SCIENTIFIC DATA SYSTEMS

Reference Manual

SDS SYMBOL and META-SYMBOL

Price: $3.50

SYMBOL and META-SYMBOL REFERENCE MANUAL

for

900 SERIES/9300 COMPUTERS

90 05 06G

March 1969

SDS

SCIENTIFIC DATA SYSTEMS/701 South Aviation Boulevard/El Segundo, California 90245

©1964, 1965, 1966, 1967, 1968, 1969, Scientific Data Systems, Inc.

Printed in U.S.A.

REVISION

This publication, SDS 90 05 06G, is a minor revision of the SYMBOL and META-SYMBOL Reference Manual, SDS 90 05 06F. Changes to the previous edition are indicated by a line at the right or left margin of the page.

RELATED PUBLICATIONS

Title of Manual	Publication Number
SDS 910 Computer Reference	90 00 08
SDS 920 Computer Reference	90 00 09
SDS 925 Computer Reference	90 00 99
SDS 930 Computer Reference	90 00 64
SDS 9300 Computer Reference	90 00 50
SDS 92 Computer Reference	90 05 05
SDS MONARCH Reference	90 05 66
SDS 9300 MONITOR Reference	90 05 13
SDS 900 Series FORTRAN II Reference	90 00 03
SDS 900 Series FORTRAN II Operations	90 05 87
SDS FORTRAN IV Reference	90 08 49
SDS FORTRAN IV Operations	90 08 82
SDS Business Language Reference	90 10 22

NOTICE

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

CONTENTS

APPENDIXES

PREFACE

This manual describes two SDS Assembly Systems: META-SYMBOL, and its compatible subset, SYMBOL. For both systems, it defines a symbolic programming language and the processor that assembles programs written in this language. Although the name SYMBOL (or META-SYMBOL) applies to both the language and the processor, context will normally clarify the distinction. Since SYMBOL is a compatible subset of META-SYMBOL, all programs written in SYMBOL may be assembled by META-SYMBOL; the converse is not true.

The introduction to META-SYMBOL is basic since, in many ways, META-SYMBOL represents a radical departure from more conventional assemblers. The description is deliberately syntax-oriented, and the details pertaining to its implementation on particular SDS computers are relegated to appendixes.

The presentation assumes that the reader is familiar with the basic theory of digital computer programming.

I. GENERAL DESCRIPTION

Basically, the solution of problems on a digital computer involves two steps:

Analysis: mathematical description of the problem, or the formulation of a mathematical model

Coding: transcription of the mathematical equations into a sequence of machine instructions

The result, called a program, operates on data specified to it (input data) and produces data which constitute the problem's solution (output data). If the mathematical description is in parametric form, a family of solutions may be obtained by varying the input data.

Both analysis and coding involve language translation: normally, the translation sequence is from verbal to mathematical to machine code. The first two forms are more familiar to humans than machine code, particularly since machine code varies from computer to computer. Although deliberately simple, the following example is illustrative:

Verbal	Mathematical	Machine (octal)
Let x be the sum of y and z	$x = y + z$	07601000
		05501001
		03501002

No wonder, therefore, that the coding phase frequently is the most time-consuming and unreliable portion of programming.

Automatic programming systems arose because of early recognition that coding itself had all the attributes of a typical programming problem. Ironically, therefore, the computer could solve the very problem it created. The creation of a program was involved that would generate machine language programs (the output data) from problem specifications (the input data), as written in some convenient non-machine language; to be convenient, the language had to be easy to teach, learn, read, and write. Since the output data form is immediately specified by the computer on which the program is to be executed (called the target machine), only the form (syntax) of the source language input to the translation program remained to be described.

Clearly, the source language would occupy a level in between mathematical notation and machine language but, unfortunately, no single language evolved. At one end of the language spectrum, several algebraically-oriented languages developed, such as FORTRAN and ALGOL. The associated language translators are known as compilers. Toward the other end of the spectrum, as languages become more machine-dependent, a new language tends to develop for each new machine. The associated language translators are called assemblers, and the input (assembly) language is generally in the form of machine instructions represented symbolically. Either language becomes more or less appropriate as the problem shifts from mathematical to machine in nature.

But the problem was not yet solved. For, once specified, the assembler and compiler in turn engendered a second problem: In what language were they to be written? Just as the proliferation of programs pointed to the first problem, the proliferation of machines and languages gave rise to the second. A programming language suitable for dealing with programming languages, that is, a programming meta-language, was required. META-SYMBOL is the outgrowth of this concept as implemented on SDS computers.

META-SYMBOL consists of two basic parts: a processing section (the processor proper) and a directive section. The directive section contains directives that describe the computer, directives that describe the assembler, and directives that instruct the meta-assembler. Since directives describe all applicable computer characteristics, only the directive section need be changed in implementing META-SYMBOL for other target machines. Similarly, alteration of the assembler-descriptive portion enables variations in the assembler's syntax, or even the implementation of entirely new programming systems. In normal usage, META-SYMBOL operates on conventional symbolic programs as a high-level symbolic assembler.

Operationally, META-SYMBOL is both faster and easier to use than conventional assembly programs. These benefits result from an advanced source language encoding scheme that makes modify-and-load assemblies not only convenient but efficient.

II. SYNTAX

A. Introduction

The syntax of a language is the set of rules governing its sentence (or statement) structure. All assembly and compiler languages possess a formal syntax.

Formerly, the syntaxes of many languages were strongly influenced by ease of implementation and/or computer hardware characteristics; they had numerous restrictions and exceptions. SYMBOL and META-SYMBOL do not have these limitations; consequently, they possess a simpler but more powerful syntax. There are fewer definitions and rules to learn because each one is more comprehensive. In learning them, however, the experienced programmer is cautioned since, in many cases, a familiar term (such as "expression") is redefined with greater generality. Proper use of the language is possible only after completely understanding the basic principles.

For convenient reference, the following definitions appear without illustration. Unless otherwise specified, all rules and definitions apply both to SYMBOL and to META-SYMBOL.

B. Characters

1. Alphabetic character: one of the characters A - Z.
2. Numeric character: one of the characters 0 - 9.
3. Alphanumeric character: any character which is either alphabetic or numeric.
4. Special character: a nonalphanumeric character (such as *, $, +). The character ⧺(internal 077) is strictly illegal in Meta-Symbol except for use in comments.

C. Program

A program is a series of one or more symbolic lines, the last of which must contain an END directive.

D. Line

A line is the unit in which the assembler processes information much as a card is the processing unit (unit record) to a keypunch.

Unlike a card, a line is a logical unit, subdivided into four parts, or fields, and may be equivalent to one or more (physical) unit records. The four fields that comprise a line are: the label field, the operation field, the operand field, and the comments field. With the exception of a line consisting entirely of comments, a line must always specify an operation. In the latter case, the presence of information in the other fields is at the programmer's option.

E. Label Field

The label field labels an operation or a value so that it can be symbolically referred to elsewhere. Labeling is accomplished by writing a symbol (see G. 1. b. i.) in the label field .

F. Operation Field

The operation field may contain a generative, such as a mnemonic machine instruction, or a non-generative, such as an assembler directive.

A directive, which always appears in the operation field, has three basic functions:

1. Describe the computer
2. Describe the assembler
3. Instruct the meta-assembler

Sections III and IV describe instructions and directives.

G. Operand Field

The operand field of a line may contain a sequence or a list of one or more expressions.

1. List

A list is a parenthetically-enclosed sequence of one or more expressions separated by commas. These expressions, called list items, are elements of the list. A list may itself be a list item. As shown below, lists are most useful in handling PROCedures and FUNCtions.

a. Expression

An expression is a series of items connected by operators (see G. 2.).The processor evaluates expressions by successively combining items, as specified by the connecting operator, in the order of decreasing operator hierarchy.

b. Items

An item may be one of the following types:

Item	Definition	Example
i. Symbol	A symbol is a string of alphanumeric characters, of which the first character is alphabetic. (Cf. VI, also Appendix J.)	ALPHA B1 X1Y
ii. Subscripted Symbol	A subscripted symbol is a symbol followed by a list of one or more expressions enclosed within parentheses.	ALPHA (2) B1 (1, N) X1Y (3*N, 4)
iii. Octal Integer	An octal integer is a string of from one to 15 octal digits preceded by a (signed or unsigned) zero.	012 01234567 077777777

	Item	Definition	Example
iv.	Decimal Integer	A decimal integer is a (signed or unsigned) string of from one to 15 decimal digits, of which the first is not zero. The legal range is $2^{47}-1 \geq N \geq -2^{47}$.	12 1234567
v.	Decimal Number	A decimal number is either a decimal integer or a (signed or unsigned) string of decimal digits and one or more of the following: decimal point, decimal scale operator, binary scale operator. When an item has a decimal point but has no binary scale operator, the item is of the floating point mode.	12 0.12 +12.0*+4 (-12.5)*+(-2)*/3
vi.	Character Data String	A character data string is a string of characters (alphabetic, numeric and/or special) surrounded by single quotes.	'B1' 'X1Y' '012' '12'
vii.	Current Location Symbol	The current location symbol represents the execution-time value of the location counter.	$
viii.	Subexpression	A subexpression is a parenthetically-enclosed expression that occurs as part of another expression.	(ALPHA + B1) (12+012) ($ +12)
ix.	Function Reference	A function reference is a symbol followed by a parenthetically-enclosed expression list. The symbol must have appeared in the label field of a NAME directive within a function definition (see IV, N).	MAX (X, Y)

2. Operators

An operator may be one of the following:

Operator	Representation	Hierarchy
Boolean		
equals	=	1
greater than	>	1
less than	<	1
sum (OR)	++	2
difference (exclusive OR)	--	2
product (AND)	**	3
Arithmetic		
sum	+	4
difference	-	4
product	*	5
truncated quotient	/	5
covered quotient	//	5
decimal scale	*+	6
binary scale	*/	6

The covered quotient operator, //, is defined: $a//b = (a+b-1)/b$.

The decimal and binary scale operators, *+ and */, respectively, may be used to combine any two expressions. Where x and y represent two expressions,

x *+y is equivalent to $(x) \cdot (10^y)$

x */y is equivalent to $(x) \cdot (2^y)$

Note that the nominal binary point of x is to the right of the least significant bit; that is, these operations use integer, not fractional notation.

Actually, */ functions as a logical shift operator, so that x */y is equal to x right (left) logical shifted y places, $y < 0$ $(y \geq 0)$. Hence, because of operator precedence, */ functions as an arithmetic operator for $\pm x$ */y but not for (-x) */y.

The use of operators is illustrated in the example which appears at the end of this section.

H. Comments Field

The comments field of a line may contain comments to annotate the program. The assembler ignores comments.

The next two sections describe instructions and directives. A format definition precedes usage description in each case. The following example illustrates the instruction format·

Format:

Label	Operation	Operand	Comments
[[$] LABEL]	LDA	[*]E1[,E2]	[LOAD A]

In this example, some of the parameters are enclosed in brackets to indicate that they are optional. All instructions must have at least an operation mnemonic and most must have an operand address. However, indexing (as indicated by E2) and indirect addressing (*) are optional. Similarly, the label and comment need not be present; if the line specifies a label, an optional dollar sign preceding the label indicates that it is an <u>external</u> <u>label</u> (Cf. V. D.).

As indicated above, the operand field of a line consists of a sequence or a list of expressions. Expressions will be represented by the symbols E, E1, E2, . . . ,EN.

The illustration on the following page is a representative (although not typical) META-SYMBOL program that uses each directive at least once.

```
                  77777777              1 A       EQU      -1
                  00000001              2 B       EQU      1
                  00000000              3 C       EQU      A=B                     FALSE
                  00000001              4 D       EQU      A=(-B)                  TRUE
                  00000001              5 E       EQU      B>A                     TRUE
                  00000000              6 F       EQU      B<A                     FALSE
                  00000003              7 G       EQU      B+B++B                  1+1.OR.1=3
                  77777776              8 H       EQU      A--B                    -1.EOR.1=-2
                  00000001              9 I       EQU      A**B                    -1.AND.1=1
                  77777777             10 J       EQU      G/H                     3/-2=-1
                  00000000             11 K       EQU      G//H                    3//-2=0
                  77777777             12 L       EQU      -3*/(-1)                -3*/(-1)=-3/2=-1
                  37777776             13 M       EQU      (-3)*/(-1)              (-3)*/(-1)=077777775*/(-1)=037777776
                  77777772             14 N       EQU      -3*/1                   -3*/1=-3*2=-6
                  00000024             15 O       EQU      2*+1                    2*(10.EXP.1)=2*10=20
                  3110375524202002     16 PI      EQU      3.1415926535            DECLARE FLOATING POINT VALUE
         00000    3110375524202002     17 PIE     DED      PI                      GENERATE FLOATING POINT DATUM
         00002    00000012             18 TEN     DATA     10,010,'10',TEN
         00003    00000010
         00004    00000100
         00005    00000002
         00006    60224321             19 TYPEOUT   TEXT < BLANKS CONVERTED TO 060>
         00007    45426260
         00010    23464565
         00011    25516325
         00012    24606346
         00013    60000600
         00014    12224321             20 PRINTOUT  BCD  < BLANKS CONVERTED TO 012>
         00015    45426212
         00016    23464565
         00017    25516325
         00020    24126346
         00021    12000102
                                       21 F       FUNC                             THIS FUNCTION COMPUTES THE ABSOLUTE
                                       22 ABS     NAME                             VALUE OF THE REFERENCE PARAMETER
                                       23         DO       F(1)<0,1,1
                                       24 RESULT  EQU      -F(1)
                                       25 RESULT  EQU      F(1)
```

```
                                26          END       RESULT
                                27 P        PROC      1
                                28 CLA      NAME      050000
                                29 STO      NAME      060100
                                30 I        FORM      3,15,3,15
                                31          I         P(0)*/(-15),P(0)**077777,P(2),P(1)
                                32          END
                                33 P        PROC                          THIS PROCEDURE MOVES THE CONTENTS
                                34 MOVE     NAME                          OF ONE MEMORY LOCATION TO ANOTHER
                                35          CLA       P(1)
                                36          STO       P(2)
                                37          END
00001                           38          AORG      1                   ESTABLISH PROGRAM REENTRY POINT
00001  0 0 01 00002             39          BRU       BEGIN               AT LOCATION 1
00000                           40          RORG      0                   REST OF PROGRAM RELOCATABLE
00000                           41 ALPHA    RES       1
00001                           42 BETA     RES       1
00002  0 50000 4 00001          43 BEGIN    CLA       1,4
00004  0 60100 4 00002          44          STO       2,4
00006  0 50000 0 00000          45          MOVE      ALPHA,BETA
00010  0 60100 0 00001
                                46 CONST    EQU       (-1,ABS(-1),-ABS(-1),ABS(-ABS(-1)))
                                47 INDEX    DO        :CONST
00012  00000001                 48          DATA      ABS(CONST(INDEX))
00013  00000001
00014  00000001
00015  00000001
       00000002                 49          END       BEGIN
```

III. INSTRUCTIONS

Instructions are represented as follows:

Format:

Label	Operation	Operand	Comments
[[$] LABEL]	LDA	[*] E1 [, E2]	[LOAD A]

All SDS computers include similar, although not identical, instruction characteristics. Among these are:

an operation field

an address field, modifiable by indexing and/or indirect addressing

an index field

an indirect address field

In the example, where the LDA instruction is used, the quantities enclosed in brackets are optional. The asterisk preceding the first operand indicates indirect addressing; the second operand, separated from the first by a comma, indicates indexing. Both E1 and E2 may be expressions, although their values may not exceed the address and index fields, respectively.

