL Directive

```
        .
        .
A       EQU     X'E1'
        .
        .
        LOCAL   A       New A, not the same as
        .               A above.
        .
A       EQU     89      Legal, since this is the
        .               local symbol A.
        .
        PROC            A PROC suspends the range
        .               of a LOCAL and reinstates
        .               any prior nonlocal symbols.
        .
B       EQU     A       Defines B as the hexa-
        .               decimal value E1.
        .
        PEND            Terminates the procedure
        .               and reinstates the prior
        .               LOCAL symbols.
        .
X       EQU     A<X'CF' Equates X to the value 1
        .               because 89 is less than
        .               X'CF'.
        .
        LOCAL Z         Terminates current local
        .               symbol region and initiates
        .               a new region.
        .
Z       EQU     A=X'E1' Equates Z to the value 1
        .               because the nonlocal sym-
        .               bol A has the hexadecimal
        .               value E1.
```

OPEN/CLOSE Symbol Control

OPEN and CLOSE control the scope of nonlocal symbols. These directives have the forms

label	command	argument
	OPEN	[symbol$_1$, . . . , symbol$_n$]
	CLOSE	[symbol$_1$, . . . , symbol$_n$]

where symbol$_i$ represent a list of nonlocal symbols that are to be opened or closed for use as unique symbols. The OPEN directive explicitly declares subsequent usage of the designated symbolic names (until closed or opened again) to be completely independent of any prior uses of the same symbolic name. See Example 53.

The CLOSE directive declares that the designated, currently opened nonlocal symbols are to be permanently closed for all subsequent usage. Once a symbol has been closed, it cannot be opened again. For example, in the sequence

```
A       EQU     15
        CLOSE   A
A       LW,4    ALPHA
        OPEN    A
```

the CLOSE directive informs Meta-Symbol that the current nonlocal symbol A may not be used again. The label A in the next statement is a valid symbol, different from the previous A. The OPEN directive informs Meta-Symbol that a new symbol A is to be used; this A is different from both of the previous A's.

If a symbol is not explicitly opened with an OPEN directive, it is considered implicitly opened the first time it appears in a program. The names of directives and intrinsic functions are opened at the start of an assembly, but it is permissible to close them or to open a new symbolic name having the same configuration. Instructions in system instruction sets may also be opened and closed (see Example 54). This enables the user to close any directive, function, or system name that may conflict with names he has used. Programmers should be very careful in using OPEN and CLOSE directives since misuse can result in an erroneous assembly or termination of assembly. In fact OPEN and CLOSE are used only in special applications; for example, communication between system procedure calls requires nonlocal symbols, because local symbols are purged at the end of each procedure.

OPEN and CLOSE are processed completely by the encoding phase (Pass 0); they are treated as comments in the two assembly phases. As such, they are unconditionally executed at the time they are first encountered within the source program. Since a GOTO or DO directive is not processed until the assembly phase, it is not possible to skip or repeat an OPEN or CLOSE directive. Also, since procedure references are not expanded until the assembly phase, an OPEN or CLOSE directive within a procedure definition is effective only when the definition is first processed; not when the procedure is referenced.

OPEN and CLOSE control all forms of usage of the symbols in a program, whether used as commands or as labels.

Example 53. OPEN/CLOSE Directives

```
        :
        OPEN    A, B, C     Declares A, B, and C open
                            for use.
        :
A       EQU     BETA        Same A as above.
        :
        LW, 2   A           Same A as above.

        OPEN    A           Opens a new A, different
                            from previous A.
        :
A       EQU     ALPHA       Legal because this A does
                            not have the same value
                            that was equated to BETA.
        :
        CLOSE   A           Closes current A. This A
                            cannot be referenced again
                            (however, ALPHA can be).
                            The previously open A – the
                            one equated to BETA – is
                            now reinstated and any
                            references to A are to it.

        STW, 2  A           Equivalent to STW, 2 BETA.
        :
        OPEN    A           This is a new A, different
                            from both A's used above.

        LW, 3   B           This is B that was opened
                            at the beginning of this
                            example.
        :
```

Example 54. OPEN/CLOSE Directives

```
        :
        SYSTEM              SIG7FDP
        :
Z       EQU     F           Legal. Equates symbol Z
                            to symbol F.

EQU     LW, 4   Z           Legal. Directive names
                            may be used as label entries
                            without conflict.
        :
        OPEN    EQU, LW     Declares EQU and LW open
                            for use.

S       EQU     T           Illegal. EQU has been
                            opened as a new symbol;
                            therefore, Meta-Symbol
                            does not recognize EQU
                            as a directive.
        :
        LW, 3   T           Illegal. LW has been
                            opened as a new symbol;
                            therefore, Meta-Symbol
                            does not recognize LW as
                            a command.
        :
```

Example 55. OPEN/CLOSE/GOTO Directives

```
        :
A       SET     2
        :
B       SET     1
        :
        GOTO, A*B/2  X, Y, Z   Begins search for label X.
        :
W       EQU     X           Legal; does not
                            terminate search.

        OPEN    X           Makes a new definition
                            of X available to the
                            assembler.
        :
X       DO      K*Z         Because of the OPEN
                            directive, this X is not
                            the same as the X for
                            which the search is be-
                            ing made and, there-
                            fore, is ignored.
        :
        CLOSE   X           Closes the new X and
                            again makes the old X
                            (i.e., X referenced to
                            in the GOTO statement)
                            available to the
                            assembler.

        FIN
        :
X                           Search is successfully
                            completed and assembly
                            resumes here.
        :
Y
        :
Z
        :
```

Example 56. OPEN/CLOSE/GOTO Directives

```
          :
          :
          OPEN      T       Opens T as a new symbol.
          :
          :
K         EQU       2
          :
          :
          GOTO, K   H, T, L Begins search for label T
                            (this is the same T that
                            was opened above).
          :
          :
          CLOSE     T       Closes the symbol T for
                            which the assembler is
                            searching. Meta-Symbol
                            continues searching until
                            the end of the program.
                            It then produces an error
                            message.
          :
          :
```

Example 57. OPEN/CLOSE/LOCAL Directives

```
          :
          :
Z         LW, 6     A       References symbol A.
          :
          :
          OPEN      A       Opens a new nonlocal
                            symbol A, different from
                            the one used above.
          :
          :
          LOCAL     A       Initiates a local symbol
                            region in which A is a
                            local symbol.
A         EQU       B       This is the local symbol A.
          :
          :
          CLOSE     A       Closes the nonlocal sym-
                            bol A that was opened
                            above and causes the pre-
                            vious nonlocal A (i.e.,
                            the one that appeared in
                            statement Z) to be rein-
                            stated when the current
                            local symbol region is
                            terminated.
          :
          :
A         EQU       ZD      Illegal. This is the
                            local A which was
                            equated to B.
          :
          :
          LOCAL     X       Terminates the previous
                            local symbol region and
                            initiates a new one in
                            which X is a local symbol.
          :
          :
          LW, 12    A       This is the same A that
                            appeared in statement Z.
A         EQU       15      Legal. This is the first
                            definition of the non-
                            local symbol A.
          :
          :
```

This example emphasizes the fact that OPEN and CLOSE directives affect only nonlocal symbols; local symbols cannot be OPENed or CLOSEd.

DEF Declare External Definitions

The DEF directive declares which symbols defined in this assembly may be referenced by other (separately assembled) programs. The form of this directive is

label	command	argument
	DEF	[symbol$_1$, . . . , symbol$_n$]

where symbol$_i$ may be the intrinsic functions, LF, CF, or AF, or any global symbolic labels that are defined within the current program. If there is no symbol$_i$, the directive is ignored.

DEF directives may appear anywhere in a program. Symbols may be declared as external definitions prior or subsequent to their use in the program.

Section names for ASECT and CSECT may be external definitions; and, if such is the case, their names must be explicitly declared external via a DEF directive. The name of a dummy section (DSECT) is implicitly an external definition and should not appear in a DEF directive; otherwise a "doubly defined symbol" error condition will be produced.

The same symbol must not be declared an external definition more than once in a program (thus the restriction on a DSECT label). Such a condition will normally be detected by the assembler, and diagnosed as a "doubly defined symbol". However, Meta-Symbol does not detect identical symbol names that have been opened or closed; this case will be diagnosed (if at all) only by the loader used to load the assembled program (see Example 59).

As stated previously, all symbols declared as external definitions via a DEF directive must be defined within the same program. However, there are restrictions on the values assigned to DEFed symbols; they may be absolute or relocatable addresses, integer constants that may be correctly represented in 32 bits, or any expression involving a combination of such terms. They may not be lists, function names, or LOCAL symbol values (see Example 60). It is permissible, however, to DEF a symbol whose value will be found via a REF or SREF directive (see Examples 61 and 65). It is not legal, however, to DEF and REF the same symbol.

All address values (absolute or relocatable) assigned to DEFed symbols are generated into the object language as byte-addresses, in order to retain any pertinent lower-order resolution (see description of REF and SREF).

Example 58. DEF Directive

```
        DEF         TAN, SUM, SORT
```

This statement identifies the labels TAN, SUM, and SORT as symbols that may be referenced by other programs.

Example 59. DEF Directive

```
        DEF         X, Y, Z         Declares symbols X, Y, and Z as external symbols that may be refer-
                                    enced by other programs.
          .
          .
          .
Y       EQU         X'1F'           Defines symbol Y.
          .
          .
          .
        OPEN        Y               To Meta-Symbol, Y is now a completely new symbol.
          .
          .
          .
Y       EQU         $+7             Defines the new symbol, Y.
          .
          .
          .
        DEF         Y               Unknown to Meta-Symbol, a second declaration and definition of
                                    the symbol, Y, will now be produced. Depending on the loader,
                                    this may be diagnosed as a load-time error.
          .
          .
          .
```

Example 60. DEF Directive

```
        DEF         O, S, U, R      Declares symbols O, S, U, and R as external symbols that may be
                                    referenced by other programs.
          .
          .
          .
O       EQU         X'1F'           Legal. Constants may be linked via external definitions.
          .
          .
          .
S, PI   EQU         FL'.314159E1'   Although this is a legal definition of both S and PI, S cannot be
                                    properly DEFed because it exceeds 32 bits in value (error).
          .
          .
          .
U       EQU         X'E8',X'D6',X'E4'  Although this is a legal definition of U, a list cannot be DEFed in
                                    the object language (error).
          .
          .
          .
R       EQU         U(2)            Legal. The value, X'D6', is generated as the external value of R.
                                    Note, however, that it is not permissible to say

                                        DEF     U(2)

                                    since the argument(s) of DEF must be unsubscripted symbols.
          .
          .
          .
```

Example 61. DEF Directive

The following DEF occurs in a root module of a large system:

```
            DEF         SUBROUTN1
            :
SUBROUTN1   CSECT       1
            :
```

The subsystems of this system are coded from a specification in which the above DEF was mistyped as SUBROUTIN, and all 27 subsystems were thus coded as:

```
            REF         SUBROUTIN
            :
            BAL,LNK     SUBROUTIN
```

As an alternate to modifying any of the existing code, the following module can be loaded into the root segment of the program. It is legal and resolves the naming conflict illustrated above:

```
            DEF         SUBROUTIN
            REF         SUBROUTN1
SUBROUTIN   EQU         SUBROUTN1
            END
```

REF Declare External References

The REF directive declares which symbols referenced in this assembly are defined in some other separately assembled program. The directive has the form

label	command	argument
	REF[,n]	[symbol$_1$, . . . , symbol$_n$]

where

n may be an (optional) constant, symbol or expression whose value is 1, 2, 4, or 8, specifying the intrinsic resolution of the associated symbols as byte, halfword, word, or doubleword, respectively. If n is omitted, word resolution is assumed. If any of symbol$_i$ reference a constant value, n is ignored by the loader.

symbol$_i$ may be the intrinsic functions, LF, CF, or AF, or any global symbolic labels that are to be satisfied at load time by other programs. If there is no symbol$_i$ reference, the directive is ignored.

Symbols declared with REF directives can be used for symbolic program linkage between two or more programs. At load time these labels must be satisfied by corresponding external definitions (DEFs) in another program.

Example 62. REF Directive

```
    REF     IOCONT, TAPE, TYPE, PUNCH
```

This statement identifies the labels IOCONT, TAPE, TYPE, and PUNCH as symbols for which external definitions will be required at load time.

Example 63. REF Directive

```
        REF         Q       Q is an external reference.
        :
B       GEN,16,16   Q,$     The value of an external
                            reference may be placed
                            in any portion of a ma-
                            chine's word.
        :
        LW,2        Q       Q is an external reference.
        :
```

SREF Secondary External References

The SREF directive is similar to REF and has the form

label	command	argument
	SREF[,n]	[symbol$_1$, . . . , symbol$_n$]

where n and symbol$_i$ have the same meaning as in REF.

REF and SREF directives may appear anywhere in a program. Symbols may be declared as external references before or after their use in the program. Symbols that are external references may be modified by the addition and subtraction of integers, relocatable symbols, and other external references. See Example 65.

SREF differs from REF in that REF causes the loader to load routines whose labels it references, whereas SREF does not. Instead, SREF informs the loader that if the routines whose labels it references are in core, the loader should satisfy the references and provide the interprogram linkage. If the routines are not in core, SREF does not cause the loader to load them; however, it does cause the loader to accept any references within the program to the names, without considering them to be unsatisfied external references.

Although all symbols are DEFed as byte addresses, a program that REFs them will use the word address unless otherwise specified. Example 64 shows two program segments that function identically.

Example 64. REF/SREF Directives

```
REF         OKE          Although low-order resolution of these symbols is available, their word
REF         FEN          address will be used unless otherwise specified.
SREF        OKEE
REF         GA
:
:
LI,7        HA(FEN)      Halfword address of FEN.
:
:
LW,5        BA(OKE)      Always word address.
:
:
DATA        GA;          Implicit word address.
            ,OKEE        Implicit word address.
:
:
END


REF,1       OKE          Each REF symbol is given an explicit intrinsic resolution that will be
REF,2       FEN          used unless otherwise specified.
SREF,4      OKEE
REF,8       GA
:
:
LI,7        FEN          Halfword address of FEN.
:
:
LW,5        OKE          Always word address.
:
:
DATA        WA(GA);      Forces word address.
            ,OKEE        Intrinsic word address.
:
:
END
```

Example 65. REF Directive

```
        .
        .
        .
        REF     Q       Q is an external reference.
        .
        .
        .
B       EQU     Q       Equates B to all attributes
                        of Q.
        .
        .
        .
        LW,2    B       Equivalent to LW,2 Q.
C       EQU     Q+2     Legal usage.
        .
        .
        .
        LW,2    C       Equivalent to LW,2 Q+2.
        .
        .
        .
M       EQU     N
        .
        .
        .
        REF     N,P     It is legal to declare N an ex-
                        ternal reference after N has
                        appeared in the program. In
                        the sequence shown here, N is
                        made an external reference by
                        the REF directive.
        .
        .
        .
        DEF     M,C     Defines M and C as externals.
                        B is not an external.
        .
        .
        .
```

DATA GENERATION

GEN Generate a Value

The GEN directive produces a hexadecimal value representing the specified bit configuration. It has the form

label	command	argument
[label$_1$, . . . , label$_n$]	GEN[,field list]	[value list]

where

label$_i$ are any valid symbols. Use of one or more labels is optional. When present, each is defined as the current value of the execution location counter and identifies the first byte generated. The location counters are incremented by the number of bytes generated.

field list is a list of evaluatable, non-negative expressions that define the number of bits comprising each field. The sum of the field sizes must be a non-negative integer value that is a multiple of eight and is less than or equal to 128. If "field list" is omitted, 32 is assumed.

value list is a list of expressions that define the contents of each generated field. This list may contain forward references. The value, represented by the value list, is assembled into the field specified by the field list and is stored in the defined location (see Example 66). If value list contains fewer elements than field list, zeros are used to pad the remaining fields.

Note: The intrinsic symbols $ and $$ always refer to the first byte generated by the GEN directive.

Example 66. GEN Directive

```
        GEN,16,16   -251,89     Produces two 16-bit
                                hexadecimal values:
                                FF05 and 0059.
```

Example 67. GEN Directive

```
        .
        .
B       EQU         X'FFFFFFFF'
        GEN,64      B           Produces: 00000000
        .                                 FFFFFFFF
        .
```

There is a one-to-one correspondence between the entries in the field list and the entries in the value list; the code is generated so that the first field contains the first value, the second field the second value, etc. The value produced by a GEN directive appears on the object program listing as eight hexadecimal digits per line.

External references, forward references, and relocatable addresses may be generated in any portion of a machine word; that is, an address may be generated in a field that overlaps word boundaries.

Example 68. GEN Directive

```
        BOUND       4           Specifies word boundary.

LAB     GEN,8,8,8   8,9,10      Produces three consecutive bytes; the first is identified as LAB and con-
                                tains the hexadecimal value 08; the second contains the hexadecimal
                                value 09; and the third byte contains the hexadecimal value 0A.
          :
          :
        LW,5        L(2)        Loads register 5 with the literal value 2.
          :
          :
        LB,3        LAB,5       Loads byte into register 3. LAB specifies the word boundary at which the
                                byte string begins, and the value of the index register (that is, the value 2
                                in register 5) specifies the third byte in the string (byte string numbering
                                begins at 0). Thus, this instruction loads the third byte of LAB (the
                                value 0A) into register 3.
          :
          :
```

Example 69. GEN Directive

```
          :
          :
ALPHA   EQU         X'F'        Defines ALPHA as the decimal value 15.
BETA    EQU         X'C'        Defines BETA as the decimal value 12.
          :
          :
A       GEN,32      ALPHA+BETA  Defines A as the current location and stores the decimal value 27 in
                                32 bits.
```

In this case, the GEN directive results in a situation that is effectively the same as:

```
A       GEN,32      27
          :
          :
```

COM Command Definition

The COM directive enables the programmer to describe subdivisions of computer words and invoke them simply. This directive has the form

label	command	argument
$label_1$[, ..., $label_n$]	COM[, field list]	[value list]

where

label$_i$ are valid symbols by which the COM may be referenced. Symbols currently declared as local may not be used as labels on a COM directive.

field list is a list of evaluatable expressions that define the number of bits comprising each field. The sum of the elements in this list must be a positive integer value that is a multiple of eight bits and is less than or equal to 128. If field list is omitted, 32 is assumed.

value list is a list of expressions or intrinsic functions (see below) that specify the contents of each field.

When the COM directive is encountered, the label, field list, and value list specifications are saved. When the label of the COM directive subsequently appears in the command field of a statement called a "COM reference line", that statement will be generated with the configuration specified by the COM directive.

The use of commands defined by a COM is restricted as follows: the COM command definition should precede all references to it.

Note: As with the GEN directive, the intrinsic symbols $ and $$, used on a COM reference line, indicate the first byte generated by the COM reference.

The COM directive differs from GEN in that Meta-Symbol generates a value at the time it encounters a GEN directive, whereas it stores the COM directive and generates a value only when a COM reference line is encountered. If the reference line is labeled, the generated value will be identified by that value.

If a COM directive is to produce four bytes, it will be preceded at reference time by an implicit BOUND 4.

Certain intrinsic functions enable the user to specify in the COM directive which fields in the reference lines will contain values that are to be generated in the desired configuration. These functions are

CF	LF[t]
AF	NUM[t]
AFA	

CF Command Field

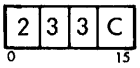
This function refers to the command field list in a reference line of a COM directive. Its format is

CF (element number)

where CF specifies the command field, and element number specifies which element in the field is being referenced.

Example 70. COM Directive and CF Function

```
          .
          .
BYT       COM, 8, 8          CF(2), CF(3)
          .
          .
XX        BYT, 35, X'3C'
          .
          .
```



The COM directive defines a 16-bit area consisting of two 8-bit fields. It further specifies that data for the first 8-bit field will be obtained from command field 2(CF(2)) of the COM reference line, and that data for the second 8-bit field will be obtained from command field 3(CF(3)). Therefore, when the XX reference line is encountered, Meta-Symbol generates a 16-bit value, so that the first eight bits contain the binary equivalent of the decimal number 35 and the second eight bits contain the binary equivalent of the hexadecimal number 3C.

AF Argument Field

This function refers to the argument field list in a reference line of a COM directive. Its format is

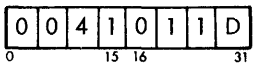
AF (element number)

where AF specifies the argument field, and element number specifies which element in the list of elements in that field is being referenced.

[t] See Chapter 5.

Example 71. COM Directive and AF Function

```
          .
          .
XYZ       COM, 16, 16        AF(1), AF(2)
          .
          .
ALPHA     EQU                X'21'
ZZ        XYZ                65, ALPHA+X'FC'
          .
          .
```



Meta-Symbol stores the COM definition for later use. When it encounters the ZZ reference line, it references the COM definition in order to generate the correct configuration. At that time, the expression ALPHA+X'FC' is evaluated. AF(1) in the XYZ line refers to 65 in the ZZ line; AF(2) refers to ALPHA+X'FC'.

AFA Argument Field Asterisk

The AFA function determines whether the specified argument in the COM reference line is preceded by an asterisk. The format for this function is

AFA (element number)

where AFA identifies the function, and element number specifies which element in the argument field of the COM reference line is to be tested. If element number is omitted, AFA(1) is assumed. The function produces a value of 1 (true) if an asterisk prefix exists on the argument designated; otherwise, it produces a zero value (false).

Example 72. COM Directive and AFA Function

```
          .
          .
STORE     COM,1,7,4,4        AFA(1),X'35',CF(2),AF(1)
          .
          .
          STORE, 4           *TOTAL
          .
          .
```

The COM directive defines STORE as a 16-bit area with four fields. The AFA(1) intrinsic function tests whether an asterisk precedes the first element in the argument field of the reference line. The first bit position of the area generated will contain the result of this test. The next seven bits of the area will contain the hexadecimal value 35. The second element in the command field of the refefence line will constitute the third field generated, while the first element in the argument field of the reference line will constitute the last field.

When the reference line is encountered, Meta-Symbol defines a 16-bit area as follows:

Bit Positions	Contents
0	The value 1 (because the asterisk is present in argument field 1).
1-7	The hexadecimal value 35.

8-11	The value 4.
12-15	The 4-bit value associated with the symbol TOTAL.

DATA Produce Data Value

DATA enables the programmer to represent data conveniently within the symbolic program. It has the form

label	command	argument
[$label_1$, ..., $label_n$]	DATA[,f]	[$value_1$, ..., $value_n$]

where

label; are valid symbols. Use of one or more labels is operational. When present, each is defined as the current value of the execution location counter and is associated with the first byte generated by the DATA directive. The location counters are incremented by the number of bytes generated.

f is the field size specification in bytes; f may be any evaluatable expression that results in an integer value in the range $0 \leq f \leq 16$.

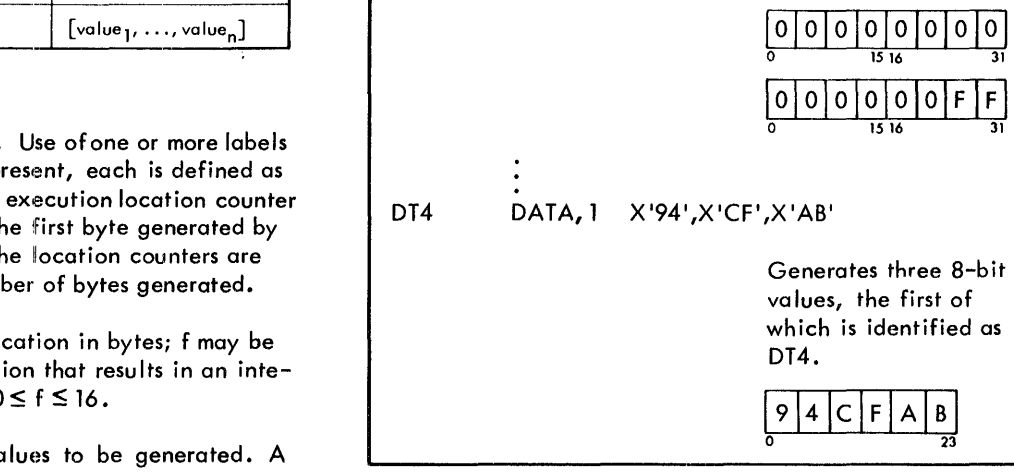
value; are the list of values to be generated. A value may be a multitermed expression or any symbol. An addressing function may be used to specify the resolution other than the intrinsic resolution of the execution location counter, if desired. Omitted values are assumed to be zero.

DATA generates each value in the list into a field whose size is specified by f in bytes. If f is omitted, four bytes are assumed.

Constant values must not exceed those specified under "Constants" in Chapter 2.

Example 73. DATA Directive

```
          :
MASK1     DATA,1    X'FF'      Produces an 8-bit value
                               identified as MASK1.
          :
          :
MASK2     DATA,2    X'1EF'     Generates the hexa-
                               decimal value 01EF as
                               a 16-bit quantity, iden-
                               tified as MASK2.
          :
          :
BYTE      DATA,3    BA(L(59))  Assembles the byte
                               address of the literal
                               value 59 in a 24-bit
                               field, identified as
                               BYTE.
          :
          :
TEST      DATA      0,X'FF'    Generates two 4-byte
                               quantities; the first
                               contains zeros and the
                               second, the hexadeci-
                               mal value 000000FF.
                               The first value is
                               identified as TEST.
          :
          :
DT4       DATA,1    X'94',X'CF',X'AB'
                               Generates three 8-bit
                               values, the first of
                               which is identified as
                               DT4.
```



S:SIN Standard Instruction Definition

The S:SIN directive provides a direct mechanism for defining the three main classes of Sigma machine instructions. It has the form

label	command	argument
$label_1$[, ..., $label_n$]	S:SIN,n	[expression]

where

label; are one or more valid global symbols which become the mnemonics by which the instruction is referenced.

n is an evaluatable, integer-valued expression which evaluates to one of the values 0, 1, or 2. This specifies a standard instruction format and a standard reference line assembly mode.

n = 0 implies the format 1, 7, 4, 3, 17 and specifies that a reference line is to be assembled "like" an LW instruction. AF(1) of any command defined via S:SIN,0 will be generated as WA(AF(1)).

n = 1 implies the format 1, 11, 3, 17 and specifies that a reference line is to be assembled "like" a BAZ/BANZ instruction. AF(1) of any command defined via S:SIN,1 will be generated as WA(AF(1)).

n = 2 implies the format 8, 4, 20 and specifies that a reference line is to be assembled "like" an LI instruction. Any command defined via S:SIN,2 is restricted to one argument field, and this argument may not have an asterisk prefix.

expression is an expression that is used as the operation code of the defined instruction. Normally this is an explicit hexadecimal constant.

