SDS

Scientific Data Systems
A XEROX COMPANY

XDS SIGMA 5/7 MACRO-SYMBOL

Reference Manual

Price: $2.75

MACRO-SYMBOL REFERENCE MANUAL

for

XDS SIGMA 5/7 COMPUTERS

PRELIMINARY EDITION

90 15 78A

October 1969

XDS

Xerox Data Systems/701 South Aviation Boulevard/El Segundo, California 90245

Printed in U.S.A.

RELATED PUBLICATIONS

Title	Publication No.
XDS Sigma 5 Computer Reference Manual	90 09 59
XDS Sigma 7 Computer Reference Manual	90 09 50
XDS Sigma 5/7 Symbol/Meta-Symbol Reference Manual	90 09 52
XDS Sigma 5/7 Real-Time Batch Monitor (RBM-2) Reference Manual	90 15 81

NOTICE

The specifications of the software system described in this publication are subject to change without notice. The availability or performance of some features may depend on a specific configuration of equipment such as additional tape units or larger memory. Customers should consult their XDS sales representative for details.

CONTENTS

APPENDIXES

ILLUSTRATIONS

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. None of the earlier programming languages 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".

Other programming languages are said to be "machine-oriented" which, consequently, have numerous restrictions and unduly complex syntax rules. Machine-oriented languages developed because the syntax rules for a given computer were strongly influenced by the hardware characteristics of that machine. The syntax of any symbolic programming language consists of the set of rules governing its sentence (that is, statement) structure; and the vocabulary consists of the permissible names, literals, operators, and other symbols that may be used to express a symbolic program.

Sigma 5/7 Macro-Symbol is neither a problem-oriented nor a machine-oriented language, and therefore there are fewer rules to learn and programming flexibility is enhanced. Macro-Symbol is a superset of the Sigma 5/7 Symbol language and a subset of the Sigma 5/7 Meta-Symbol language.

Chapters 1 through 6 of this manual describe the basic features of the Sigma 5/7 Macro-Symbol language; Chapter 7 explains the Macro-Symbol assembly listing; and Chapter 8 gives the various Macro-Symbol operating procedures under Real-Time Batch Monitor control. The Macro-Symbol directives are summarized in Appendix A; the compatibility of Macro-Symbol with Symbol and Meta-Symbol is discussed in Appendix B; and the Sigma 5 and Sigma 7 Computer instructions recognized by Macro-Symbol are given in Appendixes C and D, respectively.

1. INTRODUCTION TO MACRO-SYMBOL

PROGRAMMING FEATURES

The following list summarizes Macro-Symbol's more important features for the programmer:

- The argument field can contain both arithmetic and Boolean (logical) expressions, using constant or variable quantities.

- The DO directive allows 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.

- The call line and its individual parameters can be tested both arithmetically and logically.

- Nested procedures are used; that is, one procedure may call another.

- Complete use of arithmetic and Boolean operators in procedures is permitted.

MACRO-SYMBOL PASSES

Macro-Symbol is a two-pass assembler that runs under control of the Real-Time Batch Monitor.

PASS 1

Pass 1 reads the input program (which may be symbolic, compressed, or compressed with symbolic corrections), builds the symbol table, allocates space for each statement, generates the intermediate file (which is a slightly modified copy of the input program that will be read by Pass 2), and optionally outputs a new compressed deck.

PASS 2

Pass 2 is the final assembly pass that generates the object code. It reads the intermediate file, and using the symbol table produced by Pass 1, provides the correct addresses for all symbols. During this pass, 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 7.

[†] XDS 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. MACRO-SYMBOL 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

The Macro-Symbol character set is shown in Table 1.

Table 1. Macro-Symbol Character Set

Alphabetic:	A through Z, and $, @, #, :, ␣ (break character - prints as "underscore")	
Numeric:	0 through 9	
Special Characters:		Blank
	+	Add (or positive value)
	-	Subtract (or negative value)
	*	Multiply, indirect addressing prefix, or comments line indicator
	/	Divide
	.	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 or introduces a literal
	<=	Less Than or Equal
	>=	Greater Than or Equal
	¬=	Not Equal
	;	Continuation Code
	**	Binary Shift

The colon is an alphabetic character used in internal symbols of standard XDS 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 bv XDS 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 8 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 characters $ and $$ may be used in the argument field of a statement to represent the current value of the execution and load location counters, respectively (see Chapter 3); these characters must not be used as label field entries.

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.

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, GEN, LOC, ORG, RES, SET, TEXT, TEXTC, and USECT.

Often the programmer may 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 is considered to be an address or absolute term, depending on the value to which it is equated.

REDEFINING SYMBOLS

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

```
SYM    SET    15     SYM is assigned to 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 com-
                     pleted; i.e., 3 not 15.
```

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, and 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 two 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
   ```

   The maximum value of a decimal integer constant is limited to that which can be contained in two words (64 bits).

2. 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'7AF'      "7AF" is a hexa-
                                decimal constant
                                representing the
                                decimal value 1967
   ```

There are seven types of general constants:

Code	Type
C	Character string constant
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.

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 5 provide positioning information pertaining to the character strings used with these directives. In all other usages, character strings must not contain more than 16 characters. If the string contains less than 16 characters, the characters are right-justified and a null EBCDIC character(s) fills out the word.

Note: If any constant string enclosed by single quotation marks appears in an object program without one of the type codes listed above, it is assumed to be a character string constant and is processed as if type code C had preceded the string.

[†] A table of Extended Binary-Coded Decimal Interchange Codes can be found in the Sigma 5 and Sigma 7 Computer Reference Manuals (90 09 59 and 90 09 50).

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 8-bit mask would consist of two hexadecimal digits. The maximum value of a hexadecimal constant is limited to that which can be contained in two words (64 bits).

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

Information concerning hexadecimal arithmetic and hexadecimal-to-decimal conversions is included in the Sigma 5 and Sigma 7 Computer Reference Manuals.

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 maximum value is limited to that which can be contained in two words (64 bits). By implication, the size of the constant in binary digits is 3 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 Macro-Symbol is of the binary-coded decimal form required for Sigma 7 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:

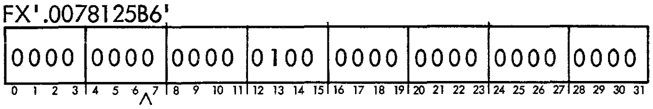
   the letter E followed optionally by an algebraic sign, followed by one or two decimal digits.

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.

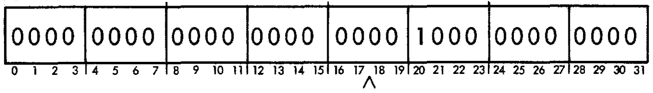
Parts 3 and 4 may occur in any relative order:

FX'.0078125B6'



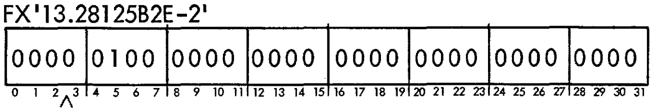
0000	0000	0000	0100	0000	0000	0000	0000
0 1 2 3	4 5 6 ∧ 7	8 9 10 11	12 13 14 15	16 17 18 19	20 21 22 23	24 25 26 27	28 29 30 31

FX'1.25E-1B17'



0000	0000	0000	0000	0000	1000	0000	0000
0 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

FX'13.28125B2E-2'



0000	0100	0000	0000	0000	0000	0000	0000
0 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

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

[†] A plus sign is a 4-bit code of the form 1100. A minus sign is a 4-bit code of the form 1101.

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 7 bits, and the second field requires 9 bits.

The number FX'3.75B4' is to be stored in the first field. The binary equivalent of this number is 11 $_\wedge$ 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	
0	1	2	3	4	5	6		Bit positions
0	0	0	1	1	1	1		

∧

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

Field 1							Field 2									
0	1	2	3	4	5	6	0	1	2	3	4	5	6	7	8	Bit positions
0	0	0	1	1	1	1	0	0	1	0	0	0	0	0	0	

∧

In generating the second number, Macro-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 Macro-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 letters 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'

3	F	1	6	8	7	2	B
0 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

Refer to the XDS Sigma 5 Computer Reference Manual or the Sigma 7 Computer Reference Manual for an explanation of floating-point format.

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'

4	2	1	D	E	0	3	1
0 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

0	C	0	E	6	E	9	4
0 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

Refer to the XDS Sigma 5 Computer Reference Manual or the Sigma 7 Computer Reference Manual for an explanation of floating-point format.

The value of a floating-point long constant is limited to that which can be stored in two words (64 bits).

LITERALS

A literal is a constant or symbol enclosed by parentheses and preceded by the letter L:

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

or a constant or symbol preceded by an equal sign:

= -185	decimal value -185
= X'5DF'	hexadecimal value 5DF
= AB	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 must not be used as a term in a multitermed expression; however, either literal form may be used in an addressing function expression. For example,

```
BA  (HA(L(S + 1)))
```

is valid.

A literal preceded by an asterisk specifies indirect addressing: (* = 10).

When a literal appears in a statement, Macro-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 unless the programmer specifies 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 symbols.

During an assembly Macro-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 Macro-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 current 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

The Macro-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 Macro-Symbol.

Table 2. Macro-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
+	4	Integer Add
-	4	Integer Subtract
<	3	Less Than
>	3	Greater Than
<=	3	Less Than or Equal
>=	3	Greater Than or Equal
=	3	Equal
¬=	3	Not Equal
&	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.

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.

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. Arithmetic operations are performed on the entire word for all expressions; Macro-Symbol does not restrict arithmetic operation to the low-order 19 bits for expressions involving addresses.

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 (*), 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.

In Macro-Symbol all arithmetic and logical operations in expressions are carried out in single precision (32 bits).

Constant expressions may be formed with any of the operators in Table 2 and operands that are constants or symbols previously equated to constants.

Operands that represent addresses may be combined only with the plus (+) and minus (-) operators to form address expressions subject to the following restrictions:

1. Two address expressions may not be added together.

2. An address expression may be subtracted from another address expression only if both address expressions are of the same resolution and section and if neither address expression is external or a local forward reference. If all of these requirements are met, the result is a constant expression.

3. An address expression may be added to a constant expression, yielding an address expression.

The address functions (BA, HA, WA, and DA) may be applied to expressions without limitation as long as the expression is not an address expression with a local forward or external reference. When an address expression is a local forward or external reference, the expression is restricted to the following form:

$af_1\ (af_2(\pm\ ax)\ \pm\ cx)$

where

af_i are address functions.

ax is a local forward or external address expression.

cx is a constant expression.