Examples:

Label	Operation	Operand	Comments
L1	LDA	M	LOAD A WITH CONTENTS OF M
LB	LDA	M, 2	LOAD A WITH CONTENTS OF M+X (900 Series) LOAD A WITH CONTENTS OF M+X2 (9300)
L2	LDA	*M	INDIRECT ADDRESSING
BL	LDA	*M, 2	INDEXING AND INDIRECT ADDRESSING

Some instructions (e.g., EOM), have more than two operands. The syntax for these instructions is covered separately.

In all these examples, the label is optional. The use of operand expressions will be illustrated following the introduction of directives.

IV. DIRECTIVES

A. Introduction

As noted previously, an assembler, like any other program, operates on input data to produce output data. The difference is that the output data from an assembler generally constitute another program which, in turn, operates on input data to produce output data. Thus, there are two levels at which the resultant program can be affected logically: at assembly time and at execution time. In the latter case, this is accomplished by input parameters to the program, and in the former case, by input parameters (called directives) to the assembler. These directives may enrich the semantics, but never the syntax, for a particular assembly. Syntactic changes may be accomplished only through reassembly of the assembler itself. Thus, directives are dynamic at assembly time, whereas instructions are dynamic at program execution time. The following directives are included in the assembly language:

Data Generation	Assembler Instruction	
DATA	AORG	PAGE
DED	RORG	DISP
TEXT	RES	INHD
BCD	DO	SUPR
Value Declaration	PROC	INHS
EQU	FUNC	MARK
FORM	NAME	SBRK
OPD	END	SIOR
RELTST		

DATA and OPD are actually system PROCs, but are included in this list because they behave similarly to directives.

Important: No forward or external reference is permitted within the operand field of a directive. Thus, the following example contains two violations.

```
F     00000                    1      RES   A      ILLEGAL, A=0 1ST PASS, 2 2ND PASS
            00000002           2  A   EQU   2
D     00002                    3  B   RES   A      HENCE B=0 1ST PASS, 2 2ND PASS
*     00004  0 43 0 00000      4      BRM   Z      LEGAL
*E                             5      END   Z      ILLEGAL
      00004                       Z
```

In the following, examples are provided to illustrate the functions of these directives, and they frequently include machine instructions. However, the role of the instruction is illustrative only, so that an understanding of the examples should not depend on an understanding of any particular machine.

B. AORG and RORG (Absolute ORiGin and Relative ORiGin)

Format:

Label	Operation	Operand	Comments
[[$] LABEL]	AORG or RORG	E	[PROGRAM ORIGIN]

The origin of a program is defined as the lowest numbered memory address occupied by (instructions or data of) the program. Generally, it is useful to allow the origin to be relocatable at execution time, so that the program can be executed equally well whether it is loaded beginning at one location or beginning at another.

Program relocatability is automatic in SYMBOL and META-SYMBOL. The assembler accomplishes it by producing relocation information together with the binary object program. Using this information, the Loader performs the relocation when the binary object program is loaded for execution.

In some cases, however, the programmer desires to control the program origin. This may be because all or part of his program must occupy fixed memory locations (for example, interrupt locations) or because, during program debugging, it is easier to relate the contents of memory to an assembly listing.

To accomplish these objectives, two directives are provided: AORG and RORG (for absolute and relocatable origin, respectively).

Example:

Label	Operation	Operand	Comments
I1	AORG	030	AT LOCATION 30 (OCTAL)
	BRM	I1INT	PLACE LINKAGE TO I1INT
	RORG	0200	I1INT IS TO BE RELOCATABLE
I1INT	HLT		
	.		
	.		
	.		
	BRR	I1INT	

In this example, all addresses except I1 are relocatable. Thus, the BRM I1INT is always loaded into location 030, but the contents of its address field, as a relocatable quantity (I1INT), is assigned at loading time. The subroutine I1INT, on the other hand, is completely relocatable, since the Loader provides the capability to override the otherwise automatic loading into location 0200.

Viewed otherwise, AORG and RORG have the function of resetting the location counter;(1) the symbol I1 has the same value (030) whether it appears on the AORG line or on the following line.

Naturally, the operand may be a completely general expression, and is not restricted to simple numeric values. Its value must, however, be defined within the program, and cannot be externally defined.

C. RES (REServe)

Format:

Label	Operation	Operand	Comments
[[$] LABEL]	RES	E	[RESERVE A BLOCK]

(1) The "location counter" is a special memory cell retained by the assembler in defining labels at assembly time.

The RES directive is primarily used to reserve and (optionally) label storage areas. It may also be used to reset the location counter; used in this manner, it is functionally redundant with respect to the RORG directive.

Example:

Label	Operation	Operand	Comments
TABLE	RES	10	RESERVE 10 LOCATIONS
	RES	0200-$	RESET LOCATION COUNTER TO 0200

Using the origin directives, these lines could have been written:

Label	Operation	Operand	Comments
TABLE	RORG	$	LABEL BLOCK
	RORG	$+10	RESERVE 10 LOCATIONS
	AORG	0200	RESET LOCATION COUNTER TO 0200

D. DATA (DATA)

Format:

Label	Operation	Operand	Comments
[[$] LABEL]	DATA	E1[,E2,...,EN]	[GENERATE DATA BLOCK]

The directive DATA enables the programmer to represent single-precision data conveniently within the symbolic program. Since operands may be general expressions, octal, decimal, binary-coded decimal, and symbolic data may all be generated with a single directive. In all cases, the translated expression is right justified within the computer word; except for negative data, unfilled bits always contain zeros.

Example:

Location	Contents	Label	Operation	Operand
01000		TENS	AORG	01000
01000	00000010		DATA	010, 10, '10', TENS
01001	00000012			
01002	00000100			
01003	00001000			
01004	02101012		DATA	(TENS+10)++('A'*0100000)

In conventional assembly programs, the manner of interpretation of the contents of the operand field depends upon the contents of the operation field. In SYMBOL and META-SYMBOL, however, this restriction does not apply since data unambiguously describe their own item type by adherence to the definitions in Section II.

E. DED (DEcimal Double Precision)

Format:

Label	Operation	Operand	Comment
[[$] LABEL]	DED	E1 [, E2, ..., EN]	[GENERATE DP DECIMAL DATA]

The directive DED enables the programmer to represent double-precision decimal data conveniently within the symbolic program. The resultant data will be generated in standard SDS double-precision fixed- or floating-point format according to the mode of the expression(s) in the operand field (Cf. II. G. 1. v). In the case of DED, only decimal numbers constitute legitimate expressions.

Example:

Label	Operation	Operand	Comment
PI	DED	3.1415926535	FLOATING
AVO	DED	6.023*+23	FLOATING
E	DED	2.7182828*/45	FIXED
LIGHT	DED	1.86*+5*/23	FIXED

Because numeric quantities are restricted to 15 digits in length, the use of "scientific" or "floating-point" notation is preferable to absolute notation (e.g., 0.0000147235821). When both a binary and a decimal scale factor are desired, the decimal scale factor should be specified first.

F. TEXT or BCD (Binary-coded character string)

Format:

Label	Operation	Operand
[[$] LABEL]	TEXT or BCD	E, character string
	or	
[[$] LABEL]	TEXT or BCD	<character string>

The programmer often needs the capability to incorporate within programs output messages in binary-coded form. This may be accomplished by subdividing the message into four-character (24-bit) strings and placing them in the operand field of a DATA directive line. For greater convenience, however, a TEXT directive is provided with which the message may be described independently of the word-size of the target computer.

Using the TEXT directive, the programmer places the character string (not enclosed in quotes) in the operand field and specifies the total message length in one of two ways: in the first, he precedes the character string by a character count, separated from the string by a comma; in the second, he encloses the character string by the characters < and >, respectively. The latter method is more convenient when it is unnecessary to know the length of the string for other reasons; but the former method is necessary when the characters < and/or > will appear within the message. In this case, the value of the expression E must be defined prior to the TEXT line. In both cases, the message is left-justified within the block of computer words allocated to it. Unfilled character positions always contain blanks (060). Note that TEXT and DATA differ in these two respects.

Example:

Location	Contents	Label	Operation	Operand
01000		MSGE	RORG	01000
01000	22232460		TEXT	8, BCD INFO
01001	31452646			
01002	22232460		TEXT	<BCD INFO>
01003	31452646			
01004	22232460		DATA	'BCD ', 'INFO'
01005	31452646			
01006	22232460		TEXT	4, BCD
01007	60222324		TEXT	4, BCD
01010	00222324		DATA	'BCD'
01011	60222324		DATA	' BCD'

Note that the first three lines result in identical code, whereas the last four do not.

The BCD directive is identical to TEXT, except that the 012 character is used for blank. The normal use of BCD, therefore, is to generate messages intended for typewriter or paper-tape output, whereas TEXT is used for all other devices.

G. EQU (EQUals)

Format:

Label	Operation	Operand	Comments
[$] LABEL	EQU	E	LABEL COMPULSORY
	or		
[$] LABEL	EQU	(E1, E2, ..., EN)	LABEL COMPULSORY

Since the directives DATA, DED, and TEXT enable the programmer to centralize and label execution-time data specifications, they contribute to both the readability and flexibility of the symbolic program. For the same reasons, it is frequently desirable to specifiy assembly-time data symbolically; or to use "parametric programming", a technique that is useful whenever a number of symbolic lines are related to one another by their common dependence upon one or more values. Using the parametric approach, the programmer labels the value(s) by an EQU directive and replaces all references to the appropriate value(s) by its (their) symbolic equivalent(s).

The EQU directive usually defines a single datum symbolically, as on the first line appearing above. Since the operand is a general expression, it is possible to pyramid parametric definition. Moreover, any single- or double-precision value may be defined by the EQU directive, whereas in conventional assembly languages, EQU (or its equivalent) can only define symbolic addresses.

Example:

Label	Operation	Operand	Comments
ONE	EQU	1	CHANGING THIS DEFINITION
TWO	EQU	ONE + ONE	WILL CHANGE THE VALUE TWO
PI	EQU	3.1415926535	FLOATING POINT DEFINITION
	DED	PI	

In Meta-Symbol the EQU directive can also define a symbolic list, similarly to the way in which the DATA directive can define a data block.

Example:

Label	Operation	Operand
SQUARE	DATA	1, 4, 9
CUBE	EQU	(1, 8, 27)

The difference is the way reference is made to list items. If LABEL is the label of a DATA block, then the address of the ith element may be symbolically referred to as LABEL +(i-1). If, however, LABEL is the label of a list definition, then the ith list element may be symbolically referred to as LABEL (i). Thus, in the above example,

SQUARE	contains	1
SQUARE +0	contains	1
SQUARE +1	contains	4
SQUARE +2	contains	9
CUBE	has the value	(1, 8, 27)
CUBE (1)	has the value	1
CUBE (2)	has the value	8
CUBE (3)	has the value	27

Note that a list definition must always be enclosed in parentheses. Because of this, it is possible to pyramid list definitions.

Label	Operation	Operand	Comments
OP1	EQU	3	OP1 IS NOT A LIST
OP2	EQU	(3)	OP2 IS A LIST OF 1 ELEMENT
A	EQU	(1, 2)	
B	EQU	(3, 4)	
C	EQU	(A, B)	EQUIVALENT TO ((1, 2), (3, 4))

The fifth line illustrates the case in which elements of the list are lists themselves. Thus:

C(1) is equivalent to (1, 2)

C(2) is equivalent to (3, 4), and

C(1)(1), written C(1, 1), is equivalent to 1

C(1)(2), written C(1, 2), is equivalent to 2

C(2)(1), written C(2, 1), is equivalent to 3

C(2)(2), written C(2, 2), is equivalent to 4

Subscripting to higher levels follows the same rules of parenthetical notation. Lists are primarily useful as they apply to PROCedures and FUNCtions, and additional list notation and examples are provided within the sections describing these two directives. In particular, the concepts of list dimension and symbolic redefinition are explored there.

H. OPD (Operation Definition)

Format:

Label	Operation	Operand	Comments
LABEL	OPD	E	LABEL COMPULSORY

OPD is the counterpart for operations of the EQU directive for values.

Example:

Label	Operation	Operand	Comments
LOC	EQU	3	
LDA	OPD	07600000	
	LDA	LOC	GENERATES 07600003

Thus, while the interpretation of the operand field of an OPD line is identical to that of the single-valued EQU directive, the reference to an OPD-defined symbol is made in the operation rather than in the operand field. Encountering a reference to the OPD-defined symbol, the assembler merges (OR, logical sum) the operation value with the address portion of the operand value. If the second line above had appeared

```
LDA          OPD          07600010
```

then the third line would have generated 07600013

OPD is preserved for compatibility with SYMBOL 4/8. The use of a FORM or a PROC definition offers greater flexibility.

In Meta-Symbol, OPD is implemented by means of a nested PROC definition. Hence, OPD may not be used within a PROC.

I. FORM (FORMat)

Format:

Label	Operation	Operand	Comments
LABEL	FORM	E1, E2, . . ., EN	DESCRIBE FORMAT

It is frequently desirable to pack multiple data within a single computer word. The computer instruction is a typical example: the computer word is divided into operation, address, index, and indirect address subfields. In processing symbolic instructions, the assembler recognizes an implicitly specified subdivision format and, upon translation to binary, packs the instruction accordingly.

The FORM directive enables the programmer to describe completely general computer word subdivisions, and to invoke them simply.