Although the same definitions may be achieved by use of command procedures (Chapter 5) or the COM directive, S:SIN provides the fastest possible processing when Meta-Symbol is used as a production assembler for Sigma machine language programs.

Example 74. S:SIN Directive

```
The following definitions of various instructions are used
in the SIG7FDP system file.

        LW      S:SIN,0     X'32'
        AND     S:SIN,0     X'48'
        B       S:SIN,1     X'680'
        LCF     S:SIN,1     X'703'
        AI      S:SIN,2     X'20'
        CI      S:SIN,2     X'21'
```

TEXT EBCDIC Character String

The TEXT directive enables the user to incorporate messages in his program. This directive has the form

label	command	argument
[label$_1$,...,label$_n$]	TEXT	'cs$_1$'[,...,'cs$_n$']

where

label$_i$ are one or more valid symbols. Use of labels is optional. When present, each label is associated with the leftmost byte of the storage area assigned to the assembled message.

'cs$_i$' are evaluatable expressions that result in character string constants.

The character string is assembled in a binary-coded form in a field that begins at a word boundary and ends at a word boundary. The first byte contains the first character of the character string, the second byte contains the second character, etc. If the character string does not require an even multiple of four bytes for its representation, trailing blanks are produced to occupy the space to the next word boundary.

When several character strings are present in the argument field of a TEXT directive, the characters are packed in contiguous bytes (see Example 75). This directive permits continuation lines, but the continuation indicator must occur between two character strings.

The TEXT directive enables the user to pass a character string as a parameter from a procedure reference line to a procedure. The character string must be written on the procedure reference line within single quotation marks. It is referenced from within the procedure via the AF, CF, or LF intrinsic functions in a TEXT directive. The intrinsic function is not written with single quotation marks (see Example 76).

Example 75. TEXT Directive

```
COL1    TEXT    'VALUE OF X'
```

generates

V	A	L	U
E		O	F
	X		

```
        :
        :
        TEXT    'A','BCDE',FGHI',;
                'JKLM'
```

generates

A	B	C	D
E	F	G	H
I	J	K	L
M			

Example 76. TEXT Directive

```
        :
        :
        TEXT    AF(1)               In a procedure
                                    definition.
        :
        :
        TEXT    'SUM OF ',AF(1),;
                ' AND ',AF(2)       In a procedure
                                    definition.
        .
        :
        PRINT1  'RESULTS ='         Procedure refer-
                                    ence line.
        .
        :
        PRINT2  'X','Y'             Procedure refer-
                                    ence line.
        :
        :
```

Assume that the first TEXT directive is in the definition of a command procedure called PRINT1, that the second TEXT directive is in the definition of a command procedure called PRINT2, and that the last two statements are procedure reference lines that call these procedures. When procedure PRINT1 is referenced, the first TEXT directive causes Meta-Symbol to generate

R	E	S	U
L	T	S	
=			

When procedure PRINT2 is referenced, the second TEXT directive causes Meta-Symbol to generate

S	U	M	
O	F		X
	A	N	D'
	Y		

Thus, entire messages or portions of messages may be used as parameters on procedure reference lines.

TEXTC Text With Count

The TEXTC directive enables the user to incorporate messages in a program where the number of characters in the message is contained as the first byte of the message. This directive has the form

label	command	argument
[label$_1$,...,label$_n$]	TEXTC	'cs$_1$'[,...,'cs$_n$']

where label$_i$ and 'cs$_i$' have the same meaning as for TEXT.

The TEXTC directive provides a byte count of the total storage space required for the message. The count is placed in the first byte of the storage area and the character string follows, beginning in the second byte. The count represents only the number of characters in the character string; it does not include the byte it occupies nor any trailing blanks. The maximum number of characters for a single TEXTC directive is 255.

In all other aspects, the TEXTC directive functions in the same manner as the TEXT directive.

Example 77. TEXTC Directive

```
ALPHA    TEXTC    'VALUE OF X';
                  ,' SQUARED'
```

generates

18	V	A	L
U	E		O
F		X	
S	Q	U	A
R	E	D	

SOCW Suppress Object Control Words

The SOCW directive causes Meta-Symbol to omit all object control bytes from the binary output that it produces during an assembly. This directive has the form

label	command	argument
	SOCW	

When Meta-Symbol encounters an SOCW directive, it sets the location counters to absolute zero, processes the program as an absolute section, and diagnoses any subsequent CSECT, DSECT, PSECT, or USECT directives. Meta-Symbol produces appropriate error messages if the directives that require control byte generation are used (REF, DEF, SREF, and LOCAL except in procedures), if an illegal object language feature is subsequently required (such as the occurrence of a local forward reference), or if the SOCW directive has been used subsequent to the generation of any object code in the program.

Once the SOCW directive is invoked, it remains in effect during the assembly of the entire program.

Normally, control words are produced to convey to the loader information concerning program relation, externally defined and/or referenced symbols, etc. In special cases, such as writing bootstrap loaders and special diagnostic programs, the programmer does not want the control words produced; he needs only the continuous string of bits that results from an assembly of statements. The SOCW directive enables the programmer to suppress the output of these control words.

Use of the ORG and RES directives is allowed, although this is a questionable practice (i.e., no code is generated for these directives, but the assembler's location counters are modified as directed.

When SOCW is specified, it is recommended that it be the first statement in the program, or at least that it precede the first generative statement.

LISTING CONTROL

Listing control directives are used to format the assembly listing and are only effective at assembly time. No object code is produced as a result of their use.

SPACE Space Listing

The SPACE directive enables the user to insert blank lines in the assembly listing. The form of this directive is

label	command	argument
	SPACE	[expression]

where expression specifies the number of lines to be spaced. The expression must not contain any external references.

If the expression is omitted, or is less than 1, its value is assumed to be 1. If the expression is greater than 16, it is set to 16. If the value of the expression exceeds the number of lines remaining on the page, the directive will position the assembly listing to top of form.

Example 78. SPACE Directive

```
        .
        .
A       SET     2
        .
        .
        SPACE   5       Space five lines.
        .
        .
        SPACE   2*A     Space four lines.
        .
        .
```

TITLE Identify Output

The TITLE directive enables the programmer to specify an identification for the assembly listing. The TITLE directive has the form

label	command	argument
	TITLE	['cs']

where cs is an expression that results in a character string constant and may include 1 to 75 EBCDIC characters.

When a TITLE directive is encountered, the assembly listing is advanced to a new page and the character string is printed at the top of the page and each succeeding page until another TITLE directive is encountered. A TITLE directive with a blank argument field causes the listing to be advanced to a new page and output to be printed without a heading.

The first TITLE directive in a program is retroactive; that is, its header will appear on the first page of the assembly listing, regardless of the placement of the first TITLE directive.

A TITLE directive with a blank argument field will suppress inclusion of the date and time in the heading; it will not suppress the assembler version number or page count (see Chapter 6).

Example 79. TITLE Directive

```
        .
        .
        TITLE   'CARD READ/PUNCH ROUTINE'
        .
        .
        TITLE   'MAG TAPE I/O ROUTINE'
        .
        .
        TITLE
        .
        .
        TITLE   "'CONTROLLER'"
        .
        .
```

The first TITLE causes Meta-Symbol to position the assembly listing to the top of the form and to print CARD READ/PUNCH ROUTINE there and on each succeeding page until the next TITLE directive is encountered. The next directive causes a skip to a new page and output of the title MAG TAPE I/O ROUTINE. The Third TITLE directive causes a skip to a new page but no title is printed because the argument field is blank. The last TITLE directive specifies the heading 'CONTROLLER'.

LIST List/No List

The LIST directive enables the user to selectively suppress and resume the assembly listing. The form of the directive is

label	command	argument
	LIST	[expression]

where expression is an evaluatable expression resulting in an integer that suppresses or resumes assembly listing. If the value of the expression is nonzero, a normal assembly listing will be produced. If the expression is zero when LIST is encountered, all listing following the directive will be suppressed until a subsequent LIST directs otherwise. If expression is omitted, zero is assumed.

Used inside a procedure, the LIST directive will not suppress printing of the procedure reference (call) line. However, LIST will suppress printing of the object code associated with the call line if the LIST directive was encountered prior to any code generation within the procedure.

Until a LIST directive appears within a source program, the assembler assumes a default convention of LIST 1, allowing a normal assembly listing.

PCC Print Control Cards

The PCC directive controls the assembly listing of directives PAGE, SPACE, TITLE, LIST, PSR, PSYS, and any subsequent PCC. The form of the directive is

label	command	argument
	PCC	[expression]

where expression is an evaluatable expression resulting in an integer that suppresses or enables assembly listing of the aforementioned directives. If the value of the expression is nonzero when PCC is encountered, all subsequent listing control directives mentioned above will be listed. This will continue in effect until canceled by a subsequent PCC directive in which the expression is zero. If expression is omitted, zero is assumed.

Until a PCC directive appears within a source program, the assembler assumes a default condition of PCC 1, allowing assembly listing of the list control directives.

PSR Print Skipped Records

The PSR directive controls printing of records skipped under control of the GOTO, DO, DO1, or WHILE directives, as well as any records skipped due to unused command or procedure definitions. The form of the directive is

label	command	argument
	PSR	[expression]

where expression is an evaluatable expression resulting in an integer that suppresses or enables assembly listing of skipped records. If rhe value of the expression is nonzero, records skipped will be listed; if the expression is zero when PSR is encountered, records skipped (not assembled), subsequent to the PSR directive, will not be listed until another PSR directs otherwise. If expression is omitted, zero is assumed.

Until a PSR directive appears within a source program, the assembler assumes a default condition of PSR 1, allowing assembly listing of skipped records.

PSYS Print System

The PSYS directive controls the assembly listing of system files. The form of the directive is

label	command	argument
	PSYS	[expression]

where expression is an evaluatable expression resulting in an integer that suppresses or enables the assembly listing of files called by the SYSTEM directive. If the value of the expression is nonzero when PSYS is encountered, the symbolic records obtained during all subsequent SYSTEM calls will be printed on the assembly listing. This will continue in effect until canceled by a subsequent PSYS directive in which the expression is zero. If expression is omitted, zero is assumed.

Until a PSYS directive appears within a source program, the assembler assumes a default condition of PSYS 0, suppressing assembly listing of system files.

DISP Display Values

The DISP directive produces a special display of the values specified in its argument list, one per line on the assembly listing. The form of the directive is

label	command	argument
	DISP	[list]

where list is any list of constants, symbols, or expressions that are to be displayed at that point in the assembly listing. The values of the argument list will be displayed one per line, beginning at the DISP directive line.

If a DISP directive is used inside a procedure, it will not display values until the procedure is called on a procedure reference line.

The value or values in list appear on the assembly listing in a special format that indicates the type of value(s) being displayed. This format is explained under "Assembly Listing" in Chapter 6.

ERROR Produce Error Message

The ERROR directive conditionally generates an error message in the assembly listing and communicates the severity level to the assembler. This directive has the form

label	command	argument
	ERROR[,level[,c]]	$'cs_1'[,\ldots,'cs_n']$

where

level is an integer-valued expression with a value from X'0' through X'F', denoting error severity level. If level is omitted, zero is assumed. If level is preceded by an asterisk, Meta-Symbol omits the error line prefix (see Chapter 6), and the message starts in column 1 of the assembly listing. In addition, a level of zero preceded by an asterisk does not enter the line number in the error line summary, providing a method for inserting true comments into the assembly listing.

c is an integer-valued expression whose value determines whether the error message is to be produced:

If c is true ($c > 0$), the error message is produced.

If c is false ($c \leq 0$), the error message is produced.

If c is omitted, the error message is unconditionally produced.

c may be a forward reference.

$'cs_i'$ are expressions which evalute to character string constants. The total number of characters must not exceed 115.

Each time an error message is generated, the assembler compares the severity level with that from the preceding ERROR message and retains the higher value. After assembling an assemble-and-execute job, Meta-Symbol communicates to the Monitor the highest error severity level encountered. This enables the programmer to control the aborting of assemble-and-execute jobs via control messages to the Monitor.

The primary purpose of ERROR is to provide the procedure writer with the capability of flagging possible errors in the use of the procedure.

Example 80. ERROR Directive

```
        .
        .
        ERROR, 3, ALPHA>5  ;
            'ARGUMENT OUT OF RANGE'
        .
        .
```

When Meta-Symbol encounters this directive, it will determine whether the value of ALPHA is greater than 5. If it is, the result is true (value of 1); therefore, the severity level (3) is compared with current highest severity level, the higher of the two is saved, and the message "ARGUMENT OUT OF RANGE" is generated for the assembly listing.

Example 81. ERROR Directive

```
        .
        .
        ERROR, 1, ABSVAL(AF(1)&1 ;
            'ODD ARGUMENT FOR LD'
        .
        .
        LD          ALPHA
        .
        .
ALPHA   EQU         5
        .
        .
```

Assume the ERROR directive is a statement within the definition of the command procedure LD and that the reference to that procedure contains a forward reference. When the procedure reference is encountered, the procedure is assembled into the object program. Since AF(1) refers to ALPHA, which is a forward reference at the time the ERROR directive is assembled during the first pass, the result of the logical AND operation is zero, and the message is not output. During the second pass of the assembly, ALPHA is no longer a forward reference but has the value 5. Therefore, when the ERROR directive is encountered the second time, the result of the logical AND operation is 1, the severity level (1) is compared with the previously encountered highest level, the higher severity level is retained, and the error message "ODD ARGUMENT FOR LD" is produced.

PAGE Begin a New Page

The PAGE directive causes the assembly listing to be advanced to a new page. This directive has the form

label	command	argument
	PAGE	

5. PROCEDURES AND LISTS

PROCEDURES

Procedures are bodies of code analogous to subroutines, except that they are processed at assembly time rather than at execution time. Thus, they primarily affect the assembly of the program rather than its execution.

Using procedures, a programmer can cause Meta-Symbol to generate different sequences of code as determined by conditions existing at assembly time. For example, a procedure can be written to produce a specified number of ADD instructions for one condition and to produce a program loop for a different condition. (See Example 116 under "Sample Procedures".)

There are two types of procedures: command procedures and function procedures. In general, either type can perform any function that the main program can perform; i.e., any machine instruction or assembler directive can be used within a procedure. A command procedure is referenced by its name appearing as the first element of the command field. A function procedure is referenced by an attempt to evaluate its name. The major difference in the two procedure types is that a function procedure returns a value to the procedure reference line (the line that calls the procedure); a command procedure does not.

Procedures allow a program written in the assembly language of one computer (e.g., Xerox 9300) to be assembled and executed on another computer (e.g., Xerox Sigma 7). If a procedure is written for each 9300 machine instruction, Meta-Symbol treats each instruction as a procedure reference, and calls in the associated procedure, thus generating Sigma 7 machine language code.

Much of the creative power of Meta-Symbol comes from four directives: GEN, DO, WHILE, and PROC. The GEN, DO, and WHILE directives were described in Chapter 4; how they are used in procedures is illustrated in the various examples in this chapter. The directives that identify procedures, those that designate the beginning and end of each procedure, and those that control the display of procedure execution are discussed in this chapter. The intrinsic functions commonly used in writing procedures are also discussed.

In this chapter, the descriptions of various directives make frequent mention of "lists". Lists are most useful in handling procedures. Value lists were described in Chapter 2; procedure reference lists are discussed in detail later in this chapter after procedures have been introduced.

PROCEDURE FORMAT

A procedure consists of two parts; the procedure identification (names) and the procedure definition. The procedure names must precede the procedure definition, and the definition in turn must precede all references to it. For this reason, procedure definitions are normally placed at the beginning of the source program; this ensures that the definitions will precede all references to them.

During an assembly, Meta-Symbol reads the procedure definition and stores the encoded symbolic lines of the procedure in core memory. When Meta-Symbol later encounters the procedure reference line, it locates the procedure it has stored and assembles those lines.

CNAME/FNAME Procedure Name

A procedure may be invoked by a command or function reference. The names that will be used to invoke a command procedure must first be designated by the CNAME directive, which has the form

label	command	argument
$label_1[, \ldots, label_n]$	CNAME[,n]	[list]

where

label; are the symbols by which the next procedure to be encountered is identified. Symbols declared to be LOCAL may not be used as labels for a CNAME directive.

n is an evaluatable, integer-valued expression that specifies the number of bytes to be allocated for a reference to this command procedure during Pass 1 of assembly. If n is present, any labels on the command reference line will be automatically defined as the current value of the execution location counter ($). In addition, if n is equal to 4, an implicit BOUND 4 will precede assembly of the procedure definition.

list is an optional list of values that are evaluated and associated with the label(s). The use of a value list is explained later in this chapter under "Multiple Name Procedures".

The names that will be used to invoke a function procedure must first be designated by the directive FNAME, which has the form

label	command	argument
$label_1[, \ldots, label_n]$	FNAME	[list]

where label and list have the same meaning as for CNAME.

A procedure may be both a command procedure and a function procedure. It may have a single name declared with both CNAME and FNAME directives, or it may have different names, one for command references and another for function references. There is no limit to the number of CNAME and/or FNAME directives that may be given for a single procedure.

The applicable CNAME/FNAME directives must precede the procedure definition; however, the definition need not follow immediately after the name lines. CNAME and FNAME directives are associated with the first procedure definition encountered following these directives. This means that one cannot put all CNAME/FNAME directives before all procedure definitions. If such a case occurred, all the "labels" would be associated with the first procedure definition, and an error notification would be produced each time another procedure definition was encountered.

The intended purpose of procedures is to allow the programmer to redefine assembly language instructions belonging to another system so they can be assembled by Meta-Symbol for operation on a Sigma computer, and also, in effect, to create new instructions, directives, and functions. However, using procedures to redefine existing Meta-Symbol directives is a questionable practice frequently leading to assembly errors. Consequently, when a Meta-Symbol directive name (GEN, ORG, etc.) is encountered in the label field of a CNAME directive, Meta-Symbol will not define a new procedure for the directive (except as noted below), and will produce the following message on the assembly listing:

DBL DEF DIR

A directive can be redefined, however, if its name is first opened with an OPEN directive. OPEN was explained in Chapter 4 along with appropriate cautions as to its use.

PROC Begin Procedure Definition

The PROC directive indicates the beginning of a procedure definition and has the form

label	command	argument
	PROC	

The first line encountered following the PROC directive begins the procedure body. Nonlocal symbols are not unique to a procedure unless they are specifically opened and closed. A procedure may contain other procedure definitions; this facilitates invoking a procedure that may itself define another procedure.

PEND End Procedure Definition

The PEND directive terminates the procedure definition. It has the form

label	command	argument
	PEND	[list]

The list in the argument field of a PEND directive is meaningful only for procedures referenced as functions, in which case list represents the resultant value of the function; that is, the value which will be substituted for the original function reference. When a procedure is called as a command, the argument field of the PEND directive is ignored. If a procedure that has an empty argument field in its PEND line is called as a function, the resultant value is null.

Generally, the format of a command procedure appears as

```
                :                       program
name      CNAME   list    identifies the procedure
          PROC
                :                       procedure definition
                :
          PEND
```

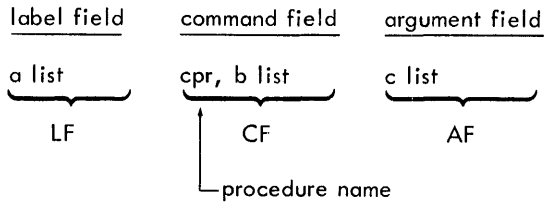
and the format of a function procedure appears as

```
                :                       program
                :
name      FNAME   list    identifies the procedure
          PROC
                :                       procedure definition
                :
          PEND    list
```

PROCEDURE REFERENCES

A procedure reference is a statement within a program that causes Meta-Symbol to assemble the procedure definition.

Command Procedure Reference. The command procedure reference line consists of a label field, a command field, an argument field, and optionally a comments field:



Within the procedure definition, the contents of the label field of the procedure reference line are referred to via the intrinsic function LF; the contents of the command field are referred to via the intrinsic function CF; and the contents of the argument field are referred to via the intrinsic function AF.

The programmer must specify in the procedure reference statement the arguments required by the procedure definition and the order in which the arguments are processed. For example, a command procedure could be written to move the contents of one area to another area of core storage. Assume that the procedure is called MOVE, and that the procedure reference line must specify in the command field which register the procedure may use. In the argument field it must specify the word address of the beginning of the current area, the word address of the beginning of the area

into which the information is to be moved, and the number of words to be moved. Such a procedure reference line could be written:

```
ANY     MOVE,2      HERE,THERE,16
```

Example 82 illustrates a command procedure and reference line.

Example 82. Command Procedure

The command procedure SUM produces the sum of two numbers and stores that sum in a specified location. The procedure reference line must consist of:

1.	label field	Use of a label is optional.
2.	command field	The name of the procedure (SUM) followed by the number of the register that the procdure may use.
3.	argument field	The word address of the first addend, followed by the word address of the second addend, followed by the word address of the storage location.
4.	comments field	Use of the comments field is optional.

The procedure definition appears as

```
SUM     CNAME
        PROC
LF      LW,CF(2)      AF(1)
        AW,CF(2)      AF(2)
        STW,CF(2)     AF(3)
        PEND
```

and the procedure reference line appears as

```
NOW     SUM,3         EARNINGS,PAY,YRTODATE
```

The resultant object code is equivalent to

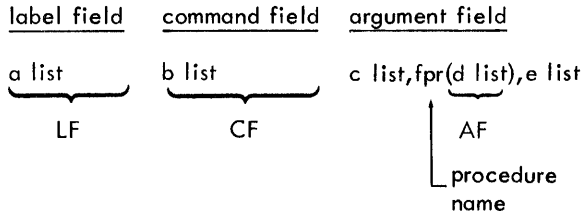
```
NOW     LW,3          EARNINGS
        AW,3          PAY
        STW,3         YRTODATE
```

Meta-Symbol defines (assembles procedure code) only for those procedure names actually referenced in the command field of command procedure reference lines. Any CNAME directive containing a procedure name not subsequently referred to by a command procedure reference line will have a skip flag (*S*) printed beside it on the assembly listing. If none of the names associated with a procedure are referenced, the same skip flag will print beside each line of the procedure as well, indicating that it has been skipped by the assembler.

The use of a label on a procedure reference line is optional. When a label is present, the procedure definition must contain the LF function in order for the label to be defined.

Conversely, if a procedure reference line is not labeled, the LF function within a procedure definition is ignored by the assembler.

Function Procedure References. A function procedure reference is different from a command procedure reference:



Within the procedure definition, the contents of the label field are referred to via the intrinsic function LF, and the contents of the command field are referred to via the intrinsic function CF. The arguments (referred to via the intrinsic function AF) of a function procedure reference consist of only those items that are enclosed by a set of parentheses and that immediately follow the name of the function procedure. Other elements may appear in the argument field of the function procedure reference line, but they are not function arguments.

The programmer must specify in the procedure reference statement what arguments are required and in what order they are processed. For example, a function procedure could be written that will return a value of the number of bit positions a given value must be shifted to right-justify it within a 32-bit field. This function procedure is shown in Example 83.

Example 83. Function Procedure

The function procedure SHIFT produces a value that indicates how many bit positions a number must be shifted in order to right-justify it within a 32-bit field. The procedure requires one argument: The rightmost bit position of the number to be shifted.

The procedure appears as

```
SHIFT    FNAME
         PROC
         PEND     AF-31
```

The function reference could appear as

```
RT       SAS,5    SHIFT(17)
```

MULTIPLE NAME PROCEDURES

The value list that appears on a particular CNAME or FNAME line can be referred to within the procedure definition via the intrinsic function NAME. This makes it possible for a procedure that can be invoked by several different names to determine which name was actually used and to modify procedure action accordingly. Example 84 illustrates this concept.

Example 84. Multiple Name Procedure

```
ALPHA    CNAME    1
BETA     CNAME    0        Identifies the procedure
         PROC
         DO       NAME
LF       GEN,32   100
         ELSE
LF       GEN,32   50
         FIN
         PEND
         .
         .
A        ALPHA
         .
         .
B        BETA
```

When this procedure is called by ALPHA at statement A the intrinsic function NAME is set to the value 1 because 1 is the value in the argument field of the CNAME directive labeled ALPHA. When the procedure is called by BETA, NAME is set to the value 0. The DO directive will cause the line

```
LF       GEN,32   100
```

to be executed if the procedure is called by ALPHA, or else the line

```
LF       GEN,32   50
```

to be executed if the procedure is called by BETA.

S:RELP Release Procedure Definitions

The S:RELP directive causes all command and function procedure definitions to be discarded, and the procedure names are set undefined. This directive has the form

label	command	argument
	S:RELP	

The S:RELP is intended for special cases where memory requirements are critical, and procedures are defined and used in such a way that they may be discarded immediately following their use. S:RELP may only be used at main program level (level 0).

PROCEDURE DISPLAY

When a procedure definition is encountered, Meta-Symbol produces on the assembly listing the symbolic code and the line numbers, but it does not output the hexadecimal equivalent of the instructions that comprise the procedure until it encounters a procedure reference line.

When a procedure reference line is encountered, Meta-Symbol produces the line number and the symbolic code for the reference line, and follows this line with the hexadecimal equivalent of the results produced by the procedure. The symbolic code defining the procedure is not shown on the assembly listing at this time. However, the user can request Meta-Symbol to display symbolic code of a procedure when assembling the procedure reference by including the directives CDISP and/or FDISP in his symbolic program.

CDISP/FDISP Command/Function Procedure Display

The command display directive CDISP has the form

label	command	argument
	CDISP	$symbol_1[,\ldots,symbol_n]$

where $symbol_i$ are the command names by which the procedure will be called.

The function display directive FDISP has the form

label	command	argument
	FDISP	$symbol_1[,\ldots,symbol_n]$

where $symbol_i$ are the function names by which the procedure will be called.

Although it is not required, it is preferred practice to place the CDISP and/or FDISP directives prior to the name declaration directives CNAME or FNAME pertaining to the procedures that are to be displayed. The display itself occurs when the procedure reference line is encountered. The format of a procedure display is shown in Figure 4.

```
nnnnn                          CDISP       SUM
nnnnn              SUM         CNAME
nnnnn                          PROC
                               :
                               :
nnnnn                          PEND
                               :
                               :
nnnnn              CALL        SUM,3       EARNINGS,PAY,YRTODATE
        *****   LEVEL 01 DISPLAY OF COMMAND SUM
nnnnn   IIIII  hhhhhhhh  LF    LW,CF(2)    AF(1)
nnnnn   IIIII  hhhhhhhh       AW,CF(2)    AF(2)
nnnnn   IIIII  hhhhhhhh       STW,CF(2)   AF(3)
        *****  END LEVEL 01 DISPLAY OF COMMAND SUM

nnnnn      next line of program
           :
           :
where

    nnnnn      is the line number.

    LEVEL 01   is the level at which the procedure is executed (see "Procedure Levels" in this chapter).

    IIIII      location counter to word level.

    hhhhhhhh   is the hexadecimal value generated for that line of code.
```

Figure 4. Command Procedure Display Format

Function procedures are displayed in a similar manner. However, because a function procedure returns a value to the procedure reference line, a function display will precede, instead of follow, the printout of its reference line. The display will include a statement identifying the level at which the procedure is executed, the fact that it is a function procedure, and the procedure's name.