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"
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
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 32-bit value (all 1's)

The logical operators & (AND), | (OR), and || (Exclusive OR) perform as follows:

AND

First Operand:	0011
Second Operand:	0101
Result of & Operation:	0001

OR

First Operand:	0011
Second Operand:	0101
Result of | Operation:	0111

Exclusive OR

First Operand:	0011
Second Operand:	0101
Result of || 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, a program statement, or a symbolic line.

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

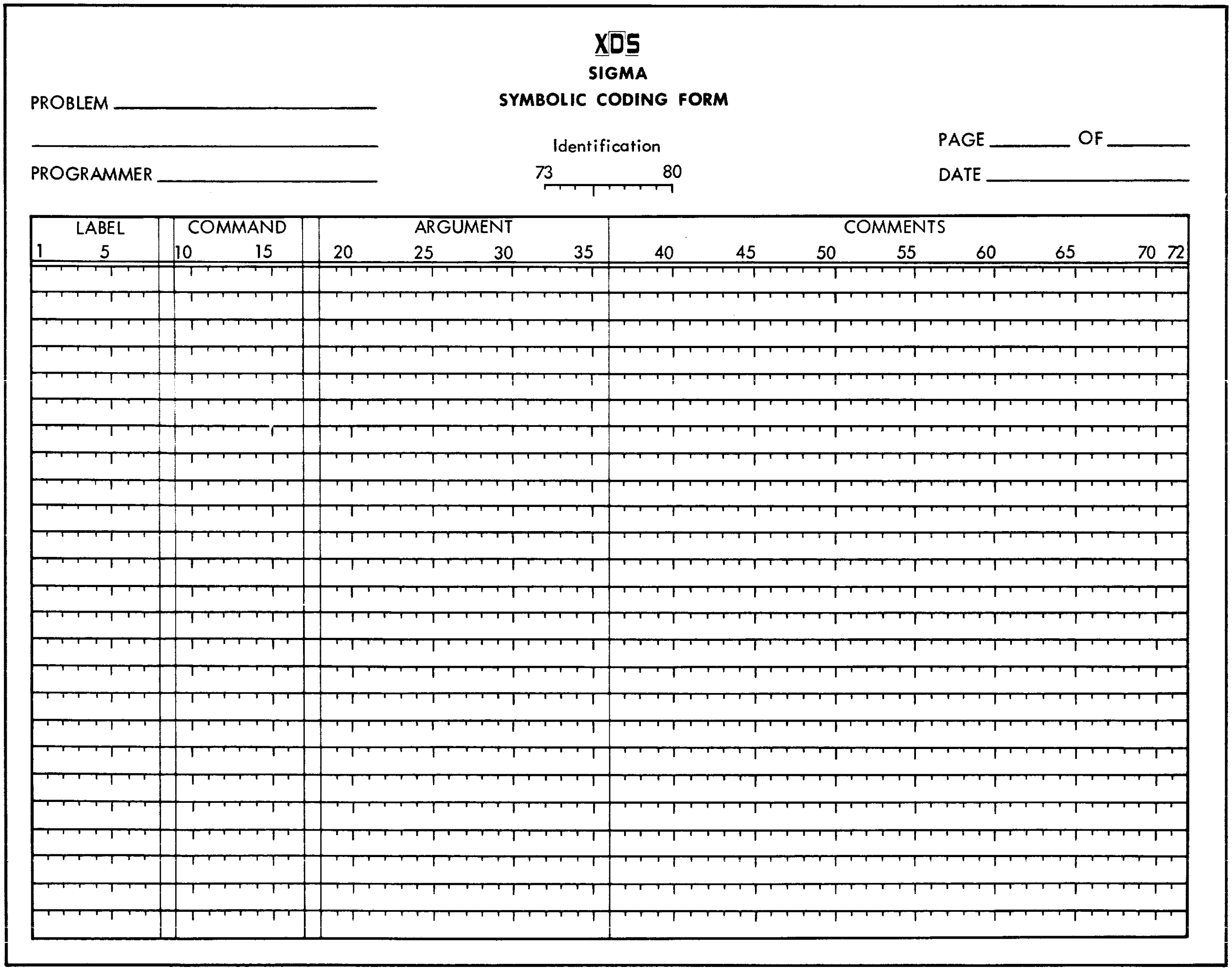
FIELDS

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.

Macro-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 as follows:

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 character on paper tape input.
3. One or more blanks at the beginning of a line specifies there is no label field entry.



XDS
SIGMA
SYMBOLIC CODING FORM

PROBLEM ____________

____

PROGRAMMER ____________

Identification
73 80

PAGE ______ OF ______
DATE ____________

LABEL	COMMAND	ARGUMENT	COMMENTS
1 5	10 15	20 25 30 35	40 45 50 55 60 65 70 72

Figure 1. XDS Sigma Symbolic Coding Form

4. The label field entry, when present, must begin in column 1.

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 omitted.

6. The argument field begins with the first nonblank column following the command field. An argument field is designated as 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 fields: a label field entry, a command field entry, an argument field entry, and a comments field entry.

A label entry is a symbol 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 of a statement may have the same configuration as an instruction, a directive, or an intrinsic function without conflict, since Macro-Symbol is able to distinguish through context which usage is intended. For example, the mnemonic code for the Load Word command is LW; LW may also appear in the label field of a statement without conflicting with the command LW in the command field.

The name of any intrinsic function that requires parentheses (BA, DA, HA, L, NUM, 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 2. Label Field Entry

```
  LABEL     COMMAND        ARGUMENT
1   5    10   15   20   25   30   35
PAY␣RATE
A
A3
COST@
```

The command entry is a mnemonic code representing a machine instruction, a procedure name, or an assembler function to be performed. A command entry is required in every active statement. Thus, if a statement is entirely blank following the label field, the assembler declares the statement in error, generates a word of all zeros in the object program, and flags the statement in the assembly listing. The same thing happens if the command entry is not an acceptable instruction, procedure, or directive. The mnemonic codes for machine instructions and the assembler directives recognized by Macro-Symbol are listed in Appendixes A, C, and D.

Example 3. Command Field Entry

```
  LABEL     COMMAND        ARGUMENT
1   5    10   15   20   25   30   35
         LW,5
 LW,5
       LW,5
                  LW,5
ALPHA    LW,5
BETA LW,5
B1         LW,5
LOOP         LW,5
```

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 Macro-Symbol to perform the designated operation.

Example 4. Argument Field Entry

```
          COMMAND        ARGUMENT
         10   15   20   25   30   35   40
         LW,5     ALPHA
         AW,2     B12
         LI,4       85
         LW,1 COUNT
         NOP                BLANK ARGUMENT
              LW,5  ANY
```

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 7 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 two 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; that is, a semicolon at the end of a comment line is treated as regular punctuation rather than as a continuation indicator.

4. Comment lines may be placed between continuation lines.

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 becomes "defined" by appearing as a label entry on machine language instruction statements and certain directives. "Defined" means that it is assigned a value. The definition, assigned to the symbol by the assembler, depends on assembly conditions when the symbol is encountered, the contents of the command field, and the current contents of the execution location counter.

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

The EQU, SET, CNAME, and COM directives require a label entry. A label entry is optional for the following directives: ASECT, CSECT, USECT, DATA, DO, DO1, DSECT, END, GEN, LOC, ORG, RES, TEXT, and TEXTC. 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.

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 use of EQU or SET. A symbol used in the label field of such a directive is assigned the value specified in the argument field.

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

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. A forward reference must not be used as a term in a multitermed expression, with one exception. The exception is that a forward reference may have a constant addend, so that the reference is of the form: reference ± exp or exp + reference. The term exp either must be a positive integer value or an expression that resolves to a positive integer value. Examples of such usage would be

```
          LW,4     HERE-2
           .
           .
HERE      EQU      $
           .
           .
FLAG      EQU      $
           .
           .
          LW,4     FLAG + 4 + SUM
           .
           .
SUM
```

Forward references may be used in any machine language instruction and in the argument field of the following directives: COM, DATA, DEF, GEN, GOTO, LOCAL, REF, and SREF.

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. An external reference must not be used as a term in a multitermed expression, with one exception. The exception is that the external reference may have a constant addend of the same kind and conforming to the same restrictions previously explained under "Forward References".

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.

Any computer instruction may contain an external reference; however, external references are not allowed in any Macro-Symbol directives except REF, SREF, GEN, DATA, EQU, and END.

CLASSIFICATION OF SYMBOLS

Symbols may be classified either as 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, external reference. Some types require additional information. For example, relocatable addresses, which are entered as 19-bit offsets from 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. Macro-Symbol enters the symbol in the table, but does not assign it a value. When the symbol is defined later in the program, Macro-Symbol assigns it a value and designates the appropriate attributes.

ABSOLUTE AND RELOCATABLE VALUES

The value of a symbol or expression may be absolute or relocatable. An absolute value, which is assigned at assembly time, is the same value that will be used by the program at execution time. A relocatable value, on the other hand, may be altered by the loader at execution time.

SYMBOL VALUES

A symbol is assigned an absolute value by one of the following methods:

1. By equating the symbol to an absolute numeric quantity:

   ```
   SUM        EQU    2
   ```

   SUM is assigned the absolute value 2.

2. By equating the symbol to an absolute symbol:

   ```
   A          EQU    -10
   ANSWER     EQU    A
   ```

   ANSWER is assigned the absolute value -10.

3. By using the symbol as a label entry in an absolute program or program section (see Chapter 3).

The value of an absolute symbol does not change, even if it is part of a relocatable program (a program that can be executed anywhere in memory).

A symbol has a relocatable value unless declared absolute as described above. The value of a relocatable symbol may be altered by the loader when the symbol is a part of a relocatable program.

EXPRESSION VALUES

An absolute expression may consist of either a single absolute term or a combination of absolute terms. An absolute term is a symbol defined in an absolute program section or a hexadecimal, octal, or decimal integer. Note that D, C, FX, FS, and FL constant types in a multiple-term expression produce unpredictable results.

A relocatable expression may consist of either a single relocatable term or a combination of relocatable terms.

The mode of an expression combining absolute terms with relocatable terms is determined as shown in Example 5.

When the assembler evaluates an expression, it determines whether the expression value is relocatable or absolute.

Example 5. Expressions Using + and - Operators

Assume R1, R2, and R3 are relocatable terms and A1 and A2 are absolute terms.