Example:

	Label	Operation	Operand	Comments
	DCHAR	FORM	4, 4, 4, 4, 4, 4	DEFINE DECIMAL SUBDIVISION
	INST	FORM	1, 2, 6, 15[†]	DEFINE INSTRUCTION FORMAT
	X2	EQU	2	
	LDA	EQU	076	
	LOC	EQU	3	
		.		
		.		
		.		
		RORG	01000	
01000	04432126	DCHAR	1, 2, 3, 4, 5, 6	PACK 6 DECIMAL CHARACTERS
01001	27600003	INST	0, X2, LDA, LOC	PACK COMPUTER INSTRUCTION

For SYMBOL, the sum of the operands on the FORM definition line must be equal to the word size (24 bits for SDS 900 Series Computers). For META-SYMBOL, the sum may range between one bit and twice the word size; when the sum is not equal to the word size (or twice the word size), the expressions in the FORM reference line are right justified in the generated single (or double) data word.

The FORM definition must precede all references to it.

J. RELTST (TeST RELocatability)

Format:

Label	Operation	Operand	Comments
L	RELTST	E	

[†]Applicable to META-SYMBOL only. 900 SYMBOL will not handle an address field greater than 14 bits.

A value is assigned to L depending upon the relocatability of E:

Relocatability of Expression E	Value assigned to L
absolute	0
relocatable	1
common relocatable (blank common)	2

K. END

Format:

Label	Operation	Operand	Comments
	END	E	(END OF PROGRAM, PROC OR FUNC)

The END directive indicates to the assembler the end of a PROCedure, FUNCtion, or of an entire program.

When the END line terminates a PROCedure, any expression in the operand field is ignored and need therefore not appear.

When the END line terminates a FUNCtion, the operand field serves to return the FUNCtional value to the functional reference line (Cf., PROC and FUNC).

When the END line terminates a program, the operand field may (but need not) be used to specify the starting address of the program.

L. DO

Format :

Label	Operation	Operana	Comments
(LABEL)	DO	E1	WITHIN PROC, FUNC OR PROGRAM
	or		
(LABEL)	DO	E1 [, E2, E3]	WITHIN PROC OR FUNC ONLY

The DO directive provides for conditional and/or repetitive code or value generation based upon the value of the first expression in the operand field of the DO line. The DO directive is valuable in conjunction with parametric programming (Cf. the EQU directive), since it enables assembly-time decisions to be made and processed.

Normally, the "range" of the DO (the number of successive statements upon which it is active) is a single statement. When used within PROCedures and FUNCtions, however, its capability is extended for action upon multiple lines. The use of this capability is described below.

The simplest use of the DO directive can be illustrated :

Label	Operation	Operand	Comments
	DO	K	
	ADD	C	ACCUMULATE SUM

Encountering this instruction sequence, the assembler generates the ADD instruction K times for $K \geq 0$, and will indicate an error for $K<0$. Thus the following sequence :

K	EQU	3
	DO	K
	ADD	C

results in the generation of three successive ADD instructions.

More typically than the above case ($\sum_K C$), the user desires to generate code to perform $\sum_K C_i$. This capability is provided by the label field of the DO directive, which becomes a dynamic index. Encountering a symbol in the label field of the DO line, the assembler assigns it the initial value zero, which is then incremented by one each time prior to the processing of the following line. Thus the sequence:

I	DO	K
	ADD	C + I

results in the generation of K ADD instructions, $K \geq 0$, which have the successive operands C+1, C+2, . . . , C+K. For $K<0$, an error indication results.

Within PROCedures and FUNCtions, the DO line may have up to two additional operands interpreted as follows.

$E1 > 0$	DO the next E2 lines E1 time(s), then skip E3 line(s).
$E1 = 0$	Skip the next E2 line(s)
$E1 < 0$	Error

When unspecified, the values of E2 and E3 are 1 and 0, respectively, to coincide with the DO that is used externally to PROCedures and FUNCtions.

Note: When counting lines, the assembler includes all symbolic lines including comments lines.

Example 1:

```
I       DO         3,2
        DATA       I
        DATA       I*I
```

generates:

```
        DATA       1
        DATA       1*1
        DATA       2
        DATA       2*2
        DATA       3
        DATA       3*3
```

Example 2:

```
        DO         TYPE < 8, 3, 2
        DATA       5
        DATA       50
        DATA       500
        DATA       17
        DATA       34
```

If TYPE < 8, E1 = 1 and the following is generated.

```
        DATA       5
        DATA       50
        DATA       500
```

If TYPE ≥ 8, E1 = 0 and the following is generated.

```
        DATA       17
        DATA       34
```

Example 3:

```
I       DO         (TYPE < 8) *3, 1, 2
        DATA       5*+ (I-1)
        DATA       17
        DATA       34
```

If TYPE < 8, E1 = 3 and the following is generated.

```
        DATA       5*+ (1-1)
        DATA       5*+ (2-1)
        DATA       5*+ (3-1)
```

If TYPE ≥ 8, E1 = 0 and the following is generated.

```
        DATA       17
        DATA       34
```

Examples 2 and 3 illustrate why E2 and E3 may be reterred to as the "true range" and "false range," respectively.

M. NAME

Format:

Label	Operation	Operand	Comments
LABEL	NAME	[E]	[CALLING NAME]

The NAME directive labels a PROCedure or a FUNCtion definition, enabling it to be called by a PROCedure reference line or a FUNCtion reference item. Just as multiple entries can be created for subroutines, multiple NAME lines can appear within a PROC/FUNC definition. In such cases, it is normally desirable to have the ability to determine internally by what NAME the PROC/FUNC was called. Since only values (not names) can be tested (as, for example, with a DO directive), the programmer may associate different values with the different calling NAMEs. This is accomplished by placing different expressions(usually integers) in the operand fields of the different NAME lines. The use of this feature is illustrated under PROCedures and FUNCtions.

The operand field of the NAME line may also contain an expression list. In this case, the expression list must be surrounded by parentheses.

N. PROC and FUNC (PROCedures and FUNCtions)

Format :

Label	Operation	Operand	Comments
LABEL	PROC		INTRODUCE PROCEDURE DEFINITION
	or		
LABEL	FUNC		INTRODUCE FUNCTION DEFINITION

PROCedures and FUNCtions are bodies of code analogous to subroutines, but which are processed at assembly time rather than at execution time. Introduced by a PROC/FUNC directive, the coding sample is always terminated by an END directive. Used without the DO directive, the PROCedure is similar to the simpler "macro", in which a single line of code (the reference line) is replaced by one or more lines specified in a macro definition. Used together with the DO, however, the PROCedure provides a more powerful capability than simple line replacement. This capability is illustrated in examples which follow.

The FUNCtion, like the PROCedure, is a generator; whereas the PROCedure generates code, and is invoked by placing its name in the operation field of a line, the FUNCtion generates values and is invoked by placing its name in the operand field.

The PROCedure or FUNCtion definition must always precede the first reference line.

The following examples of an ordinary macro are provided for illustrative purposes only. There is no MACRO directive in META-SYMBOL.

Label	Operation	Operand	Comments
MOVE	MACRO	A, B	MACRO DEFINITION
	LDA	A	
	STA	B	
	END		
	.		
	.		
	.		
	MOVE	C, D	MACRO REFERENCE LINE

The macro definition defines an instruction sequence in terms of dummy parameters A and B that appear on the MACRO definition line. Encountering the MOVE line, the assembler generates the LDA, STA sequence, but replaces the dummy parameters A and B by the reference parameters C and D. The macro is said to operate on a "call by name" principle.

The PROCedure operates, on the other hand, on a "call by value" basis.

Example :

```
 Label      Operation  Operand               Comments

  P         PROC                         PROCEDURE DEFINITION
 MOVE       NAME
            LDA         P(1)
            STA         P(2)
            END
             .
             .
             .
            MOVE        C, D             PROCEDURE REFERENCE LINE
```

In this case, the reference parameters are named implicitly in terms of the symbol P that appears in the label field of the PROC line. They are evaluated before the PROC/FUNC is processed, and it is only these values, not their names, that can be determined within the sample. If the PROCedure reference lines were :

```
            OP          E1, E2, . . . , EN
```

then the correspondence between the symbol P and the parameters E1, E2, . . . , EN exists as though the reference line had been :

```
     P          EQU             (E1, E2, . . . , EN)
```

Thus,

P(1) has the value E1

P(2) has the value E2

.
.
.

P(N) has the value EN

Note that the reference parameters constitute a list even though they are not enclosed in parentheses.

If any of the parameters E1, E2, . . . , EN is in turn a list, the elements can be referred to by subscripting further the symbol which appears in the PROC line.

Example :

```
 Q      PROC                          }    PROC DEFINITION
 OP     NAME                          }
         .                            }
         .                            }
         .                            }
        END                           }
         .
         .
         .
          OP   (A, (B, C), (D, (E,F)))     PROC REFERENCE
```

This reference line contains only one operand, viz., Q. Thus,

Q	= ((A, (B, C), (D, (E,F))))	list of one element
Q(1)	= (A, (B, C), (D, (E,F)))	list of three elements
Q(1, 1)	= A	not a list unless A is
Q(1, 2)	= (B, C)	list of two elements
Q(1, 2, 1)	= B	not a list unless B is
Q(1, 2, 2)	= C	not a list unless C is
Q(1, 3)	= (D, (E, F))	list of two elements
Q(1, 3, 1)	= D	not a list unless D is
Q(1, 3, 2)	= (E, F)	list of two elements
Q(1, 3, 2, 1)	= E	not a list unless E is
Q(1, 3, 2, 2)	= F	not a list unless F is

It is frequently desirable that the PROC/FUNC definition be written without restricting the list structure of the PROC/FUNC reference line, although the list structure of the reference line must be determinable within the definition. Notationally, this problem is resolved by the convention that, if L is the name of a list, then : L has the value "the number of elements in the list L". Thus, in the above example :

:Q	=1
:Q(1)	=3
:Q(1, 1)	=0
:Q(1, 2)	=2
:Q(1, 2, 1)	=0
:Q(1, 2, 2)	=0
:Q(1, 3)	=2
:Q(1, 3, 1)	=0
:Q(1, 3, 2)	=2
:Q(1, 3, 2, 1)	=0
:Q(1, 3, 2, 2)	=0

In general, there are two additional quantities of interest within a procedure : the identity of the operation on the calling (reference) line and the knowledge whether any of the reference operands was indirectly addressed.

Example :

Label	Operation	Operand	Comments
P	PROC		
LDA	NAME	076	900 SERIES LOAD/STORE
LDB	NAME	075	INSTRUCTION SET
LDX	NAME	071	
STA	NAME	035	
STB	NAME	036	
STX	NAME	037	
INST	FORM	3, 6, 1, 14	
	INST	P(2), P(0), P(*1), P(1)	
	END		
	.		
	.		
	.		
	LDA	100	GENERATES 07600144
	STA	*0200, 2	GENERATES 23540200

The above example illustrates how several reference lines may invoke the same PROCedure. In the first case, the INST line will generate an 076 for the six-bit instruction code, since the operation field on the calling line corresponds to the label field of the first NAME line, which, in turn, contains the value 076 in its operand field. For the same reasons, the INST line generates an 035 for the six-bit instruction code of the STA line. The correspondence is established via the subscripted symbol P(0), which stands for the value on the NAME line whose label field agrees with the operation field of the reference line.

The example also illustrates how a procedure may determine whether or not a reference parameter was indirectly addressed: If the reference parameter P(i) was indirectly addressed (preceded by an asterisk), then the item P(*i) will have the value 1; otherwise, P(*i) will have the value 0.

More generally, if P(E1, E2, . . . , EN) is a subscripted symbol, then the subscripted symbol flag corresponding to this item, written P(E1, E2, . . . , *EN), has the value 1 if an asterisk preceded the expression that defined this item. Otherwise, the subscripted symbol flag has the value 0. Note that the subscripted symbol flag corresponding to an element is notationally indicated by an asterisk preceding the last subscript of the element.

Normally, the programmer does not have to make this identification. The MOVE PROCedure, for instance, generates correct code regardless of whether one or both of the reference parameters is indirectly addressed. This is true because the MOVE PROC invokes the LDA/STA PROC, which does make the determination. Since all META-SYMBOL instructions are defined by PROCs, indirect address determination by the programmer is necessary only when he uses non-machine instructions defined by himself. Note the implication that PROCs may be defined and/or called within other PROCs, which is illustrated in the third example below.

Example:

Label	Operation	Operand	Comments
LOAD	OPD	07600000	
STORE	OPD	03500000	
P	PROC		
MOVE	NAME		
	DO	P(*1), 1, 1	
	LOAD	*P(1)	
	LOAD	P(1)	
	DO	P(*2), 1, 1	
	STORE	*P(2)	
	STORE	P(2)	
	END		

In this example, indirect address determination is necessary because LOAD and STORE are defined by OPD and not by instruction PROCs. By the above means, all the attributes of the operation and operand fields at the reference line can be determined and tested within a PROC. It is also useful to operate within procedures on the contents of the label field of the reference line.

Example:

It is desired to define two procedures, one a BSS (Block Started by Symbol) PROC, and one a BES (Block Ended by Symbol) PROC:

Label	Operation	Operand	Comments
P	PROC		
BSS	NAME	0	
BES	NAME	1	
	DO	P(0), 2, 1	DO NEXT 2 FOR BES
	RES	P(1)	RESERVE P(1) LOCATIONS
$	RES	0	THEN DEFINE SYMBOL
$	RES	P(1)	BSS IDENTICAL TO RES
	END		
	.		
	.		
	.		
	RORG	0	
LABEL 1	BSS	1	LABEL 1=0, BUT
LABEL 2	BES	1	LABEL 2=2

Normally, when the reference label is not manipulated within the PROCedure, it is equated to the value of the location counter when the PROCedure is called. A lone dollar sign placed in the label field of a line within a PROC, however, has the effect of "postponing" the definition of the reference label from the beginning of the PROC to the processing of the $-labeled line. In no other case can a dollar sign appear alone in the label field of a symbolic line.

The following example illustrates an interesting use of this feature in a nested PROC. The example, which is the OPD PROC as it actually appears in the system, shows again how a PROC may be used to simulate a directive.

Label	Operation	Operand
P	PROC	
OPD	NAME	
A	EQU	P(1)*/(-21)**7
B	EQU	P(1)*/(-15)**077
C	EQU	P(1)*/(-14)**1
D	EQU	P(1)**037777
Q	PROC	1
$	NAME	(A, B, C, D)
Z	EQU	Q(0)
I	FORM	3, 6, 1, 14
	DO	Z(4)=0, 1, 1
	I	Z(1)++ Q(2), Z(2), Z(3)++Q(*1), Q(1)
	I	Z(1)++Q(2), Z(2), Z(3)++Q(*1), Q(1)++Z(4)
	END	
	END	

When an OPD line is encountered, the OPD PROC is processed, resulting simply in the definition of another PROC, which takes its NAME from the label field of the OPD line. This PROC in turn is processed when its reference line is encountered.