PROCEDURE LEVELS

As mentioned in connection with CDISP and FDISP, Meta-Symbol assemblies involve various "levels" of execution. The main program is arbitrarily defined as level 0. A procedure referenced by the main program is designated as level 1; a procedure referenced from a level 1 procedure is designated as level 2; and so forth. For example, assume that command procedure C is to be displayed and that the main program references procedure A which references procedure B which references procedure C. Command procedure C would be displayed at level 3.

For each assembly a maximum of 32 levels is allowed, which are numbered 0 through 31 for display purposes.

INTRINSIC FUNCTIONS

Intrinsic functions are functions that are built into the assembler. The intrinsic functions BA, HA, WA, DA, concerned with address resolution, were discussed in Chapter 3. The functions CF, AF, and AFA were introduced in Chapter 4; therefore, only the extended features that are applicable to procedures are described here. The Meta-Symbol addressing function ABSVAL was discussed in Chapter 3.

The intrinsic functions discussed in this section include

LF	AFA	SCOR	S:NUMC	S:UFV
CF	NAME	TCOR	S:UT	S:IFR
AF	NUM	CS	S:PT	

Intrinsic functions may appear in any field of any instruction or assembler statement with the following exception: they must not be used in the argument field of the DEF, REF, SREF, CDISP, and FDISP directives nor in the label field of the DSECT directive.

LF Label Field

This function refers to the label field list in a COM directive or a procedure reference line. Its format is

LF(subscript list)

where LF specifies the label field, and subscript list specifies which element in that field is being referenced. If subscript list is omitted, or is zero the function references the entire label field.

Each LF reference causes Meta-Symbol to process the designated argument. That is, if the designated argument is an expression, it will be evaluated when it is used and at each point it is used, not at the time of call.

Example 85. LF Function

```
          .
          .
          .
A         SET                 LF
          .
          .
          .
TEST      TOTAL, SUM<5        (7*XYZ/SUM+57);
                              ,(5*XYZ/SUM+57)
          .
          .
          .
```

Assume that line A is a statement within a procedure definition and that line TEST is a procedure reference line. The SET directive defines the symbol A as the value of the label field of the reference line. In this example, therefore, the result would be the same as

```
A         SET                 TEST
```

CF Command Field

This function refers to the command field list in a COM directive or a procedure reference line. Its format is

CF(subscript list)

where CF specifies the command field, and subscript list specifies which element in that field is being referenced. If subscript list is omitted, or is zero the function references the entire command field.

As for LF, each CF reference causes Meta-Symbol to process the designated argument. That is, if the designated argument is an expression, it will be evaluated when it is used and at each point it is used, not at the time of the call.

Example 86. CF Function

```
          .
          .
          .
CFVALUE   SET                 CF
          .
          .
          .
ALPHA     STORE, 3, Z*Y       HOLD, 4*(A/C+8)
          .
          .
          .
```

Assume that line CFVALUE is within a procedure definition and that line ALPHA is a reference to that procedure. When the CFVALUE line is executed, Meta-Symbol will evaluate all expressions in the command field of the reference line and equate CFVALUE to the resultant value.

AF Argument Field

This function refers to the argument field list in a COM directive or a procedure reference line. Its format is

AF(subscript list)

where AF specifies the argument field, and subscript list specifies which element in that field is being referenced. If subscript list is omitted, or is zero, the function references the entire command field.

Example 87. AF Function

```
          .
          .
          .
AA        SET       AF
          .
          .
          .
XX        AOP       50, BETA/SUM
          .
          .
          .
```

Assume that statement AA is within a procedure definition and that the XX statement is the procedure reference line. In the argument field of the procedure reference line is a list of two elements. The first element consists of the value 50 and the second element consists of the value BETA/SUM. In statement AA the construct AF refers to the entire argument field list because no specific element is designated.

AFA Argument Field Asterisk

The AFA function determines whether the specified argument in a COM directive or procedure reference line is preceded by an asterisk. The format for this function is

AFA(subscript list)

where AFA identifies the function, and subscript list specifies which element in the argument field list is to be tested. If subscript list is omitted, AFA(1) is assumed.

In the case where an argument may be passed down several procedure levels, any occurrence of the argument with an asterisk prefix will satisfy the existence of the prefix.

Example 88. AFA Function

```
          .
          .
          .
          BOUND     4
          GEN, 8    AFA(1)
          .
          .
          .
XYZ       STORE, 5  *ADDR, 3
          .
          .
          .
```

Assume that the BOUND and GEN directives are within a procedure definition and that the XYZ statement is a procedure reference line. The GEN directive will generate the value 1 if the first element in the argument field of the procedure reference line (i.e., ADDR) is preceded by an asterisk. If an asterisk is not present, the GEN directive will generate a zero value.

NAME Procedure Name Reference

This function enables the programmer to reference (from within the procedure) any element of the CNAME/FNAME argument lists. Its format is

NAME(subscript list)

where NAME identifies the function, and subscript list specifies which element in the CNAME/FNAME list is being referenced. If subscript list is not specified, or is zero, NAME refers to the entire list.

A programmer can write a procedure with several entry points and assign the procedure several names via CNAME or FNAME directives. Each name may be given a unique value in the argument field of the CNAME/FNAME directive. Then, within the procedure definition the programmer can use the NAME function to determine which entry point was referenced.

Example 89. NAME Function

```
                :
                :
SINE      FNAME          1
COSINE    FNAME          2
                :
                :
          GOTO, NAME     SINE, COSINE
                :
                :
SINE
                :
                :
COSINE
                :
                :
```

Assume this represents a function procedure with two entry points: SINE and COSINE. The NAME function is set to the value 1 when the procedure is referenced as SINE and to the value 2 when the procedure is referenced as COSINE. Thus, different code will be produced depending on which name is used to reference the procedure.

Example 90. NAME Function

```
          :
          :
B         CNAME          X'68',0
BGE       CNAME          X'68',1
BLE       CNAME          X'68',2
          PROC
          BOUND          4
LF        GEN,1,7,4,3,17 AFA(1),NAME(1),NAME(2),AF(2),WA(AF(1))
          PEND
          :
          :
NOW       BLE            RETRY
```

Declares three names for the following command procedure, each with an associated list of values.

Bound on a fullword boundary.

Generates a 32-bit word with the configuration for a Branch, Branch if Greater Than or Equal to, or Branch if Less Than or Equal to instruction.

End of procedure definition.

Procedure reference line. If condition codes contain the "less than" setting (as the result of a prior operation), branch to location RETRY.

When the procedure reference line is encountered, Meta-Symbol processes the procedure. In this instance, the label NOW is defined, and Meta-Symbol generates a 32-bit word as follows:

Bit Positions	Contents
0	The value 0 because no asterisk precedes the first element in the argument field of the procedure reference line.
1-7	The hexadecimal value 68.
8-11	The hexadecimal value 2.
12-14	The hexadecimal value 0 because there is no second argument field element (i.e., no indexing specified).
15-31	The first argument field element in the procedure reference line, evaluated as a word address.

NUM Determine Number of Elements

The NUM function yields the number of elements in the designated list. Its format is

NUM(list name)

where NUM identifies the function, and list name identifies the list whose elements are to be counted. List name enclosed by parentheses is required.

The NUM function may also be used to determine the number of subfields in the label, command, and argument fields of a procedure reference line (as in NUM(LF), NUM(CF), and NUM(AF)).

Example 91. NUM Function

```
          :
          :
A         SET       8, 16, 19, 28
          :
          :
I         DO        NUM(A)
          :
          :
```

List A is composed of the elements 8, 16, 19, and 28. Because there are four elements in list A, the count for the DO-loop will be 4.

SCOR Symbol Correspondence

This function enables the programmer to test for the presence of a specified symbol on a procedure reference line.

The format of this function is

SCOR (symbol, $test_1$, $test_2$, . . . , $test_n$)

where SCOR identifies the function, symbol is the symbol to be tested, and the $test_i$ are the items with which symbol is to be compared.

Symbol can be an explicit symbol name or one of the intrinsic functions designating an element on the procedure reference line. The $test_i$ can likewise be explicit symbol names or intrinsic functions.

SCOR compares the symbol with each of the test items. The result of the comparison is the value k, where the kth item is identical to symbol. The result of the comparison is zero if there is no correspondence.

Example 92. SCOR Function

```
          :
          :
J         DO        SCOR(AF(3), MIN, LIMIT, MAX)
          :
          :
A         TALLY, 2, 3   HOLD, TEMP, LIMIT
          :
          :
```

Assume line J is within a procedure definition and that line A is a reference line to that procedure. When line J is processed, Meta-Symbol compares the third element in the argument field of the reference line (LIMIT) with the symbols MIN, LIMIT, and MAX. The resultant value is 2 since LIMIT is the second symbol listed for the SCOR function, and the DO-loop will be executed twice.

SCOR has many possible applications in procedures. To fully understand its use it is important to note that Meta-Symbol first substitutes designated items from the procedure reference line for any intrinsic functions used as SCOR arguments, and then evaluates the SCOR function. This is made clearer by the following example:

Example 93. SCOR Function

```
          :
          :
SUM       CNAME
          PROC
          :
          :
X         SET       SCOR(C, AF(2))
          :
          :
Y         SET       SCOR(AF(1), AF(2))
          :
          :
Z         SET       SCOR(AF(2))
          :
          :
          PEND
          :
          :
K         SUM       A, (B, C, A, D)
```

Lines X, Y, and Z are within the definition of procedure SUM, and line K is a reference to that procedure. When the procedure is called and line X is subsequently processed, its argument field will have the internal configuration

SCOR(C, B, C, A, D)

SCOR will therefore produce the value 2, since C corresponds to the second test item, and X will be set to 2. When line Y is processed, its argument field will have the internal configuration

SCOR(A, B, C, A, D)

SCOR will produce the value 3, since A corresponds to the third test item, and Y will be set to 3. When line Z is processed, its argument field will have the internal configuration

SCOR(B, C, A, D)

SCOR will produce the value zero, since B does not correspond to any of the test items, and Z will be set to zero.

TCOR Type correspondence

The TCOR function compares the value type of a specified item with the value types of a given list of test items. The format of this function is

$$TCOR(item, test_1, test_2, \ldots, test_n)$$

where TCOR identifies the function, item designates which item is to be compared, and the $test_i$ are elements whose value types are to be compared with that of the designated item. Item may be any symbol, constant, evaluatable expression, or any element on a procedure reference line. The $test_i$ may be the same kind of elements as item or any of the following value type intrinsic symbols:

Symbol	Type
S:RAD	Relocatable address
S:LIST	List
S:AAD	Absolute address
S:EXT	External reference
S:FR	Forward reference to global symbol
S:LFR	Forward reference to local symbol
S:UND[†]	Undefined global symbol
S:SUM	Expression involving relocatable addresses, externals, or forward references
S:INT	Integer constant
S:DPI	Double precision integer constant
S:C	Character constant
S:D	Decimal constant
S:FX	Fixed decimal constant
S:FS	Floating short constant
S:FL	Floating long constant

TCOR is most commonly used to determine the value type of an item by comparing it with one or more of the above list of value type intrinsic symbols. If the value type of the item corresponds to the type of one of the given symbols, TCOR returns the value k, where the kth symbol's type is the same as that of the item. If there is no correspondence, a zero value is produced by the function.

It is important to note, however, that TCOR is not restricted to using only the value type intrinsic symbols as $test_i$. Any symbol, constant, or evaluatable expressions may be given, and TCOR will return a value indicating which one corresponds in type to "item".

Example 94. TCOR Function

```
          :
          :
A         CNAME
          PROC
          :
          :
K         DO              TCOR(AF, S:FL, S:DPI)>0
L         DATA,8          AF
          ELSE
M         DATA            AF
          FIN
          PEND
          :
          :
N         A               FL'5'
P         A               16
          :
          :
```

Lines K, L, and M are within the definition of procedure A, and lines N and P are references to the procedure. When line N is processed, Meta-Symbol compares its argument field (FL'5') with the list of value type intrinsic symbols on line K. The argument FL'5' is a floating long constant and corresponds to intrinsic symbol S:FL. The TCOR function therefore produces the value 1 (since the correspondence is to the first test item on line K). This value is then compared against zero, and since the result of this logical operation (1>0) is "true", line L is processed. Line L produces a 64-bit (8-byte) data word containing the value 5 as a floating-point long constant.

Meta-Symbol performs the same kind of operation when line P is processed. But since 16 is a decimal integer constant corresponding to neither S:FL nor S:DPI, TCOR returns a value of zero, the result of the logical operation 0 > 0 is "false", and line M is processed instead of line L. Line M produces a 32-bit data word containing the value 16 as a decimal integer constant.

[†]Use extreme care with S:UND, as its misuse makes the program sensitive to the two different assembly passes of Meta-Symbol. Pass 1 of the assembler cannot detect truly undefined symbols; it must assume that an undefined symbol is a forward reference (S:FR). Pass 2 of the assembler detects truly undefined global symbols; thus the value of TCOR(UNDEF, S:UND) is zero on Pass 1 and one on Pass 2 if UNDEF is truly undefined.

Example 95. TCOR Function

```
          :
A         CNAME
          PROC
          :
B         SET             TCOR(AF(1),$,5,'A')
          :
          :
          PEND
          :
C         A               17,'PDQ'
          :
          :
D         A               FL'75'
          :
          :
```

Line B is within the definition of procedure A, and lines C and D are references to the procedure. When line C is processed, its first argument field is compared against the list of test items on line B. Since 17 does correspond in type to the second test item (both are decimal integer constants), TCOR produces the value 2, and B is SET to 2. When line D is processed, its first argument field does not correspond to any of the test items on line B; B is therefore SET to zero.

S:UFV Use Forward Value

S:IFR [t] Inhibit Forward Reject

The S:UFV intrinsic function is used to alter the manner in which the assembler processes global forward references. Its format is

S:UFV(item)

where

S:UFV identifies the function.

item represents any valid Meta-Symbol construct (symbol, intrinsic function, expression, list, etc.).

---

[t] S:IFR is simply an alternate name for the S:UFV function; there is no difference in the action of the two. S:UFV is used in the examples because it seems more descriptive of the actual use of the function.

Example 96. S:UFV Function

At a point prior to the definition of SWITCH, it is desired to generate a data word in one of three formats, depending on the value of SWITCH. Since only one word will be generated in any case, the correct format should be selected during Pass 2. The S:UFV function makes this simple to accomplish.

```
START     CSECT
          :
          :
          GOTO,S:UFV(SWITCH)    X,Y
          GEN,3,10,19     SWITCH,X'13',BA($)-START   Selected on Pass 1
          GOTO            Z
X         BOUND           1
          GEN,3,11,18     SWITCH,X'7',HA($)-START
          GOTO            Z
Y         BOUND           1
          GEN,3,12,17     SWITCH,X'3',WA($)-START    Selected (and generated) on Pass 2
Z         BOUND           1
          :
          :
SWITCH    EQU             2
          :
          :
```

Example 97. S:UFV and TCOR Functions

Normally, the TCOR function will match any global forward reference with S:FR. Use of S:UFV allows the actual type to be found during Pass 2 assembly.

```
          CSECT
          :
          :
          DATA            TCOR (X, S:FR, S:RAD)          Generates DATA 1
          :
          :
          DATA            TCOR(S:UFV(X), S:FR, S:RAD)    Generates DATA 2
          :
          :
X         EQU             $
          :
          :
```

Meta-Symbol is a two-pass assembler. In order to maintain identical address assignments and to calculate the same values on both assembly passes, certain restrictions are placed on the use of forward references to symbols. For instance, directives that may directly or indirectly affect address assignment (RES, BOUND, ORG, LOC, DO, WHILE, etc.) may not contain a forward reference in their argument field. If a forward reference is used with such directives, the value zero is used on both passes, and a diagnostic is given on Pass 2 of the assembly.

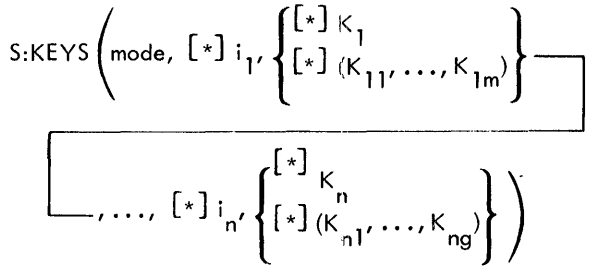
The "normal" processing of forward references (as in the argument field of a DATA or GEN directive) is for Pass 1 to ignore forward references and eventually define all global symbols, and for Pass 2 to then use the value assigned to the symbol during Pass 1. In certain cases, this behavior may be desired even in directives where forward references are normally illegal. The S:UFV function is used to achieve this.

During Pass 1 of the assembly, S:UFV returns an integer zero if its argument is a forward reference; otherwise, its value is the argument itself. During Pass 2 of the assembly, S:UFV returns the value assigned by Pass 1 and inhibits the diagnostic that would occur if the global forward reference was used in a normally illegal context (see Example 96).

The S:UFV function may be used in conjunction with the TCOR intrinsic function in order to determine the type of a global forward reference (see Example 97).

S:KEYS Keyword Scan

This intrinsic function, which may be used only within procedures, permits one to easily scan a procedure reference argument field for the presence of specified keywords. This scan can return information specifying how many and which keywords are present as well as where in the argument field each keyword appears. The value returned by S:KEYS is a linear list of two or more elements. The first element is a keyword "hit" count. The second element is a parameter/flag presence word that indicates which keywords (up to a maximum of 32) were hit. The remaining elements are indexes that specify where in the reference line argument field the various parameter keywords occurred. The form of the function is:



where

mode is an expression that evaluates to $0 \leq \text{integer} \leq 7$.

(mode&1)>0 specifies that AF(1) of the PROC reference argument field should not be scanned.

(mode&2)>0 specifies use of NUM(AF)+1 as a default index for parameter misses.

(mode&4)>0 specifies suppression of "unrecognized key" error reporting.

[*] i_k is an explicit integer ($0 \leq i_k$) which specifies that the i_kth bit of the parameter/flag presence word is to be associated with the keyword K_k or the keywords ($K_k1, K_k2, \ldots, K_{km}$). If $i_k > 31$, subsequent keywords will not affect the parameter/flag presence word.

If i_k is preceded by an asterisk, then any subsequent keyword occurring prior to [*] i_{k+1} is considered a parameter, in which case a hit on the first or any subsequent keyword causes the specified bit in the parameter/flag presence word to be turned on and causes the concatenation to the S:KEYS list of an element that specifies which subfield in the reference line contained the specified word.

If i_k is not preceded by an asterisk, then any subsequent keyword occurring prior to [*] i_{k+1} is considered a flag, in which case only the specified bit of the parameter/flag presence word and hit count are affected. If more than one keyword is specified for a given presence bit, then a hit on the first keyword turns the presence bit on while a hit on any other keyword has no effect.

[*]K_k and [*]($K_k1, \ldots, K_k3$) are any legal symbols. These are the keywords associated with the specified bit position. A leading asterisk indicates that a hit is required, provided that K_k is a parameter.

ABBREVIATED SYNTAX

If [*] i_1, is omitted, *0 is assumed.

If [*] i_{k+1} is omitted, [*] i_k +1 is assumed.

Example 98. S:KEYS Abbreviated Syntax

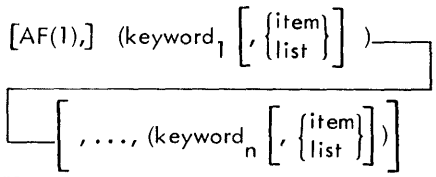
S:KEYS(1, *0, A, *1, (B, C), *17, D, 18, E, 19, F)

may be abbreviated

S:KEYS (1, A, (B, C), *17, D, 18, E, F)

SYNTAX OF THE SCANNED ARGUMENT FIELD

S:KEYS, evaluated within a PROC, causes a scan of the argument field of the PROC reference. That argument field is expected to have the form



where

AF(1) is not scanned if (mode&1)>0; hence its structure is not significant to S:KEYS.

keyword is a keyword that will be looked at by S:KEYS and compared with the K_n and K_{nm} in the S:KEYS argument field.

item/list is any item or list of items that are to be associated with a given keyword. When present, the keyword is normally used as a parameter rather than a flag. The term "item" is used because there are no restrictions, other than syntactic, on what an item may be.

Notice that S:KEYS interrogates only the first subelement of each subfield of the scanned argument field.

If a given argument of the scanned argument field contains a keyword without an associated item (or list), then as far as S:KEYS is concerned, the parentheses around that argument field are redundant.

That is,

(KEY1, 25), (KEY2), (KEY3, 17, 42)

could be written

(KEY1, 25), KEY2, (KEY3, 17, 42)

USAGE EXAMPLES

Example 99. S:KEYS Usage Example

Assume a PROC reference line as follows:

HERE PROC$REF (D, 9), (A)

Equivalent notation is

HERE PROC$REF (D, 9), A

Assume the PROC PROC$REF contains the line

P SET S:KEYS(0, 26, A, 27, B, 28, C, 29, D, 30, E, 31, F)

then

mode = 0

all keywords are flags

hits occur on A and D

P will be defined as the list of two elements formed by S:KEYS

P(1) = 00000002 (hit count)

P(2) = 00000024 (in binary 0000...0010 0100)

bit 29

bit 26

Equivalent notation is

P SET S:KEYS(0, 26, A, B, C, D, E, F)

Example 100. S:KEYS Usage Example

Suppose the PROC from Example 99 contained the line

Q SET S:KEYS(0, *26, A, *27, B, *28, C, *29, D, *30, E, *31, F)

then

mode = 0

all keywords are parameters

hits occur on A and D

Q will be defined as the list of four elements

Q(1) = 00000002
Q(2) = 00000024 } same as P(1) and P(2) above
Q(3) = 00000002 parameter A is in AF(2)
Q(4) = 00000001 parameter D is in AF(1)

Note the power gained by having this list. Without knowing where in the scanned argument field the keyword D is written, references to the keyword associated value, 9, can be parameterized as AF(Q(4), 2).

Equivalent notation is

Q SET S:KEYS(0, *26, A, B, C, D, E, F)

Example 101. S:KEYS Usage Example

Suppose the PROC from Example 99 contained the line

```
R    SET    S:KEYS(2, *26, A, B, C, D, E, F)
```

then

mode = 2 (use default indexes for parameter misses)

all keywords are parameters

hits occur on A and D

misses occur on B, C, E, and F

R will be defined as the list of eight elements

R(1) = 00000002 R(2) = 00000024	same as P and Q above
R(3) = 00000002	A - hit in AF(2)
R(4) = 00000003	B - miss, point at null argument which evaluates to 0
R(5) = 00000003	C - miss
R(6) = 00000001	D - hit in AF(1)
R(7) = 00000003	E - miss
R(8) = 00000003	F - miss

An advantage of default parameter indexes is that they permit a less complex parameterization since, for example, R(5) may always be associated with the parameter C, regardless of how many and which parameters are hit. If NUM(AF(R(5)))>0 (i.e., not null), then C is present. It is also true, since C is a parameter, that bit 28 of R(2) will be on if and only if C is present.

Example 102. S:KEYS Usage Example

Assume the function PROC reference line

```
NOW    SET    SUMTHIN((H, (4, 3)), K,
     (L, F:THERE), (M, 4), N)
```

where the function PROC SUMTHIN contains the line

```
Z      SET      S:KEYS(0, *17, L, H, 4, N, *(A, K),
     *8, (S, D), M)
```

then

mode = 0

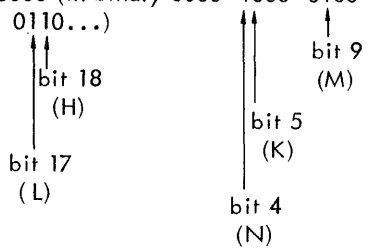
the keywords L, H, S, D, and M are parameters

the keywords N, A, and K are flags

hits occur on L, H, N, K and M

Z will be defined as the list of five elements

Z(1) = 00000005 (hit count)
Z(2) = 08406000 (in binary 0000 1000 0100 0000 0110...)



Note that bit 5 is off. K is not the first flag listed for this bit.

Z(3) = 00000003 the parameter L is in AF(3)
Z(4) = 00000001 the parameter H is in AF(1)
Z(5) = 00000004 the parameter M is in AF(4)

Note that the order in which the indexes appear in list P is not the bit-number order of P(2), but instead the order of left-to-right occurrence of the parameter keywords in the S:KEYS argument field.

Example 103. S:KEYS Usage Example

Suppose the PROC SUMTHIN from Example 102 contained the line

```
T      SET      S:KEYS(1, *17, L, H, 4, N, *(A, K),
                    *8, (S, D), M)
```

then

mode = 1 (AF(1) should not be scanned)

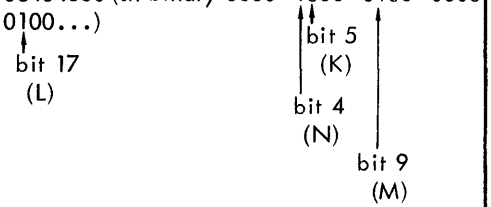
the keywords L, H, S, D, and M are parameters

the keywords N, A, and K are flags

hits occur on L, N, K, and M but not on H

T will be defined as the list of four elements.

T(1) = 00000004 hit count
T(2) = 08404000 (In binary 0000 1000 0100 0000 0100...)