Expression:	R1±A1	Legal, relocatable
Expression:	R1-R2-R3	Legal, relocatable
Expression:	R1-R2+A1	Legal, absolute
Expression:	R1-R2+R3-A1+A2	Legal, relocatable
Expression:	R1+R2	Illegal, diagnostic error
Expression:	R1+R2-R3	Illegal, diagnostic error
Expression:	R1+(R2-R3)	Legal, relocatable
Expression:	A1±A2	Legal, absolute

3. MACRO-SYMBOL ADDRESSING

Sigma 5/7 computer addressing techniques require a register designation and an argument address which 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 Chapters 5 and 6.

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 6. $, $$ Functions

```
        ⋮
A       EQU     $           Equate A to the current value
                            of the execution location
        ⋮                   counter.
Z       EQU     $$          Equate Z to the current value
        ⋮                   of the load location counter.
TEST    BCS,3   $+2         Branch 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 7. 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 8. 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 9. 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 10. 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.
.
.
```

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 11.

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)

Macro-Symbol restricts the number of nested resolution functions and addends that may be applied to a forward or external reference with an addend. Only one such change of address resolution may be made. For example, the following usage of a forward reference is permissible:

BA(2+WA(reference))

while the following usage cannot be processed by Macro-Symbol and will be flagged as an error:

WA(BA(2 + WA(reference)))

Similarly, once a forward or external reference has been given an addend followed by a change of resolution, it may not be given another addend. For example, the following forward reference usage will again be flagged as an error:

BA(2 + WA(reference)) + 1

Example 11. Address Resolution

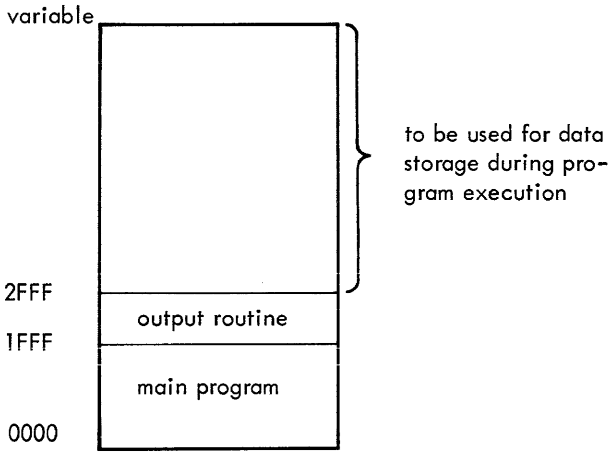
```
            Generated
Location    Code
                                CSECT
00000                           ORG        0                Sets value of location counters to zero with
                                                            word resolution.
00000       FFFB        A       GEN,16     -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               Define 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

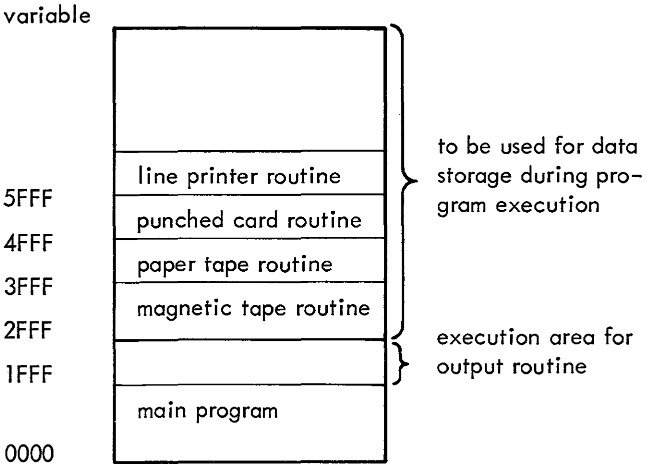
A location counter is a memory cell the assembler uses to record the storage location it assigned last and, thus, what location it should assign next. Each program has two location counters associated with it during assembly: the load location counter (referenced symbolically as $$) and the execution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

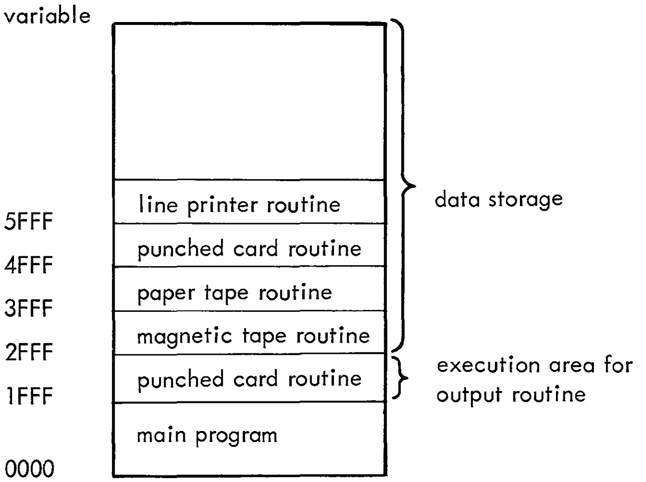
In the "normal" case both counters are stepped together as each instruction is assembled, and both contain the same location value. However, the ORG and LOC directives make it possible to set the two counters to different initial values to handle a variety of programming situations. The load location 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



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:

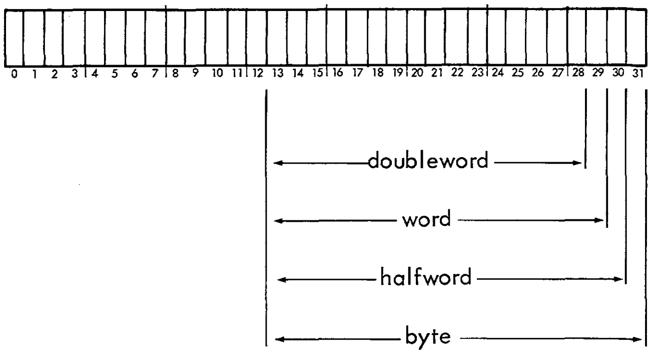


When the main program has determined which output routine is to be used, during program execution, it moves the routine to the execution area. No address modification to the routine is required since all routines were originally assembled to be executed in that area. If the punched card output routine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

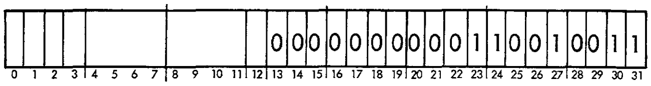


The user should not assume from this example that the execution location counter must be controlled in the manner indicated in order for a program to be relocated. By properly controlling the loader and furnishing it with a "relocation bias", any Macro-Symbol program, unless the programmer specifies 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	Hex. 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, Macro-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]	ORG[,n]	location

where

label may be any valid symbol. 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 may be a constant, symbol, or 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 may be relocatable or an evaluatable expression resulting in a positive integer value.

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 $$ are encountered, the values they represent are expressed according to the currently applicable intrinsic resolution.

LOC Set Program Execution

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

label	command	argument
[label]	LOC [,n]	location

where

label is any valid symbol. 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 may be a constant, symbol, or 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 may be relocatable or an evaluatable expression resulting in a positive integer value.

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

Example 12. ORG Directive

```
         .
         .
         .
AA       ORG      8          This directive sets the location counters to 8 and assigns that location 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 13. 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.
         .
         .
         .
```

Example 14. 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 origin 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 and ≤ 32. Halfword addresses are multiples of 2 bytes, fullword addresses are multiples of 4 bytes, and doubleword addresses are multiples of 8 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.

Example 15. 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]	RES[,n]	u

where

label is any valid symbol. 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 expression designating the size in bytes of the units to be reserved. The value of n must be a positive integer. Use of n is optional; if omitted, its value is assumed to be four bytes.

u is an evaluatable expression designating the number of units to be reserved. The value of u may be a positive or negative integer.

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

Example 16. 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; that is, 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[†] 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.

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

The program section directives have the following form:

label	command	argument
[label]	ASECT	
[label]	CSECT	[exp]
[label]	DSECT	[exp]
[label]	USECT	name

where

label is any valid symbol. The label is 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 and DSECT the label value becomes relocatable zero in the appropriate program section. For USECT the label value is the saved value of the execution location counter for the specified section. The label on ASECT, CSECT, 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 usually set up so that they can be referred to by other programs.

exp is an expression whose value must be from 0 to 3. This value, applicable only to CSECT and DSECT, 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 "exp" is optional. When it is omitted, the assembler assumes the value 0 for the entry. The expression "exp" may not contain an external reference.

name is a 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 1 to 127 relocatable control sections, which Macro-Symbol numbers sequentially. At the beginning of each assembly, Macro-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 or DSECT directive.

The execution of a CSECT or DSECT directive always opens a new section. Therefore, if control section 1 has been opened by generating values in, or referencing or manipulating

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

the initial location counters, the first CSECT or DSECT 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 contain 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, Macro-Symbol does the following when a new section is declared (ASECT, CSECT, or DSECT):

1. Saves the current value of the execution location counter ($) in the SAVED $ TABLE, and
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 Macro-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.

Example 17. 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, Macro-Symbol saves the value of the execution location counter in the SAVED $ TABLE. The value of the load location counter is not saved. Macro-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. Macro-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 Macro-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.

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 indicates the resolution (BA, HA, WA, DA).

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

ADDR specifies that the value is an address.

VALUE is the assigned symbol value.

3. The USECT directive, which specifies a previously declared section that Macro-Symbol is to use in assembling generative statements.

Example 18. USECT Directive

Current Location Counters				Program			SAVED $			SAVED MAX. $$	
$	Section	$$	Section				ABS	CS1	CS2	CS1	CS2
0	CS1	0	CS1		CSECT			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, Macro-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.

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 19. 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			:						

$	Section	$$	Section				ABS	CS1	CS2	CS1	CS2
					⋮						
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, Macro-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 20. 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, Macro-Symbol sets both counters to the value of the saved execution location counter for CS1 (1100). The ASECT directive causes Macro-Symbol to save the value of the execution location counter for CS1 and to replace the SAVED MAX. $$ value (100) with 1200.

Example 21. 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, Macro-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 CS1 (300).

DUMMY SECTIONS

In any load module dummy sections of the same name must be the same size and have the same memory protection. 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:

```
SUBR1   .      | SUBR2   .      | SUBR3   .
        :      |         :      |         :
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 Macro-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 22.