There is one restriction on the nested PROC: All NAME lines in the internal PROC must be rendered external by the appearance of a "$" alone or preceding the symbol in the label field, therefore, PROC definitions may be nested only one level.

The example also illustrates a use of a one-pass PROC, conveyed to the assembler by the appearance of the value 1 in the operand field of the PROC line. When, as in the preceding examples, the operand field of the PROC line is vacant, the assembler performs a two-pass "assembly" on the PROC when the reference line is encountered. This is necessary whenever a PROC contains an internal forward reference, but unnecessary otherwise. For example, the following PROC can be changed to a "PROC 1" only if the reference to the symbol A is replaced by a reference to $+2.

Label	Operation	Operand	Comments
P	PROC		
SUM	NAME		
	BRU	A	BRANCH AROUND RESULT
DUMMY	RES	0	LABEL RESULT WITH
$	RES	1	REFERENCE LINE LABEL
A	LDA	P(1)	
I	DO	:P-1	SUM REFERENCE PARAMETERS
	ADD	P(I+1)	
	STA	DUMMY	STORE RESULT
	END		

Since FUNCtions only generate values, and do not influence storage allocation, they are always processed in one pass.

Because of the flexible list structure, it is possible to write very general PROCedures and FUNCtions where the operands can be indexed and/or indirectly addressed. They may, in addition, be literals (Cf. V. C) as the following example illustrates:

Label	Operation	Operand	Comments
P	PROC	1	
ATANF	NAME		
	DO	:P(1)=0, 1, 1	
	LDF	P(1)	
	LDF	P(1, 1), P(1, 2)	
	BMA	ATAN	
	DO	:P(2)=0, 1, 1	
	PZE	P(2)	
	PZE	P(2, 1), P(2, 2)	
	END		
	.		
	.		
	.		
	ATANF	(*ARGS, 2), =1.0	

The code generated by the PROC reference line is equivalent to:

```
LDF     *ARGS,2
BMA     ATAN
PZE     =1.0
```

It can be inferred from this example that PROCedure/FUNCtion NAMEs are defined externally to the PROC/FUNC sample. However, no symbols appearing on other than NAME lines are defined externally unless they are preceded by a dollar sign. Similarly, any PROC/FUNC may refer to symbols defined externally to it. In cases of conflict (where the same symbol is defined both externally and internally with respect to a PROC/FUNC sample), the ambiguity is resolved in favor of the innermost (internal) definition level. However, this convention applies only within the sample, and cannot affect the reference line.

Example:

Label	Operation	Operand	Comments
X	EQU	3	
P	PROC	1	PROC DEFINITION
LOADX	NAME		
X	EQU	2	
	LDX	P(1), X	
	END		
	.		
	.		
	.		
	LOADX	X	PROC REFERENCE

The PROCedure reference line will become LDX 3, 2.

The writing of FUNCtions follows the same rules as PROCedures except that:

1. The FUNCtion call occurs in the operand field of the reference line and not in the operation field.
2. The function generates a value—not code—and only nongenerative lines may be used within FUNCtions.
3. The reference label symbol, $, has no meaning within a FUNCtion.

Example: The following FUNCtion will determine the maximum or the minimum of two given arguments.

Label	Operation	Operand
F	FUNC	
MAX2	NAME	1
MIN2	NAME	0
	DO	F(0)--(F(1)<F(2)), 1, 1
M	EQU	F(1)
M	EQU	F(2)
	END	M

A general MAX/MIN FUNCtion can now be written:

Label	Operation	Operand
G	FUNC	
MAX	NAME	1
MIN	NAME	0
M	EQU	G(1)
N	DO	:G-1, 3
	DO	G(0), 1, 1
M	EQU	MAX2(M, G(N+1))
M	EQU	MIN2(M, G(N+1))
	END	M

This example illustrates the use of symbolic redefinition, which is permissible only when none of the multiple definitions equates the symbol to a relocatable quantity. Either a symbol or a list may be redefined as a symbol (list), but a list may not be redefined as a non-list.

The entire FUNCtion could instead have been written recursively:

Label	Operation	Operand
F	FUNC	
MAX	NAME	1
MIN	NAME	0
	DO	F(0)--(F(1)<F(2)), 1, 1
M	EQU	F(1)
M	EQU	F(2)
N	DO	:F-2, 3
	DO	F(0), 1, 1
M	EQU	MAX(M, F(N+2))
M	EQU	MIN(M, F(N+2))
	END	M

Evidently, a FUNCtion can be written to execute any computation that can be stated algorithmically.

From the foregoing, we can define four new quantities:

	Quantity	Definition	Example
1.	List Dimension	The dimension of a list is the number of elements contained within the list	:L
2.	Subscripted Symbol Flag	The subscripted symbol flag corresponding to a subscripted symbol is notationally identical to the element, but with an asterisk preceding the last subscript.	P(2, *3)
3.	Reference NAME Value	The reference NAME value is the value of the expression on the NAME line summoned by a PROC/FUNC reference line	P(0)
4.	Reference Label Symbol	The Reference Label Symbol represents the symbol that appears in the label field of the PROC reference line	$

Of these quantities, all but the last are items. The last two have meaning only within PROCedures and FUNCtions.

O. PAGE (Eject PAGE)

Format:

Label	Operation	Operand	Comments
	PAGE		(EJECT PAGE)

When the PAGE directive is encountered, the assembler causes a page eject to occur on the output listing medium. The PAGE line is the first line on the new page.

P. DISP (DISPlay)

Format:

Label	Operation	Operand	Comments
	DISP		

The DISP directive is used within a procedure. If during the "expansion" of a procedure a DISP directive is encountered, those statements encountered in expanding the remainder of the procedure are listed in a format similar to the lines of the main program code. The two listings differ in that for those produced as the result of DISP line numbers are not given and lines skipped under control of DO statements are not displayed.

DISP governs the display of the internal structure of only those procedures in which it occurs. Thus, if a procedure containing DISP calls a procedure that does not contain its own DISP directive, the called procedure will not be displayed.

The DISP directive is ignored when encountered in a function or a procedure called by a function. It is also ignored when encountered in a procedure that is being executed under the influence of a main program DO. An M flag is given in the latter case (see "Error Flags" in Section VII.)

The DISP directive is invaluable both for debugging procedures and as a tutorial device.

Q. INHD (INHibit Display)

Format:

Label	Operation	Operand	Comments
	INHD		

The INHD directive causes the assembler to ignore all succeeding DISP directives.

After a set of procedures has been debugged, an INHD may be inserted early in the main program to cancel the effect of all subsequent DISP directives. Then if further errors occur, removal of that one statement allows complete display for continued debugging. After full confidence in the procedure is gained, the DISP directives may be removed from the individual procedures.

R. SUPR (SUPpRess octal listing of binary output)

Format:

Label	Operation	Operand	Comments
	SUPR		

The SUPR directive is used within a procedure to suppress the octal listing of the binary output. It does not suppress the first generated work. Its primary use is to provide more compact listings for procedure-mechanized higher level languages, where the user is not expected to be interested in the mechanization. The action of the SUPR directive may be overridden by the INHS (inhibit suppression) directive.

A side effect of an active DISP directive within a procedure containing an active SUPR directive is to neutralize the effect of the SUPR directive. However, a DISP directive is not the converse of the SUPR directive.

S. INHS (INHibit Suppression)

Format:

Label	Operation	Operand	Comments
	INHS		

The INHS directive causes the assembler to ignore succeeding SUPR directives for the remainder of the program.

T. MARK (insert character in flag region on listing)

Format:

Label	Operation	Operand	Comments
	MARK	E	

The MARK directive causes the last six bits of the expression in the operand field to be inserted in the flag field of the next line to be listed. The primary purpose of MARK is to provide the procedure writer with the capability to flag possible errors in the use of the procedure.

The MARK directive can generate only one flag per listed line. Thus, the use of a MARK directive before a preceding MARK has generated its character causes that first character to be lost.

If any flags are waiting to be listed when the assembler processes a procedure END line, they are listed on an otherwise blank line. This increases the clarity of flagging within the assembly.

U. SBRK (Set BREAK1)

Format:

Label	Operation	Operand	Comments
	SBRK	E	

SBRK causes the assembler to modify, at assembly time, its own working storage memory allocation scheme. The need for this ability and its effect on memory allocation are described in Section VIII.

SBRK should be the first non-comment line in the program. It must be used before the first external reference and before the first procedure reference. An illegal use of SBRK will be flagged with an M and ignored.

V. SIOR (Set special I/O Relocation)

Format:

Label	Operation	Operand	Comments
	SIOR		

The next form reference line encountered will have the special I/O relocation bit set if its address is load relocatable. This directive is used within the IORD, IORP, IOSD, IOSP, and IOCT procedures to mechanize special I/O relocation.

V. ADDITIONAL PROGRAMMING FEATURES

A. Comments Line

Format:

Label	Operation	Operand
*THIS IS A COMMENTS LINE		

Whenever an asterisk introduces a symbolic line, the assembler ignores its contents. Such lines are used to annotate the source program, and are preserved on the output listing. The comments line may contain a maximum of 63 characters beginning with the first non-blank following the asterisk. Additional characters are discarded by the Encoder.

B. Free Form and Continuation

SYMBOL and META-SYMBOL provide for free-form symbolic lines; that is, it is unnecessary to begin each field at a prespecified column of the source input record. The rules for writing such a record are:

The label field begins in column one.

A blank character terminates any field.

One or more blanks written at the beginning of a line specify that there is no label field.

A maximum of 15 blanks may be written following a symbol in the label field to specify that the next field is absent.

More than eight blanks written following a symbol in the operation field specify that the next field is blank.

When the input record contains 80 columns (that is, a card), the assembly processor ignores columns 73-80 and terminates the physical record at column 72. However, when the operand field contains a list, the list may continue to subsequent physical records. The user specifies continuation by ending each to-be-continued operand field with a comma followed by all blanks to column 72. Such continuable lists include a list definition line or a PROC/FUNC reference line.

Example:

```
LABEL           OPERATION            OPERAND
1   5           10      15      20   25   30   35
LOAD LDA TEMP (COMMENT)
 STA TEMP (COMMENT)
LOAD1              LDA                (COMMENT)
LIST     EQU                (1,2,
                             3,4)
                MOVE        LOC1,
LOC2
```

C. Literals

Format:

Label	Operation	Operand	Comments
[[$] LABEL]	OP	=E	OPERAND SPECIFIED BY VALUE, NOT NAME

In a typical program, machine instructions serve two basic purposes: they operate on variables and on constants. When operating on variables, the location of the variable is important, but its value is unknown until tested. When operating on constants, the converse is generally true in that only the value of the operand is important.

Symbolic programming facilitates the representation of both types of operation. Operands of the first category (variables) can be given symbolic names (such as X, ALPHA, etc.), and can be referred to by these names throughout the symbolic program. For operating with constants, however, it is generally desirable to refer to the constant by value rather than by name; literals provide this capability.

In order to use literals, the programmer writes the value of the expression, rather than a name, in the operand field of the symbolic line, and precedes the expression by an equals sign (=). Detecting the leading equals sign, the assembler computes as usual the value of the expression that follows, but it then stores this value in a literal table which it constructs following the program. The address portion of the generated instruction is then made to refer to the literal table entry rather than to contain the value of the computed expression.

Examples:

Location	Contents	Label	Operation	Operand
00144			RORG	100
00144	07600152	TENS	LDA	=010
00145	07600153		LDA	=10
00146	07600154		LDA	='10'
00147	07600155		LDA	=TENS
00150	07600156		LDA	=010*/15+TENS+10
00151	07600152		LDA	=1*8
			END	
00152	00000010			
00153	00000012			
00154	00000100			
00155	00000144			
00156	01000156			

As shown in this example, the processor detects the multiple equal values (010=1*8) and enters them only once into the literal table.

D. External Definitions and References

One of the most powerful features of SYMBOL and META-SYMBOL is the provision for separate assembly of interdependent programs. This feature not only permits programs to refer by name to standard library programs such as subroutines, but it also allows large programs to be segmented, without in either case shifting the burden of memory allocation to the programmer. As a result, considerable economies accrue both in reduced assemblies and in debugging.

Symbolic inter-program communication is achieved by means of <u>external labels</u>. Most labels are <u>internal</u> (or local) labels in that they are defined only internally to a program. This means that the assembler recognizes a symbolic reference in the operand field of a line only when the symbol is defined elsewhere in the program by its appearance in the label field of a line. When a symbolic reference cannot be satisfied within a program, references to the symbol are said to be <u>external references</u> (that is, the symbol is assumed to be defined within some context external to the program in which the symbolic reference occurs).

The counterpart of the external reference is the external definition; a symbolic definition is made external by preceding it by a dollar sign ($). The programmer may establish an external definition either on the line that defines the symbol, or on a subsequent line. In the latter case, where the entire line is simply an external definition line, it is possible to define additional symbols as external by listing them following the first symbol. Although additional dollar signs are not required, commas must separate one symbol from another.

External references may appear only in the address field of an instruction or FORM reference line. External definitions and references are restricted to six characters in SYMBOL and to eight in META-SYMBOL. Relative external references (e.g., Symbol±n) are not permitted.

Example:

Label	Operation	Operand	Comments
X	EQU	01000	ADDRESS OF DATA
$START	LDA	X	
	BRM	SIN	EXTERNAL REFERENCE
	STA	SINX	
	LDA	X	
	BRM	COS	EXTERNAL REFERENCE
	STA	COSX	
$SINX	DATA	0	EXTERNAL DEFINITION
$COSX	DATA	0	EXTERNAL DEFINITION
	END	START	
SIN	DATA	0	SIN ENTRY
	⋮		
COS	DATA	0	COS ENTRY
	⋮		
$SIN,COS			
	END		RENDER DEFINITIONS EXTERNAL

The above example illustrates both methods of external definition. In the first program, their definition lines make SINX and COSX external. In the second, SIN and COS are made external after their definition.