T(3) = 00000003 the parameter L is in AF(3)
T(4) = 00000004 the parameter M is in AF(4)

Example 104. S:KEYS Usage Example

Suppose the PROC SUMTHIN from Example 102 contained the line

```
Y     SET     S:KEYS(3,*17,L,H,4,N,*(A,K),
                  *8,(S,D),M)
```

then

mode = 3 (AF(1) should not be scanned; and default indexes are to be used for parameter misses.

the keywords L, H, S, D, and M are parameters

the keywords N, A, and K are flags

hits occur on L, N, K, and M

misses occur on H, A, S, and D

Y will be defined as the list of six elements

```
Y(1) = 00000004
Y(2) = 08404000   same as T(1) and T(2) above
Y(3) = 00000003   L - hit in AF(3)
Y(4) = 00000006   H - miss, point at AF(6), a null
Y(5) = 00000006   S or D - miss
Y(6) = 00000004   M - hit in AF(3)
```

Example 105. S:KEYS Usage Example

Assume the PROC Definition

```
A$PROC   CNAME
         PROC
P        SET     S:KEYS(2,W,X,Y,Z)
         DATA    AF(P(3),2),AF(P(4),2),
                 AF(P(5),2),AF(P(6),2)
         PEND
```

Now assume the PROC reference line

```
         A$PROC  (Z,7),(X,-1)
```

P will be defined, for this reference of A$PROC, as the list

```
P(1) = 00000002
P(2) = 50000000
P(3) = 00000003
P(4) = 00000002
P(5) = 00000003
P(6) = 00000001
```

This reference to A$PROC will cause four words of data to be generated as follows:

```
00000000   (AF(3,2) is null)
FFFFFFFF   (AF(2,2) is -1)
00000000   (AF(3,2) is null)
00000007   (AF(1,2) is 7)
```

CS Control Section

This function returns the control section number of any item whose value is a relocatable address. The format of this function is

```
CS(item)
```

where CS specifies control section, and item is the element whose control section is to be determined. Control section, a value ranging from 1 to the total number of control sections, was discussed in the previous chapter under "Program Section Directives", and is the same as that appearing on the assembly listing for SET and EQU directives. If the value of the item given is not a relocatable address, a zero value is returned.

Example 106. CS Function

```
          .
          .
          CSECT
A         DATA    7
          CSECT
B         DATA    14
          .
          .
C         DATA    CS(A),CS(B),CS(-85)
```

When line C is processed, the first CS function returns a value of 1 because item A is a relocatable address within a control section 1; Meta-Symbol generates a 32-bit data word containing the value 1. The next CS function is evaluated and returns a value of 2 because item B is a relocatable address within control section 2; Meta-Symbol generates a 32-bit data word containing the value 2. The last CS function is evaluated and returns a value of zero because item -85 is not a relocatable address; Meta-Symbol generates a 32-bit data word containing the value zero.

S:NUMC Number of Characters

This function returns an integer count of the total number of characters found in its evaluated argument. Its format is

```
S:NUMC(item)
```

where S:NUMC identifies the function, and item designates the element or list for which a character count is to be calculated. Any element in the evaluated argument other than a character string is ignored in calculating the total count. Note that an element in the list which is itself a list (i.e., a sublist) is thus ignored in the count.

If no character constants are found in the evaluated argument, S:NUMC returns a count of zero. No restriction is imposed on the magnitude of the final count, although no one character string may have a character count greater than 255.

Example 107. S:NUMC Function

```
If A is defined as

     A     SET     'THESE', 'ARE', 'STRINGS'

then

     Q     SET     S:NUMC(A)

assigns the value 15 to Q.

However, if A were defined as

     A     SET     'THESE', ('ARE', 'STRINGS')

then

     Q     SET     S:NUMC(A)
     R     SET     S:NUMC(A(1), A(2))

assigns the value 5 to Q and the value 15 to R.
```

Example 108. S:UT Function

```
If A is defined as

     A     SET     'THIS', 'IS', 'A', 'STRING'

then

     Q     SET     S:UT(A(1), A(2), A(3), 'NEW', A(4))

creates a string Q as if Q had been defined as

     Q     SET     'T', 'H', 'I', 'S', 'I', 'S', 'A', ;
                   'N', 'E', 'W', 'S', 'T', 'R', 'I', 'N', 'G'

Suppose that A had been defined as

     A     SET     ('THIS', 'IS', 'A'), 'STRING'

then

     Q     SET     S:UT(A)

creates a string Q as if Q had been defined as

     Q     SET     ('THIS', 'IS', 'A'), ;
                   'S', 'T', 'R', 'I', 'N', 'G'
```

S:UT Unpack Text

This function provides the facility for manipulating character strings of arbitrary length. It unpacks a character string into a sequence of single-character elements. Its format is

S:UT(item)

where S:UT identifies the function, and item designates the element or list which is to have its text-valued elements "unpacked". Any element in the argument list other than a character constant remains unchanged, although its relative position in the value list may change as a result of other unpacking operations. Note that an element in the argument list which is itself a list (i.e., a sublist) is thus left unchanged.

Care should be taken that the value list contains no more than 255 elements as a result of unpacking several text elements.

Note that, for a given list, Q, the relationship NUM(S:UT(Q)) = S:NUMC(Q) holds only if Q is a linear list composed entirely of character constants.

S:PT Pack Text

This function transforms any sequence of character constants and nulls into a single character string. Its format is

S:PT(item)

where S:PT identifies the function, and item designates the list to be "packed". During packing, any null elements are discarded. After all nulls are eliminated, any contiguous character constants are concatenated to form a single character string, provided that the resultant string contains no more than 255 characters. If it does contain more, an error message is given, and only the leftmost 255 characters are used. This does not terminate packing; the remaining characters are simply discarded.

Any element in the argument list other than a character constant or a null is left unchanged, although its relative position in the value list may change as a result of other packing operations. Note that an element in the list which is itself a list (i.e., a sublist) is thus left unchanged.

If the argument consists only of a null or a list of nulls, the value of S:PT is a single null.

Example 109. S:PT Function

Assume that the following definitions are made:

```
A     SET     'THIS'
B     SET     ' IS A '
C     SET     'STRING'
```

then

```
Q     SET     S:PT(A, B, 'BIGGER ', C)
```

assigns the same value to Q as if Q had been defined as

```
Q     SET     'THIS IS A BIGGER STRING'
```

Example 110. Character String Functions

This function procedure is called with three arguments. The first argument is a string that is to be searched for occurrences of the character in the second argument. If such a match is found, that character in the string is replaced by the character in the third argument. The value of the function is the new string after substitution. The definition is

```
REPL        FNAME       Defines function REPL
            PROC
            LOCAL I, Q

Q     SET       S:UT(AF(1))  Forms character list

I     DO        NUM(Q)

      DO1       Q(I)=AF(2)

Q(I)  SET       AF(3)        Substitutes on match

      FIN

      PEND      S:PT(Q)      Returns new string
```

Now, if A is defined as

```
A     SET       '- THIS IS A STRING -'
```

a call on the function such as

```
STR1  TEXT      REPL(A, ' ', '.')
```

generates the text string

```
'-.THIS.IS.A.STRING.-'
```

while the following call

```
STR2  TEXT      REPL(REPL(A, '-', '$'), ' ', '-')
```

generates the text string

```
'$-THIS-IS-A-STRING-$'
```

Notice that, in the above example, had the function nesting been reversed, as

```
STR3  TEXT      REPL(REPL(A, ' ', '-'), '-', '$')
```

the resulting text string would have been

```
'$$THIS$IS$A$STRING$$'
```

PROCEDURE REFERENCE LISTS

A list composed only of elements that are evaluated when Meta-Symbol encounters the list in a statement is referred to as a "value list", as discussed in Chapter 2. A list having at least one element that cannot be evaluated when first encountered is called a "procedure reference list". For example, the directives SET, EQU, GEN, and COM require value lists, because the elements must be evaluated before the assembler can process the directives. Command and function procedure reference lines require procedure reference lists, because the list elements are not evaluated at the time the reference line is encountered, but are acted upon within the procedure.

A list used in a procedure reference line cannot be distinguished from a value list merely by appearance. That is, the list may be either a procedure reference list or a value list depending on its use in a program. If it appears in a directive such as SET or GEN

```
R     SET             5, A

      GEN, 16, 16     5, A
```

the list is a value list and is evaluated by Meta-Symbol at the time it is encountered. However, if the list appears in a command or function procedure reference line, it is a procedure reference list. For example, if there were a command procedure name SUM, the reference line could appear as

```
NOW   SUM       TABLE, 15*(TABLE2+;
                TABLE)/4
```

When Meta-Symbol encounters this line, it will execute the SUM procedure, and the elements of the named lists will be evaluated depending on their use within the procedure. That is, if LF is referenced within the procedure, NOW becomes a defined symbol and is stored in the symbol table. If LF does not appear within the procedure, the label on the reference line is lost. The same principle applies to the elements of command field and argument field lists.

Example 111. Procedure Reference Lists

```
          :
          :
ALL       SET     AF              Assumes these statements
AF(1)     SET     AF(2,2)         are within a procedure
AF(3)     SET     ALL(2,2)        definition called LST.
          :
          :
A         SET     (11,12,13)
B         SET     (21,22,23)      Main program.
C         SET     (31,32,33)
          :
          :
          LST     A,B,C           Procedure reference line.
```

The three elements (A, B, C) on the procedure reference line may be referred to within the procedure as

AF(1) = A
AF(2) = B
AF(3) = C

Notice, however, that the functions AF(1), AF(2), and AF(3) apply only to the symbols that actually appear on the procedure reference line (i.e., A, B, and C) and not to the values that have been equated to them. Thus, the statement

```
AF(1)     SET     AF(2,2)
```

results in AF(1) – which is A – being set to null because there is no element AF(2,2) on the procedure reference line.

On the other hand, the statement

```
ALL       SET     AF
```

causes Meta-Symbol to evaluate the symbols A, B, and C, and to assign ALL as

```
ALL       SET     (11,12,13),(21,22,23),(31,32,33)
```

Therefore, the element AF(3) – which is C – can be set to ALL(2,2) which has the value 22.

Example 112. Procedure Reference Lists

The procedure OUT generates a 32-bit value equal to the number of elements in the list of the procedure reference line:

```
OUT       CNAME                       Declares the command name of the procedure to be OUT.
          PROC                        Identifies a procedure.
LF        GEN,32          NUM(AF)     Generates 32 bits containing the number of elements in the argu-
          :                           ment field of the procedure reference line.
          :
          PEND                        Signifies the end of the procedure.
```

The following reference lines could call the procedure:

```
FIRST     OUT     3,6,(4,7)           Generates 00000003 (hexadecimal).
A         SET     3,6
B         SET     (4,7)
TWO       OUT     A,B                 Generates 00000002 (hexadecimal).
```

The list in line FIRST consists of three elements: 3,6, and (4,7); therefore, the procedure OUT generates the value 3. Next, A is defined as a value list of two elements: 3 and 6; and B is defined as a value list of one element: (4,7). The list in line TWO consists of two elements: A and B. Meta-Symbol does not determine what values A and B have because there is no statement within the procedure that causes Meta-Symbol to evaluate the argument field list.

```
OUT       CNAME
          PROC
          LOCAL     COUNT             Declares COUNT to be a local symbol within this procedure.
COUNT     SET       AF                COUNT is SET to the value of the list in the argument field of the
                                      procedure reference line.
LF        GEN,32    NUM(COUNT)
```

Since COUNT is declared to be a local symbol within this procedure, it cannot be confused with any previously defined symbol "COUNT". When the SET directive is executed, Meta-Symbol must evaluate the list in the argument field of the procedure reference line in order to assign a value to COUNT. With this procedure, the reference lines

```
FIRST    OUT    3,6,(4,7)      Generates 00000003 (hexadecimal).
A        SET    3,6
B        SET    (4,7)
TWO      OUT    A,B            Generates 00000003 (hexadecimal).
```

now generate the same value. When the procedure is called at line TWO, the list consists of A, B. The directive

```
COUNT    SET    AF
```

executed within the procedure, causes Meta-Symbol to evaluate A and B and to assign COUNT as

```
COUNT = 3,6,(4,7)
```

Thus, NUM(COUNT) yields the value 3.

Notice that although NUM(COUNT) now equals 3, NUM(AF) still equals 2. This is true because the elements A and B in the reference line are not replaced by their values (3,6, and (4,7)). Thus a procedure can refer to the elements that actually appear on the procedure reference line as well as the values of the elements.

Example 113. Procedure Reference Lists

Assume the command procedure CHECK

```
CHECK    CNAME
         PROC
         LOCAL    CNT
CNT      SET      AF
         :
         :
H        DO       NUM(CNT)
         :
         :
J        DO       NUM(AF)
         :
         :
```

is called as follows:

```
UPPER    SET      16,24,32
LOWER    SET      9,11,13
LIMIT    SET      12,18
         :
         :
FIELD    CHECK    UPPER, LOWER, LIMIT
         :
         :
```

In the CHECK procedure CNT is defined as

```
CNT = 16,24,32,9,11,13,12,18
```

Therefore, the DO directive at line H has a count of 8 because CNT is a list of eight elements. On the other hand, the DO directive at line J has a count of 3 because NUM(AF) determines how many elements are in the argument field list of the reference line, and there are three: UPPER, LOWER, and LIMIT.

The use of procedure reference lists is not limited to the argument field. A list appearing in any field in a procedure or function reference line is a procedure reference list.

Example 114. Procedure Reference Lists

The statement

```
A,C,D    TABSIZ,S,T,U    X,Y,Z
```

could be a reference line for a command procedure that adds the items identified in the label field to those identified in the command field and stores the results in the locations identified in the argument field: i.e.,

A+S ⟶ X, C+T ⟶ Y, D+U ⟶ Z

All three lists are evaluated inside the procedure when the actual addition occurs:

```
TABSIZ   CNAME
         PROC
I        DO       NUM(LF)
AF(I)    SET      LF(I)+CF(I+1)
         FIN
         PEND
```

The loop is to be executed NUM(LF) or 3 times. Each time through the loop, I is incremented by 1, so AF(I) references element X, Y, and Z; LF(I) references element A, C, and D; and CF(I + 1) references element S, T, and U. Therefore, the SET directive is equivalent to

```
X        SET      A+S
Y        SET      C+T
Z        SET      D+U
```

PROCs are frequently used to define machine instructions. In this manner, a programmer can use any mnemonic code he wishes for an instruction by writing a procedure definition that will generate the appropriate bit configuration. This is another instance when it is necessary for the programmer to remember that lists in procedure reference lines are not evaluated at the time they are encountered but rather at the time they are used inside the procedure.

Example 115. Lists in Procedures

Assume a procedure LOAD is to be written that produces the same bit configuration as a Load Word instruction. The procedure definition could be written

```
LOAD    CNAME           X'32'
        PROC
        LOCAL           P
P       SET             AF
LF      GEN,1,7,4,3,17  AFA(1),NAME;
                        ,CF(2),P(2),P(1)
        PEND
```

If the procedure is called by

```
        LOAD,4          *Z,5
```

the procedure functions as follows:

1. P is declared a local symbol.

2. P is SET to the value of the argument field of the procedure reference line; i.e.,

   P = Z,5

3. In the GEN directive

   a. LF causes Meta-Symbol to determine whether a label exists on the procedure reference line and, if one does, to define it.

   b. AFA(1) tests to determine whether an asterisk appeared as the first symbol in the argument field of the reference line. If an asterisk did appear, a 1 is generated for bit position zero of the instruction word; if an asterisk did not appear, a 0 is generated for that bit position.

   c. NAME causes Meta-Symbol to place the value X'32' (from the argument field of the CNAME directive) in bits 1 through 7 of the word being formed.

   d. CF(2) specifies that the second entry in the command field of the reference line is to be assembled into the next four bits (i.e., bit positions 8 through 11).

   e. P(2) designates the second element of list P. Since P = Z,5, its second element is 5. This value is assembled into bit positions 12 through 14 of the word.

   f. P(1) designates the first element of list P, i.e., Z. This value is assembled as a 17-bit address.

The same procedure will operate properly when called in this fashion:

```
Q       EQU             Z,5
        LOAD,4          *Q
```

because inside the procedure the directive

```
P       SET             AF
```

forces Meta-Symbol to evaluate the argument field of the procedure reference line and, therefore, to SET P:

P = Z,5

If the procedure were written

```
LOAD    CNAME           X'32'
        PROC
LF      GEN,1,7,4,3,17  AFA(1),NAME;
                        ,CF(2),AF(2),AF(1)
        PEND
```

and called by

```
Q       EQU             Z,5
        LOAD,4          *Q
```

it would not operate properly. There is no directive within this procedure definition to cause Meta-Symbol to evaluate the argument field of the procedure reference. Thus, when the GEN directive is processed, the asterisk, the NAME entry, and the command field item are handled correctly, but there is no AF(2) entry on the procedure reference line since the argument field consists only of *Q.

Thus, it can be seen that lists in procedure reference lines are conditional in that Meta-Symbol evaluates them only if there is an instruction or directive within the procedure that causes it to do so; otherwise, the lists are passed directly from the reference line to the procedure.

SAMPLE PROCEDURES

The following examples illustrate various uses of procedures, such as how one procedure may call another, and how a procedure can produce different object code depending on the parameters present in the procedure reference.

Example 116. Conditional Code Generation

This procedure tests element N in the procedure reference line to determine whether straight iterative code or an indexed loop is to be generated. If N is less than 4, straight code will be generated; if N is equal to or greater than 4, an indexed loop will be generated. In either case, the resultant code will sum the elements of a table and store the result in a specified location.

The procedure definition is

```
ADDEM   CNAME
        PROC
LF      SW, AF(3)     AF(3)
IND     DO            (AF(2)\4)*AF(2)
        AW, AF(3)     AF(1) + IND -1
        ELSE
        LW, AF(5)     L(-AF(2))
        AW, AF(3)     AF(1) + AF(2), AF(5)
        BIR, AF(5)    $ - 1
        FIN
        STW, AF(3)    AF(4)
        PEND
```

The general form of the procedure reference is

```
        ADDEM     ADDRS, N, AC, ANS, X
```

where

ADDRS represents the address of the initial value in the list to be summed.

N is the number of elements to sum.

AC is the register to be used for the summation.

ANS represents the address of the location where the sum is to be stored.

X is the register to be used as an index when a loop is generated.

For the procedure reference

```
XYZ     ADDEM     ALPHA, 2, 8, BETA, 3
```

machine code equivalent to the following lines would be generated in-line at assembly time.

```
XYZ     SW, 8     8          Clears the register.
        AW, 8     ALPHA      Adds contents of ALPHA to register 8.
        AW, 8     ALPHA+1    Adds contents of ALPHA + 1 to register 8.
        STW, 8    BETA       Stores answer.
```

If the procedure reference were

```
        ADDEM     ALPHA, 5, 8, BETA, 3
```

the generated code would be equivalent to

```
        SW, 8     8          Clears the register.
        LW, 3     L(-5)      The value -5 would be stored in the literal table and its address
                             would appear in the argument field of this statement. Thus, load
                             index with the value -5.

        AW, 8     ALPHA+5, 3 Register 3 contains -5, ∴ ALPHA+5-5  ALPHA.
        BIR, 3    $-1        Increments register 3 by 1 and branches.
        STW, 8    BETA       Stores answer.
```

Example 117. Function Procedures

Assume that a 32-bit element of data consists of three fields: Field A occupies bits 0 through 6, field B occupies bits 7 through 17, and field C occupies bits 18 through 31. The program that uses this data will frequently need to alter the contents of the fields. Two function procedures could be written to facilitate this process: SHIFT and MASK. The procedure SHIFT returns a value equal to the number of bit positions that a quantity must be shifted to right-justify it within the 32-bit area. The procedure MASK produces a field of all 1's that occupy the required number of bits to mask a given field.

The procedure definitions could be:

```
SHIFT   FNAME
        PROC
        LOCAL   SYM
SYM     SET     AF
        PEND    31-SYM(2)
        .
        .
        .
MASK    FNAME
        PROC
        LOCAL   VAL,ARG
ARG     SET     AF
VAL     SET     (1**(ARG(2)-ARG(1)+1)-1)**(31-ARG(2))
        PEND    L(VAL)
        .
        .
        .
```

The sequences of code needed to reference these procedures include:

```
        .
        .
        .
A       EQU     0,6
B       EQU     7,17            Defines fields A, B, and C.
C       EQU     18,31
        .
        .
        .
        LW,4    L(5)
        SAS,4   SHIFT(B)
        LW,5    MASK(B)         Stores the value 5 into field B of data area Q.
        STS,4   Q
        .
        .
        .
```

The EQU directives define the bits that comprise each of the three data fields.

The first Load Word instruction uses a literal constant for the value 5. The Arithmetic Shift instruction references the SHIFT procedure, using as its argument the list B (defined as 7, 17). The SHIFT function procedure will return the value 14, because an integer must be shifted 14 bit positions in order to right justify it in the B field (i.e., in bits 7 through 17).

The second Load Word instruction references the MASK procedure with an argument of B. The MASK procedure first determines the number of bits in the specified field: ARG(2) - ARG(1) + 1 = 17 - 7 + 1 = 11. Then, the number 1 is shifted left that number of bit positions. Next, the value 1 is subtracted from the shifted value, forming the desired mask of eleven 1-bits. To position the mask for the correct data field requires shifting it left 14 positions. This is determined by subtracting the value ARG(2) (i.e., 17) from 31. The correctly positioned mask is assigned to the label VAL. On the PEND line, VAL appears as a literal, so the mask is stored in the literal table and its address is returned to the procedure reference. Thus, the second Load Word instruction loads a mask for the B data field into register 5.

The Store Selective instruction stores the contents of register 4 into location Q under the mask in register 5.

Because Meta-Symbol allows one procedure to call upon another procedure, the MASK procedure could have been written to call upon the SHIFT procedure to position the mask it developed. The MASK procedure could have been written:

```
        .
        .
        .
MASK    FNAME
        PROC
        LOCAL   VAL,ARG
ARG     SET     AF
```

```
VAL     SET     (1**(ARG(2)- ARG(1)+1)-1)**SHIFT(ARG)
        PEND    L(VAL)
        :
        :
```

which would produce the same result.

Example 118. Recursive Function Procedure

As pointed out in the previous example, Meta-Symbol allows one procedure to call another. Meta-Symbol also allows recursion; that is, a procedure may call itself. This is illustrated in the following function procedure that produces the factorial of the argument.

```
        :
        :
FACT    FNAME
        PROC
        LOCAL   S,R
S       SET     AF
        DO      S(1)>1
R       SET     S * (FACT(S - 1))
        ELSE
R       SET     1
        FIN
        PEND    R
        :
        :
```

Because the explanation of a recursive procedure necessarily refers to procedure levels and the use of identical symbols on various levels, subscript notation is used to denote levels: S_1 refers to level 1 symbol S; S_2 to level 2 symbol S; etc.

The procedure reference in the main program could be

```
        :
        :
Q       SET     8
        :
        :
        LI,4    FACT(Q-5)
```

Within the procedure, S_1 and R_1 are declared to be local symbols. Next, S_1 is set to the value of the argument field at level 0; therefore, Q -5 is evaluated and S_1 is SET to 3. The DO directive determines whether the first element of list S_1 is greater than 1. Since S_1 consists of only one element and it is greater than 1, the statement following the DO directive is processed. The statement on line R_1 calls the FACT procedure. So, the process begins again.

The symbols S_2 and R_2 are declared to be local symbols. (This time, they are local to the level 2 procedure and will not be confused with the S and R that were local to the level 1 procedure.) S_2 is set to the value of the argument field, which is S_1 - 1 (3 - 1); that is, S_2 is set to the value 2. The DO statement determines whether the first element of list S_2 is greater than 1. Because S_2 consists of only one element and that element is greater than 1, the line following the DO directive is processed. The statement on line R_2 calls the FACT procedure again — this time at level 3.

The LOCAL directive declares S_3 and R_3 to be local symbols. Next, S_3 is set to the value of the argument field. This time the argument field is S_2 - 1, which is the value 1. The DO directive determines whether the first element of list S_3 is greater than 1. S_3 consists of only one element and it is not greater than 1, so control passes to the statement following the ELSE directive. R_3 is set to the value 1. The FIN directive terminates the DO-loop. The PEND directive terminates the procedure at level 3 and returns control to the procedure reference at level 2. Then, the processing of line R_2 is completed. The value 1, returned by the FACT procedure, is multiplied by $S_2(2)$ and equated to the label R_2. The ELSE directive terminates the DO-loop, and control passes to the statement following the FIN directive. The PEND directive terminates the procedure at level 2 and returns control to the procedure reference at level 1.

The value of $R_2(2)$ is returned to level 1, where it is multiplied by $S_1(3)$, and the product 6 is equated to the label R_1. The ELSE directive terminates the DO-loop, and control passes to the statement following the FIN directive. The PEND directive terminates the procedure at level 1 and returns control to the procedure reference in the main program.

Thus, the Load Immediate instruction loads the value 6 into register 4.

Example 119. Recursive Command Procedure

Recursion can also occur in command procedures. This SUM procedure produces the sum of the values of the elements of a list.

```
SUM     CNAME
        PROC
        LOCAL   R, I
R       SET     AF
LF      SET     0
I       DO      NUM(R)
        DO      NUM(R(I)) > 1
R(I)    SUM     R(I)
        FIN
LF      SET     R(I)+LF
        FIN
        PEND
```

Inner Loop

Outer Loop

Assume the procedure reference is

```
        :
Q       SET     5, (3, 4), (3, (7, 8), 4)
        :
Z       SUM     Q                          Procedure Reference (level 01)
        :
```

(As in Example 118, subscript notation is used to denote levels.) The resulting code is equivalent to

level 01

R_1	SET	5, (3, 4), (3, (7, 8), 4)	Equates local symbol R_1 to list.
Z_1	SET	0	
I_1	DO	NUM(R_1)	Does the loop 3 times; increments counter of outer DO-loop by 1; I_1 counter; $\therefore I_1$ 1.
	DO	NUM($R_1(1)$) > 1	False; $R_1(1)$ 5; $\therefore$ NUM($R_1(1)$) 1, so skip to FIN.
	FIN		Terminates inner loop.
Z_1	SET	$R_1(1) + Z_1$	$\therefore Z_1$ 5 + 0 5
	FIN		Increments counter of outer DO-loop by 1 and sets I_1 counter $\therefore I_1$ 2.
	DO	NUM($R_1(2)$) > 1	True; $R_1(2)$ 3, 4; $\therefore$ NUM($R_1(2)$) > 1.
$R_1(2)$	SUM	$R_1(2)$	Procedure Reference (level 02)

level 02

R_2	SET	3, 4	Equates local symbol R_2 to sublist.
$R_1(2)$	SET	0	
I_2	DO	NUM(R_2)	Does this loop 2 times; increments counter of outer DO-loop by 1; I_2 counter; $\therefore I_2$ 1.
	DO	NUM($R_2(1)$) > 1	False; $R_2(1)$ 3; $\therefore$ NUM($R_2(1)$) 1, so skips to FIN.
	FIN		Terminates inner loop.