Example 22. Program Sections and Literals

Example 22a:

```
AREA    CSECT
        :  }           Literals declared
BAY     CSECT
        :  }           Literals declared
        END            Literals generated as
                       part of section BAY.
```

Example 22b:

```
GATE    CSECT
        :  }           Literals declared
        ASECT
        ORG     100
        END            Literals generated be-
                       ginning in absolute lo-
                       cation 100.
```

Example 22c:

```
REAL    CSECT
        :  }           Literals declared
LAST    RES            0
LOOP    CSECT
        :  }           Literals declared
        USECT   REAL
        END            Literals generated as part
                       of section REAL immedi-
                       ately following the loca-
                       tion assigned to LAST.
```

Example 22d:

```
NOW     DSECT
        :  }           Literals declared
HERE    RES     25
        :  }
        ORG     HERE
        END            Literals generated as part
                       of section NOW, begin-
                       ning at location HERE.
```

4. INSTRUCTIONS

Sigma 5 and Sigma 7 computer instructions may be written in symbolic code and combined with other assembly language elements to form symbolic instruction statements.

The four fields of a symbolic instruction statement are

Field	Contents
label	Any valid symbol. Use of a label entry is optional; when present, the label symbol may also appear in the argument field of other instructions and directives.
command	Any mnemonic operation code listed in Appendixes C and D. The entry may consist of several subfields, the first of which is always the operation mnemonic code. The subsequent subfields may be a register expression, a count expression, or a value expression, depending on the requirements of the particular instruction.
argument	One or more subfields such as an address expression, an indirect addressing designator, or a displacement expression, depending on the requirements of the specific instruction.
comments	Any remark explaining the specific purpose of the statement of the overall function of the program.

Machine language instructions are automatically aligned on word boundaries by the assembler. The address expressions in the argument fields of these instructions are assembled according to the dictates of the specific instruction and the dictates of any addressing functions in the argument. (See Example 13 in Chapter 3.)

Appendixes C and D contain a summary of machine language instruction mnemonics specifying the requirements of each field. The XDS Sigma 5 and Sigma 7 Computer Reference Manuals (90 09 59 and 90 09 50) contain complete decriptions of these instructions.

Example 23. Sigma 5 and Sigma 7 Instructions

label	command	argument	comments
L1	LW,4	HOLD	Load Word from location HOLD into register 4.
L2	LW,4	HOLD,2	Indexed Load Word instruction using register 2 as an index register.
L3	LW,4	*HOLD,2	A Load Word Instruction that specifies both indexing and indirect addressing.
L4	LI,3	X'F3E'	Load the hexadecimal value F3E from the argument field into register 3.
L5	AW,12	L(32)	Add the decimal value 32 to the contents of register 12.
L6	B	LOOP	Branch unconditionally to location LOOP.

Although the general registers and index registers are specified only by digits in these examples, they may be arithmetic expressions whose values are 0-15 for general registers and 0-7 for index registers. They also may be symbols that have been assigned values within that range.

5. MACRO-SYMBOL 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 four directives: CNAME, COM, EQU, and SET. EQU and SET equate the symbol in the label field to the value of the expression in the argument field. The label field entries for COM and CNAME identify the generated command procedure. The location counters are not altered by these directives.

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 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, USECT, and DO1 identify the first word of the area affected by the directive.

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 on the following directives will be ignored unless it is the target label of a GOTO search: BOUND, DEF, ELSE, FIN, GOTO, LOCAL, PAGE, PEND, PROC, REF, SPACE, SREF, SYSTEM, TITLE.

A label for the DO directive is handled in a special manner that will be 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 and LOCAL 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 Macro-Symbol language includes these directives:

Assembly Control

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

Symbol Manipulation

EQU
SET
LOCAL
DEF
REF
SREF

Data Generation

GEN
COM
DATA
TEXT
TEXTC

Listing Control

PAGE
SPACE
TITLE

Procedure Control

CNAME[††]
PROC[††]
PEND[††]

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

ASSEMBLY CONTROL

SYSTEM Define System

SYSTEM directs the assembler to define the subset of computer instructions that are to be valid during this portion of the assembly. This directive has the form

label	command	argument
	SYSTEM	name

where "name" identifies the instruction set, and must be one of the following:

Name	Instruction Set
SIG7	Basic Sigma 7
SIG7F	Sigma 7 with Floating-Point Option
SIG7D	Sigma 7 with Decimal Option

[†] Discussed in Chapter 3.

[††] Discussed in Chapter 6.

Name	Instruction Set
SIG7P	Sigma 7 with Privileged Instructions
SIG7FD	Sigma 7 with Floating-Point and Decimal Option
SIG7FP	Sigma 7 with Floating-Point Option and Privileged Instructions
SIG7DP	Sigma 7 with Decimal Option and Privileged Instructions
SIG7FDP	Sigma 7 with Floating-Point, Decimal Option, and Privileged Instructions
SIG5	Basic Sigma 5
SIG5F	Sigma 5 with Floating-Point Option
SIG5P	Sigma 5 with Privileged Instructions
SIG5FP	Sigma 5 with Floating-Point Option and Privileged Instructions

None of the instruction sets omits any of the intrinsic commands or functions. Macro-Symbol assumes a default specification of SIG7FDP when SYSTEM is not specified.

END End Assembly

The END directive terminates the assembly of the object program. It has the form

label	command	argument
[label]	END	[exp]

where

label is any valid symbol. When present, the label is assigned (i.e., associated with) the location immediately following the last location in the literal table.

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

As explained under "Program Sections and Literals" at the end of Chapter 3, Macro-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 22). If the END directive is labeled, the label is associated with the first location immediately following the literal table. Thus, in Example 22c, 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.

DO1 Iteration Control

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

label	command	argument
[label]	DO1	exp

where

label is any valid label. Use of a label is optional. When present, it is defined as the current value of the execution location counter and identifies the first byte generated as a result of the DO1 iteration.

exp is an evaluatable expression resulting in a positive integer 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 in the DO1 directive is not evaluatable, Macro-Symbol produces an error notification, and processes the DO1 directive as if the expression had been evaluated as zero.

Example 24. 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
      .
      .
      .
```

GOTO Conditional Branch

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

label	command	argument
	GOTO[,k]	label$_1$[,label$_2$, ..., label$_n$]

where

k is an absolute, evaluatable expression whose value refers to the kth label in the argument field. If k is omitted, 1 is assumed.

label$_i$ are forward references.

A GOTO statement is processed at the time it is encountered during the assembly. Macro-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, Macro-Symbol resumes assembly at the line immediately following the GOTO directive.

Although a label on BOUND, DEF, GOTO, LOCAL, PAGE, REF, SREF, and SYSTEM is normally ignored by the assembler, it will be recognized if it is the target label of a GOTO search.

While Macro-Symbol is searching for the statement whose label corresponds to the kth label, it operates in a skipping mode during which it ignores all machine language instructions and directives except END and LOCAL. Skipped statements are produced on the assembly listing in symbolic form, preceded by an *S*.

If Macro-Symbol encounters the END directive before it finds the target label of a GOTO search, it produces an error notification and terminates the assembly. If a LOCAL directive is encountered while searching for a local label, then an error notification is produced and the search is terminated.

When Macro-Symbol encounters the first of a logical pair of directives[†] 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, Macro-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 or a DO/FIN loop. Furthermore, it is not permissible to write a GOTO directive that might branch out of a procedure definition. If such a case occurred, Macro-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 25. GOTO Directive

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

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

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]	DO	exp
	ELSE	
	FIN	

where

label is any valid symbol. Use of a label is optional. When present, it is initially assigned the value zero and incremented by one each successive time through the loop.

exp is an evaluatable expression that produces an integer. This integer represents the count of the number of time the DO-loop is to be processed.

Figure 2 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 a label is present on the DO line, sets its 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 a label is present on the DO line, sets it 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

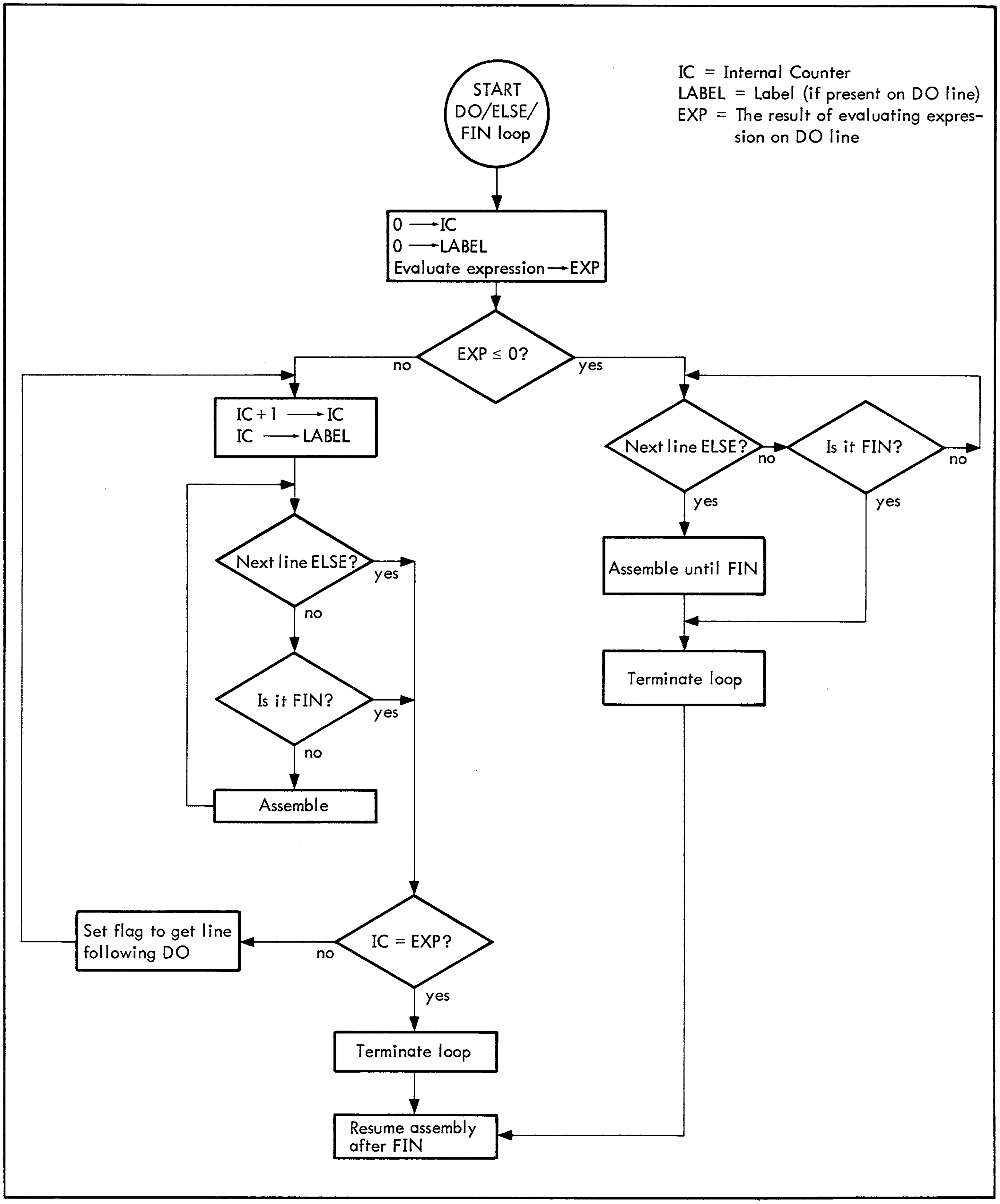


Figure 2. Flow Chart of DO/ELSE/FIN Loop

count. (The repeat count must not exceed one if the number of lines to be repeated exceeds one and the loop is not within a procedure.)

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     }
         :      }  block 1
         ELSE   }
         :      }  block 2
         FIN    }

Form 2.  DO     }
         :      }  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 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, Macro-Symbol sets the label (if present) to the value zero, produces an error notification, and processes the DO directive as if the expression had been evaluated as zero.

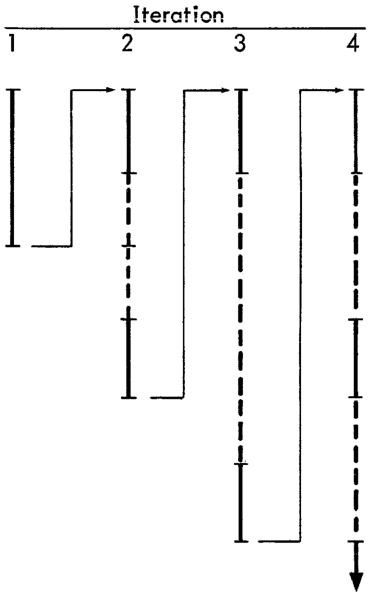
An iteration block may contain other iteration blocks but they must not overlap.

The label for the DO directive is redefinable and its 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, 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 must be completed on the same program level on which they originate. That is, if a DO 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 directive occurs within a procedure definition, the ELSE and/or FIN for that directive must also be within the definition.

Example 26. 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.