As indicated above, program segmentation may be useful for maintenance or debugging reasons. Segmenting can also be used to facilitate the assembly of programs which contain large numbers of symbols. For especially large programs, the situation may arise that the number of symbols used in a program overflows the capacity of the assembler's symbol table (approximately 250 symbols for a 4K 900 Series SDS Computer). In this event, segmentation can be accomplished mechanically in the following manner:

1. The program is divided into as many physical segments as desired. Each of these segments is separately assembled.
2. A set of external definition lines is prepared from the external reference lists output at the end of each assembly listing. Relative external references are eliminated.
3. The set of external definition lines is duplicated for each program segment and included before each END card.
4. The program segments are reassembled. The loader can now fulfill all external references.

To communicate external definition and reference information to the loader, the assembler outputs the former prior to the binary output (called "text") and the latter following the text. The external definition table consists of the alphanumeric symbols accompanied by their (relocatable or not) binary values . Each entry in the external reference table consists of the alphanumeric symbol accompanied by the (relocatable or not) binary address of the last location in which the external reference occurred. The address portion of that location will point, in turn, to (contain the address of) the last previous location where an external reference was made to the same symbol. The chain terminates when the address portion of an instruction contains 0.

Example:

Location	Contents	Label	Operation	Operand
00100			ORG	0100
00100	07600000	START	LDA	X
00101	07500000		LDB	Y
00102	03500101		STA	Y
00103	03600100		STB	X
00104	07600103		LDA	X
00105	06400104		MUL	X
00106	03500107		STA	XSQ
00107	00000000	XSQ	DATA	0
			END	START
00105	X			
00102	Y			

Consulting the external reference information, which appears following the END line on the assembly listing, the programmer can easily find all references to the external label by threading his way backward through the listing. As a result, octal corrections for undefined symbols can be made more reliably than with conventional assemblers. However, octal corrections are seldom required, since it is simpler to assemble a separate "program" consisting solely of lines to define the symbols.

E. Relocation

Particularly because of program segmentation capability, it is normally desirable to assemble a symbolic program without being required to allocate the program to any particular memory area or starting location. When a program is written such that it can be executed independently of its origin (that is, independently of where it is physically located within the computer), the program is said to be relocatable. All instructions are relocatable that are not affected by an AORG directive (see IV. B.).

All decimal and octal numbers are clearly non-relocatable. Assuming the absence of an AORG directive, all symbols, however, are relocatable that are not equated to a non-relocatable expression by an EQU directive. As a symbol, $ is always relocatable.

When an expression consists of at least one relocatable item, the expression is:

1. Relocatable if R, the sum of the added relocatable items minus the sum of the subtracted relocatable items, is equal to 1, and non-relocatable if R=0.

2. Illegal if R≠0, 1 or if the expression involves any operations other than addition and subtraction upon two relocatable items.

Example:

Label	Operation	Operand	Comments
R1	DATA	0	
R2	DATA	0	
NON	EQU	1	
A	EQU	R1+R2	ILLEGAL
B	EQU	R1-R2	NON-RELOCATABLE
C	EQU	R1+NON	RELOCATABLE
D	EQU	R1*NON	RELOCATABLE
E	EQU	R1*R1	ILLEGAL
	END		

The assembler provides relocation information in the text section of the binary output. Detecting a relocation flag for any instruction, the loader adds a bias (the loading origin) to the address portion of the instruction. Further details concerning the binary format are available in Appendix K.

F. Concordance Listing

The 900 Series META-SYMBOL system has been extended to include an optional program concordance listing. The option is selected by the presence of the parameters CONC or EXCP in the MONARCH METAXXXX control message. The use of CONC results in a standard concordance being generated; the use of EXCP results in a concordance being generated with exceptions from the standard. The exceptions from the standard concordance must be specified on INCLUDE or EXCLUDE cards.

The standard concordance includes all symbols that occur in the user's program except:

1. Operation codes.
2. Symbols appearing as part of a function or procedure sample unless the symbols, including procedure or function names, are available for reference by code not occurring within any PROC or FUNC.

The format of the concordance listing is

T DLN SYMBOL RLN RLN RLN RLN RLN

where:

T is the symbol type code:

Code	Interpretation
A	absolute
R	relocatable
*	undefined
$	externally defined
O	operation code
L	list

DLN is the line number of the definition.

SYMBOL is the user's symbol. Symbols are listed in alphanumeric sort sequence with the collating sequence:

$_\wedge$(blank), 0 through 9, A through Z,\

RLN are the line numbers on which the symbol is referenced. Reference line numbers appear in ascending sequence for each symbol.

When a concordance is to be generated with the exceptions from the standard, the user must supply INCLUDE and/or EXCLUDE records specifying the exceptions. (The INCLUDE and/or EXCLUDE records must be followed by an end-of-file record, △EOF.)

EXCLUDE records must precede the INCLUDE records. The format of these records is:

$\wedge$EXCLUDE$\wedge$SYM, SYM, SYM, ..., SYM$\wedge$

$\wedge$INCLUDE$\wedge$SYM, SYM, SYM, ..., SYM$\wedge$

or

$\wedge$EXCLUDE$\wedge$ *ALL$\wedge$

$\wedge$INCLUDE$\wedge$ *ALL$\wedge$

where

$\wedge$ represents one or more blanks. Note that $\wedge$ is the only legal terminator.

SYM represents:

1. In the case of EXCLUDE, the specific symbols to be excluded from the concordance.
2. In the case of INCLUDE, the specific symbols to be included in the concordance (this enables the user to specify symbols that would not be included in a standard concordance).

*ALL specifies:

1. In the case of EXCLUDE, no symbol is to be listed unless it appears on a subsequent INCLUDE.
2. In the case of INCLUDE, every symbol in the user's program is to be listed regardless of where it appears in the code unless it is present on a previous EXCLUDE.

If a symbol is both excluded and included, the exclusion takes precedence.

Examples:

$\wedge$EXCLUDE$\wedge$ P, X2, X0, A, B, LDA$\wedge$

$\wedge$INCLUDE$\wedge$ *ALL$\wedge$

$\triangle$EOF.

In this case all mnemonic codes, labels, and symbol references will be listed except those indicated by the EXCLUDE record.

$\wedge$EXCLUDE$\wedge$ *ALL$\wedge$

$\wedge$INCLUDE$\wedge$ BRX, LDX, STX$\wedge$

$\triangle$EOF.

This would result in only three symbols (BRX, LDX, and STX) appearing in the concordance.

The concordance subroutine takes the exception records from the symbolic input device. It is assumed that the unit assigned as X2 is available as a scratch tape. The user's program is scanned from the intermediate output tape X1. The concordance is produced on the device assigned for listing output. All assignments must be made prior to calling META-SYMBOL.

G. System Procedures

The user is not restricted as to the number of system procedure decks he may have in the procedure library on the system tape. He is free to add specialized procedures to the META-SYMBOL library either as modifications to an existing system procedure deck or as an entirely new segment on the MONARCH or MONITOR system tape.

Since system procedures are handled somewhat differently than procedures occurring in the user's program, caution must be exercised in putting a group of user procedures with the standard system procedures for a given machine. For instance, system procedures are selectively loaded by the Preassembler. Therefore, if a new system procedure is to use any of the other system procedures, it must precede those other procedures in the system procedure deck. Also, only the parts of the system procedure deck that are within the scope of procedure lines are processed by the Preassembler. Thus, while a simple FORM definition line, external to all the procedures using it, will suffice at the user-program level, such a FORM must be internal to all these procedures at the system-procedure level. Also, a name line in a system PROC may not have a list in the operand field.

MONARCH

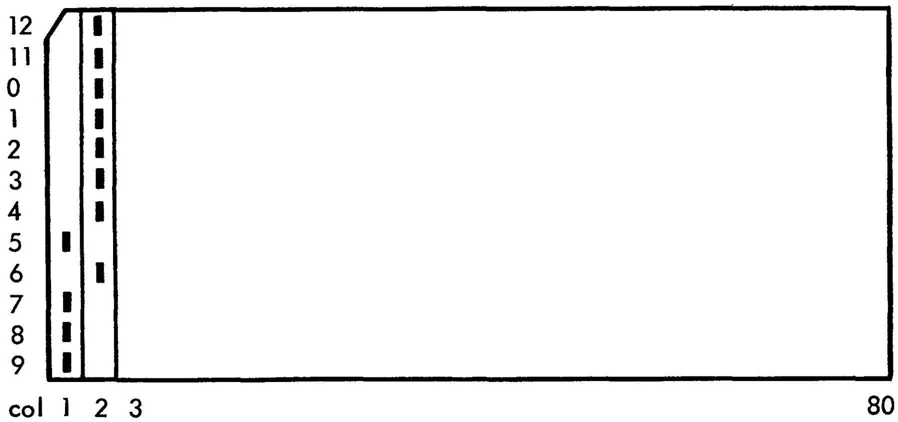
Each system procedure is inserted on the MONARCH tape between PREASSEM and SHRINK. It must be preceded by its △2 MONARCH identification record and its machine series identification record.

The MONARCH ID record has the following format:

△2 PROCXXXX

Column	Contents
1	△ (delta)
2	2 (identifying a level 2 record)
3 - 8	Blank
9 - 16	The letters PROC occupy columns 9 through 12. The four alphanumeric characters that identify the procedure occupy columns 13 through 16. These four characters are used in place of XXXX in a MONARCH METAXXXX control message. Blank is a terminator.
17 - 25	Blank
26 - 72	Comments

The machine series identification record consists of a 2-word, encoded record of the following configuration:



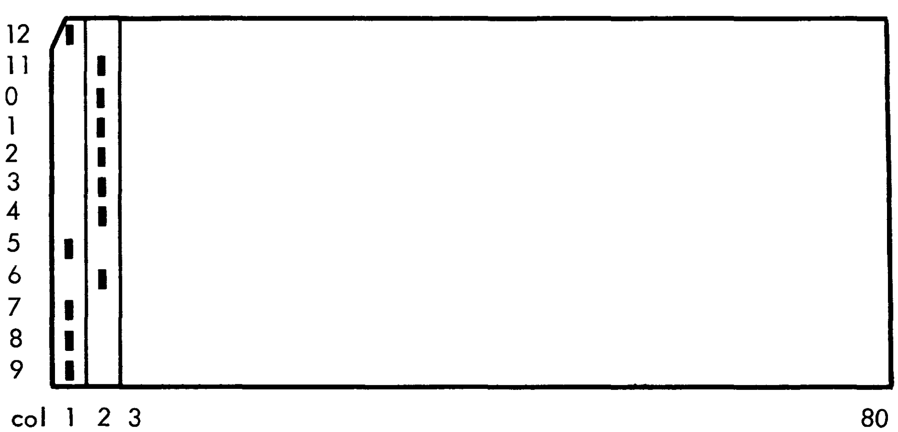
Column	Row	Contents
1	12	Punched if the object code is to be run on a 9300 Computer.
		Not punched if the object code is for a 900 Series machine.
	11 - 0	Blank
	1 - 6	Word count (This value is always 2, indicating a 2-word record.)
	7 - 9	Punched to indicate an encoded card.
2	12 - 9	Checksum for the card. (Each row contains the opposite – punch or non-punch – of the same row in column 1.)
3 - 72		Blank

Example:

To add the procedure deck, labelled CUBE, for 9300 machines to the META-SYMBOL system procedure library, the user must prepare an encoded deck for the procedures, a MONARCH level 2 ID record:

```
△2      PROCCUBE.
        column 9
column 1
```

and the machine series identification record:



The procedure is inserted on the system tape via the System Update Routine (see the SDS MONARCH Reference Manual, publication number 90 05 66). It is called into core with a MONARCH control message:

△METACUBE $P_1, P_2, P_3, P_4, P_5, P_6$.

See the MONARCH Reference Manual for an explanation of the parameters for this control message.

MONITOR

Changing a MONITOR system tape is the function of System-Make, a free-standing program. A description of System-Make is contained in SDS Library Program, Catalog Number 860692.

VI. COMPATABILITY: SYMBOL/META-SYMBOL

The preceding sections described the programming language as though it were identical in SYMBOL and META-SYMBOL. Actually, META-SYMBOL requires a larger hardware configuration than SYMBOL, and SYMBOL does not therefore include all of the features of the meta-assembler. Aside from the lack of PROCedures and FUNCtions, these differences are slight, and it is entirely possible to write programs in a common subset of the language.

The differences between SYMBOL and META-SYMBOL are:

1. In META-SYMBOL, symbols may be from 1 to 15 characters in length. External definitions may not exceed 8 characters in length.

   In SYMBOL, no symbol may exceed 6 characters in length.

2. Symbol does not include the Boolean operators >, =, and <.

3. SYMBOL does not include PROCedures and FUNCtions. Therefore, it does not include the following directives:

   PROC
   FUNC
   NAME
   DO

   It also does not include lists (Cf. IV. G.).

4. In SYMBOL, the sum of the expressions in the operand field of a FORM definition line must be equal to the number of bits in a single computer word. In META-SYMBOL, the sum may have any value between 1 and twice the word size (in bits). Double precision is also excluded in literals and as operands of a SYMBOL EOM line.

5. In META-SYMBOL, but not in SYMBOL, an OPD line may override a system definition.

VII. COMPATIBILITY: 92 SYMBOL/META-SYMBOL

92 SYMBOL is a 1-pass assembly program for the SDS 92. It operates on a minimal SDS 92 with 4K memory and a Teletype, model 35ASR, and processes a language which is very similar to SYMBOL and META-SYMBOL for other SDS computers. For this reason, the 92 SYMBOL language will be defined in terms of compatibility with META-SYMBOL.

A. Symbols

A symbol is a string of from one to eight alphanumeric characters of which the first is alphabetic. Operation symbols (instruction mnemonics, directives, etc.) are restricted to four characters.

92 SYMBOL provides for the definition, and possible subsequent discarding, of local symbols which retain value only within a certain region of the program. (See B.3, below.) A local symbol is a symbol preceded by the character \ (internal code 076); for example:

```
\TEMP 1
```

The current location counter, indexing, indirect addressing, and literals (which are immediate) are indicated as in 900 Series/9300 SYMBOL.

B. Directives

Data Generation	Assembler Instruction
DATA	AORG
TEXT	RORG
BCD	RES
	DO
Value Declaration	REG
	DEF
EQU	REF
FORM	PAGE
OPD	END

1. DED and floating-point items are not implemented.

2. The DO directive is used to process a line a given number of times. The expression in the operand field indicates the number of times the line is to be processed.

3. REG is used to declare the beginning of a local symbol region. When this line is encountered, all currently defined local symbols are discarded. New local labels may now be defined which will not conflict with previous local symbols.

Example:

	RORG 0100	
TEMP1	RES 1	TEMP1 is a valid symbol.
\TEMP1	RES 1	\TEMP1 is a local symbol and is distinct from TEMP1.
	REG	All previous local symbols purged.
\TEMP1	RES 1	\TEMP1 redefined as present location counter, a value which it will retain until next REG directive.