$R_1(2)$	SET	$R_2(1) + R_1(2)$	$\therefore R_1(2)$ 3 + 0 3
	FIN		Increments counter of outer DO-loop by 1 and sets I_2 counter; $\therefore I_2$ 2.
R_2	SET	3, 4	Equates local symbol R_2 to sublist.
$R_1(2)$	SET	0	
I_2	DO	$NUM(R_2)$	Does this loop 2 times; increments counter of outer DO-loop by 1; I_2 counter; $\therefore I_2$ 1.
	DO	$NUM(R_2(1)) > 1$	False; $R_2(1)$ 3; $\therefore NUM(R_2(1))$ 1, so skips to FIN.
	FIN		Terminates inner loop.
$R_1(2)$	SET	$R_2(1) + R_1(2)$	$\therefore R_1(2)$ 3 + 0 3
	FIN		Increments counter of outer DO-loop by 1 and sets I_2 counter; $\therefore I_2$ 2.
	DO	$NUM(R_2(2)) > 1$	False; $R_2(2)$ 4; $\therefore NUM(R_2(2))$ 1, so skips to FIN.
	FIN		Terminates inner loop.
$R_1(2)$	SET	$R_2(2) + R_1(2)$	$\therefore R_1(2)$ 4 + 3 7
	FIN		Terminates outer DO-loop.
	PEND		Terminates level 02 procedure and returns to level 01.
level 01			
	FIN		Terminates inner loop.
Z_1	SET	$R_1(2) + Z_1$	$\therefore Z_1$ 7 + 5 12
	FIN		Increments counter of outer DO-loop by 1 and sets I_3 counter; $\therefore I_3$ 3.
	DO	$NUM(R_1(3)) > 1$	True; $R_1(3)$ 3, (7, 8), 4; $\therefore NUM(R_1(3))$ 3.
$R_1(3)$	SUM	$R_1(3)$	Procedure Reference (level 02)
level 02			
R_2	SET	3, (7, 8), 4	Equates local symbol R_2 to list. Note that R2 is a new symbol; it is not to be confused with the previous level 2 symbol R.
$R_1(3)$	SET	0	
I_2	DO	$NUM(R_2)$	Does this loop 3 times; increments DO-loop counter by 1; I_2 counter; $\therefore I_2$ 1.
	DO	NUM(R2(1)) > 1	False; $R_2(1)$ 3; $\therefore NUM(R_2(1))$, 1, so skips to FIN.
	FIN		Terminates inner DO-loop.
$R_1(3)$	SET	$R_2(1) + R_1(3)$	$\therefore R_1(3)$ 3 + 0 3
	FIN		Increments counter of outer DO-loop by 1 and sets I_2 counter; $\therefore I_2$ 2.

	DO	NUM($R_2(2)$) > 1	True; $R_2(2)$ 7,8; ∴ NUM($R_2(2)$) > 1.
$R_2(2)$	SUM	$R_2(2)$	Procedure Reference (level 03).
level 03			
R_3	SET	7,8	Equates local symbol R_3 to list.
$R_2(2)$	SET	0	
I_3	DO	NUM(R_3)	Does this loop 2 times; increments DO-loop counter by 1; I_3 counter; I_3 1.
	DO	NUM($R_3(1)$) > 1	False; $R_3(1)$ 7; ∴ NUM($R_3(1)$) 1, so skips to FIN.
	FIN		Terminates inner loop.
$R_2(2)$	SET	$R_3(1) + R_2(2)$	∴ $R_2(2)$ 7 + 0 7
	FIN		Increments counter of outer DO-loop by 1 and sets I_3 counter; ∴ I_3 2.
	DO	NUM($R_3(2)$) > 1	False; $R_3(2)$ 8; ∴ NUM($R_3(2)$) 1, so skips to FIN.
	FIN		Terminates inner DO-loop.
$R_2(2)$	SET	$R_3(2) + R_2(2)$	∴ $R_2(2)$ 8 + 7 15
	FIN		Terminates outer DO-loop.
	PEND		Terminates level 03 procedure and returns to level 02.
level 02			
	FIN		Terminates inner DO-loop.
$R_1(3)$	SET	$R_2(2) + R_1(3)$	∴ $R_1(3)$ 15 + 3 18
	FIN		Increments counter of outer DO-loop by 1 and sets I_2 counter; ∴ I_2 3.
	DO	NUM($R_2(3)$) > 1	False; $R_2(3)$ 4; ∴ NUM($R_2(3)$) 1, so skips to FIN.
	FIN		Terminates inner DO-loop.
$R_1(3)$	SET	$R_2(3) + R_1(3)$	∴ $R_1(3)$ 4 + 18 22
	FIN		Terminates outer DO-loop.
	PEND		Terminates level 02 procedure and returns to level 01.
level 01			
	FIN		Terminates inner DO-loop.
Z_1	SET	$R_1(3) + Z_1$	∴ Z_1 22 + 12 34
	FIN		Terminates outer DO-loop.
	PEND		Terminates level 01 procedure and returns to main program at level 0.

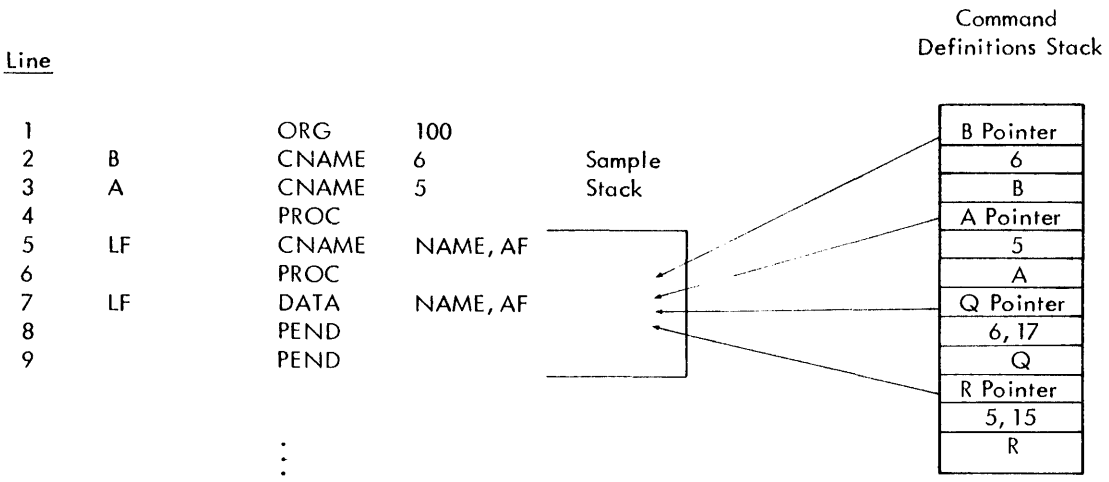
Thus, the main program statement

Z	SUM	Q

results in the value 34 being assigned to label Z.

Example 120. Procedure that Defines a Procedure

The following procedure is assigned two names (B and A) and defines a procedure when referenced.



When Meta-Symbol encounters the A/B procedure definition, it creates two stacks: the Sample Stack (lines 5-9) and the Command Definitions Stack. In the latter stack, it enters B, assigning it the value 6, and A, assigning it the value 5. These entries also contain a pointer to the Sample Stack, indicating the lines that will be processed when B or A is subsequently referenced.

The procedure reference lines

```
20    Q    B    17
21    R    A    15
22    J    Q    77
```

cause Meta-Symbol to do the following:

1. Line 20 causes Meta-Symbol to process lines 5 and 6 as

```
Q    CNAME    6, 17
     PROC
```

   which is a procedure definition. Therefore, Q is entered in the Command Definitions Stack, assigned the values 6, 17, and associated with line 7. Line 8 terminates the definition of Q, and line 9 returns control to the main program.

2. Line 21 causes Meta-Symbol to process lines 5 and 6 as

```
R    CNAME    5, 15
     PROC
```

   which is a procedure definition. Therefore, R is entered in the Command Definitions Stack, assigned the values 5, 15, and associated with line 7. Line 8 terminates the definition of R, and line 9 returns control to the main program.

3. Line 22 references Procedure Q which processes line 7 as

```
J    DATA    6, 17, 77
```

   Label J is assigned the value 100 (ORG 100).

6. ASSEMBLY LISTING

The Xerox Meta-Symbol assembler produces listing lines according to the format shown in Figure 5. The page count, a decimal number, appears in the upper right-hand corner of each page.

EQUATE SYMBOLS LINE

Each source line that contains an equate symbols or display directive (EQU, SET, or DISP) contains the following information:

NNNNN* Source image line number in decimal. An asterisk identifies an update line.

and

XXXXXXXX Value of argument field as a 32-bit value.

or

CC Current section number in hexadecimal. The first control section of an assembly is arbitrarily assigned the value 1, and subsequent sections are numbered sequentially.

LLLLL Value of the argument field as a hexadecimal word address.

B Blank, 1, 2, or 3 specifying the current byte displacement from a word boundary.

or

TTTT A one- to four-character value type indicator when the value of the item in the argument field is other than an address or a single precision integer. This is discussed below.

and

SSS . . . Source image. If source input was from a keyed file, columns 73-80 (print positions 100-117) will be replaced by the key number.

When the argument field of an EQU, SET, or DISP directive specifies a value that is neither a single precision integer nor an address that is evaluatable when the directive is encountered, the assembler will print a one- to four-character value type indicator in the value field of the listing (print positions 19-26). If the argument field of these directives specifies more than one value, or if one of the arguments is itself the name of a list, the values or value type indicators will be printed singly beginning with the value field of the directive line and continuing for successive lines. The information listed in the value field for various kinds of EQU, SET, and DISP arguments is shown in the following listing:

SET, EQU, DISP Argument Type	Display in Listing Value Field
Single precision integer	value of integer
Address	value of address
Fixed decimal constant	FX
Floating short constant	FS

Print Position	1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39
Equate symbols line	NNNNN C C L L L L L B S... NNNNN X X X X X X X X S... NNNNN T T T T S...
Assembly listing line	NNNNN C C L L L L L B X X X X X X X X A S...
Ignored source image line	NNNNN * S *
Error line	* * * * m e s s a g e S...
Literal listing line	C C L L L L L B X X X X X X X X A

Figure 5. Meta-Symbol Listing Format

SET, EQU, DISP Argument Type	Display in Listing Value Field
Floating long constant	FL
Decimal constant	D
Character string constant	TEXT (For EQU and SET directives) first 8 bytes of constant in EBCDIC (for DISP directive)
Undefined global symbol	UND
Forward reference	FR
Local forward reference	LFR
External reference	EXT
Double precision integer	DPI
! expression involving a relocatable item	value of integer addend ⌴ S.
List, i.e., $value_1, \ldots, value_n$	LIST followed by: $value_1$ ⋮ $value_n$ **** } only for DISP directive

Note: Any of the list items might itself be a list. In that case LIST and **** will print to define the elements of such a sublist.

ASSEMBLY LISTING LINE

Each source image line containing a generative statement prints the following information:

NNNNN* Source image line number in decimal. An asterisk identifies an update line.

CC Current section number in hexadecimal. See CC under "Equate Symbols Line".

LLLLL Current value of execution location counter to word level in hexadecimal.

B Blank, 1, 2, or 3, specifying the byte displacement from word boundary.

XX, XXXX, XXXXXX, XXXXXXXX Object code in hexadecimal listed in groups of one to four bytes.

A Address classification flag:

- blank — denotes a relocatable field.
- A — denotes an absolute address field.
- F — denotes an address field containing a forward reference.
- X — denotes an address field containing an external reference.
- N — indicates that the object code produced for the source line contains a relocatable item (i.e., an address, a forward reference, or external reference) in some field other than the address field.
- NN — specifies intersection reference number.

SSS... Source image.

IGNORED SOURCE IMAGE LINE

A skip flag indication

S

is printed in columns 33-35 for each statement skipped by the assembler during a search for a GOTO label or while processing a DO or DO1 directive with an expression value of zero. It will also be printed for a system directive that specifies a file included in the standard definition file. NNNNN and SSS... have the same meanings as in an assembly listing line.

The *S* flag is also printed in columns 33-35 beside any CNAME directive containing a procedure name that was not subsequently referred to in a command procedure reference line. If none of the names for a procedure are referred to, the entire procedure will be skipped and so indicated on the assembly listing.

ERROR LINE

When an error is detected in the source image line, the line immediately following begins with the error indication

****

This line may contain one or more single-character syntax error codes or an error message. The error codes print beneath the part of the source image line that is erroneous. The error codes and their meaning are listed in Table 6; the error messages are given in Chapter 7, "Meta-Symbol Operations".

LITERAL LINE

Any literals evaluated during an assembly are printed immediately following the END statement. Literals are listed in the order in which they were evaluated, and the listing line contains

CC Current section number in hexadecimal. See CC under "Equate Symbols Line".

LLLLL Current value of execution location counter to word level in hexadecimal.

Table 6. Meta-Symbol Syntax Error Codes

Code	Severity[†]	Significance
A	4	Arithmetic operand error. Arithmetic has been attempted with an operand on which arithmetic is not syntactically meaningful, probably a list. For example: (B, C) * 2 A
B	4	Column 1 of a continuation card does not contain a blank. For example: MASK DATA ; X'3FFF' B
C	4	Constant string error. A constant contains an illegal character or is improperly formed. For example: X'ABCDEFG' C
E	4	Expression error. An arithmetic expression is malformed (a missing operand, an unknown operator, etc.). For example: 5 + / 2 E
F	4	Invalid system file name. The argument field of a SYSTEM directive contains other than a legal Meta-Symbol name. System SIG7FDP is substituted. For example: SYSTEM ALPHA + 2 F
L	4	Label error. The argument field of one of the directives CDISP, FDISP, DEF, REF, SREF, LOCAL, OPEN, or CLOSE contains other than a well-formed Meta-Symbol symbol. For example: OPEN 50 L
M	4	Missing command field.
N	4	Parentheses nesting error. For example: A * (B / (C + 1) N
O	4	Arithmetic overflow during constant conversion. For example: FX'1.5B2E3' O

[†]The highest error severity level encountered in the assembly is passed to the loader and appears at the end of the assembly listing.

Table 6. Meta-Symbol Syntax Error Codes (cont.)

Code	Severity[†]	Significance
Q	4	An apparent constant qualifier other than C, D, O, X, FS, FL, or FX has been encountered. For example: G'FFAB' Q
S	4	A general violation of syntax structure has been encountered, usually a juxtaposition of characters that provides no indication of intended meaning. For example: BA(A)B S
V	4	A character not in the recognized character set has been encountered outside a constant string. For example: ALPHA.R2 V
X	4	The previous list has more than 255 elements.

[†]The highest error severity level encountered in the assembly is passed to the loader and appears at the end of the assembly listing.

B	Blank, 1, 2, or 3, specifying the byte displacement from word boundary.
XXXXXXXX	Value of literal as a hexadecimal memory word.
A	Address classification flag. See "Assembly Listing Line".

SUMMARY TABLES

Immediately following the literal table, the following eight summaries are printed as a standard part of the assembly listing. Each summary is preceded by an identifying heading.

1. Control Section Summary. Shows, in hexadecimal, the section number, size, and protection type of all control sections in the program. A typical item has the form

   01 005B4 2 PT 1

   where 01 is the control section number, and 005B4 is the number of words in the section, plus two additional bytes. PT denotes "protection type" and 1 means that protection type 1 is assigned to this section. Protection type, an integer from 0 to 3, is specified by a CSECT or DSECT, or PSECT directive (see Chapter 2). The control section summary is listed four items per line.

   If an asterisk follows the control section number, as in

   03*001D3 PT 0

   it indicates that control section 3 was declared to be of size X'1D3' words in the object language (if generated), but that the listing indicates a different maximum size for control section 3. This may indicate that a different amount of data was generated by pass 1 of the assembler than was generated by pass 2. This condition causes an error severity level of 3.

   A page eject follows the control section summary and the following summaries then print. Items 2 through 5 below may be omitted by including the option NS (no summaries) on the METASYM control card. Items 6 through 8, the error summaries, always print, however.

2. Symbol Value Summary. Shows all nonexternal symbols in the program, except those designated as LOCAL or closed. A typical item has the form

   SCALE/01 001B5

   where SCALE is a symbol name, 01 is its control section and 001B5 is the hexadecimal word address at which it

is defined. In place of a control section and word address, some symbols will have a 32-bit value displayed as an eight-digit hexadecimal number or may have a one- to four-character value type indicator. In other words, the information following a symbol name may have the same format as described previously under "Equate Symbols Line". On some items, the slash is replaced by an asterisk if the SD option has been included in the METASYM control card. The SD option specifies that symbolic debugging code (i.e., a symbol table) is to be included in the relocatable object module.

The symbol values are printed four per line except where an entry is too long for its allotted print field and overflows into the field to its right.

3. External Definition Summary. Shows all symbols in the program declared to be external definitions. Format is the same as the Symbol Value Summary.

4. Primary External Reference Summary. Shows all symbols declared to be external references. Only symbol names are listed, formatted seven per line where possible.

5. Secondary External Reference Summary. Shows all symbols declared to be secondary external references. Format is the same as for the Primary External Reference Summary.

6. Undefined Symbol Summary. Shows all symbols used but not defined nor declared to be external references. Format is the same as for the Primary External Reference Summary.

7. Error Severity Level. This line shows the highest error severity level encountered in the program.

8. Error Line Summary. Shows the line numbers of all lines in the source program on which errors were encountered.

7. OPERATIONS

Meta-Symbol has been designed to run under control of the Sigma Batch Processing Monitor and Batch Time-Sharing Monitor, thereby making available to it all facilities of the Monitor system. This document provides a discussion of the Monitor interface that affects the typical user. In particular, Monitor control commands necessary to assemble a program are described. Consult the reference manuals for the above mentioned Monitors for further information about their operation and use.

BATCH MONITOR CONTROL COMMANDS

To assemble a Meta-Symbol program, a run deck containing the necessary Monitor commands must first be prepared. This chapter describes those commands. A large variety of other Monitor commands exist; for these, the user is referred to the appropriate Monitor Reference Manual.

JOB CONTROL COMMAND

The first card in each Meta-Symbol run deck must be a JOB card, which has the format shown below.

```
!JOB  account number, name, priority
```

where

account number is a 1- to 8-character alphanumeric string identifying the account or project to which the run is to be charged.

name is a 1- to 12-character alphanumeric string identifying the user.

priority is a number between X'1' and X'F' specifying the priority of the job. Although the priority is not used by the unscheduled Monitor, it must be specified; otherwise, the Batch Monitor will reject the entire job.

LIMIT CONTROL COMMAND

Immediately following the JOB card there should be a LIMIT card, which has the format shown below. The order of the three limit options is immaterial.

```
!LIMIT  (LO,limit),(PO,limit),(TIME,limit)
```

where limit is a decimal integer specifying maximum operational limits, as follows:

LO, limit specifies the maximum number of pages that can be listed.

PO, limit specifies the maximum number of cards that can be punched.

TIME, limit specifies, in minutes, the maximum time that the job can take.

These three limits are applied, respectively, to the sums of LO output, PO output, and time used across the entire job. Thus, these limits should reflect the maximum expected usage for all assemblies within the job, not just the first assembly. If the LIMIT card is omitted, the installation default limits are used.

ASSIGN CONTROL COMMAND

Appearing next in the run deck are any ASSIGN cards relating to the assembly. Normally, ASSIGN cards will not be needed, since the system has the following standard default assignments.

Logical Device or File	Physical Device
BO	Card punch
CI	Card reader
CO	Card punch
DO	Line printer
GO	Magnetic disk
LO	Line printer
SI	Card reader
SO	Card punch
X1 (Intermediate file)	Magnetic disk
X2 (Update file)	Magnetic disk
X2KF (Update file)	Magnetic disk
X3 (Concordance file)	Magnetic disk

If the user desires to reassign any of these I/O options, an appropriate ASSIGN card is necessary. For most users, the only change from the above standard assignments will be the assignment to magnetic tape of CI and/or CO or the intermediate files.

Meta-Symbol does not protect against conflicts that may arise when two or more output options are assigned to the same physical device. Such conflicts (such as SO and LS to a line printer, or SO and CO to a card punch) may cause the output to be interspersed in an irregular manner. Such conflicts should be resolved by reassigning one or more of the output options to a different device, or by calling the assembler more than once, specifying only one such option per assembly.

METASYM CONTROL COMMAND

The next card in the run deck will be the METASYM card, which has the following format:

```
!METASYM  option1,option 2,...,option n
```

where any number of options, or none, may be specified. The options and their meanings are given below.

AC(ac_1,ac_2,...,ac_n)	Account number specification, where each ac is a Batch Monitor account number.
BA	Batch assembly mode.
BO	Binary output.
CI	Compressed input.
CN	Concordance output.
CO	Compressed output.
DC	Default concordance.
GO	Output GO file.
LD	List standard definition file.
LO	List assembly output.
LS	List source.
LU	List updates.
ND	No standard definition file.
NS	No summaries.
PD (sn_1,sn_2,...,sn_n)	Produce standard definition file.
SD	Symbolic debugging output.
SB or SB(...)	Sequence binary.
SC or SC(...)	Sequence compressed.
SI	Source input.
SO	Source output.
SU	Sequential update.

Options may be specified in any order. Except for AC, repetitions of the same option are ignored, that is, the effect is that of a single occurrence. If no options are specified, the following options are assumed:

SI, LO, GO

The METASYM card is free form; blanks may appear anywhere except between the two letters of an option name or between the option name and the left parenthesis for AC, SB, and SC[†]. At least one blank must separate the METASYM command from the first option. The option list may be continued on one or more cards following the METASYM card. Continuation is specified by placing a semicolon at any point where a blank is legal. Processing of the METASYM card is then resumed at the first nonblank column. METASYM continuation cards must not have an ! in column one.

If the program is on cards, it must immediately follow the METASYM card. However, if CN has been specified, the METASYM card must be followed immediately by Concordance Control Command cards, the last one of which must be a .END card (see "Concordance Control Commands and Listing" later in this Chapter). The Meta-Symbol program deck is considered terminated by the first card containing an END directive in the command field. Any cards after the END directive are ignored by the Meta-Symbol assembler.

A sample METASYM card is shown below.

```
!METASYM  SI, LO, CI, BO, SB(BIN)
```

The meanings of the various options are as follows:

AC(ac_1,ac_2,...,ac_n) where $n \leq 9$. This option is used in conjunction with the SYSTEM directive of Meta-Symbol. With this directive, the user has the capability of calling system files that have been placed on disk. Normally the only system files used are those provided by the assembler (namely, SIG7FDP,...,SIG5). However, when a user wants to provide his own system files, a minor problem arises in that, typically, he will have access to only a limited number of account numbers[†] provided by the Monitor with which to identify these files when they are entered onto the disk. In order for Meta-Symbol to access these files, the assembler must be told what their account numbers are. The AC option provides this information.

If the AC option is specified and Meta-Symbol later encounters a SYSTEM directive, it will ask the Monitor to search for the system name in the Monitor's account number and name table, under the account numbers given in the AC option. The search will be performed according to the order of the numbers in the AC option, from left to right, until the specified system is found or the account numbers are exhausted. If the system is not found under the user-specified account numbers, the systems filed under the "system account number"[††] are then searched. If the AC option is not specified, the system specified by the SYSTEM directive is searched for only under the "system account number". Since all standard Meta-Symbol systems are filed under the "system account number", they will be found correctly even when the AC option is not used. If more than one AC option is specified, the search is performed from left to right across the card.

Thus,

```
!METASYM  AC(1),...,AC(2,3),...,AC(4),...
```

is equivalent to

```
!METASYM  AC(1,2,3,4),...
```

and both will cause a system search to be performed, first under account number 1, then 2, 3, 4, and, finally, under the "system account number".

[†]Note that blanks may be significant characters for the SB and SC options.

[†]See "JOB Control Command" for the definition of account numbers.

[††]The "system account number" is the account number (:SYS) under which all standard Xerox Sigma software is filed.

A system is identified by the name under which it is entered on the disk. This name must correspond to the name specified on the SYSTEM directive line used to reference the system. Further, a system name must constitute a legal "symbol" according to the Meta-Symbol syntax rules, whereas the Monitor's rule for naming files is somewhat broader (see "Creating System Files").

BA Selects the batch assembly mode. In this mode, successive assemblies may be performed with a single METASYM card. The assembler will read and assemble successive programs until a double end-of-file is read. In the batch mode current device assignments and options on the METASYM card are applied to all assemblies within the batch.

A program is considered terminated when an END directive is processed. Successive programs may or may not have an end-of-file indicator separating them.

With input from the card reader, an end-of-file is indicated by an EOD card. Two successive EOD cards or any other Monitor control card terminates the job.

With input from unlabeled magnetic tape, standard tape end-of-files provide job termination.

With input from labeled files on disk or tape, the job will terminate when all programs in the file have been assembled.

When batch assemblies consist of successive updates from the card reader, to compressed programs from disk or tape, the update packets are considered terminated by a +END card, and should not be separated by EOD cards. There must be a one-to-one correspondence of update packets to compressed programs. End of job is signaled by end-of-file conventions applied to the CI device.

Output may be to any device, or labeled files on disk or tape. Output on a device will be separated by ends-of-file, and terminated by a double end-of-file. All output to labeled files specified on the original ASSIGN card will be written as successive records in the appropriate output file, without module or program end-of-file separators.

BO This option specifies that binary output is to be produced on the BO device.

CI This option specifies that compressed input is to be taken from the CI device.

CN This option specifies that a concordance, or symbolic name cross-reference listing, is to be produced on the LO device. One or more Concordance Control Commands will follow the METASYM card on the C device. These commands specify the range of names to be included in the concordance (see "Concordance Control Commands and Listing" later in this chapter). The concordance listing is produced at the end of the assembler's encoding phase, and does not require a full assembly. It may be produced in conjunction with an LS listing.

CO This option specifies that compressed output is to be produced on the CO device.

DC This option specifies that a "standard" concordance is to be produced on the LO device. The DC option differs from the CN option in that no attempt is made to read the C device for concordance control commands. If both DC and CN are specified, the DC option takes priority, and the CN option is ignored.

GO This option specifies that the binary object program is to be placed in a temporary file from which it can later be loaded and executed. The resultant GO file is always temporary and cannot be retained from one job to another. To retain the binary object program for a subsequent job, the BO option (with BO assigned to disk or magnetic tape) must be used.

LD This option specifies that a listing of the standard definition file (if any) is to immediately precede the normal program listing. LD will have no effect if LO is not also specified.

LO This option specifies that a listing of the assembled object program is to be produced on the LO device.

LS This option specifies that a listing of the source program is to be produced on the LO device. This listing consists of an image of columns 1 to 72 of each input line (after updates have been incorporated) with its line number.

LU This option specifies that a listing of the update decks (if any) is to be produced on the LO device. This listing consists of an image of each update line with the number of the lines in the update deck.

ND This option specifies that no standard definition file is to be input for this assembly. Note that PD implies the ND option, so that ND is redundant if PD is also specified.

NS This option specifies that summaries following the assembly listing are to be omitted for symbol values, external definitions, and primary and secondary external references.