```
      :
I     DO          4
      :
      GOTO,I-1    S,T,S
      :
      ELSE
      :
S
      :
      ELSE
      :
T
      :
      FIN
      :
```

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. 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. Thus, the expression for GOTO has the value 1 so Macro-Symbol will skip until it finds a statement labeled S. Starting with S, Macro-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. Thus, the expression for GOTO has the value 2 so Macro-Symbol will skip until it finds a statement labeled T. Starting at T, Macro-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. Thus, the expression for GOTO has the value 3 so Macro-Symbol will skip until it finds a statement labeled S. Starting at S, Macro-Symbol assembles code until it encounters the ELSE directive which terminates the fourth – and last – iteration of the loop. Then, Macro-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 expression in the argument field. This directive has the form

label	command	argument
label	EQU	exp

where

label is a valid symbol.

exp is an evaluatable expression whose value is to be associated with "label". The expression may not contain forward references. Only previously defined or external reference ± addend expressions are legal. The mode (absolute or relocatable) of "exp" is assigned to label.

When EQU is processed by Macro-Symbol, "label" is defined as the value of "exp". For example, the statement

```
VALUE     EQU     8+5
```

assigns the absolute value 13 to VALUE, and

```
ALPHA     EQU     $-10
```

assigns the relocatable value $-10 to ALPHA.

A symbol defined with an EQU cannot be redefined:

```
A     EQU     X'F'     Legal
       :
A     EQU     O'2'     Illegal because A has already
                       been equated to a value.
```

Example 27. EQU Directive

```
A     EQU     10       A=10
       :
B     EQU     A+4      B=14
       :
      LW,A    DELTA    Loads the contents of loca-
                       tion DELTA into register 10.
```

SET Set a Value

The SET directive, like EQU, enables the user to define a symbol by assigning to it the attributes of the expression in the argument field. SET has the form

label	command	argument
label	SET	exp

where "label" and "exp" are the same as for EQU, except that the expression may not be external reference ± addend.

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.

Example 28. 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 nonlocal 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	$\text{symbol}_1,\ldots,\text{symbol}_n$

where the "symbol_i" are declared to be local to the current region.

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 or subtraction. This does not limit the use of defined local symbols in 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 30. (PROC and PEND define the beginning and end, respectively, of a procedure definition. See Chapter 6.)

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 29. 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 30. 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 non-
                            local 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 initi-
                            ates a new region.
        :
Z       EQU      A=X'E1'    Equates Z to the value 1
                            because the nonlocal
                            symbol A has the hexa-
                            decimal value E1.
        :
```

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	$\text{symbol}_1[,\ldots,\text{symbol}_n]$

where the "symbol_i" may be any symbolic labels that are defined within the current program.

DEF directives must precede any statement that causes code to be generated; this includes all machine language instructions and directives BOUND, DATA, GEN, TEXT, and TEXTC. DEF directives must precede all REF and SREF directives.

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.

Example 31. DEF Directive

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

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

REF Declare External References

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

label	command	argument
	REF	symbol$_1$ [, symbol$_2$, . . . , symbol$_n$]

where "symbol$_i$" may be any labels that are to be satisfied at load time by other programs.

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

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.

REF directives must precede any statements that cause code to be generated; this includes all machine language instructions and directives BOUND, DATA, GEN, TEXT, and TEXTC. REF directives must not precede DEF directives.

Example 32. 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 33. 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 machine'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	symbol$_1$[,symbol$_2$, . . . , symbol$_n$]

where the "symbol$_i$" have the same meaning as for REF.

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

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, then 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 symbols without considering them to be unsatisfied external references.

Like REF, SREF directives must precede any statements that cause code to be generated and must follow all DEF directives.

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]	GEN, field list	value list

where

label is any valid symbol. Use of a label is optional. When present, it 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 words generated.

field list is a list of evaluatable expressions that define the number of bits comprising each field. The sum of the field sizes must be a positive integer value that is a multiple of eight and is less than or equal to 128.

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 34).

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 34. GEN Directive

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

Example 35. GEN Directive

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

Example 36. 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 contains
                                     the hexadecimal value 08; the second contains the hexadecimal value 09;
                                     and the third byte contains the hexadecimal value 0A.
           .
           .
           .
         LW,5         L(2)           Load register 5 with the literal value 2.

           .
           .
           .
         LB,3         LAB,5          Load 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 37. 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
name	COM,field list	value list

where

name is any valid symbol and identifies the command being defined. The "name" must not be a local symbol nor the same as a Sigma 5/7 machine instruction or Macro-Symbol 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.

value list is a list of constants 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.

In Macro-Symbol, an asterisk preceding a field list element on the COM definition line specifies that the absence of a corresponding parameter on the COM reference line is to be flagged as an error. See Example 41.

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

The COM directive differs from GEN in that Macro-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.

In Macro-Symbol, 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[†]
AF	NUM[†]
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)

The "CF" specifies the command field, and "element number" specifies which element in the field is being referenced. "Element number" enclosed in parentheses is required.

Example 38. COM Directive and CF Function

```
        .
        .
        .
BYT     COM,8,8             CF(2),CF(3)
        .
        .
        .
XX      BYT,35,X'3C'        |2|3|3|C|
        .                   0      15
        .
        .
```

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, Macro-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)

The "AF" specifies the argument field, and "element number" specifies which element in the list of elements in that field is being referenced. "Element number" enclosed in parentheses is required.

[†] See Chapter 6.

Example 39. COM Directive and AF Function

```
          .
          .
XYZ       COM,16,16       AF(1),AF(2)
          .
          .
ALPHA     EQU             X'21'
ZZ        XYZ             65,ALPHA+X'FC'
          .               |0|0|4|1|0|1|1|D|
          .               0        15 16       31
```

Macro-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. "Element number" is required, and must be enclosed in parentheses. 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 40. 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 reference 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, Macro-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

Example 41. COM Directive's Error Notification

```
          .
          .
MAP       COM,*16,*16     CF(2),AF(1)
          .
          .
R         MAP,3           7
          .               |0 0 0 3|0 0 0 7|
          .               0               31
X         MAP,5           |0 0 0 5|0 0 0 0|
          .               0               31
          .
```

When the first reference line is encountered, Macro-Symbol defines a location R and generates a 32-bit data word with the values 3 and 7 in the left and right halfwords, respectively.

When the second reference line is encountered, an error notification is produced because the argument field entry is missing. However, the assembly is not terminated; Macro-Symbol will define a location X and generate a 32-bit data word with the values 5 and 0 (for the missing entry) in the left and right halfwords, respectively.

DATA Produce Data Value

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

label	command	argument
[label]	DATA[,f]	value_1[,value_2,...,value_n]

where

label is any valid symbol. Use of a label is optional. When present, it 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 words generated.

f is the field size specification in bytes; f may be any evaluatable expression that results in an integer value in the range $1 \le f \le 16$.

value_i 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 of a value when an address resolution other than the intrinsic resolution of the execution location counter is desired.

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 42. DATA Directive

```
          .
          .
MASK1     DATA,1    X'FF'        Produces an 8-bit value
                                 identified as MASK1.
                                 |F|F|
                                 0   7
          .
          .
MASK2     DATA,2    X'1EF'       Generates the hexa-
                                 decimal value 01EF as
                                 a 16-bit quantity,
                                 identified as MASK2.
                                 |0|1|E|F|
                                 0      15
          .
          .
BYTE      DATA,3    BA(L(59))    The byte address of the
                                 literal value 59 is as-
                                 sembled 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 iden-
                                 tified as TEST.
                                 |0|0|0|0|0|0|0|0|
                                 0      15 16     31
                                 |0|0|0|0|0|0|F|F|
          .                      0      15 16     31
          .
DT4       DATA,1    X'94',X'CF',X'AB'
                                 Generates three 8-bit
                                 values, the first of
                                 which is identified as
                                 DT4.
                                 |9|4|C|F|A|B|
          .                      0          23
          .
```

TEXT EBCDIC Character String

The TEXT directive enables the user to incorporate messages in his program that are to be output on some device other than the typewriter via the Monitor's standard output subroutines or output on the typewriter by some routine other than the Monitor's standard one. This directive has the form

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

where

label is any valid symbol. Use of a label is optional. When present, the label is associated with the leftmost byte of the storage area assigned to the assembled message.

'cs' are character string constants. The total number of characters must not exceed 255.