4. DEF is used to declare external symbols. The symbols to be made external are listed as operands. All such symbols must have been previously defined, non-local symbols. The DEF line is analogous to the $ line in META-SYMBOL.

Example:

Label	Operation	Operand
	DEF	ALPHA, BETA

as opposed to

$ALPHA, BETA

in META-SYMBOL.

5. REF is used to declare explicitly external references. All external references and undefined symbols are published at the end of the assembly; undefined symbols are preceded by a U diagnostic. The REF line must precede the first external reference.

Example:

Label	Operation	Operand
	REF	ALPHA, BETA

6. The OPD directive has two operands separated by commas. The second operand may have the value 6 or 12 to define the field size in which the OPD is effective.

Second Operand	Interpretation
6	The OPD-defined symbol is treated similarly to a computer instruction (e.g., LDA).
12	The subsequent reference line causes a 12-bit, single word to be generated. The value of the OPD definition line is added to the value of the reference line operand.

The OPD definition line must precede all references. All OPD lines must precede the first local symbol definition. An OPD line may not override a system definition.

7. The syntax for TEXT and BCD lines is

```
TEXT/BCD     character count, string
```

The option

```
TEXT/BCD     <string>
```

is not implemented.

8. PROC, FUNC, NAME, FORT2, FORT4, DISP, INHD, SUPR, INHS, MARK, SBRK, RELTST, and SIOR are not implemented.

9. All other directives are implemented as in META-SYMBOL. The sum of the operands on a FORM line must equal 12.

C. Expressions

The operations =, >, <, ++, --, **, +, -, */ are implemented, and occupy the same relative hierarchy, as in META-SYMBOL. The operations *, /, //, *+ are not implemented. Parenthetical expressions are not allowed.

Examples:

```
A     EQU     TYPE>0
\A    EQU     A--0**A
```

D. Instruction Generation

The typical instruction line may be represented as

[label] operation [*] operand 1 , [operand 2]

where the brackets denote "optional." If the value of the first expression is absolute, greater than zero, and less than 32, the address is considered to be a Scratch Pad address (unless a literal was indicated).

If the value of the second expression is not zero, indexing is applied. In this case, the address may not indicate a literal.

If the first operand is a symbol (not an expression composed of a symbol plus one or more items connected by operations), and the symbol has not been previously defined, the reference will be treated as a forward or external reference. A 2-word instruction will be generated. The value of the address will be determined when the program is loaded.

Relative forward or external references are not permitted except when they are relative to the location counter symbol ($), such as BRU $+5.

The instruction mnemonics recognized by 92 SYMBOL are those provided in the SDS 92 Instruction List (Appendix E) plus the EOM/SES instructions that address the typewriter/keyboard, paper tape reader/punch, and card/reader punch. The mnemonics for magnetic tape and other devices are not recognized. The syntax for SDS 92 device EOMs and SESs is identical to that for the corresponding 900 Series/9300 operations (see Appendix G) with the exception that no channel designation is required, and an asterisk does not denote interlace.

<u>Example:</u>

RPT 1, 1	Read paper tape unit 1 in 1-character mode (EOM 02104)
CRT 1	Card reader 1 ready test (SES 012106)

For programming convenience there are two additional instruction mnemonics NOP (No Operation—07340) and XAB (Exchange A and B—03040).

E. Remarks

Although the assembler's space requirements are modest, table overflows can occur in a minimal configuration whenever many and/or long symbols are used. Short symbols and local symbols are to be encouraged to alleviate overflows.

Because 92 SYMBOL is a 1-pass assembler, forward references are "chained" on the assembly listing and binary output and are satisfied at load time. This means that the address portion of an instruction involving a forward reference will not, after loading, correspond to the assembly listing. Therefore, forward references should be used as sparingly as possible. A good programming practice is to allocate all data at the beginning of the program and to use forward references only in branch instructions.

Relative forward references are not permitted in any case except where they are relative to the current location counter ($).

VIII. OPERATIONAL PROCEDURES

A. Error Flags

Certain errors are detected by the assembler and are indicated, during the listing of the program, by special symbols. These symbols appear at the left-hand margin of the output listing, preceding the instruction that contains the error(s). Errors, flagged in this manner, do not cause the assembler to terminate the job.

Symbol	Interpretation
*	External address reference. (May or may not be an actual error.)
D	1. Duplicate definition of a main program symbol.
	2. Multiple use of a variable name within COMMON statements.
E	1. Operand field expression error.
	2. Directive syntax error. Examples (not exhaustive):
	a. TEXT – if the first symbol is a value and the second symbol is not a comma.
	b. DO – more than one expression or improper nesting.
	c. END – external reference in END line.
	3. Procedure syntax error. Examples (not exhaustive):
	a. LDX, BRX, STX – no index field given.
	b. Shifts – indirect addressing.
F	Illegal forward references in directive.
G	Generative code in function.
I	Unknown operation code (on 900 Series Computers all POP's are flagged with an I).
L	1. Illegal label (special characters).
	2. Exceeding PROC or FUNC level.
M	Improper use of SBRK or DISP.
N	Missing END line.
P	Exceeding maximum parenthesis nesting level. May occur during use of function.
R	1. Primitive relocation error. See Appendix V section E of reference manual.
	2. Use of relocatable address in extended mode I/O procedure calls other than IORD, IORP, IOSD, IOSP, IOCT.
T	1. Truncation. Attempt to use a value exceeding the capacity of the specified field.
	2. Request COPY not available in hardware.
U	1. Undefined symbol used in manner which does not allow possibility of external reference.
	2. Use of labeled common name in directive or procedure other than COMMON.

Notes:

1. Error and MARK flags generated within PROCs may appear in three possible places:
   a. On call line if generated during pass 1 of a 2-pass procedure.
   b. On the next generated line.
   c. On a blank line following the procedure if no generative line follows error.

2. Labels appearing on PROC reference lines are not defined until the end of the PROC. This is necessary to mechanize the lone $ feature. Therefore, if such a label is doubly defined, the D flag will be printed on a blank line following the procedure.

3. Machine instructions (LDA, etc.) are procedures.

B. META-SYMBOL Error Messages

9300 COMPUTERS

The 9300 META-SYMBOL abort messages are of the form

!META ERROR α xx

where α indicates which overlay segment of the assembler was last loaded:

α	Interpretation
E	Encoder
P	Preassembler
A	Assembler

xx identifies the type of error:

xx	Interpretation
01	Insufficient space to complete encoding of input.
02	Corrections to encoded deck but encoded input file is empty.
03	End of file detected before an end card while reading encoded input.
04	Insufficient space to complete preassembly operations.
05	Insufficient space to complete the assembly.
06	Data error. META-SYMBOL does not recognize the data as anything meaningful.
07	Requested output on a device which is not available.
08	Corrections out of sequence.
09	End of file detected by ENCODER when trying to read intermediate tape X1.
10	Request for non-existent system procedures.
11	Byte larger than dictionary (bad encoded deck).
12	Not encoded deck.
13	Checksum error reading system tape.
14	Preassembler overflow (ETAB). Try using 'SET' option in META Control Card.
15	Not used.
16	Data error causing META-SYMBOL to attempt to process procedure sample beyond end of table.
17	Shrink overflow.
18	Improperly formatted or missing PROC deck series-specification card.
19	End of file encountered while reading system procedures.
20	Irrecoverable error in attempting to read X1 or X2.
21	Symbol table overflow.
22	Abnormal condition (probably end of tape) on X2.

xx	Interpretation
23	End of file on X1.
24	Input is not encoded.
25	Checksum error on encoded deck.
26	End of file on X1.
27	Irrecoverable error in attempting to read INCLUDE, EXCLUDE, or SI.
28	Irrecoverable error in attempting to read X2.
29	Both SI and EI were specified on the META card, but the first card of EI does not have a + in column 1 (i.e., is not a correction card). Note that an empty SI file (a △EOF only) will not cause an error 29 abort.
30	The first SI card is a + card, but no EI parameter appears on the META card.
31	No SI or EI parameter has been specified on the META card.

For example, an improperly nested DO pair would cause the printout

!META ERROR A 06

900 SERIES COMPUTERS

The standard abort message for 900 Series Computers is

META-SYMBOL ERROR xx

where xx has any of the values 01 through 19 as described above for 9300 META-SYMBOL.

For both 9300 and 900 Series Computers errors 05, 06, and 16 are accompanied by a printout that shows the value of certain internal parameters at the time of the abort:

LINE NUMBER	yyyyy	
BREAK1	yyyyy	
LOCATION COUNTER	yyyyy	
UPPER	yyyyy	
LOWER	yyyyy	
BREAK	yyyyy	
SMPWRD	yyyyy	
LTBE	yyyyy	second pass only
LTBL	yyyyy	

(yyyyy represents the value of the particular item.) The last six of these are useful in determining the nature of the assembly overflow and are defined in paragraph E of this section. After the appropriate message has been typed, control is transferred to the system Monitor.

9300 I/O ERROR MESSAGES AND HALTS

When an I/O error is detected, a message is typed, and control is returned to MONITOR. The message will be either

!META ERROR α IOC

indicating checksum error, or

!META ERROR α IOE

indicating buffer error. (α has the same meaning as for abort messages.)

A checksum error is considered to be irrecoverable.

900 SERIES I/O ERROR MESSAGES AND HALTS

When an I/O error is detected, a simple message is typed and the computer halts. The message consists of a 2-letter indication of the type of error and a 2-digit indication of the I/O device. The letter indicators are defined below; the 2-digit number is the unit address number used in EOM selects (see applicable computer reference manual). The action taken if the halt is cleared depends upon the type of error and the device involved. There are three types of error.

BUFFER ERROR (BE)

1. Examples:

   BE11 buffer error while reading magnetic tape 1.

   BE52 buffer error while writing magnetic tape 2.

2. Action upon clearing the halt:

   a. Magnetic tape input - since ten attempts are made to read the record before the halt occurs, continuing causes META-SYMBOL to accept the bad record.

   b. Paper tape or card input - try again.

   c. Magnetic tape output - try again.

   d. Output other than magnetic tape - continues.

CHECKSUM ERROR (CS)

1. Examples:

   CS06 checksum error card reader.

   CS11 checksum error reading magnetic tape 1.

2. Action upon clearing the halt:

   Accepts bad record.

WRITE ERROR (FP)

1. Example:

   FP12 magnetic tape 2 file protected.

2. Action upon clearing the halt:

   Checks again.

C. SYMBOL Error Halts

Input/output errors during a SYMBOL assembly result in a halt with the relative location of the halt displayed in the P register. The recovery procedure depends on the type of error and the device involved.

1. Paper tape reader or typewriter symbolic input - Upon detection of a buffer error, a halt occurs with relative location 032 displayed in the P register. To continue the assembly, one can branch to relative location 025. To reread the record, one must reposition the paper tape and branch to relative location 03.

2. Magnetic tape input - Input records are required to be card images (20 words). A premature termination is treated as being equivalent to an end-of-file. One end-of-file mark is allowed to separate input files on a tape reel and is ignored by the assembler at the beginning of the first pass. An additional end-of-file mark or one occurring after the first symbolic line but before the END line causes a halt in relative location 050. Clearing the halt causes a branch to location 01, which reinstates the assembly process.

   In case of tape read errors, ten recovery attempts are made after which a halt occurs in relative location 021. Clearing the halt causes the record to be accepted.

3. Line printer listing - In the event of a printer fault, a halt occurs in relative location 023. To continue the assembly, clear the fault on the printer and then clear the halt.

D. Concordance Routine Error Messages (META-SYMBOL only)

If an error occurs while a concordance is being output, a message is produced on the output listing device.

Message	Meaning	Action
Write error on magnetic tape.	Unable to write on magnetic tape.	Clear the halt to try again.
Tape file protected	Write ring removed from tape.	Insert ring; clear the halt to continue.
Magnetic tape read error	Read failure on magnetic tape.	Clear the halt to accept record as read.

Message	Meaning	Action
Symbol table overflow	Insufficient space to retain all symbols requested.	Run is aborted.
End-of-file error	End of file detected on X1.	Run is aborted.
Input is not encoded	A non-encoded record is detected on X1.	Run is aborted.
Checksum error	An erroneous checksum is detected on X1.	Clear the halt to read next record.
EXCLUDE follows INCLUDE	An EXCLUDE card follows an INCLUDE card.	EXCLUDE card is ignored.
Concordance control card not recognized	Control card is not INCLUDE, EXCLUDE, or △EOF.	Card is ignored.
Printer fault	Error on printing.	Run continues.
Print buffer error	Buffer error while printing.	Run continues.
Typewriter buffer error	Buffer error while typing listing.	Run continues.

E. Use of SBRK

The SBRK directive gives the user the capability of modifying, at assembly time, the assembler's working storage memory allocation scheme. To understand how SBRK may be useful, one must first understand how META-SYMBOL's table storage is arranged.

After the particular system procedures required for the job have been read in and properly arranged, all of memory from PACKL (the next available cell above the system procedures) to TOP (the highest available location) will be used for building the tables required for assembly. At this time—immediately prior to assembly—the value BREAK, which determines the relative sizes of the various tables, is set. For 8K machines BREAK is set to PACKL + 600_8. For larger machines the increment between PACKL and BREAK is progressively greater. This increment is BREAK1. In the case of an abort due to lack of table storage, the value of BREAK1 for that run is given in the error printout.

During pass 1 of the assembly, user sample or procedures are assigned storage starting at PACKL and progressing upward toward BREAK. The next available cell above the user sample is SMPWRD. Main code symbols and odd procedure level[†] symbols are assigned storage starting at the highest available address and

[†]In discussing META-SYMBOL storage, items are referred to by "levels." The main program is arbitrarily defined as "level 1," external definitions to be satisfied at load time are designated as "level 0," and procedures take on level values 2, 3, 4, etc. (and thus are referred to as "odd" or "even" level procedures).

expanding downward. The next available address is contained in UPPER. Even procedure level symbols and main code definitions are stored starting at BREAK and expanding upward. The next available address for this purpose is contained in LOWER.

Two possibilities for pass 1 overflow exist: (1) if LOWER is greater than UPPER, processing must cease, as no more symbols may be defined; (2) if SMPWRD is greater than BREAK, there are too many user procedures for available storage.

At the start of pass 2, SMPWRD has attained its final value. The amount of memory left between SMPWRD and BREAK is used for two purposes. Table storage for literals starts at SMPWRD and expands upward with the next available address in LTBL. External reference storage starts at BREAK - 1 and expands downward, where LTBE points to the next location for this purpose.