PD (sn_1,...,sn_n) This option specifies that a standard definition file is to be produced. The file will be written through the F:STD DCB, which contains a built-in file name of $:STDMET. Thus, if F:STD is not reassigned, the PD option will cause creation (or overwriting) of a file, $:STDMET, in the current job account. F:STD may be assigned to a different file name in the current account, and a standard definition file of that name will then be created.

The optional sn_i are names which, if used as arguments of a SYSTEM directive in a program that subsequently uses this file, will cause that SYSTEM directive to be ignored. This allows programs to reference SYSTEM directives as usual, yet take advantage of a standard definition file that includes the designated system.

SB,SC These options specify, respectively, that binary or compressed card images are to be sequence numbered in columns 77 to 80. The form SB(...) or SC(...) also may be used, where the ellipsis represents a string of alphanumeric characters.[†] With this form the leftmost four characters of

[†] All legal EBCDIC characters are permitted except for commas, semicolons, and left or right parentheses. Blanks are treated as significant characters.

the string are punched as identification in columns 73 to 76. If fewer than four characters are specified, they are left-justified and blank-filled in the remaining columns. If SB or SC is specified without a corresponding output option (BO or CO, respectively), the SB or SC option has no effect.

SD This option specifies that symbolic debugging code (i.e., a symbol table) is to be included in the relocatable object module produced by the assembler. Inclusion of this symbol table allows a debug subsystem to associate symbolic names and type information with specified memory cells. This allows run-time debugging and modification of a program in a symbolic format similar to the actual assembly listing.

If the SD option appears on the METASYM card, BO or GO must be given also.

When a symbol value summary is produced at the end of the assembly listing, any symbols entered into the object code will be identified in the summary by an asterisk (*) instead of a slash (/) preceding their value, word address, or type indicator.

SI This option specifies that symbolic input is to be taken from the SI device.

SO An EBCDIC card image representation of the input program is to be produced. The symbolic records will be written on the SO device. The full range of assignments may be made when obtaining source output.

Creation of source output does not require a complete assembly, but rather is done during the encoding phase.

SU This option specifies that the update control commands (see "Updating a Compressed Deck") within any update deck must be given in sequential order. Normally, the order of these commands is immaterial, the assembler ordering them as required; but if SU is specified, only update control commands in sequential order are permitted. This option is provided to accommodate those systems whose disk storage is too small to hold the intermediate file necessary to perform updating. In such a case, this file could be placed on magnetic tape.

EOD CONTROL COMMAND

In the batch mode (that is, when the BA option is specified), programs on cards may optionally be separated by EOD cards, which have the format !EOD.

Each EOD card will normally be placed immediately after the END card in a program deck. Any cards between the first END card of the program and the EOD card are ignored.

FIN CONTROL COMMAND

Another Monitor control card should allow the last program deck of the assembly. This may be a FIN card, which has the format !FIN.

Since the FIN command returns the Monitor to the idle state, however, the program deck will be followed in most cases by a LOAD card, METASYM card, or JOB card.

UPDATING A COMPRESSED DECK

By the use of the CO option on the METASYM card, Meta-Symbol may be directed to produce a compressed deck of a source program which can then be used as input during a later assembly. Since a typical compressed deck contains one-fourth to one-fifth as many cards as the corresponding source deck, the use of compressed decks offers significant operating advantages in both manageability and speed. The following discussion explains how to update a compressed deck with an "update packet". An update packet is considered to be the set of cards between the first ! (update) command and the compressed deck. If any symbolic cards precede the first ! command they are termed a "symbolic deck"; however, they are treated as if they were preceded by a !0 card (see !k below); that is, they are inserted before the first line of the program.

Meta-Symbol recognizes four update control commands:

!k where k is a line number corresponding to a line number on the source or assembly listing produced from the compressed deck. The !k control card designates that all cards following the !k card, up to but not including the next update control card, are to be inserted after the kth line of the source program. The command !0 designates an insertion before the first line of the program.

!j,k where j and k are line numbers corresponding to line numbers on the source or assembly listing produced from the compressed deck, and $j \leq k$. This form designates that all cards following the !j,k card, up to but not including the next update control card, are to replace lines j through k of the source program. The number of lines to be inserted does not have to equal the number of lines removed; in fact, the number of lines to be inserted may be zero. In this case, lines j through k are deleted.

!* identifies a comment card that will be displayed within the update listing (LU specified), but will have no other effect upon the update process. Comments may begin in column 3 of the !* command. Comment cards are not inserted into the program being updated.

!END designates the physical end of an update packet. If the SI and CI devices are the same, this command is optional, since, if omitted, Meta-Symbol will terminate the update packet automatically on encountering the first compressed card. If the SI and CI devices are different, this command is required.

The + character of each update control command must be in column 1, followed immediately by the control information, with no embedded blanks. The control command is terminated by the first blank column encountered. Optionally, the blank may be followed by comments. Unless the SU option has been specified, the update control commands, with their associated update records, may occur in any order; Meta-Symbol will order them as required.

The ranges of successive insert and/or delete control commands must not overlap, except that the following case is permissible: +j,k followed by +k, where j<k.

Overlapping or otherwise erroneous control commands will cause an abort error.

PROGRAM DECK STRUCTURES

Meta-Symbol accepts two basic types of input decks: symbolic decks and compressed decks preceded by optional update packets. Meta-Symbol will accept any number of alternating symbolic and compressed (with update) decks until a deck is found that contains an END directive; any cards remaining after the END directive are ignored up to the next Meta-Symbol control card. These decks, up to and including the END directive, are combined into one program. Five basic deck structures are possible, as shown in Figure 6.

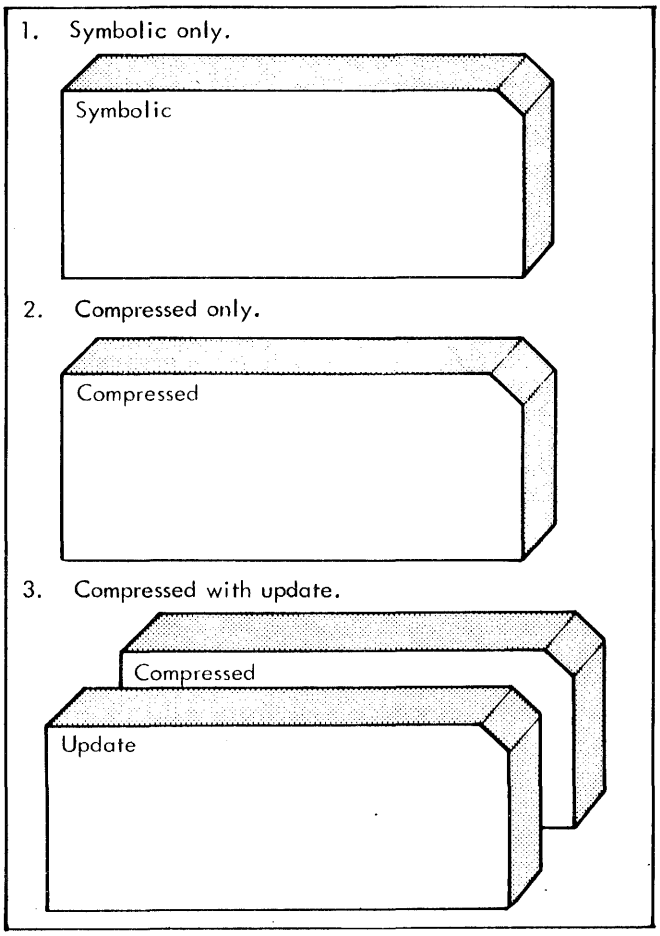


Figure 6. Basic Symbolic and Compressed Deck Structures

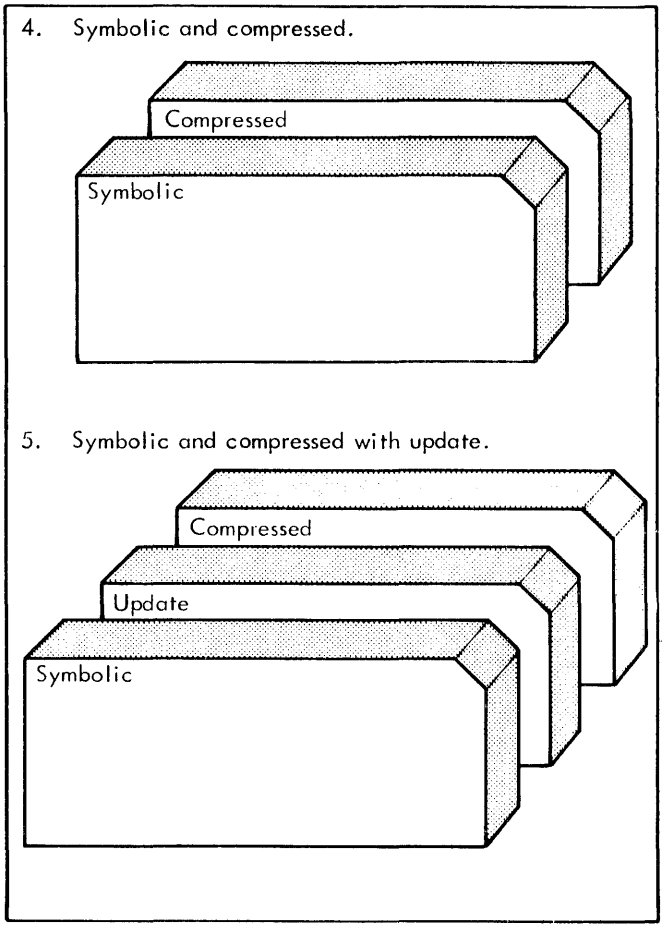


Figure 6. Basic Symbolic and Compressed Deck Structures (cont.)

Any of these five kinds of deck structures may be combined with any other kind, as required. Various legal deck structures are shown below in Figure 7.

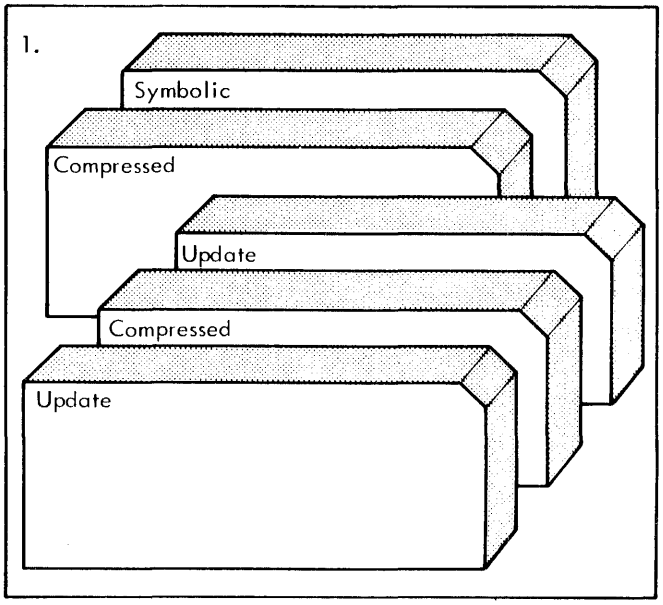


Figure 7. Sample Legal Deck Structures

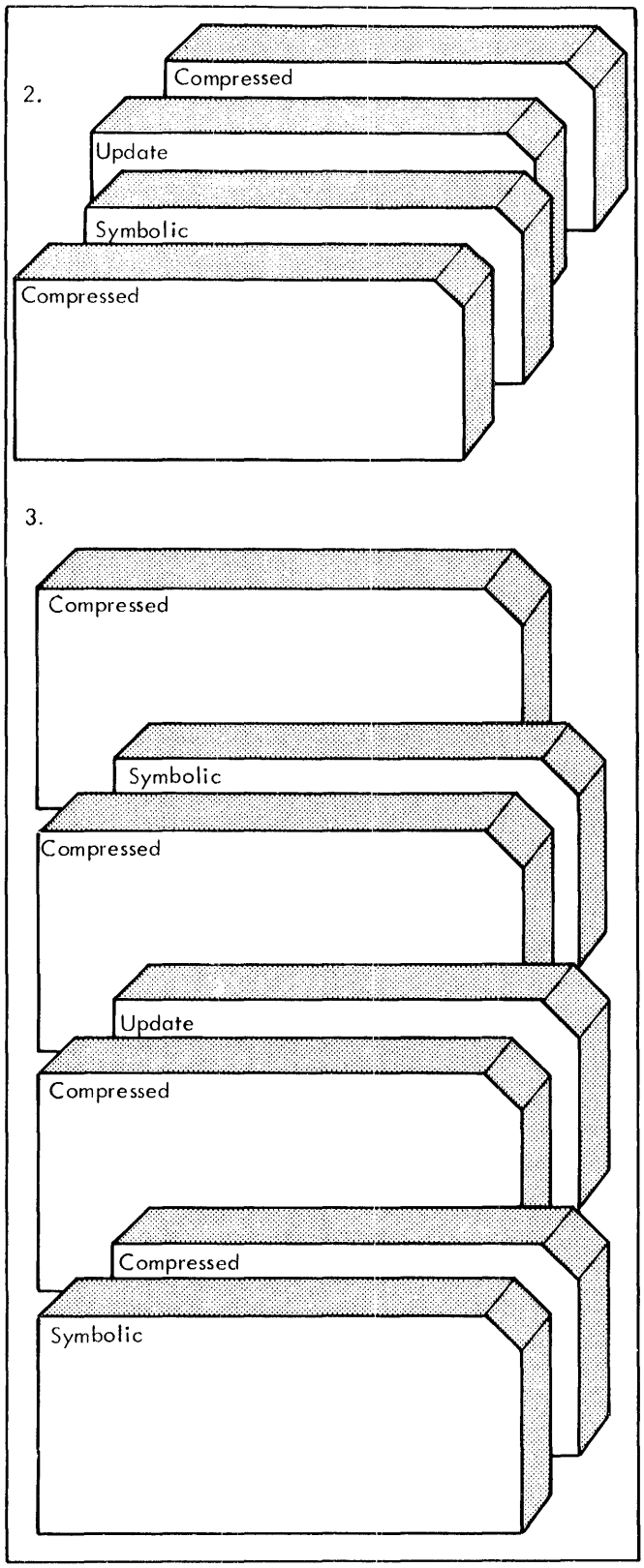


Figure 7. Sample Legal Deck Structures (cont.)

If the SI and CI devices are different and it is desired to read compressed input from the CI device, then the only permissible structure is that shown in Figure 8.

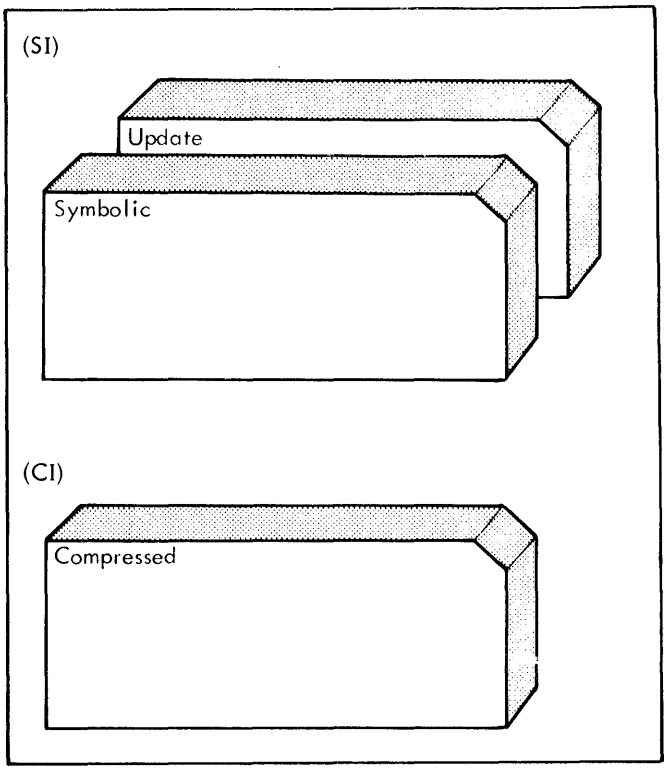


Figure 8. Deck Structure for SI and CI on Different Devices

The symbolic deck on the SI device may be omitted. If the SI device contains a compressed deck preceded by an update packet that does not terminate with a +END card, that compressed deck will be updated and assembled, and the CI device will not be read.

If an update packet and its associated compressed deck are to come from different devices, both SI and CI must be specified on the METASYM card. If both are on the same device (or if only symbolic input or only compressed input is to be processed), the assembler can distinguish among the three types of decks from the deck structure and card format, and is therefore not dependent on the options specified. The action of the assembler for each combination of options is as follows:

SI (or CI) only. The assembler will read from the SI (CI) device and process whatever structure it finds. In either case, this may consist of any legal combination of symbolic, update, and compressed decks. The only difference between the SI and CI option in this situation is that SI causes input to be read from the SI device and CI from the CI device.

SI and CI. The assembler will read the SI device first; it must contain some information. If the input deck on the SI device consists of an update deck terminating with a +END control command, the CI device is then read. If there is no +END card on the update deck, and this is followed by a compressed deck, it is assumed to be the compressed input specified by the CI option and no attempt is made to read the CI device.

(The section headed "Maintaining Compressed Files on Magnetic Tape" has been deleted.)

CREATING SYSTEM FILES

To place a system deck on disk, the Monitor control command sequence shown in Figure 9 could be used.

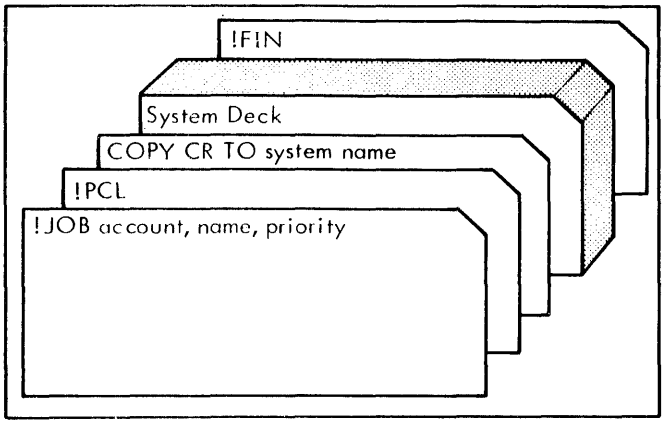


Figure 9. Example of System File Creation

This sequence will cause the deck following the PCL COPY card to be entered onto the disk with the "system name" given on the COPY card and under the account number specified on the JOB card. The system deck may consist of either a single symbolic or a single compressed Meta-Symbol program. If the account number is the "system account number" (:SYS), this system deck can be referenced by other programs without using the AC option on a METASYM card. Otherwise, any assembly that needs to reference this deck must use the appropriate AC option. That is the case in the typical usage shown in Figure 10.

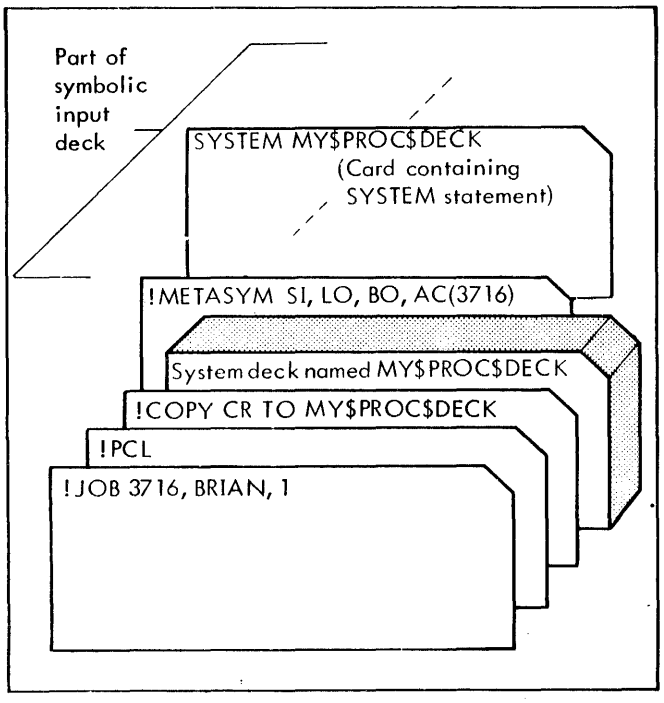


Figure 10. Use of the AC Option

Note that use of the AC option was necessary even though the assembly was done under the same account number as that under which the system deck MY$PROC$DECK was entered. This number was not the system account number.

CREATING AND USING A STANDARD DEFINITION FILE

Although system files provide the most general and flexible means for including common source libraries into the program, they can cause a processing overhead that is unacceptable for short programs or on-line terminal use. A standard definition file is similar in function to a system file, but exists in a Meta-Symbol internal format that is directly usable by the assembler, thus avoiding the processing time required for source or compressed system files. Unlike system files, a standard definition file is not invoked by name but is automatically read in prior to starting an assembly. Only one standard definition file is available to a single program, but SYSTEM directives may still be used to include other required source library files in the program.

Since standard definition files are installation-specific in content, a program that may be assembled at different installations should still use a SYSTEM directive to identify each required system file. The process of producing a standard definition file allows specification of system file names that are included in that file; Meta-Symbol will then ignore any SYSTEM directives whose file name is one of those included in the standard definition file. Figure 11 illustrates a typical standard definition file creation.

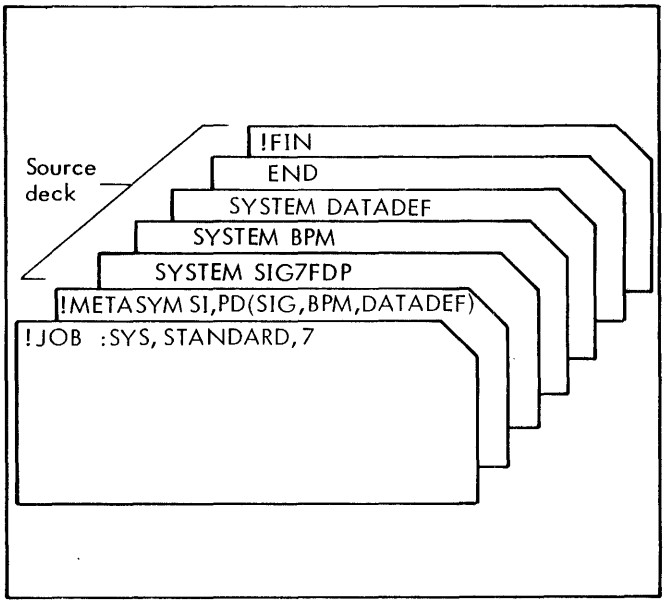


Figure 11. Creation of a Standard Definition File

Note that the Sigma instruction procedures are identified in the PD option with the special keyword SIG. When a SYSTEM directive is included in the program that specifies any of the instruction subsets (SIG5P, SIG7, etc.), the directive is ignored and treated as if it specified the same set of instruction procedures which was used to create the standard definition file.

By default, use of the PD option will produce a file named $:STDMET in the current job account. Also by default, it will attempt to open the $:STDMET file (unless the ND option is used) first in the current job account, and if not found, in the :SYS account. If neither account contains a $:STDMET file, the assembly proceeds as if ND had been specified. Note that the AC option does not apply to standard definition files.

It is possible to create and use standard definition files with other names. This requires an ASSIGN card for the F:STD DCB prior to calling Meta-Symbol. Figure 12 illustrates this method.

Unlike the default case when F:STD is preassigned by Meta-Symbol, a user assignment of F:STD will cause an abort if the file cannot be located.

There is no restriction on the language elements that may be included in a standard definition file, although it is expected that the common use will be for procedure definitions and certain symbol definitions. No output occurs on the listing for inclusion of a standard definition file; if the file causes code to be generated, the location counters will start with the values last set by the standard definition file.

CONCORDANCE CONTROL COMMANDS AND LISTING

When the CN option is included on the METASYM card, the assembler will access the C device for additional control records describing the data to be included in the concordance (symbolic name cross-reference) listing.

An alphanumeric string, such as R2, B, or RES is considered by Meta-Symbol to be an operation code when used in the first command field of a statement. When used elsewhere in a statement it is considered to be a symbol.

If desired, a "standard" concordance can be produced by entering the DC option on the METASYM command and omitting all concordance control records on the C device.

The "standard" concordance listing does not include operation code names, but otherwise includes all symbol references, including function and command procedure names and intrinsic functions such as $, L, AFA, etc.

LOCAL symbols or symbols appearing as arguments of a SYSTEM directive do not appear on any concordance listing. Except for this restriction, all symbols and operation codes used in a program can be listed by selective use of the concordance control commands.

CONCORDANCE CONTROL COMMANDS

The concordance subsystem provides the following commands for specifying the contents of a concordance listing:

IO	Include all or a selected set of operation codes.
SS	Suppress all or a selected set of symbols.
OS	Include only a selected set of symbols.
DS	Produce a modified LS listing, displaying only lines that reference a selected set of names.
END	Terminate concordance control commands.

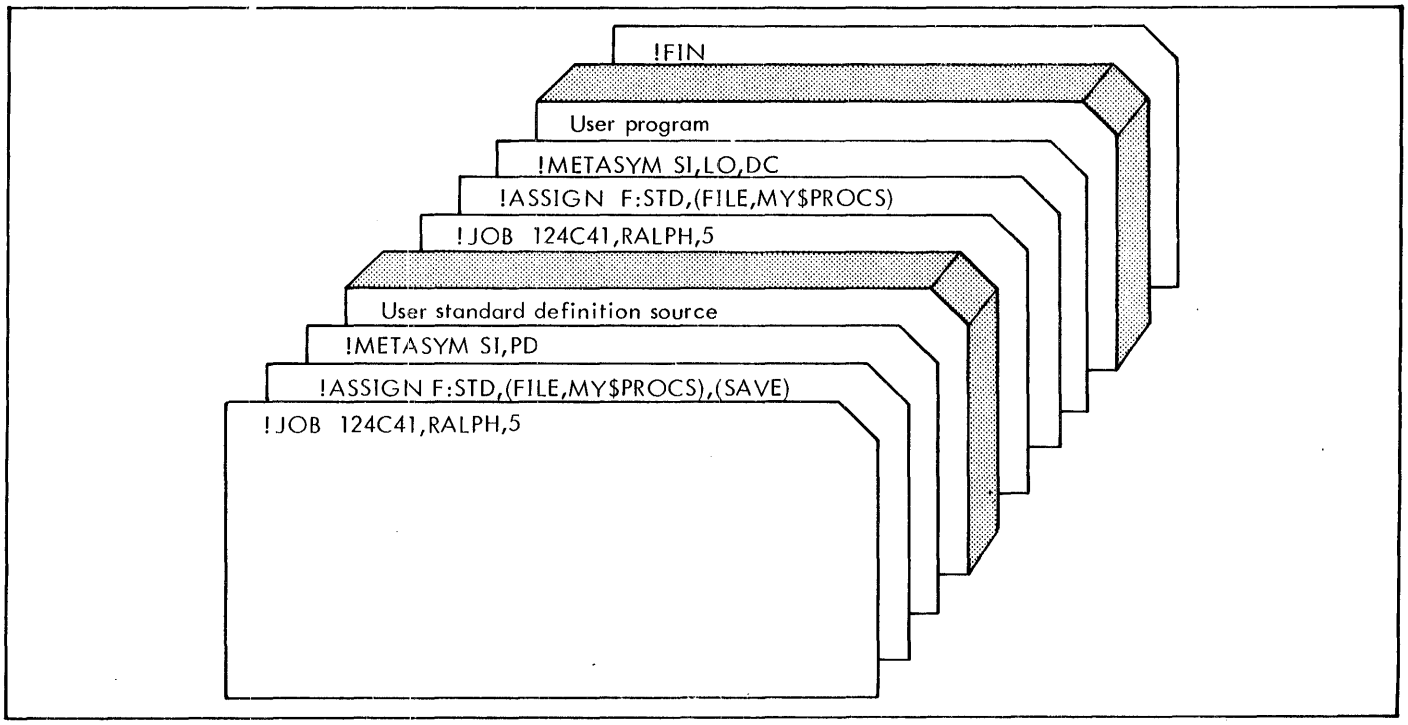


Figure 12. Creation and Use of a Named Standard Definition File

The control records must have a period (.) in column 1 and the selection code (i.e., command name) in columns 2-4. After a space of one or more blanks, a name list of the form $name_1, name_2, \ldots$ may follow the selection code. Embedded blanks between names in the list are not allowed. The name list may be continued for several physical records by using the Meta-Symbol semicolon continuation convention. Furthermore any number of records containing the same selection code may be used.