When several character strings are present in the argument field of a TEXT directive, the characters are packed in contiguous bytes. See Example 43. This directive permits continuation lines, but the continuation indicator must occur between two character strings. The characters are assembled in a binary-coded form in a field that begins at a word boundary and ends at a word boundary. 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.

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 intrinsic function in a TEXT directive. The AF function is not written with single quotation marks. See Example 44.

Example 43. 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 44. TEXT Directive

```
        .
        .
        TEXT      AS(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 Macro-Symbol to generate

R	E	S	U
L	T	S	
=			

When procedure PRINT2 is referenced, the second TEXT directive causes Macro-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 functions essentially the same as the TEXT directive except that a byte count of the data bytes is generated prior to the first text 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 form of the TEXTC directive is

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

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

Example 45. TEXTC Directive

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

                         generates
```

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

LISTING CONTROL

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	

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

The PAGE directive is effective only at assembly time. No code is generated for the object program as a result of its 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	[u]

where u is an expression whose value specifies the number of lines to be spaced. The expression u must not contain any external references.

If u is omitted, its value is assumed to be 1. If u is greater than 15, u is set to 15. If the value of u exceeds the number of lines remaining on the page, the directive will position the assembly listing to the top of the form.

The SPACE directive is effective only at assembly time. No code is generated in the object program as a result of its use.

Example 46. SPACE Directive

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

TITLE Identity 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	['character string']

where "character string" is a character string constant and may include 1-75 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.

Example 47. TITLE Directive

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

The first TITLE causes Macro-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'.

6. PROCEDURES

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 Macro-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. A procedure is referenced by its name appearing as the first element of the command field.

Procedures allow a program written in the assembly language of one computer (e.g., XDS 9300) to be assembled and executed on another computer (e.g., XDS Sigma 7). If a procedure is written for each 9300 machine instruction, Macro-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 Macro-Symbol comes from three directives: GEN, DO, and PROC. The GEN and DO directives were described in Chapter 5; how they are used in procedures is illustrated in the various examples in this chapter. The directives that identify procedures and the directives that designate the beginning and end of each procedure are discussed in this chapter. The intrinsic functions commonly used in writing procedures are also discussed.

All procedure definitions must appear prior to the first generative statement (GEN, DATA, TEXT, TEXTC, or machine instruction).

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 must be placed at the beginning of the source program; this ensures that the definitions will precede all references to them.

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

CNAME Procedure Name

A procedure may be invoked by a command 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	CNAME	[exp]

where

label is the symbol 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.

exp is an optional value that is evaluated and associated with the label. The expression may not contain forward references. Only previously defined or external reference ± addend expressions are legal. The use of the value is explained later in this chapter under "Multiple Name Procedures".

There is no limit to the number of CNAME directives that may be given for a single procedure.

The applicable CNAME directives must precede the procedure definition and the definition must follow immediately after the name lines. CNAME directives are associated with the first procedure definition encountered following these directives. This means that one cannot put all CNAME 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.

Procedures allow the programmer to create new instructions and directives. The programmer is not permitted to redefine the existing assembly language instructions and directives.

PROC Begin Procedure Definition

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

label	command	argument
	PROC	

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

The first line encountered following the PROC directive begins the procedure body. Nonlocal symbols are not unique to a procedure. A procedure may not contain other procedure definitions.

PEND End Procedure Definition

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

label	command	argument
	PEND	

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

Generally, the format of a command procedure appears as

```
         :
         .
name     CNAME exp      identifies the procedure
         PROC
         :              procedure definition
         .
         PEND
```

PROCEDURE REFERENCES

A procedure reference is a statement within a program that causes Macro-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:

```
label field    command field    argument field

   label        cpr, b list      c list
  └──┬──┘      └─────┬─────┘    └───┬────┘
    LF        ↑      CF             AF
              │
              └── procedure name
```

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 48 illustrates a command procedure and reference line.

Example 48. 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 procedure 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
```

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.

MULTIPLE NAME PROCEDURES

The expression that appears on a particular CNAME 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 49 illustrates this concept.

Example 49. 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.

PROCEDURE DISPLAY

When a procedure definition is encountered, Macro-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, Macro-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.

INTRINSIC FUNCTIONS

Intrinsic functions are functions that are built into the assembler. The intrinsic functions BA, HA, WA, and DA, concerned with address resolution, were discussed in Chapter 3, and are valid for Macro-Symbol.

The intrinsic functions discussed in this section include

LF	AFA
CF	NAME
AF	NUM

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, and SREF.

LF Label Field

This function refers to the label field in a COM directive or a procedure reference line. Its format is

LF(subscript)

or

LF

If present, the subscript must be 1; whether or not the subscript is present, the reference is to the symbol or expression in the label field. More than one label is not permitted.

Each LF reference causes Macro-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 50. 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)

The "CF" specifies the command field, and "subscript" specifies which element in that field is being referenced. "Subscript" enclosed in parentheses is required.

As for LF, each CF reference causes Macro-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 51. CF Function

```
                 .
                 .
  CFVALUE        SET              CF(3)
                 .
                 .
  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, Macro-Symbol will evaluate the third expression 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)

The "AF" specifies the argument field, and "subscript" specifies which element in that field is being referenced. "Subscript" enclosed in parentheses is required.

Example 52. AF Function

```
         .
         .
  AA     SET      AF(2)
         .
         .
  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(2) refers to the second element of the argument field.

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)

where "AFA" identifies the function, and "subscript" specifies which element in the argument field list is to be tested. "Subscript" enclosed by parentheses is required.

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 53. 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 (that is, 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) the expression in the CNAME argument field. Its format is

NAME

where "NAME" identifies the function.

A programmer can write a procedure with several entry points and assign the procedure several names via the CNAME directives. Each name may be given a unique value in the argument field of the CNAME directive. Then, within the procedure definition the programmer can use the NAME function to determine which entry point will be referenced (see Example 54).

NUM Determine Number of Elements

The NUM function yields the number of elements in the designated field. Its format is

NUM(AF), NUM(LF), or NUM(CF)

The NUM function is thus 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)).

SAMPLE PROCEDURES

Example 55 illustrates 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 54. NAME Function

```
         :
B        CNAME            0                                  Declares three names for the following com-
BGE      CNAME            1                                  mand procedure, each with an associated
BLE      CNAME            2                                  value.
         PROC
         BOUND         4                                     Bound on a fullword boundary.
LF       GEN,1,7,4,3,17   AFA(1),X'68',NAME,AF(2),WA(AF(1))  Generate a 32-bit word with the configura-
                                                             tion for a Branch, Branch if Greater Than or
                                                             Equal, or Branch if Less Than or Equal
                                                             instruction.
         PEND                                                End of procedure definition.
         :
NOW      BLE           RETRY                                 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, Macro-Symbol processes the procedure. In this instance, the label NOW is defined, and Macro-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 (that is, no indexing specified).
15-31	The first argument field element in the procedure reference line, evaluated as a word address.

Example 55. 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 table 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             Clear the register.
            AW,8     ALPHA         Add contents of ALPHA to register 8.
            AW, 8    ALPHA+1       Add contents of ALPHA+1 to register 8.
            STW,8    BETA          Store answer.
```

If the procedure reference were

```
            ADDEM    ALPHA,5,8,BETA,3
```

the generated code would be equivalent to

```
            SW,8     8             Clear 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           Increment register 3 by 1 and branch.

            STW,8    BETA          Store answer.
```

7. ASSEMBLY LISTINGS

MACRO-SYMBOL ASSEMBLY LISTING

The XDS Macro-Symbol assembler produces listing lines according to the format shown in Figure 3. 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 symbol (EQU or SET) contains the following information:

NNNNN   Source image line number in decimal.

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.

and

SSS...   Source image.

ASSEMBLY LISTING LINE

Each source image line containing a generative statement prints the following information:

NNNNN   Source image line number in decimal.

CC   Current section number in hexadecimal. See CC under "Equate Symbols Line".

LLLLL   Current value of load 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 1 to 4 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

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				N	N	N	N	N										C	C		L	L	L	L	L		B										S	S	S...
				N	N	N	N	N										X	X	X	X	X	X	X	X												S	S	S...
				N	N	N	N	N										T	T	T	T																S	S	S...
Assembly listing line				N	N	N	N	N				C	C		L	L	L	L	L		B			X	X	X	X	X	X	X	X		A				S	S	S...
Ignored source image line				N	N	N	N	N																									*	S	*		S	S	S...
Error line				*	*	*	*								m	e	s	s	a	g	e																		
Literal listing line												C	C		L	L	L	L	L					X	X	X	X	X	X	X	X		A						

Figure 3. Macro-Symbol Listing Format

processing a DO directive with an expression value of zero. NNNNN and SSS... have the same meanings as in an assembly listing line.

ERROR LINE

When an error is detected in the source image line, the line begins with up to three error code characters (any three of the codes listed in Table 3).

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 load location counter to word level in hexadecimal.
XXXXXXXX	Value of literal as a hexadecimal memory word.
A	Address classification flag. See "Assembly Listing Line".

Table 3. Macro-Symbol Error Flags

Code	Severity	Significance
C	3	Constant string error. A constant contains an illegal character or is improperly formed.
D	3	Duplicate symbol or command.
E	3	Expression error. The statement contains an expression that cannot be properly evaluated or that does not have a legal value.
F	3	Illegal system name.
I	3	Illegal or unknown command.
K	7	Program structure error.
L	3	Label is missing or an external name exceeds 63 characters.
M	3	A required field is missing from the statement.
N	3	A DO or DO1 nesting error has occurred.
S	3	A general violation of syntax structure was encountered in the statement.
T	3	Truncation error.
U	3	Undefined symbol.

8. MACRO-SYMBOL OPERATIONS

REAL-TIME BATCH MONITOR CONTROL COMMANDS

To assemble a Macro-Symbol Program, a run deck containing certain Monitor commands must be prepared. These commands are described in this chapter. (Other Monitor commands that are not needed for Macro-Symbol are described in the Sigma 5/7 Real-Time Batch Monitor Reference Manual.)

JOB CONTROL COMMAND

The first card in each Macro-Symbol run deck must be a JOB card, which has the following format:

```
!JOB  account number,name
```

where

account number is an optional alphanumeric string, from 1 to 8 characters, that identifies the account or project to which the run is to be charged.

name is an optional alphanumeric string, from 1 to 12 characters, that identifies the user.

ASSIGN CONTROL COMMAND

Appearing next in the run deck are any ASSIGN cards relating to the assembly. ASSIGN cards normally are not needed, because the system assumes the standard default assignments that are specified at SYSGEN. (For a description of the ASSIGN format, see the Sigma 5/7 Real-Time Batch Monitor Reference Manual.)

MACRSYM CONTROL COMMAND

The next card in the run deck will be the MACRSYM card, which has the following format:

```
!MACRSYM  option1,option2,...,option n
```

where any number of options, or none, may be specified. The options available are listed below:

BA	Batch assembly mode
BO	Binary output
CI	Compressed input
CO	Compressed output
GO	Output GO file
LO	List assembly output
SI	Source input
LU	List updates

Options may be specified in any order. If the same option occurs more than once, only the first option has any effect; that is, repetitions are ignored. If no options are specified, the following options are assumed: SI, CI, LU, LO, and BO.

The MACRSYM control command obeys the format rules for Monitor control commands, except that a semicolon is not allowed. (See the Sigma 5/7 Real-Time Batch Monitor Reference Manual for the format rules.)

If the program is on cards, these cards must immediately follow the MACRSYM card. The Macro-Symbol program deck is terminated by the first card that has an END directive in the command field. Any cards after the END directive are ignored by the Macro-Symbol assembler, except in the batch mode where the cards are treated as successive programs (see "EOD Control Command").

A sample MACRSYM card is shown below:

```
!MACRSYM  SI,LO,CI,BO
```

The various options are as follows:

BA selects the batch assembly mode. In this mode, successive assemblies may be performed with a single MACRSYM card. The assembler reads and assembles successive programs until a double end-of-data (!EOD) is encountered. In the batch mode, current device assignments and options on the MACRSYM card are applied to all assemblies within the batch. A program is terminated when an END directive is processed. Successive programs may or may not have an end-of-data 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 magnetic tape, standard tape end-of-files provide job termination. When batch assemblies consist of successive updates from the card reader to compressed programs from disc or tape, the update packets are terminated by a +END card and they should not be separated by EOD cards. There must be a one-to-one correspondence of update packets to compressed programs. The end of a job is signaled by end-of-file conventions applied to the CI device.

BO specifies that binary output is to be produced on the BO device.

CI specifies that compressed input is to be taken from the CI device.

CO specifies that compressed output is to be produced on the CO device.

GO 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 disc or magnetic tape) must be used.

LO specifies that a listing of the assembled object program is to be produced on the LO device.

LU specifies that a listing of the update decks (if any) is to be produced on the LO device. This listing consists of the image of each update line with the number of the line in the update deck.

SI specifies that symbolic input is to be taken from the SI device.

EOD CONTROL COMMAND

An EOD card is required only with a symbolic deck and is placed immediately after the END directive. If the BA (batch mode) option is not specified, then any cards between the END directive and the EOD card are ignored. In the batch mode, EOD cards are optional between successive symbolic program decks. The END directive terminates each assembly. If EOD cards separate successive assemblies, then any cards between an END directive and an EOD card are assembled as the next program. If an EOD card is not preceded by an END directive, Macro-Symbol automatically provides an END directive. The EOD card has the following form