Above BREAK, the situation in pass 2 is the same as in pass 1 with the exception that since no external definitions are being processed, the difference between BREAK and LOWER becomes only as large as is necessary to define even procedure level symbols.

Again, two possibilities for pass 2 overflow exist: (1) if LOWER is greater than UPPER or (2) if LTBL is greater than LTBE, processing must cease.

The SBRK directive enables the user to set, at assembly time, the value of BREAK1. As indicated earlier, the directive must be used before the first external definition or procedure reference; i.e., before the pointers have begun to move. In this manner, the value of the expression E in the operand field of the directive is used as BREAK1, and BREAK is set to PACKL + E.

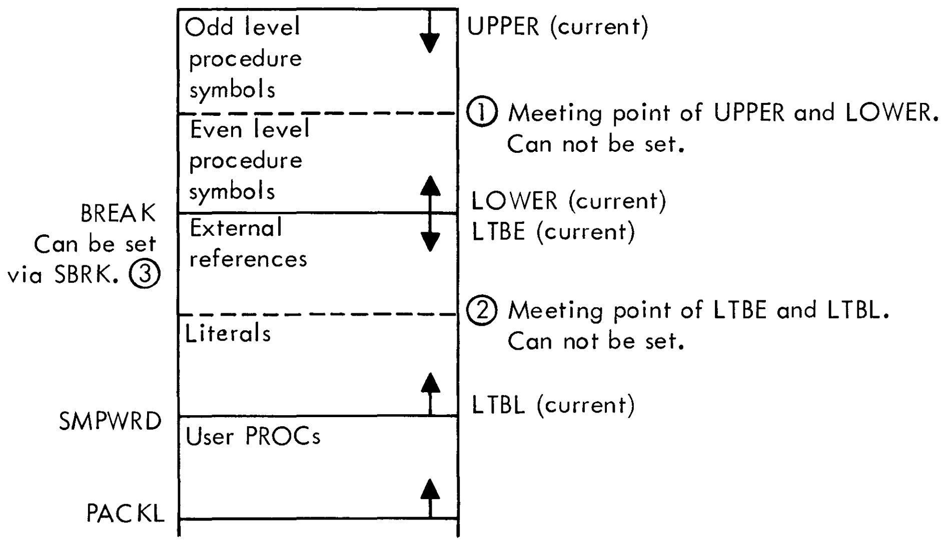
This is useful primarily in attempting to recover from an assembler overflow. For example, suppose one receives the error printout:

META SYMBOL ERROR 05	
LINE NUMBER	1090
BREAK1	01300
LOCATION COUNTER	00737
UPPER	24155
LOWER	17326
BREAK	17304
SMPWRD	17017
LTBE	17077
LTBL	17077

In this case, a pass 2 overflow (indicated by the presence of LTBE and LTBL in the diagnostic), the assembler has run out of storage for literals and references. However, it is apparent that at this point in the assembly considerable memory is still available for symbol storage. The only solution short of program modification, or a larger machine, is to attempt to recover by increasing the amount of pass 2 table storage for literals and references through an initial increase in the value of BREAK1, currently 1300_8. Inserting the card:

```
SBRK    01700
```

at the start of the program would accomplish this. In any case, the exact value to be used in the directive is based upon an evaluation of such immediate considerations as the pass and the point in the program at which overflow occurred, the amount of user sample, and the number of literals and external references that can subsequently be expected to be encountered.



① When UPPER and LOWER meet, an overflow error occurs.
② When LTBE and LTBL meet, an overflow error occurs.
③ BREAK is preset to a fixed point. It can be changed at assembly time via the directive SBRK.

F. Making Symbolic Changes to Encoded Programs

Symbolic changes are accomplished by a series of insertions and deletions controlled by specially formatted symbolic records. The encoded program is interpreted as a series of logical lines as indicated by the line numbers given on the assembly listing for that program. Note that the continuation feature allows two or more cards to be considered as one logical line.

The format of the symbolic change control record is

$+\alpha\wedge$

$+\alpha,\beta\wedge$

\+ must be in column 1.

α is a decimal integer corresponding to the line number given on the assembly listing and specifying the line following which an insertion is to be made, or the first line of a group of sequential lines to be deleted or replaced.

β has the same interpretation as α except that it specifies the last line of a group of sequential lines to be deleted or replaced.

$\wedge$ indicates a space which terminates the scan of the + card.

1. Insertion

Insert $S_1, \ldots, S_n$ following line α:

$+\alpha$

S_1

S_2

.

.

.

S_n

The S_i are symbolic cards for assembly.

To insert before the first line, use:

+0

S_1

S_2

.

.

.

S_n

2. Deletion

Delete lines α through β inclusively:

$+\alpha, \beta$

(Note that if $\alpha = \beta$, only one line is deleted.)

3. Replacement

Replace lines α through β, inclusively, with $S_1, S_2, \ldots, S_n$:

$+\alpha, \beta$

S_1

S_2

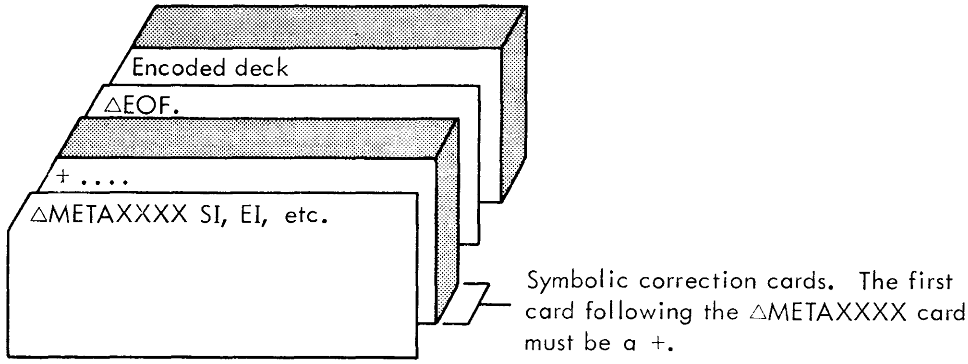
.

.

.

S_n

4. Deck Structure



Note: An encoded deck may not be corrected by merging or juxtaposing other encoded decks.

APPENDIX A. SDS 900 SERIES PROGRAMMED OPERATORS

In 900 Series SYMBOL and META-SYMBOL, non-machine instructions are treated similarly to external references. This enables Programmed Operator definitions and linkages to be established at execution rather than at assembly time. As a result, the entire 64 Programmed Operator instructions are at the disposal of the programmer.

To define a Programmed Operator, the Programmer precedes the POP subroutine by a line which has the following format:

Label	Operation	Operand	Comments
[$] LABEL	POPD		OPERAND IGNORED. POP FOLLOWS

A dollar sign preceding the symbol in the label field causes the POP to be externally defined (so that it can be referred to in other, separately assembled programs).

To refer to a Programmed Operator, the programmer places its name in the operation field of a symbolic line. If a corresponding POP definition does not exist within the same program, the assembler assumes that the operation is a reference to an external POP.

POP assignments are established at assembly time in order of appearance, beginning at 0100, and corrected at loading time if necessary.

EXAMPLE:

Location	Contents	Label	Operation	Operand
00000	10000000	PROG1	LDP	X
00001	10100000		FLA	X
00002	10200001		FLM	X
00003	10300002		STD	X
			END	
00000		$X	RES	2
00002	10000000	PROG2	LDP	X
00003	10100000		FLM	X
00004	10200000		FLA	X
00005	10300000		STD	X
		$LDP	POPD	
00006	03700014		STX	TEMP
00007	07740000		EAX	*0
00010	27600001		LDA	1, 2

EXAMPLE (continued)

Location	Contents	Label	Operation	Operand
00011	27500000		LDB	0, 2
00012	07100014		LDX	TEMP
00013	05100000		BRR	0
00014		TEMP	RES	1
			END	

If PROG1 is loaded first, the operation assignment of PROG1 overrides those of PROG2; if PROG2 is loaded first, the converse is true.

Loading PROG1 first, the loader inherits from the assembler a table equivalent to the following:

LDP: 0100
FLA: 0101
FLM: 0102
STD: 0103

Upon subsequently loading PROG2, the loader detects the mnemonic coincidence(but binary conflict) of FLA and FLM. It therefore changes all 0101 instructions (FLM) in PROG2 to 0102 to agree with PROG1, and all 0102 to 0101.

Also, the loader establishes the necessary POP linkages in locations 0100-01XX.

APPENDIX B. SDS 910/925 INSTRUCTION LIST

Instruction syntax is indicated where non-standard (Cf. III.).

Mnemonic	Instruction Code	Function
LOAD/STORE		
LDA	76	LOAD A
STA	35	STORE A
LDB	75	LOAD B
STB	36	STORE B
LDX	71	LOAD INDEX
STX	37	STORE INDEX
EAX	77	COPY EFFECTIVE ADDRESS INTO INDEX
ARITHMETIC		
ADD	55	ADD M TO A
MIN	61	MEMORY INCREMENT
SUB	54	SUBTRACT M FROM A
MDE	60	MEMORY DECREMENT
MUS	64	MULTIPLY STEP
DIS	65	DIVIDE STEP
LOGICAL		
ETR	14	EXTRACT
MRG	16	MERGE
EOR	17	EXCLUSIVE OR
REGISTER CHANGE		
RCH	46	REGISTER CHANGE
XAB	0 46 00000	EXCHANGE A AND B
BAC	0 46 10000	COPY B INTO A, CLEAR B
ABC	0 46 20000	COPY A INTO B, CLEAR A
CLR	0 46 30000	CLEAR A, B

Mnemonic	Instruction Code	Function
BRANCH		
BRU	01	BRANCH UNCONDITIONALLY
BRX	41	INCREMENT INDEX AND BRANCH
BRM	43	MARK PLACE AND BRANCH
BRR	51	RETURN BRANCH
TEST/SKIP		
SKS	40	SKIP IF SIGNAL NOT SET
SKG	73	SKIP IF A GREATER THAN M
SKN	53	SKIP IF M NEGATIVE
SKA	72	SKIP IF M AND A DO NOT COMPARE ONES
SKM	70	SKIP IF A = M ON B MASK
SHIFT		
RSH	0 66 000XX	RIGHT SHIFT AB
RCY	0 66 200XX	RIGHT CYCLE AB
LSH	0 67 000XX	LEFT SHIFT AB
LCY	0 67 200XX	LEFT CYCLE AB
NOD	0 67 100XX	NORMALIZE AND DECREMENT X
CONTROL		
HLT, PZE	00	HALT
NOP	20	NO OPERATION
EXU	23	EXECUTE
BREAKPOINT TESTS (Breakpoints specified as expression list in operand field.)		
BPT	0 49 20XX0	BREAKPOINT TEST
OVERFLOW (No operand.)		
OVT	0 40 20001	OVERFLOW INDICATOR TEST AND RESET
ROV	0 02 20001	RESET OVERFLOW

Mnemonic	Instruction Code	Function
INTERRUPT (No operand)		
EIR	0 02 20002	ENABLE INTERRUPT SYSTEM
DIR	0 02 20004	DISABLE INTERRUPT SYSTEM
IET	0 40 20004	INTERRUPT ENABLED TEST
IDT	0 40 20002	INTERRUPT DISABLED TEST
AIR	0 02 20020	ARM INTERRUPT
CHANNEL CONTROL (Channel designated by expression in operand field)		
ALC	X 0X 50X00	ALERT CHANNEL (925 only)
DSC	X 0X 00X00	DISCONNECT CHANNEL
ASC	X 0X 12X00	ALERT TO STORE ADDRESS IN CHANNEL (925 only)
TOP	X 0X 14X00	TERMINATE OUTPUT ON CHANNEL
CHANNEL TESTS (925 only - Channel designated by expression in operand field)		
CAT	X 40 X4X00	CHANNEL ACTIVE TEST
CET	X 40 X1X00	CHANNEL ERROR TEST
CZT	X 40 X2X00	CHANNEL ZERO COUNT TEST
CIT	X 40 X0X00	CHANNEL INTER-RECORD TEST
INPUT/OUTPUT		
MIW	12	M INTO W BUFFER WHEN READY
WIM	32	W BUFFER INTO M WHEN READY
MIY	10	M INTO Y BUFFER WHEN READY
YIM	30	Y BUFFER INTO M WHEN READY
BRTW, BRTY	0 40 2X000	BUFFER READY TEST
BETW, BETY	0 40 200X0	BUFFER ERROR TEST
POT	13	PARALLEL OUTPUT
PIN	33	PARALLEL INPUT
BPO	11	BLOCK PARALLEL OUTPUT (925 only)
BPI	31	BLOCK PARALLEL INPUT (925 only)
EOM[†]	02	ENERGIZE OUTPUT M
EOD[†]	06	ENERGIZE OUTPUT TO DIRECT ACCESS CHANNELS (925 only)

[†] Indirect address flag (*) interpreted as interlace control flag.