Symbols specified on concordance control commands are implicitly OPENed when the command is processed. The symbols may subsequently be OPENed and CLOSEd within the program and the command will control all such symbols with the same name. However, if a CLOSE balances the initial implicit OPEN, that symbol is effectively removed from further concordance control at the point of the CLOSE.

Concordance control records are printed, as read, on the LO device.

IO This command specifies that all operation codes, or only those given, are to appear on the concordance listing. The form of the command is

```
.IO[name1,name2, ...,namen]
```

If the name list is given, only the operation codes it specifies will be listed. If the name list is absent, all operation codes will be listed. (The brackets do not appear on the control record; they are shown above only to indicate that the name list is optional.)

SS This command specifies that all symbols, or only those given, are to be suppressed on the concordance listing. The form of the command is

```
.SS [name1,name2, ...,namen]
```

If the name list is given, only the symbols it specifies will be suppressed. If the name list is absent, all symbols will be suppressed. The SS and OS commands (explained below) may not both be used in a given set of concordance control commands. (The brackets do not appear on the control record; they are shown above only to indicate that the name list is optional.)

OS This command specifies that only a given list of symbols is to appear on the concordance listing. The form of the command is

```
.OS name1,name2, ...,namen
```

The name list is mandatory. Only the symbols it specifies will appear on the concordance listing. The SS and OS commands may not both be used in a given set of concordance control commands.

DS This command specifies that a given list of symbols is to be displayed by producing a modified LS listing. (The LS option was explained previously under "METASYM Control Command".) The format of the DS command is

```
.DS name1,name2, ...,namen
```

The name list is mandatory. Only the symbols it specifies will appear on the modified LS listing. Instead of the entire source program, the LS listing will display only lines containing names – in any context – specified in the DS name list. The DS command is independent of the IO, SS, and OS commands. The DS command overrides a request for a full LS listing.

END This command identifies the end of a set of concordance control commands. Its format is

```
.END
```

The END command is mandatory if the CN option is specified. If only the END command appears on the C device, a "standard" concordance listing will be produced.

CONCORDANCE LISTING

The concordance listing precedes the regular assembly listing. Names are printed on the concordance listing in alphabetical order, sorted on the first seven characters. Appearing on the lines below each name are one or more name reference items. The general format of each name reference item is

$$\text{reference line number} \begin{Bmatrix} \text{- op. code} \\ \$ \\ \text{/ op. code [*]} \end{Bmatrix}$$

where

reference line number is the source program line number in which the name appears. The largest reference line number that may be correctly processed is 32767. If update records appear in the concordance in the form "n.n", the largest update record number (".n") that may be correctly processed is 16383.

- op. code indicates that the name occurs in the label field of the reference line, and op. code is the operation code name used on that line.

$ indicates that the name occurs in the first command field of the reference line. In this case, $ terminates the reference item.

/ op. code [*] indicates that the name occurs in other than the label or first command field of the reference line, and op. code is the operation code name used on that line. The operation code name may be followed by an asterisk if the name specified occurred in argument field 1 and was indirectly addressed.

A sample name might appear on the concordance listing as

```
A
    372 - DATA          459/LW*
```

This display means that symbol A was used at line 372 in the label field of a DATA statement, and at line 459 of an indirectly addressed Load Word instruction.

Reference line numbers can appear in the form "n" or "n.n", depending on the form of the source program. The form n.n appears for those lines that are in an update record format and for which a new compressed file has not been produced.

The reference items following each name are formatted seven per line and are sorted by reference line number. Unusually long operation code names will cause fewer reference items per line to be printed.

LIMITATIONS

The largest reference line number that may be correctly processed is 32,767. If update records appear in the concordance in the form "n.n", the largest update record number (".n") that may be correctly processed is 16,383.

META-SYMBOL ERROR MESSAGES

Meta-Symbol has two basic phases: (a) the "encoder" phase, during which the input file is read and updated, system files are read, and syntax analysis is performed; and (b) the "assembly" phase, during which the input is assembled, and a listing and binary object program are produced. Both phases of Meta-Symbol may generate various error messages and diagnostics. This chapter explains these error messages.

TERMINAL ERRORS

Certain unusual conditions cause Meta-Symbol to terminate an assembly prematurely. In such a case, an explanation for the termination is given, followed by the message

```
METASYMBOL ABORT ERROR
```

If an abort occurs during a batch assembly (BA) run, the standard termination message is

```
METASYMBOL ABORT ERROR
PROCESSING PROGRAM NO. nnn
```

where nnn is the sequence number of the program within the batch.

Following either of the above messages, Meta-Symbol causes an error exit (M:ERR) to the monitor.

ENCODER PHASE ERROR MESSAGES

Errors detected during the "encoder" phase fall into five classes:

SYNTAX ERRORS

The encoder detects a variety of syntax errors during its pass over the input file. These errors are noted for later inclusion in the assembly listing, and will also appear in the source listing if one is produced (i.e., if the LS option is specified). Syntax error identification is in the form of a one-character flag displayed beneath the character in the symbolic image at which the error was detected. These one-character flags and their meanings were given in Table 6, Chapter 6.

ERRORS ENCOUNTERED DURING PROCESSING OF AN UPDATE PACKET

Update Syntax Errors

```
  record number          erroneous control record
                                      :
ILLEGAL SYNTAX

METASYMBOL ABORT ERROR
```

The update control record displayed has a syntax error in the position indicated by the colon. The position of the erroneous record in the update packet is indicated by the record number. For example:

```
100          +5,Z
                :
```

Update Record Sequence Errors

```
  record number 1        erroneous control record 1

  record number 2        erroneous control record 2
ILLEGAL UPDATE SEQUENCE

METASYMBOL ABORT ERROR
```

The update control records displayed are not in sequential order and the SU option has been specified.

Update Command Value Errors

```
record number          erroneous control record
ILLEGAL UPDATE SEQUENCE
METASYMBOL ABORT ERROR
```

The update control record displayed is of the form +j,k where j > k.

Update Record Overlap Errors

```
record number 1        erroneous control record 1
record number 2        erroneous control record 2
OVERLAPPING SEQUENCE NUMBERS
METASYMBOL ABORT ERROR
```

The update control records displayed are overlapping in an illegal manner. For example:

```
10      +13,26
27      +3,15
```

Update Line Number Errors

```
UPDATE CONTROL NUMBERS EXCEED COMPRESSED
FILE
METASYMBOL ABORT ERROR
```

A line number specified in an update control record is greater than the number of lines in the program.

ERRORS ENCOUNTERED WHILE PROCESSING AN INPUT FILE

Input File Control Byte Errors

```
ERROR RECORD CONTROL BYTES xx/xx/xx/xx.
ID/SEQUENCE/CHECKSUM/BYTE COUNT.
PROCESSING SYSTEM – system name.      } If error oc-
AT LEVEL – level of system file nesting. } curs while
                                          processing a
                                          system file.
COMPRESSED RECORD { ID / SEQUENCE / CHECKSUM }  ERROR
   SHOULD BE – correct { identification / sequence / checksum }
METASYMBOL ABORT ERROR
```

While reading a compressed input or system file, a compressed record was encountered with the indicated erroneous control byte.

Compressed Input File Missing

```
COMPRESSED FILE MISSING AFTER UPDATE PACKET
METASYMBOL ABORT ERROR
```

After processing an update packet, the encoder expected but did not find a compressed file on the appropriate input device.

Compressed Input File Incomplete

```
INCOMPLETE COMPRESSED FILE
PROCESSING SYSTEM – system name.      } If error oc-
AT LEVEL – level of system call nesting. } curs while
                                          processing a
                                          system file.
METASYMBOL ABORT ERROR
```

A symbolic record or an end-of-file was encountered while processing a compressed file, prior to having encountered the compressed end-of-file byte. Typically, this error occurs when cards have been lost from the end of a compressed deck.

Input File END Directive Missing

```
EOF ENCOUNTERED. END DIRECTIVE SUPPLIED BY
ENCODER.
PROCESSING SYSTEM – system name.      } If error oc-
AT LEVEL – level of system call nesting. } curs while
                                          processing a
                                          system file.
```

While processing the input file, an end-of-file was encountered prior to encountering an END directive. The missing END directive is supplied by the encoder.

ERRORS ENCOUNTERED DURING THE OPENING OR PROCESSING OF SYSTEM FILES

System Missing

```
UNABLE TO FIND SYSTEM – system name.
AT LEVEL – level of system nesting.
METASYMBOL ABORT ERROR
```

A SYSTEM directive specifying the system name displayed has been encountered, but there is no system filed with this name under any of the account numbers specified by the AC option (if any), under the current job account, or under the "system account number".

Monitor-Detected Errors

```
ERROR IN OPENING – system name.
AT LEVEL – level of system nesting.
METASYMBOL ABORT ERROR
```

An error has occurred while trying to find the system filed under the system name displayed.

OTHER ABNORMAL CONDITIONS ENCOUNTERED BY THE ENCODER

No Input Option Specified

```
NO INPUT SPECIFIED.
METASYMBOL ABORT ERROR
```

Neither SI nor CI was specified on the METASYM card, but the card contained other options.

Monitor-Detected Abnormal Conditions

```
BAD I/O.  ABNORMAL CODE – xx.
PROCESSING SYSTEM – system name.   } If error occurs while
AT LEVEL – level of system call nesting. } processing a system file.
METASYMBOL ABORT ERROR
```

An abnormal condition has been signaled by the Monitor.

Encoder Abort Errors

```
ENCODER ABORT.          symbolic image
METASYMBOL ABORT ERROR
```

A machine error or an assembler error during the encoder's syntax analysis was encountered in processing the line displayed.

ASSEMBLY PHASE ERROR MESSAGES

In the "assembly" phase, a variety of syntactical, logical, and functional errors are detected during the two passes over the input. These errors are normally included in the assembly listing, but will be listed on the LO device even if the LO option is not specified. The non-fatal errors cause severity of 3.

Checksum Errors

```
X1 CHECKSUM ERROR
METASYMBOL ABORT ERROR
```

A hardware malfunction has occurred while reading the intermediate file.

Duplicate Definitions of Program Symbols

```
DBL DEF
```

This error message is caused by one of the following conditions:

1. A non-redefinable symbol is defined more than once within the program, or a symbol is defined in both a redefinable and a non-redefinable context.

2. The same symbol is declared as an external definition more than once within the program, or the symbol is declared as both an external definition and an external reference.

Unterminated Loops

```
PEND/END BEFORE FIN
```

The assembler has detected an unterminated DO or WHILE loop (i.e., a PEND or END directive was encountered before the FIN directive that should have terminated the loop).

Unterminated Procedures

```
END BEFORE PEND
```

The assembler has detected an unterminated procedure (i.e., an END directive was encountered before the PEND directive that should have terminated the procedure).

Illegal Placement of a Directive

```
INVALID DIRECTIVE
```

This error message is produced when a directive occurs in a context where it either is meaningless or cannot be processed consistently.

1. An ELSE or FIN directive occurs outside a DO or WHILE loop, or an extra ELSE was encountered inside a DO or WHILE loop.

2. A PEND directive occurs outside a procedure definition.

3. A DO1, END, or SYSTEM directive immediately follows a DO1 directive that has a repeat count greater than one.

4. An S:RELP directive was encountered within a procedure.

Illegal Argument Fields

ILLEGAL AF

This error message is caused by one of the following conditions:

1. The argument field for SCOR or TCOR is not a list.
2. The argument field for BOUND contains other than an integer from 1 to X'8000'.
3. The argument field for DO, DO1, RES, SPACE, or WHILE contains other than a single-precision integer.
4. The argument field for ORG or LOC contains other than an integer or an address.
5. The argument field for USECT contains other than an address.
6. The argument field for CSECT, DSCET, or PSECT contains other than an integer between 0 and 3.
7. The argument field of a standard instruction is blank, or contains more than two fields.
8. The argument field for DEF, REF, SREF, CDISP, or FDISP contains other than unsupscripted global symbols.
9. The (nonblank) argument field for TITLE contains other than a single character string constant of 0 to 75 characters.
10. The argument field of an immediate class instruction is indirect, or specifies indexing.
11. The (nonblank) argument field for LIST, PCC, PSR, PSYS contains other than a non-negative, single-precision integer.
12. The argument field for ERROR, TEXT, or TEXTC contains other than character string constants.
13. The argument field for COM or GEN contains more values than the number of fields specified in the command field.

Illegal Command Fields

ILLEGAL CF

This error message is caused by one of the following conditions:

1. The severity level for ERROR is other than an integer from 0 to 15, or the condition is other than an integer.
2. The command field for ORG, LOC, REF, or SREF contains other than the integers 1, 2, 4, or 8.
3. The command field for DATA contains other than an integer from 0 to 16.
4. The command field (for class 0 or 2) of a standard instruction is blank.
5. The (nonblank) command field list for COM and GEN contains other than non-negative, single-precision integers, or the total is not a multiple of 8, or is greater than 128.
6. The (nonblank) command field for CNAME, EQU, GOTO, RES, S:SIN, or SET contains other than a non-negative, single-precision integer, or the command field for S:SIN is greater than 2.

Illegal Forward References

ILLEGAL FORWARD

A symbol or literal was used in a directive in such a way that core allocation could not be determined at the time that the directive was processed (e.g., a forward reference in the field list of a GEN directive or in the command or argument field of a RES directive).

Unsatisifed Local GOTO Searches

INVALID LOCAL GOTO

The assembler has encountered a LOCAL directive while a GOTO search was being made for a local symbol.

Illegal Use of GOTO

ILLEGAL GOTO

This error message is caused by one of the following conditions:

1. Command field two of the GOTO directive specifies a number greater than the number of symbols in the argument field.
2. The selected argument is not a symbol.
3. The selected argument is a local symbol passed into the procedure from the reference line.

Illegal Labels

ILLEGAL LABEL

This error message is caused by one of the following conditions:

1. The label field for CNAME, COM, FNAME, or S:SIN contains other than an unsubscripted global symbol or a list of such symbols.

2. The label field for DSECT contains other than a single unsubscripted global symbol.

3. The label field for a directive that enters values into the symbol table contains other than a blank, a symbol, a subscripted symbol, or a list of symbols or subscripted symbols.

Illegal Subscripts

ILLEGAL SUBSCRIPT

A subscripted definition is not an integer from 1 to 255, or a subscripted reference is not an integer from 0 to 255.

Maximum Procedure Level Exceeded

PROC LEVEL > 31

More than 31 levels of procedure referencing have been encountered.

New Literals in Pass 2

NEW LITERAL IN PASS 2

Most commonly, the argument of a literal is itself a literal, i.e., the literal of a literal.

Illegal Operand Types

OPERAND TYPE ERROR

An operand that is illegal for the associated operator (or, possibly, for all operators) has been encountered.

Excessive Number of List Elements

LIST TOO LONG

The indicated operation would create a list containing more than 255 elements. The list is truncated to the first 255 elements.

Unterminated Skips

SKIP TERMINATED BY PEND/END

The assembler has detected an unterminated skip in a conditional assembly sequence in a procedure (i.e., a PEND or END directive was encountered before the termination condition was satisfied).

Memory Overflows

INSUFFICIENT CORE META-SYMBOL ABORT ERROR

The program being assembled is too large to assemble in the amount of core memory available. (This error can also occur during the "encoder" phase.)

Overlong Text Strings

TEXT TOO LONG

A single text string contains more than 255 EBCDIC characters.

Excessive Generated Data Lengths

TRUNCATION

The assembler has encountered a generated data value that is too long for the specified field.

Undefined Local Symbols

UNDEFINED LOCALS

A symbol declared to be local was used, but not defined, within the previous local region. (This message appears at the end of a local region.)

Undefined Symbols

UNDEF SYM

An attempt was made to evaluate a global symbol that was not defined on Pass 1 of assembly, and has not yet been defined on Pass 2.

Unrecognized Commands

UNDEF COM

The assembler has encountered a command procedure reference containing an unrecognized command procedure name. The command is evaluated as if it were a class 0 instruction with an op-code of X'00'.

Circularly Defined Symbols

CIRCULAR DEF

The assembler has encountered a symbol that is defined in terms of itself, either directly or indirectly.

Use of Doubly Defined Symbols

USE OF DBL DEF SYM

The assembler has encountered an instruction in which a doubly defined program symbol is used.

Illegal Object Language Value

```
VALUE TYPE ERROR
```

The argument cannot be expressed in the standard object language.

Doubly Defined Commands

```
DBL DEF COM
```

The assembler has encountered a CNAME, COM, or S:SIN statement label that is identical to the label of another CNAME, COM, or S:SIN statement.

Arithmetic Operand Precision Exceeded

```
ARITHMETIC TRUNCATION
```

The assembler has encountered an arithmetic operation in which the precision of one or more of the operands exceeds the limits allowed.

Illegal Use of CNAME

```
DBL DEF DIR
```

An attempt has been made to redefine a Meta-Symbol directive with CNAME, COM, or S:SIN.

Excessive Number of Control Sections

```
TOO MANY CS
```

A CSECT, PSECT, or DSECT directive has been encountered after 127 relocatable control sections have been generated. The directive is ignored.

Illegal Use or Placement of SOCW

```
SOCW ERROR
```

An illegal object language feature is required, or one of the directives DEF, REF, SREF, CSECT, DSECT, or USECT has been encountered, while the assembly is under SOCW control, or the SOCW directive has been encountered after the assembler has begun generating object code.

Unrecognized Key in S:KEYS Function

```
UNRECOGNIZED KEY
```

The scanned argument field of the PROC reference contains a keyword which is not specified in the S:KEYS reference. Reporting of this condition is suppressed if mode&4>0.

Missing Key in S:KEYS Reference

```
MISSING KEY
```

A required hit has not occurred.

Key Conflict in S:KEYS Function

```
KEY CONFLICT
```

More than one hit has occurred for a single bit specification.

Illegal Use of S:KEYS

```
ILLEGAL S:KEYS
```

S:KEYS is not being used within a PROC or the S:KEYS argument contains illegal syntax.

METASYM CONTROL COMMAND ERROR MESSAGES

Errors can also be detected on the METASYM control card:

Unrecognized METASYM Options

```
METASYM card image
          :
ILLEGAL OPTION IGNORED
```

The option in the position indicated by the colon is unknown.

Illegal Account Numbers

```
METASYM card image
          :
ILLEGAL ACCOUNT NO. IGNORED
```

The account number in the position indicated by the colon contains more than eight alphanumeric characters, or more than nine account numbers have been specified.

Syntax Errors in METASYM Commands

```
METASYM card image
          :
ILLEGAL SYNTAX
```

The character in the position indicated by the colon is erroneous syntactically.

CONCORDANCE CONTROL COMMAND ERROR MESSAGES

Control Command Conflict

```
*CONTROL CONFLICT. ABOVE STATEMENT IGNORED
```

Both the SS and OS commands have been encountered. Both may not be used.

Incorrect Symbol List

```
*IMPROPER SYMBOL LIST
```

The name list in the preceding control command is improperly formatted: blanks between names, no commas, etc.

Missing Symbol List

```
*MISSING SYMBOL LIST
```

The mandatory list of names is missing after the OS or DS control command.

Incorrect Control Command

```
*IMPROPER CN CONTROL.  END OF PROCESSING
```

The previous record was not a concordance control record. The sequence of control records is considered terminated, but a concordance listing will be produced according to any legal commands received prior to the error.

Concordance Overflow

```
*INSUFFICIENT SPACE TO PRODUCE CN
```

Less than one page (512 words) of computer memory, over and above the assembly's symbol tables, is available to the assembler when the concordance listing is to be produced. The concordance listing is aborted.

Extended Memory Required

```
*CONCORDANCE EXTENDED MEMORY MODE
*REFERENCE COUNT – xxxx.  DISC OVERFLOW – yyyy.
```

Maximum efficiency in producing the concordance listing is attained when all its reference data may be co-resident in memory with the assembly's symbol tables. If this is not possible, part of the data must remain on the intermediate file during concordance listing, causing repeated accesses to the RAD. When this is the case, the above message prints just before the concordance listing.

The reference count is the total number of reference items, and disk overflow count is the number of items that are not memory resident. If output speed reaches an unacceptable level, the disk overflow figure indicates the approximate amount the data should be reduced by modifying the concordance control commands.

As another alternative, the programmer might wish to consider use of the DS command. The DS command requires no reference item storage.

EXAMPLES OF RUN DECKS

Shown below in Figures 13 through 16 are examples of legal run deck structures of varying complexity. In all the following examples, a blank has been inserted after the character "!" preceding some commands. This is permissible on all but the input control commands; namely, EOD, BIN, BCD, DATA, and FIN.

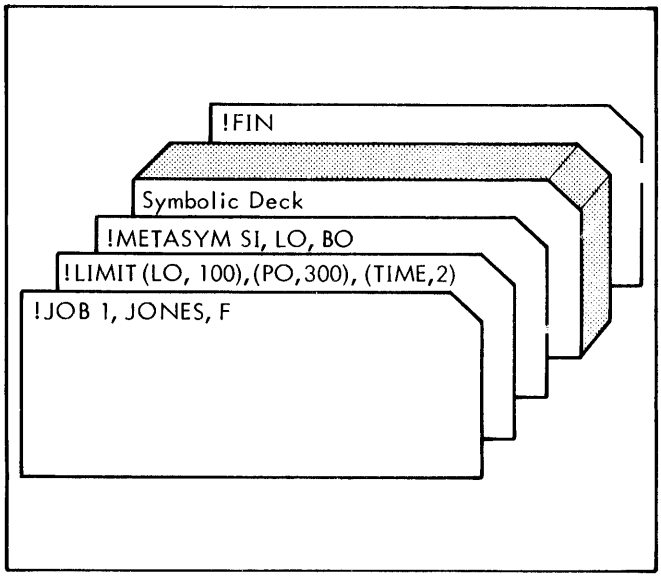


Figure 13. Sample Run Deck – Single Symbolic Assembly

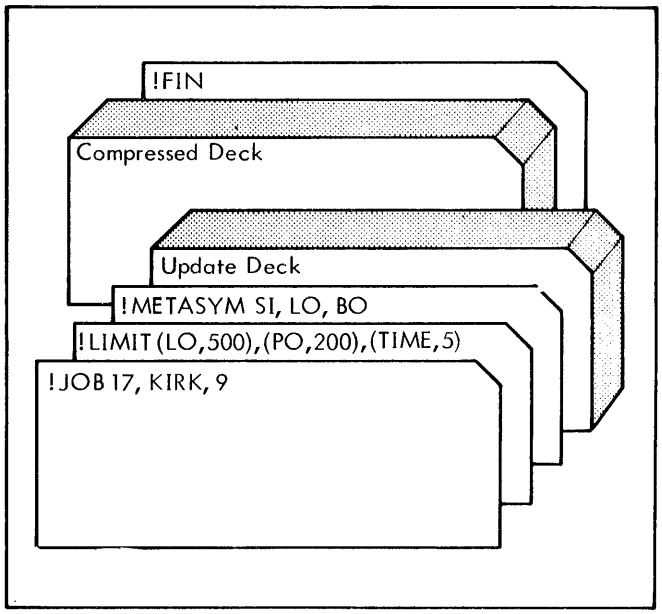


Figure 14. Sample Run Deck – Single Assembly with Update

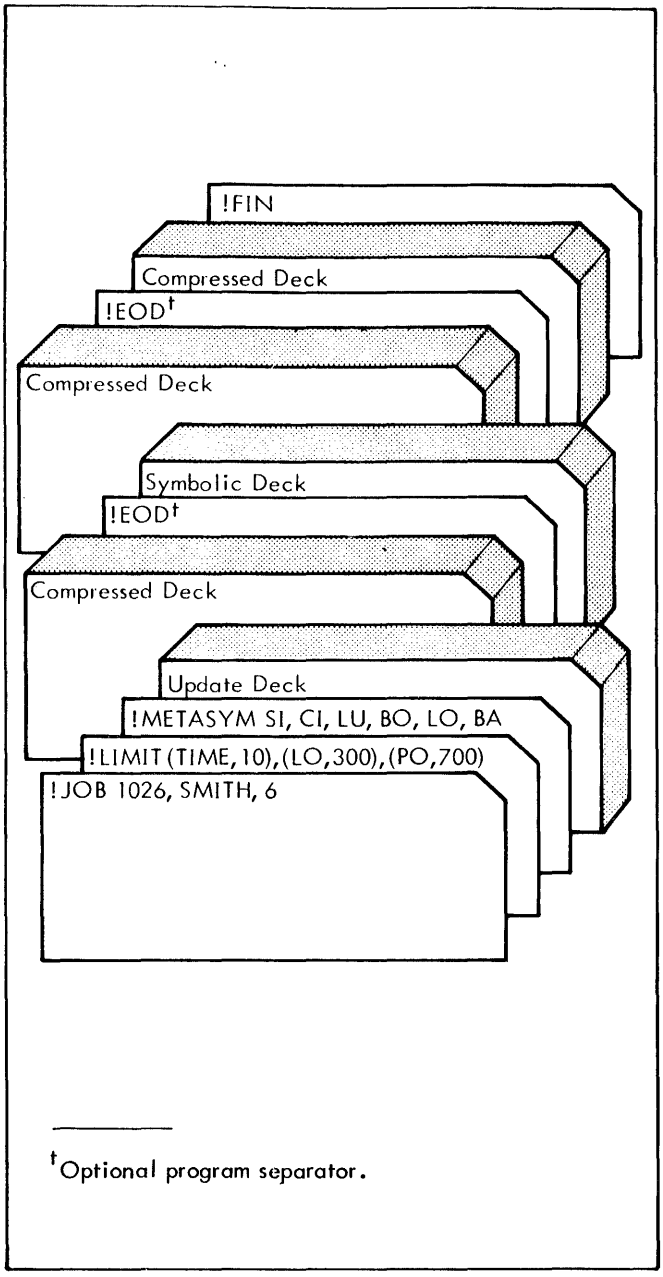


Figure 15. Sample Run Deck – Batch Assembly

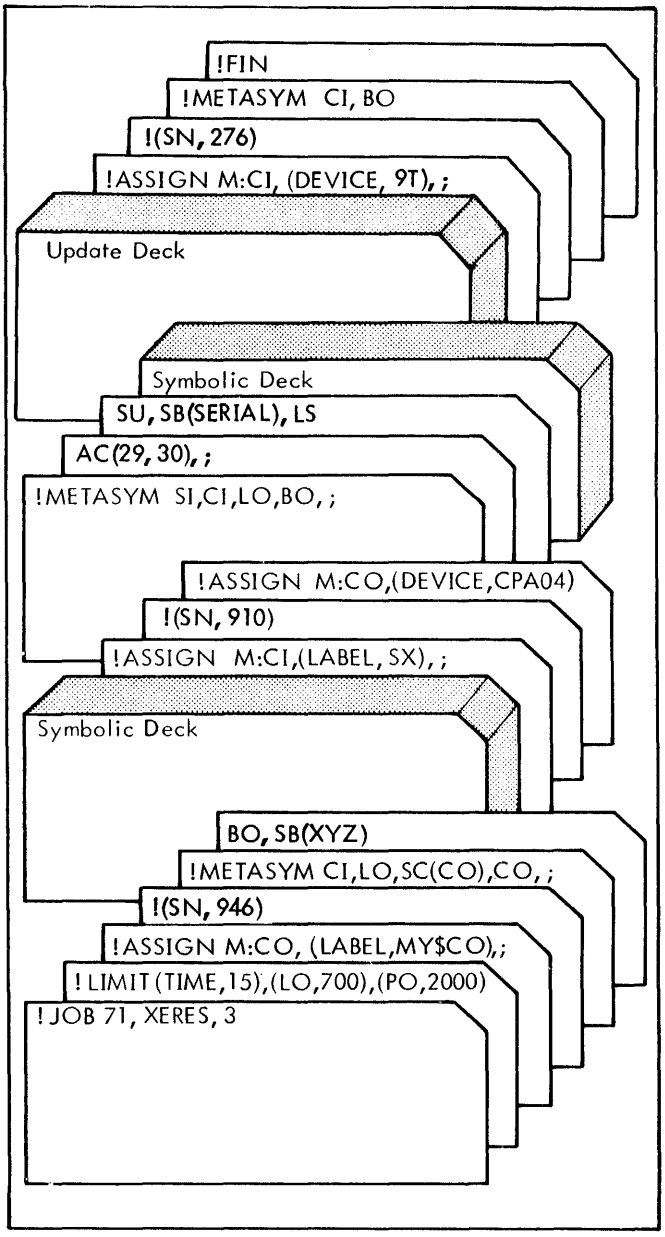


Figure 16. Sample Run Deck – Multiple Assembly with Compressed Input and Output on Magnetic Tape

APPENDIX A. SUMMARY OF META-SYMBOL DIRECTIVES

In this summary brackets are used to indicate optional items.