```
!EOD
```

UPDATING A COMPRESSED DECK

The use of the CO option on the MACRSYM card directs Macro-Symbol to produce a compressed deck from a source program, which can then be used as input during a later assembly. Because compressed decks contain one-fourth to one-fifth as many cards as the source deck, they are significantly more manageable and faster to handle.

Compressed decks can be updated via an "update packet", which is the set of cards between the first + (update) command and the compressed deck. If any symbolic cards (a "symbolic deck") precede the first + command, they are treated as if they were preceded by an a+0 card (see +k below); that is, they are inserted before the first line of the program.

Macro-Symbol recognizes three update control commands:

+k where k is a line number corresponding to a line 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 is less than or equal to 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.

+END where END designates the physical end of an update packet. If the SI and CI devices are the same, this command is optional, since Macro-Symbol terminates the update packet automatically on encountering the first compressed card. If the SI and CI devices are different, this command is necessary.

The + character of each update control command must be in column 1, followed immediately by the control information, with no embedded blanks. The first blank column terminates the control command, and comments may optionally follow the blank. The update control commands, with their associated update records, must occur in numeric sequence.

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 control commands cause an abort error.

PROGRAM DECK STRUCTURES

Macro-Symbol accepts two basic types of input decks: symbolic decks and compressed decks preceded by optional update packets. Five basic deck structures are possible (see Figure 4).

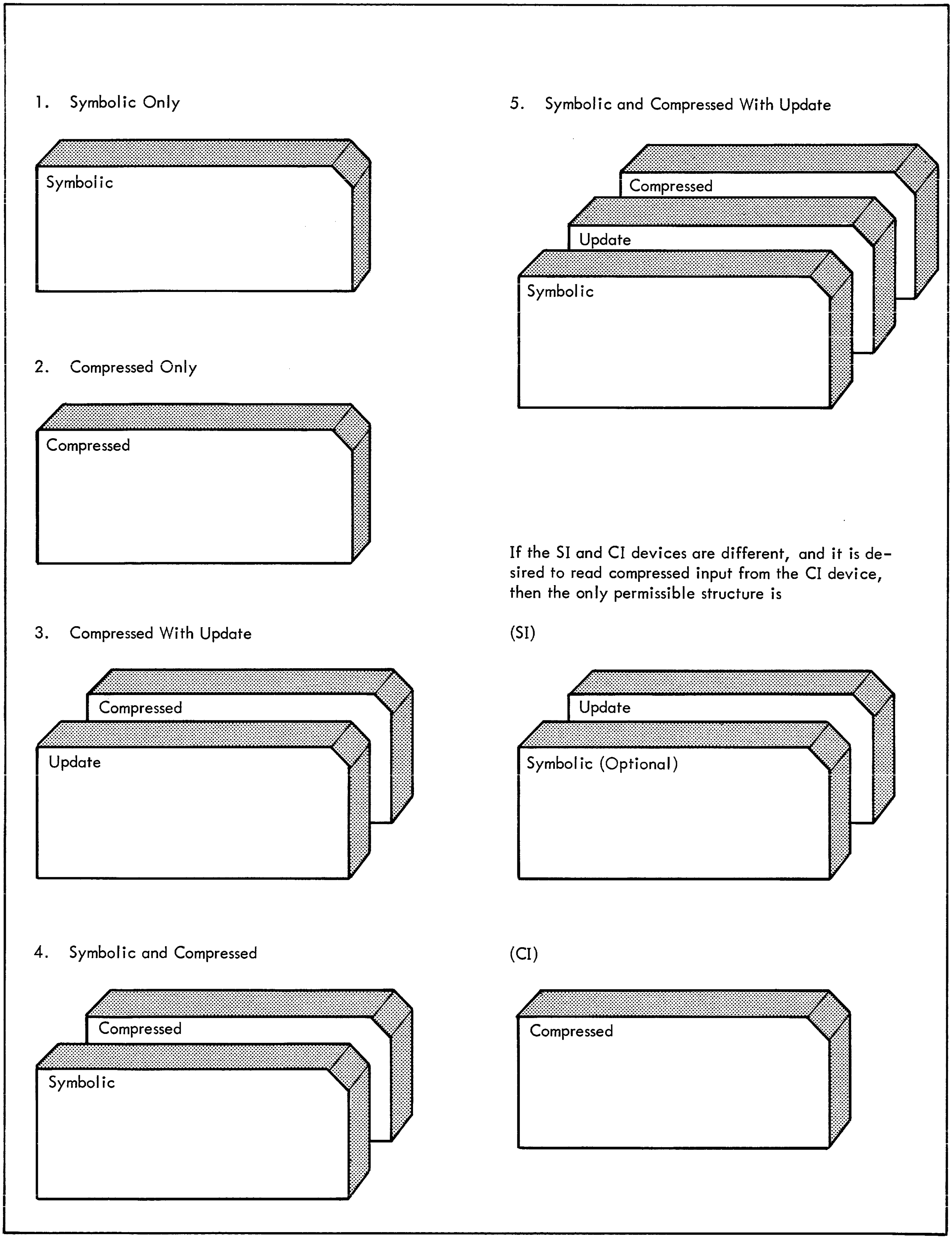
ERROR MESSAGES

Macro-Symbol outputs two types of error messages: (1) flags and error messages pertaining to the assembled program and (2) operational error messages. The error flags are described in Table 3. If Macro-Symbol encounters an undefined local symbol, the following error message is output:

UNDEFINED LOCAL "SYMBOL"

The operational error messages are as follows:

ILLEGAL OPTION '0000' This error message indicates that the unknown option '0000' has been specified in the Macro-Symbol command. The option is ignored.



1. Symbolic Only

2. Compressed Only

3. Compressed With Update

4. Symbolic and Compressed

5. Symbolic and Compressed With Update

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

(SI)

(CI)

Figure 4. Basic Deck Structures

NO INPUT SPECIFIED
MACRO-SYMBOL ABORT ERROR

This error message indicates that neither SI nor CI has been specified in the Macro-Symbol command.

IO ERROR CODE XX FILE AA
MACRO-SYMBOL ABORT ERROR

This error message indicates that the Monitor has signaled I/O error XX on file AA.

UPDATE DECK ERROR
MACRO-SYMBOL ABORT ERROR

This error message indicates that the last output update control command is incorrect.

COMPRESSED DECK ERROR
MACRO-SYMBOL ABORT ERROR

This error message indicates a sequence, checksum, or format error in the compressed deck.

SPACE OVERFLOW
MACRO-SYMBOL ABORT ERROR

This error message indicates the program being assembled is too large to assemble in the core memory available.

APPENDIX A. SUMMARY OF MACRO-SYMBOL DIRECTIVES

In this summary brackets are used to indicate optional items.

Form			Function	Page
[label]	ASECT		Declares generative statements will be assembled to be loaded into absolute locations.	18
	BOUND	boundary	Advances the execution location counter to a byte multiple of "boundary" and advances the load location counter the same number of bytes.	17
label	CNAME	[exp]	Designates a command name ("label") for the next procedure definition and specifies the value ("exp") associated with "label".	38
label	COM,field list	v_1 [,...,v_n]	Describes a command skeleton; v_i specifies the contents of each "field"; "label" is the symbol by which the command skeleton is referenced.	31
[label]	CSECT	[exp]	Declares program section "label" as a relocatable control section with memory protection specified by "exp" where $0 \leq \text{exp} \leq 3$. If "exp" is omitted, the value 0 (no memory protection) is assumed.	18
[label]	DATA[,f]	v_1 [,...,v_n]	Generates each value in the list of v_i into a field whose size is specified by f in bytes. If f is omitted, a field size of 4 bytes is assumed.	34
	DEF	symbol_1 [,...,symbol_n]	Declares that the "symbol_i" may be referenced by other separately assembled programs.	30
[label]	DO	exp	If the DO is within a procedure and the value of "exp" is greater than zero, processes the code from DO to ELSE or FIN (if ELSE is absent) "exp" times. If the DO is outside a procedure and the value of "exp" is equal to one, processes the code from DO to ELSE or FIN (if ELSE is absent) once. If the DO is outside a procedure and the value of "exp" is greater than one, processes the statement following DO "exp" times. Then continues assembly at the statement following FIN. If "exp" ≤ 0, skips all code from DO to ELSE or FIN (if ELSE is absent); resumes assembly at that point.	26
[label]	DO1	exp	If the value of "exp" is greater than zero, processes the one statement following the DO1, "exp" times, then continues the assembly at the next statement. If "exp" ≤ 0, skips the statement following DO1 and resumes assembly.	25
[label]	DSECT	[exp]	Declares a dummy program section "label" with memory protection specified by "exp"	18

Form			Function	Page
			where 0 ≤ exp ≤ 3. If "exp" is omitted, the value 0 (no memory protection) is assumed.	
	ELSE		Terminates the range of an action DO loop, or identifies the beginning of the alternate sequence of code for an inactive DO loop.	26
[label]	END	[exp]	Terminates a program. 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.	25
label	EQU	[exp]	Equates "label" to the value of "exp" (nonredefinable).	29
	FIN		Terminates a DO loop.	26
[label]	GEN,field list	v_1 [,...,v_n]	Produces a hexadecimal value representing v_i in the number of bits specified by "field" in "field list".	31
	GOTO[,k]	label_1 [,..., label_n]	Resumes assembly at the statement whose label corresponds to the kth "label".	25
[label]	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.	16
	LOCAL	[symbol_1,...,symbol_n]	Terminates existing local symbol region and initiates a new region where the "symbol_i" are local symbols.	29
[label]	ORG [,n]	location	Sets both the current load location counter ($$) and the current execution location counter ($) to the value "location" and sets their resolution specifications to n, where the value of n is 1, 2, 4, or 8.	15
	PAGE		Upspaces assembly listing to the top of form.	36
	PEND		Terminates procedure definition.	39
	PROC		Identifies the beginning of a procedure definition.	38
	REF	[symbol_1,...,symbol_n]	Declares that the "symbol_i" are references to externally defined symbols.	31
[label]	RES[,n]	u	Advances both location counters ($ and $$) by u n-sized units. If n is omitted, a size of 4 bytes is assumed.	17

Form			Function	Page
label	SET	[exp]	Equates "label" to the value of "exp" (redefinable).	29
	SPACE	[u]	Upspaces the assembly listing u lines. If u is omitted, 1 is assumed.	36
	SREF	symbol$_1$ [,...,symbol$_n$]	Declares that the "symbol$_i$" are secondary external references.	31
	SYSTEM	name	Defines system as "name".	24
[label]	TEXT	'cs$_1$'[,...,'cs$_n$']	Assembles the "cs$_i$" (character string constant) in binary-coded format for use as an output message.	35
[label]	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.	36
	TITLE	['cs']	Prints "cs" (character string constant) as a heading on each page of assembly listing.	37
[label]	USECT	name	Specifies that the control section of which label "name" is part is to be used in assembling subsequent statements.	18

APPENDIX B. MACRO-SYMBOL COMPATIBILITY

Macro-Symbol is designed to be compatible with Symbol and Meta-Symbol. However, minor differences do exist, and these differences are discussed below.

1. In a procedure, the statements

   ```
   LF      EQU     exp
   ```

   or

   ```
   LF      SET     exp
   ```

   cause a label error if the procedure reference statement does not contain a label field.

2. DO/ELSE/FIN and DO1 loop processing does not consider comment lines to be statements. The following input causes the DATA statement to be repeated n times:

   ```
       DO1   n
   *COMMENT
       DATA
   ```

3. An illegal argument field expression for a RES, ORG, or LOC directive generates a RES, ORG, or LOC 0.

4. The listing line for a USECT directive contains the control section number and the value of the execution and load location counters in the new control section. Macro-Symbol and Meta-Symbol assign this value to the label field of the USECT statement. Meta-Symbol, however, prints the section number and the saved value of the execution location counter of the previous section.

5. The listing lines for an ORG or LOC directive contain the value of the load location counter followed by the value of the execution location counter. This is the reverse order of the Meta-Symbol format.

6. The evaluation of an expression involving explicit address resolution functions applied to an external or local forward reference ± addend, assumes that at load time the reference is an address and not a constant.

APPENDIX C. SUMMARY OF SIGMA 5 INSTRUCTION MNEMONICS

Required syntax items are underlined whereas optional items are not. The following abbreviations are used:

m	mnemonic	a	address expression
r	register expression	x	index expression
v	value expression		
*	indirect designator		

Codes for required options are

P	Privileged
F	Floating-Point Option
L	Lock 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
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,x	Add Halfword		
AW	m,r	*a,x	Add Word		
AD	m,r	*a,x	Add Doubleword		
SH	m,r	*a,x	Subtract Halfword		
SW	m,r	*a,x	Subtract Word		
SD	m,r	*a,x	Subtract Doubleword		
MI	m,r	v	Multiply Immediate		

Mnemonic	Syntax	Function	Equivalent to:	Required Options
FIXED-POINT ARITHMETIC (cont.)				
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		
SF	m,r *a,x	Shift Floating		
SFS	m,r v,x	Shift Floating, Short		
SFL	m,r v,x	Shift Floating, Long		
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
FDS	m,r *a,x	Floating Divide Short		F
FDL	m,r *a,x	Floating Divide Long		F
PUSH DOWN				
PSW	m,r *a,x	Push Word		
PLS	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		

Mnemonic	Syntax			Function	Equivalent to:		Required Options
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	For Use After Comparison Instructions	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	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	BCR,2	*a,x	
BNE	m	*a,x		Branch if Not Equal	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	
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,C	*a,x	
BWO	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	
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,A	*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,F	*a,x	
BSO	m	*a,x		Branch if 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	

[†] See CW instruction in XDS Sigma 5 Computer Reference Manual.

Mnemonic	Syntax		Function	Equivalent to:	Required Options
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	Load Program Status Doubleword		P
XPSD	m,r	*a	Exchange Program Status Doubleword		P
LRP	m	*a,x	Load Register Pointer		P
MMC	m,r	v	Move to Memory Control		P
WAIT	m	*a,x	Wait		P
RD	m,r	*a,x	Read Direct		P
WD	m,r	*a,x	Write Direct		P
NOP[†]	m	*a,x	No Operation		
PZE	m	*a,x	Positive Zero		
INPUT/OUTPUT					
SIO	m,r	*a,x	Start Input/Output		P
HIO	m,r	*a,x	Halt Input/Output		P
TIO	m,r	*a,x	Test Input/Output		P
TDV	m,r	*a,x	Test Device		P
AIO	m,r	*a,x	Acknowledge Input/Output Interrupt		P

[†] Equivalent to LCFI instruction with r = 0.

APPENDIX D. SUMMARY OF SIGMA 7 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

7	Sigma 7
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 Halfwaord		
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
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,x	Add Halfword		
AW	m,r	*a,x	Add Word		
AD	m,r	*a,x	Add Doubleword		
SH	m,r	*a,x	Subtract Halfword		
SW	m,r	*a,x	Subtract Word		

MACRO-SYMBOL DIRECTIVES

[label]	ASECT	
	BOUND	boundary
label	CNAME	[expression]
label	COM,field list	value$_1$[,...,value$_n$]
[label]	CSECT	[expression]
[label]	DATA[,f]	value$_1$[,...,value$_n$]
	DEF	symbol$_1$[,..., symbol$_n$]
[label]	DO	expression
[label]	DO1	expression
[label]	DSECT	[expression]
	ELSE	
[label]	END	[address]
label	EQU	[expression]
	FIN	
[label]	GEN,field list	value$_1$[,...,value$_n$]
	GOTO[,k]	label$_1$[,...,label$_n$]
[label]	LOC $\left[,\begin{matrix}1\\2\\4\\8\end{matrix}\right]$	location
	LOCAL	[symbol$_1$,...,symbol$_n$]
[label]	ORG $\left[,\begin{matrix}1\\2\\4\\8\end{matrix}\right]$	location
	PAGE	
	PEND	
	PROC	
	REF	symbol$_1$[,...,symbol$_n$]
[label]	RES[,k]	number of locations
label	SET	[expression]
	SPACE	[number of spaces]
	SREF	symbol$_1$[,...,symbol$_n$]
	SYSTEM	instruction set
[label]	TEXT	'character string$_1$'[,...,character string$_n$]
[label]	TEXTC	'character string$_1$'[,...,character string$_n$]
	TITLE	['character string']
[label]	USECT	name

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			
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	For Use After Comparison Instructions	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	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	BCR,2	*a,x	
BNE	m	*a,x		Branch if Not Equal	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	

[†] See CW instruction in XDS 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,C	*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 Digits	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,A	*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,F	*a,x	
BSO	m	*a,x		Branch if 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		Load Program Status Doubleword			P
XPSD	m,r	*a		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
LPC	m,r			Load Program Control			7MP
LLOCKS	m,r			Load Locks			LP
WAIT	m	*a,x		Wait			P
RD	m,r	*a,x		Read Direct			P
WD	m,r	*a,x		Write Direct			P

Mnemonic	Syntax		Function	Equivalent to:	Required Options
CONTROL (cont.)					
NOP[†]	m	*a,x	No Operation		
PZE	m	*a,x	Positive Zero		
INPUT/OUTPUT					
SIO	m,r	*a,x	Start Input/Output		P
HIO	m,r	*a,x	Halt Input/Output		P
TIO	m,r	*a,x	Test Input/Output		P
TDV	m,r	*a,x	Test Device		P
AIO	m,r	*a,x	Acknowledge Input/Output Interrupt		P

[†] Equivalent to an LCFI instruction with r = 0.

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.

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.

Publication Revision Sheet

February 1970

CORRECTIONS TO XDS SIGMA 5/7 MACRO-SYMBOL REFERENCE MANUAL,

PUBLICATION NO. 90 15 78, OCTOBER 1969

Replacement pages for pp 53-54 and 57-58 are attached. Lines that have been changed in the manual are indicated by revision bars in the margin of the page. Pages 54 and 58 have been altered to indicate that the LAS and LMS instructions do not apply to all machines. Item 7 has been deleted from page 53 in the interest of compatibility between Macro-Symbol and Meta-Symbol.