Form			Function	Page
[label$_1$,...,label$_n$]	ASECT		Declares generative statements will be assembled to be loaded into absolute locations.	26
	BOUND	boundary	Advances the execution location counter to a byte multiple of "boundary" and advances the load location counter the same number of bytes.	25
	CDISP	symbol$_1$[,...,symbol$_n$]	Displays the command procedure identified by "symbol$_i$".	62
	CLOSE	[symbol$_1$,...,symbol$_n$]	Declares that "symbol$_i$" are to be permanently closed for all subsequent usage.	44
label$_1$[,...,label$_n$]	CNAME[,n]	[list]	Designates a command ("label") for the next procedure definition and specifies the values ("list") associated with "label".	59
label$_1$[,...,label$_n$]	COM[,field list]	[value list]	Describes a command skeleton; specifies the contents of each "field"; "label" is the symbol by which the command skeleton is referenced.	51
[label$_1$,...,label$_n$]	CSECT	[expression]	Declares program section "label" as a relocatable control section with memory protection specified by "expression" where $0 \leq$ expression ≤ 3. If "expression" is omitted, the value 0 (no memory protection) is assumed.	26
[label$_1$,...,label$_n$]	DATA[,f]	[value$_1$,...,value$_n$]	Generates each value in the list of "value$_i$" into a field whose size is specified by f in bytes. If f is omitted, a field size of 4 bytes is assumed.	53
	DEF	[symbol$_1$,...,symbol$_n$]	Declares that the "symbol$_i$" may be referenced by other separately assembled programs.	46
	DISP	[list]	Displays each value specified in "list" on the assembly listing.	57
[label$_1$,...,label$_n$]	DO	[expression]	If the value of "expression" is greater than zero, processes the code from DO to ELSE or FIN (if ELSE is absent) "expression" times. Then continues assembly at the statement following FIN. If "expression" ≤ 0, skips all code from DO to ELSE or FIN (if ELSE is absent); resumes assembly at that point.	37
[label$_1$,...,label$_n$]	DO1	[expression]	If the value of "expression" is greater than zero, processes the one statement following the DO1, "expression" times, then continues the assembly at the next statement. If "expression" ≤ 0, skips the statement following DO1 and resumes assembly.	34

Form			Function	Page
label	DSECT	[expression]	Declares a dummy program section "label" with memory protection specified by "expression" where $0 \leq$ expression ≤ 3. If "expression" is omitted, the value 0 (no memory protection) is assumed.	26
	ELSE		Terminates the range of an active DO or WHILE loop, or identifies the beginning of the alternate sequence of code for an inactive DO or WHILE loop.	35, 37
[label$_1$,..., label$_n$]	END	[expression]	Terminates a program or system file. Optionally provides the starting address of the program. If a label is given, associates it with the location immediately following the literal table, which is generated at the end of the currently active program section.	34
[label$_1$,..., label$_n$]	EQU[,s]	[list]	Equates "label" to the value of "list". (Non-redefinable)	42
	ERROR[, s[, c]]	'cs$_1$'[,...,'cs$_n$']	If c >0, s is compared with the current highest severity level, the higher value is retained, and "message" is output. If c $\leq$ 0, ERROR is ignored.	57
	FDISP	symbol$_1$[,..., symbol$_n$]	Displays the function procedure identified by the "symbol$_i$".	62
	FIN		Terminates a DO or WHILE loop.	35, 37
label$_1$[,..., label$_n$]	FNAME	[list]	Designates a function name ("label" for the next procedure definition and specifies the values ("list") associated with "label".	59
[label$_1$,..., label$_n$]	GEN[,field list]	[value list]	Produces a hexadecimal value representing "value list" in the number of bits specified by "field" in "field list".	50
	GOTO[, k]	label$_1$[,..., label$_n$]	Resumes assembly at the statement whose label corresponds to the kth "label".	34
	LIST	[expression]	Suppresses or resumes assembly listing depending on value of "expression". If "expression" is zero, assembly listing following LIST will be suppressed until resumed by another LIST directive; if "expression" is nonzero, assembly listing is enabled.	56
[label$_1$,..., label$_n$]	LOC[, n]	[location]	Sets the execution location counter ($) to the value "location" and sets its resolution specification to n, where the value of n is 1, 2, 4, or 8.	25
	LOCAL	[symbol$_1$,..., symbol$_n$]	Terminates existing local symbol region and initiates a new region where the "symbol$_i$" are local symbols.	43
	OPEN	[symbol$_1$,...,symbol$_n$]	Declares that the "symbol$_i$" are to be open for use as symbols until another OPEN or a CLOSE directive is encountered.	44

Form			Function	Page
[label$_1$, . . . , label$_n$]	ORG[,n]	[location]	Sets both the current load location counter ($$) and the current execution location counter ($) to the value "location" and sets their resolution specifications to n, where the value of n is 1, 2, 4, or 8.	24
	PAGE		Upspaces assembly listing to the top of form.	58
	PCC	[expression]	Suppresses or resumes assembly listing of directives PAGE, SPACE, TITLE, LIST, PSR, PSYS, and PCC, depending on value of "expression". If "expression" is zero, assembly listing of these directives will be suppressed until resumed by another PCC directive; if "expression" is nonzero, these directives will be listed.	56
	PEND	[list]	Terminates procedure definition.	60
	PROC		Identifies the beginning of a procedure definition.	60
[label$_1$,. . .,label$_n$]	PSECT	[expression]	Declares program section "label" as a relocatable control section to be loaded on a page boundary with memory protection specified by "expression" where expression is in the range 0 to 3. If "expression" is omitted, the value 0 (no memory protection) is assumed.	27
	PSR	[expression]	Suppresses or resumes assembly listing of lines skipped under control of GOTO, DO, or WHILE, depending on value of "expression". If "expression" is zero, assembly listing of lines skipped subsequent to PSR will be suppressed until resumed by another PSR directive; if "expression" is nonzero, skipped lines will be listed.	56
	PSYS	[expression]	Suppresses or resumes assembly listing of files called by the SYSTEM directive. If "expression" is zero, assembly listing of all files called by SYSTEM subsequent to PSYS will be suppressed until resumed by another PSYS; if "expression" is nonzero, system files will be listed.	57
	REF [,n]	symbol$_1$[, . . .,symbol$_n$]	Declares that the "symbol$_i$" are references to externally defined symbols.	48
[label$_1$,. . .,label$_n$]	RES[,n]	[expression]	Adjusts both location counters ($ and $$) by the number of n-sized units indicated by the value of expression. If n is omitted, a size of four bytes is assumed.	26
[label$_1$,. . .,label$_n$]	SET[,s]	[list]	Equates "label" to the value of "list". (Redefinable.)	43
	S:RELP		Releases all command and function procedure definitions.	62
label$_1$[,. . .,label$_n$]	S:SIN,n	[expression]	Defines standard instruction, "label", to be of format "n", with opcode "expression".	53

Form			Function	Page
	SOCW		Suppresses the automatic generation of object control words.	55
	SPACE	[expression]	Upspaces the assembly listing the number of lines indicated by expression. If expression is omitted, 1 is assumed.	55
	SREF[,n]	[symbol$_1$,..., symbol$_n$]	Declares that the "symbol$_i$" are secondary external references.	48
	SYSTEM	name	Calls system "name" from the library storage media.	33
[label$_1$,...,label$_n$]	TEXT	'cs$_1$'[,...,'cs$_n$']	Assembles the "cs$_i$" (character string constant) in binary-coded format for use as an output message.	54
[label$_1$,...,label$_n$]	TEXTC	'cs$_1$'[,...,'cs$_n$']	Assembles the "cs$_i$" (character string constant) in binary-coded format, preceded by a byte count, for use as an output message.	55
	TITLE	['cs']	Prints "cs" (character string constant) as a heading on each page of assembly listing.	56
[label$_1$,...,label$_n$]	USECT	name	Specifies that the control section of which label "name" is part is to be used in assembling subsequent statements.	27
[label$_1$,...,label$_n$]	WHILE	[expression]	If "expression" ≤ 0, skips all code from WHILE to ELSE or FIN (if ELSE is absent), and resumes assembly at that point. If "expression" > 0, performs the comparison ($0 \leq$ expression) again, and proceeds accordingly.	35

APPENDIX B. SUMMARY OF SIGMA INSTRUCTION MNEMONICS

Required syntax items are underlined whereas optional items are not. The following abbreviations are used:

m	mnemonic
r	register expression
v	value expression
*	indirect designator
a	address expression
x	index expression
d	displacement expression

Codes for required options are

9	Sigma 9
7	Sigma 7 (or 9)
P	Privileged
D	Decimal Option
F	Floating-Point Option
L	Lock Option
MP	Memory Map Option
SF	Special Feature—not implemented on all machines

Mnemonic	Syntax		Function	Equivalent to:	Required Options
LOAD/STORE					
LI	m, r	v	Load Immediate		
LB	m, r	*a, x	Load Byte		
LH	m, r	*a, x	Load Halfword		
LW	m, r	*a, x	Load Word		
LD	m, r	*a, x	Load Doubleword		
LCH	m, r	*a, x	Load Complement Halfword		
LAH	m, r	*a, x	Load Absolute Halfword		
LCW	m, r	*a, x	Load Complement Word		
LAW	m, r	*a, x	Load Absolute Word		
LCD	m, r	*a, x	Load Complement Doubleword		
LAD	m, r	*a, x	Load Absolute Doubleword		
LS	m, r	*a, x	Load Selective		
LM	m, r	*a, x	Load Multiple		
LCFI	m	v, v	Load Conditions and Floating Control Immediate		
LCI	m	v	Load Conditions Immediate		
LFI	m	v	Load Floating Control Immediate		
LC	m	*a, x	Load Conditions		
LF	m	*a, x	Load Floating Control		
LCF	m	*a, x	Load Conditions and Floating Control		
LAS	m, r	*a, x	Load and Set		SF
LMS	m, r	*a, x	Load Memory Status		SF
LRA	m, r	*a, x	Load Real Address		9P
XW	m, r	*a, x	Exchange Word		
STB	m, r	*a, x	Store Byte		
STH	m, r	*a, x	Store Halfword		
STW	m, r	*a, x	Store Word		
STD	m, r	*a, x	Store Doubleword		
STS	m, r	*a, x	Store Selective		
STM	m, r	*a, x	Store Multiple		
STCF	m	*a, x	Store Conditions and Floating Control		
ANALYZE AND INTERPRET					
ANLZ	m, r	*a, x	Analyze		
INT	m, r	*a, x	Interpret		
FIXED-POINT ARITHMETIC					
AI	m, r	v	Add Immediate		
AH	m, r	*a, r	Add Halfword		
AW	m, r	*a, x	Add Word		
AD	m, r	*a, x	Add Doubleword		
SH	m, r	*a, x	Subtract Halfword		

Mnemonic	Syntax		Function	Equivalent to:	Required Options
FIXED-POINT ARITHMETIC (cont.)					
SW	m, r	*a, x	Subtract Word		
SD	m, r	*a, x	Subtract Doubleword		
MI	m, r	v	Multiply Immediate		
MH	m, r	*a, x	Multiply Halfword		
MW	m, r	*a, x	Multiply Word		
DH	m, r	*a, x	Divide Halfword		
DW	m, r	*a, x	Divide Word		
AWM	m, r	*a, x	Add Word to Memory		
MTB	m, v	*a, x	Modify and Test Byte		
MTH	m, v	*a, x	Modify and Test Halfword		
MTW	m, v	*a, x	Modify and Test Word		
COMPARISON					
CI	m, r	v	Compare Immediate		
CB	m, r	*a, x	Compare Byte		
CH	m, r	*a, x	Compare Halfword		
CW	m, r	*a, x	Compare Word		
CD	m, r	*a, x	Compare Doubleword		
CS	m, r	*a, x	Compare Selective		
CLR	m, r	*a, x	Compare with Limits in Register		
CLM	m, r	*a, x	Compare with Limits in Memory		
LOGICAL					
OR	m, r	*a, x	OR Word		
EOR	m, r	*a, x	Exclusive OR Word		
AND	m, r	*a, x	AND Word		
SHIFT					
S	m, r	*a, x	Shift		
SLS	m, r	v, x	Shift Logical, Single		
SLD	m, r	v, x	Shift Logical, Double		
SCS	m, r	v, x	Shift Circular, Single		
SCD	m, r	v, x	Shift Circular, Double		
SAS	m, r	v, x	Shift Arithmetic, Single		
SAD	m, r	v, x	Shift Arithmetic, Double		
SSS	m, r	a, x	Shift Searching, Single		9
SSD	m, r	a, x	Shift Searching, Double		9
SF	m, r	*a, x	Shift Floating		
SFS	m, r	v, x	Shift Floating, Short		
SFL	m, r	v, x	Shift Floating, Long		
CONVERSION					
CVA	m, r	*a, x	Convert by Addition		7
CVS	m, r	*a, x	Convert by Subtraction		7
FLOATING-POINT ARITHMETIC					
FAS	m, r	*a, x	Floating Add Short		F
FAL	m, r	*a, x	Floating Add Long		F
FSS	m, r	*a, x	Floating Subtract Short		F
FSL	m, r	*a, x	Floating Subtract Long		F
FMS	m, r	*a, x	Floating Multiply Short		F
FML	m, r	*a, x	Floating Multiply Long		F

Mnemonic	Syntax		Function	Equivalent to:	Required Options
FLOATING-POINT ARITHMETIC (cont.)					
FDS	m, r	*a, x	Floating Divide Short		F
FDL	m, r	*a, x	Floating Divide Long		F
DECIMAL					
DL	m, v	*a, x	Decimal Load		D
DST	m, v	*a, x	Decimal Store		D
DA	m, v	*a, x	Decimal Add		D
DS	m, v	*a, x	Decimal Subtract		D
DM	m, v	*a, x	Decimal Multiply		D
DD	m, v	*a, x	Decimal Divide		D
DC	m, v	*a, x	Decimal Compare		D
DSA	m	*a, x	Decimal Shift Arithmetic		D
PACK	m, v	*a, x	Pack Decimal Digits		D
UNPK	m, v	*a, x	Unpack Decimal Digits		D
BYTE STRING					
MBS	m, r	d	Move Byte String		7
CBS	m, r	d	Compare Byte String		7
TBS	m, r	d	Translate Byte String		7
TTBS	m, r	d	Translate and Test Byte String		7
EBS	m, r	d	Edit Byte String		D
PUSH DOWN					
PSW	m, r	*a, x	Push Word		
PLW	m, r	*a, x	Pull Word		
PSM	m, r	*a, x	Push Multiple		
PLM	m, r	*a, x	Pull Multiple		
MSP	m, r	*a, x	Modify Stack Pointer		
EXECUTE/BRANCH					
EXU	m	*a, x	Execute		
BCS	m, v	*a, x	Branch on Conditions Set		
BCR	m, v	*a, x	Branch on Conditions Reset		
BIR	m, r	*a, x	Branch on Incrementing Register		
BDR	m, r	*a, x	Branch on Decrementing Register		
BAL	m, r	*a, x	Branch and Link		
B	m	*a, x	Branch	BCR, 0	*a, x
BE	m	*a, x	Branch if Equal	BCR, 3	*a, x
BG	m	*a, x	Branch if Greater Than	BCS, 2	*a, x
BGE	m	*a, x	Branch if Greater Than or Equal to	BCR, 1	*a, x
BL	m	*a, x	Branch if Less Than	BCS, 1	*a, x
BLE	m	*a, x	Branch if Less Than or Equal to	BCR, 2	*a, x
BNE	m	*a, x	Branch if Not Equal to	BCS, 3	*a, x
BEZ	m	*a, x	Branch if Equal to Zero	BCR, 3	*a, x
BNEZ	m	*a, x	Branch if Not Equal to Zero	BCS, 3	*a, x
BGZ	m	*a, x	Branch if Greater Than Zero	BCS, 2	*a, x
BGEZ	m	*a, x	Branch if Greater Than or Equal to Zero	BCR, 1	*a, x
BLZ	m	*a, x	Branch if Less Than Zero	BCS, 1	*a, x
BLEZ	m	*a, x	Branch if Less Than or Equal to Zero	BCR, 2	*a, x
BAZ	m	*a, x	Branch if Implicit AND is Zero[†]	BCR, 4	*a, x
BANZ	m	*a, x	Branch if Implicit AND is Nonzero[†]	BCS, 4	*a, x

(For Use After Comparison Instructions)

[†] See CW instruction in Xerox Sigma 7 Computer Reference Manual.

Mnemonic	Syntax			Function	Equivalent to:	Required Options
EXECUTE/BRANCH (cont.)						
BOV	m	*a,x	For Use After Fixed-Point Arithmetic Instructions	Branch if Overflow	BCS,4	*a,x
BNOV	m	*a,x		Branch if No Overflow	BCR,4	*a,x
BC	m	*a,x		Branch if Carry	BCS,8	*a,x
BNC	m	*a,x		Branch if No Carry	BCR,8	*a,x
BNCNO	m	*a,x		Branch if No Carry and No Overflow	BCR,12	*a,x
BWP	m	*a,x		Branch if Word Product	BCR,4	*a,x
BDP	m	*a,x		Branch if Doubleword Product	BCS,4	*a,x
BEV	m	*a,x	For Use After Fixed-Point Shift Instructions	Branch if Even (number of 1's shifted)	BCR,8	*a,x
BOD	m	*a,x		Branch if Odd (number of 1's shifted)	BCS,8	*a,x
BID	m	*a,x	For Use After Decimal Instructions	Branch if Illegal Decimal Digit	BCS,8	*a,x
BLD	m	*a,x		Branch if Legal Decimal Digit	BCR,8	*a,x
BSU	m	*a,x	For Use After Push Down Instructions	Branch if Stack Underflow	BCS,2	*a,x
BNSU	m	*a,x		Branch if No Stack Underflow	BCR,10	*a,x
BSE	m	*a,x		Branch if Stack Empty	BCS,1	*a,x
BSNE	m	*a,x		Branch if Stack Not Empty	BCR,1	*a,x
BSF	m	*a,x		Branch if Stack Full	BCS,4	*a,x
BSNF	m	*a,x		Branch if Stack Not Full	BCR,15	*a,x
BSO	m	*a,x		Branch is Stack Overflow	BCS,8	*a,x
BNSO	m	*a,x		Branch if No Stack Overflow	BCR,8	*a,x
BIOAR	m	*a,x	For Use After Input/Output Instructions	Branch if I/O Address Recognized	BCR,8	*a,x
BIOANR	m	*a,x		Branch if I/O Address Not Recognized	BCS,8	*a,x
BIODO	m	*a,x		Branch if I/O Device Operating	BCS,4	*a,x
BIODNO	m	*a,x		Branch if I/O Device Not Operating	BCR,4	*a,x
BIOSP	m	*a,x		Branch if I/O Start Possible	BCR,4	*a,x
BIOSNP	m	*a,x		Branch if I/O Start Not Possible	BCS,4	*a,x
BIOSS	m	*a,x		Branch if I/O Start Successful	BCR,4	*a,x
BIOSNS	m	*a,x		Branch if I/O Start Not Successful	BCS,4	*a,x
CALL						
CAL1	m,v	*a,x		Call 1		
CAL2	m,v	*a,x		Call 2		
CAL3	m,v	*a,x		Call 3		
CAL4	m,v	*a,x		Call 4		
CONTROL						
LPSD	m,r	*a,x		Load Program Status Doubleword		P
XPSD	m,r	*a,x		Exchange Program Status Doubleword		P
LRP	m	*a,x		Load Register Pointer		P
MMC	m,r	v		Move to Memory Control		P
LMAP	m,r			Load Map		7MP
LMAPRE	m,r			Load Map (Real Extended)		9MP
LPC	m,r			Load Program Control		7MP
LLOCKS	m,r			Load Locks		LP
WAIT	m	*a,x		Wait		P
RD	m,r	*a,x or (v,v),x		Read Direct		P
WD	m,r	*a,x or (v,v),x		Write Direct		P

Mnemonic	Syntax		Function	Required Options
CONTROL (cont.)				
NOP[t]	m	a,x	No Operation	
PZE	m	*a,x	Positive Zero	
INPUT/OUTPUT				
SIO	m,r	*a,x or (v,v),x or (v,v,v),x	Start Input/Output	P
HIO	m,r	*a,x or (v,v),x or (v,v,v),x	Halt Input/Output	P
TIO	m,r	*a,x or (v,v),x or (v,v,v),x	Test Input/Output	P
TDV	m,r	*a,x or (v,v),x or (v,v,v),x	Test Device	P
AIO	m,r	*a,x	Acknowledge Input/Output Interrupt	P
RIO	m,r	*a,x	Reset Input/Output	9P
POLP	m,r	*a,x	Poll Processor	9P
POLR	m,r	*a,x	Poll and Reset Processor	9P

[t] Generates an LCFI instruction with neither C nor F specified.

INDEX

Note: For each entry in this index, the number of the most significant page is listed first. Any pages thereafter are listed in numerical sequence.

Note: For each entry in this index, the number of the most significant page is listed first. Any pages thereafter are listed in numerical sequence.

E

F

G

H

I

J

K

L

Note: For each entry in this index, the number of the most significant page is listed first. Any pages thereafter are listed in numerical sequence.

M

N

O

P

Q

R

Note: For each entry in this index, the number of the most significant page is listed first. Any pages thereafter are listed in numerical sequence.

S

T

U

V

W

X

XEROX

Reader Comment Form

We would appreciate your comments and suggestions for improving this publication

Publication No.	Rev. Letter	Title	Current Date

How did you use this publication?

☐ Learning ☐ Installing ☐ Sales
☐ Reference ☐ Maintaining ☐ Operating

Is the material presented effectively?

☐ Fully Covered ☐ Well Illustrated ☐ Well organized ☐ Clear

What is your overall rating of this publication?

☐ Very Good ☐ Fair ☐ Very Poor
☐ Good ☐ Poor

What is your occupation?

Your other comments may be entered here. Please be specific and give page, column, and line number references where applicable. To report errors, please use the Xerox Software Improvement or Difficulty Report (1188) instead of this form.

Your name & Return Address

Thank You For Your Interest. (fold & fasten as shown on back, no postage needed if mailed in U.S.A.)

CUT ALONG LINE

FOLD ALONG LINE

PLEASE FOLD AND TAPE—
NOTE: U. S. Postal Service will not deliver stapled forms

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 59153 LOS ANGELES,CA 90045

POSTAGE WILL BE PAID BY ADDRESSEE

HONEYWELL INFORMATION SYSTEMS
5250 W. CENTURY BOULEVARD
LOS ANGELES, CA 90045

ATTN: PROGRAMMING PUBLICATIONS

FOLD ALONG LINE

Honeywell

Honeywell Information Systems
In the U.S.A.: 200 Smith Street, MS 486, Waltham, Massachusetts 02154
In Canada: 2025 Sheppard Avenue East, Willowdale, Ontario M2J 1W5
In Mexico: Avenida Nuevo Leon 250, Mexico 11, D.F.

21506, 5C878, Printed in U.S.A.

XG48, Rev. 0