APPENDIX C. SDS 920/930 INSTRUCTION LIST

Mnemonic	Instruction Code	Function
LOAD/STORE		
LDA	76	LOAD A
STA	35	STORE A
LDB	75	LOAD B
STB	36	STORE B
LDX	71	LOAD INDEX
STX	37	STORE INDEX
EAX	77	COPY EFFECTIVE ADDRESS INTO INDEX
XMA	62	EXCHANGE M AND A
ARITHMETIC		
ADD	55	ADD M TO A
ADC	57	ADD WITH CARRY
ADM	63	ADD A TO M
MIN	61	MEMORY INCREMENT
SUB	54	SUBTRACT M FROM A
SUC	56	SUBTRACT WITH CARRY
MUL	64	MULTIPLY
DIV	65	DIVIDE
LOGICAL		
ETR	14	EXTRACT
MRG	16	MERGE
EOR	17	EXCLUSIVE OR
REGISTER CHANGE (Cf. Appendix E – no operand except for RCH and COPY)		
RCH, COPY	46	REGISTER CHANGE
CLA	0 46 00001	CLEAR A
CLB	0 46 00002	CLEAR B
CLR	0 46 00003	CLEAR AB
CAB	0 46 00004	COPY A INTO B

Mnemonic	Instruction Code	Function
REGISTER CHANGE (continued)		
CBA	0 46 00010	COPY B INTO A
XAB	0 46 00014	EXCHANGE A AND B
BAC	0 46 00012	COPY B INTO A, CLEAR B
ABC	0 46 00005	COPY A INTO B, CLEAR A
CXA	0 46 00200	COPY INDEX INTO A
CAX	0 46 00400	COPY A INTO INDEX
XXA	0 46 00600	EXCHANGE INDEX AND A
CBX	0 46 00020	COPY B INTO INDEX
CXB	0 46 00040	COPY INDEX INTO B
XXB	0 46 00060	EXCHANGE INDEX AND B
STE	0 46 00122	STORE EXPONENT
LDE	0 46 00140	LOAD EXPONENT
XEE	0 46 00160	EXCHANGE EXPONENTS
CNA	0 46 01000	COPY NEGATIVE INTO A
BRANCH		
BRU	01	BRANCH UNCONDITIONALLY
BRX	41	INCREMENT INDEX AND BRANCH
BRM	43	MARK PLACE AND BRANCH
BRR	51	RETURN BRANCH
TEST/SKIP		
SKS	40	SKIP IF SIGNAL NOT SET
SKE	50	SKIP IF A EQUALS M
SKG	73	SKIP IF A GREATER THAN M
SKR	60	REDUCE M, SKIP IF NEGATIVE
SKM	70	SKIP IF A = M ON B MASK
SKN	53	SKIP IF M NEGATIVE
SKA	72	SKIP IF M AND A DO NOT COMPARE ONES
SKB	52	SKIP IF M AND B DO NOT COMPARE ONES
SKD	74	DIFFERENCE EXPONENTS AND SKIP

Mnemonic	Instruction Code	Function
SHIFT		
RSH	0 66 000XX	RIGHT SHIFT AB
LRSH	0 66 240XX	LOGICAL RIGHT SHIFT AB
RCY	0 66 200XX	RIGHT CYCLE AB
LSH	0 67 000XX	LEFT SHIFT AB
LCY	0 67 200XX	LEFT CYCLE AB
NOD	0 67 100XX	NORMALIZE AND DECREMENT X
CONTROL		
HLT, PZE	00	HALT
NOP	20	NO OPERATION
EXU	23	EXECUTE
BREAKPOINT TESTS (Breakpoints specified as expression list in operand field)		
BPT	0 40 20XX0	BREAKPOINT TEST
OVERFLOW (No operand)		
OVT	0 40 20001	OVERFLOW INDICATOR TEST AND RESET
ROV	0 02 20001	RESET OVERFLOW
REO	0 02 20010	RECORD EXPONENT OVERFLOW (930 only)
INTERRUPT (No operand)		
EIR	0 02 20002	ENABLE INTERRUPT SYSTEM
DIR	0 02 20004	DISABLE INTERRUPT SYSTEM
IET	0 40 20004	INTERRUPT ENABLED TEST
IDT	0 40 20002	INTERRUPT DISABLED TEST
AIR	0 02 20020	ARM INTERRUPTS
CHANNEL CONTROL (Channel designated by expression in operand field)		
ALC	X 0X 50X00	ALERT CHANNEL (930 only)
DSC	X 0X 00X00	DISCONNECT CHANNEL
ASC	X 0X 12X00	ALERT TO STORE ADDRESS IN CHANNEL (930 only)
TOP	X 0X 14X00	TERMINATE OUTPUT ON CHANNEL

Mnemonic	Instruction Code	Function
CHANNEL TESTS (930 only - Channel designated by expression in operand field)		
CAT	X 40 X4X00	CHANNEL ACTIVE TEST
CET	X 40 X1X00	CHANNEL ERROR TEST
CZT	X 40 X2X00	CHANNEL ZERO COUNT TEST
CIT	X 40 X0X00	CHANNEL INTER-RECORD TEST
INPUT/OUTPUT		
MIW	12	M INTO W BUFFER WHEN READY
WIM	32	W BUFFER INTO M WHEN READY
MIY	10	M INTO Y BUFFER WHEN READY
YIM	30	Y BUFFER INTO M WHEN READY
BRTW, BRTY	0 40 2X000	BUFFER READY TEST
BETW, BETY	0 40 200X0	BUFFER ERROR TEST
POT	13	PARALLEL OUTPUT
PIN	33	PARALLEL INPUT
EOM[t]	02	ENERGIZE OUTPUT M
EOD[t]	06	ENERGIZE OUTPUT TO DIRECT ACCESS CHANNELS (930 only)

[t]Indirect address flag (*) interpreted as interlace control flag.

APPENDIX D. SDS 9300 INSTRUCTION LIST

Mnemonic	Instruction Code	Function
LOAD/STORE		
LDA	16	LOAD A
STA	76	STORE A
LDB	14	LOAD B
STB	74	STORE B
LDX	X - 17	LOAD INDEX
STX	X - 77	STORE INDEX
STZ	0 - 77	STORE ZERO
LDP, LDF	26	LOAD DOUBLE PRECISION (FLOATING)
STD, STF	75	STORE DOUBLE PRECISION (FLOATING)
XMA	36	EXCHANGE M AND A
XMB	34	EXCHANGE M AND B
XMX	X - 37	EXCHANGE MEMORY AND INDEX
LDS	06	LOAD SELECTIVE
STS	70	STORE SELECTIVE
EAX	15	COPY EFFECTIVE ADDRESS INTO INDEX REGISTER 1
ARITHMETIC		
ADD	05	ADD M TO A
DPA	25	DOUBLE PRECISION ADD
SUB	04	SUBTRACT
DPS	24	DOUBLE PRECISION SUBTRACT
MPO	71	MEMORY PLUS ONE
MPT	72	MEMORY PLUS TWO
MUL	63	MULTIPLY
DIV	62	DIVIDE
ADM	35	ADD A TO M
TMU	61	TWIN MULTIPLY
DPN	27	DOUBLE PRECISION NEGATE

Mnemonic	Instruction Code	Function
FLOATING POINT		
FLA	65	FLOATING ADD
FLS	64	FLOATING SUBTRACT
FLM	67	FLOATING MULTIPLY
FLD	66	FLOATING DIVIDE
LOGICAL		
ETR	11	EXTRACT
MRG	13	MERGE
EOR	12	EXCLUSIVE OR
REGISTER CHANGE (Cf. Appendix E)		
Mode I		
RCH, COPY	0 40 XXXXX	
Mode II		
RCH, COPY	X 40 XXXXX	
Mode III		
AXB	4X 40 XXXXX	ADDRESS TO INDEX BASE
BRANCH		
BRU	01	BRANCH UNCONDITIONALLY
BRX	X - 57	INCREASE INDEX AND BRANCH
BRC	0 - 57	BRANCH AND CLEAR INTERRUPT
BRM	03	MARK PLACE AND BRANCH
BMA	43	BRANCH AND MARK PLACE OF ARGUMENT ADDRESS
BRR	41	RETURN ADDRESS

Mnemonic	Instruction Code	Function
TEST/SKIP		
SKE	45	SKIP IF A EQUALS M
SKU	47	SKIP IF A UNEQUAL TO M
SKG	46	SKIP IF A GREATER THAN M
SKL	44	SKIP IF A LESS THAN OR EQUAL TO M
SKR	73	REDUCE M, SKIP IF NEGATIVE
SKM	55	SKIP IF A = M ON B MASK
SKN	53	SKIP IF M NEGATIVE
SKA	54	SKIP IF M AND A DO NOT COMPARE ONES
SKB	52	SKIP IF M AND B DO COMPARE ONES
SKP	51	SKIP IF BIT SUM EVEN
SKS	20	SKIP IF SIGNAL NOT SET
SKF	50	SKIP IF FLOATING EXPONENT IN B $\geq$ M
SKQ	56	SKIP IF MASKED QUANTITY IN A GREATER THAN M
SHIFT		
SHIFT	60	SHIFT (Used in conjunction with indirect addressing)
ARSA	60-20	ARITHMETIC RIGHT SHIFT A
ARSB	60-10	ARITHMETIC RIGHT SHIFT B
ARSD	60-00	ARITHMETIC RIGHT SHIFT DOUBLE
ARST	60-30	ARITHMETIC RIGHT SHIFT TWIN (A AND B)
LRSA	60-21	LOGICAL RIGHT SHIFT A
LRSB	60-11	LOGICAL RIGHT SHIFT B
LRSD	60-01	LOGICAL RIGHT SHIFT DOUBLE
LRST	60-31	LOGICAL RIGHT SHIFT TWIN (A AND B)
CRSA	60-22	CIRCULAR RIGHT SHIFT A
CRSB	60-12	CIRCULAR RIGHT SHIFT B
CRSD	60-02	CIRCULAR RIGHT SHIFT DOUBLE
CRST	60-32	CIRCULAR RIGHT SHIFT TWIN (A AND B)

Mnemonic	Instruction Code	Function
SHIFT (continued)		
ALSA	60-24	ARITHMETIC LEFT SHIFT A
ALSB	60-14	ARITHMETIC LEFT SHIFT B
ALSD	60-04	ARITHMETIC LEFT SHIFT DOUBLE
ALST	60-34	ARITHMETIC SHIFT TWIN (A AND B)
LLSA	60-25	LOGICAL LEFT SHIFT A
LLSB	60-15	LOGICAL LEFT SHIFT B
LLSD	60-05	LOGICAL LEFT SHIFT DOUBLE
LLST	60-35	LOGICAL LEFT SHIFT A AND B
CLSA	60-26	CIRCULAR LEFT SHIFT A
CLSB	60-16	CIRCULAR LEFT SHIFT B
CLSD	60-06	CIRCULAR LEFT SHIFT DOUBLE
CLST	60-36	CIRCULAR LEFT SHIFT TWIN (A AND B)
NORA	60-64	NORMALIZE A
NORD	60-44	NORMALIZE DOUBLE
FLAG REGISTER (Single operand expression)		
FLAG	22	FLAG
FIRS	22-0	FLAG INDICATOR RESET/SET
FSTR	22-1	FLAG INDICATOR SET TEST/RESET
FRTS	22-2	FLAG INDICATOR RESET TEST/SET
FRST	22-3	FLAG INDICATOR RESET/SET TEST
SWT	22-4	SENSE SWITCH TEST
CONTROL		
HLT, PZE	00	HALT
NOP	10	NO OPERATION
EXU	21	EXECUTE
INT	07	LOAD OP CODE INTO INDEX 2, SKIP ON BIT 1
REP	23	REPEAT INSTRUCTION IN M

Mnemonic	Instruction Code	Function
INTERRUPTS (No operand)		
EIR	0 02 20002	ENABLE INTERRUPT SYSTEM
DIR	0 02 20004	DISABLE INTERRUPT SYSTEM
AIR	0 02 20020	ARM INTERRUPTS
IET	0 20 20004	INTERRUPT ENABLED TEST
IDT	0 20 20002	INTERRUPT DISABLED TEST
CHANNEL CONTROL (Channel designated by expression in operand field)		
DSC	X X2 00X00	DISCONNECT CHANNEL
ALC	X X2 50X00	ALERT CHANNEL
ASC	X X2 12X00	ALERT TO STORE ADDRESS IN CHANNEL
TOP	X X2 14X00	TERMINATE OUTPUT ON CHANNEL
CHANNEL TEST (Channel designated by expression in operand field)		
CAT	X 20 X4X00	CHANNEL ACTIVE TEST
CET	X 20 X1X00	CHANNEL ERROR TEST
CIT	X 20 X0X00	CHANNEL INTER-RECORD TEST
CZT	X 20 X2X00	CHANNEL ZERO COUNT TEST
INPUT/OUTPUT		
EOM[†]	02	ENERGIZE OUTPUT M
EOD[†]	42	ENERGIZE OUTPUT TO DIRECT ACCESS CHANNEL
PIN	33	PARALLEL INPUT
POT	31	PARALLEL OUTPUT
MIA	30	MEMORY INTO CHANNEL A BUFFER
AIM	32	CHANNEL A BUFFER INTO MEMORY

[†] Indirect address flag (*) interpreted as interlace control flag.

APPENDIX E. SDS 92 INSTRUCTION LIST

Mnemonic	Instruction Code	Function
LOAD/STORE		
LDA	64	LOAD A
LDB	24	LOAD B
STA	44	STORE A
STB	04	STORE B
XMA	74	EXCHANGE M AND A
XMB	34	EXCHANGE M AND B
FLAG		
XMF	17	EXCHANGE M AND F
LDF	57	LOAD F
SFT	0044	SET FLAG TRUE
SFF	0042	SET FLAG FALSE
INF	0046	INVERT FLAG
ARITHMETIC		
ADA	62	ADD TO A
ADB	22	ADD TO B
ACA	63	ADD WITH CARRY TO A
ACB	23	ADD WITH CARRY TO B
SUA	60	SUBTRACT TO A
SUB	20	SUBTRACT TO B
SCA	61	SUBTRACT WITH CARRY TO A
SCB	21	SUBTRACT WITH CARRY TO B
MPA	76	MEMORY PLUS A TO MEMORY
MPB	36	MEMORY PLUS B TO MEMORY
MPO	16	MEMORY PLUS ONE TO MEMORY
MPF	56	MEMORY PLUS FLAG TO MEMORY
MUA	13	MULTIPLY A (OPTIONAL)
MUB	53	MULTIPLY B (OPTIONAL)
DVA	52	DIVIDE AB (OPTIONAL)
DVB	12	DIVIDE BA (OPTIONAL)

Mnemonic	Instruction Code	Function
CONTROL		
EXU	73	EXECUTE
HLT	0041/00000000*	HALT
TRAPPING (no operand)		
SCT	0061	SET PROGRAM-CONTROLLED TRAP
RCT	0060	RESET PROGRAM-CONTROLLED TRAP
TCT	0160	TEST PROGRAM-CONTROLLED TRAP
BREAKPOINT TESTS (single operand)		
BRT 1	0144	BREAKPOINT NUMBER 1 TEST
BRT 2	0145	BREAKPOINT NUMBER 2 TEST
BRT 3	0146	BREAKPOINT NUMBER 3 TEST
BRT 4	0147	BREAKPOINT NUMBER 4 TEST
INTERRUPTS (no operand)		
EIR	0051	ENABLE INTERRUPT
DIR	0050	DISABLE INTERRUPT
IET	0150	INTERRUPT ENABLED TEST; SET FLAG IF INTERRUPT SYSTEM ENABLED
AIR	00020001	ARM INTERRUPTS
CHANNEL CONTROL AND TESTS (no operand)		
DSC	00000100	DISCONNECT CHANNEL
TOP	00012100	TERMINATE OUTPUT ON CHANNEL
TIP	00012100	TERMINATE INPUT ON CHANNEL
ALC	00050100	ALERT CHANNEL INTERLACE
ASC	00010500	ALERT TO STORE INTERLACE COUNT
CAT	01004100	CHANNEL ACTIVE TEST; SET FLAG IF NOT ACTIVE
CET	01001100	CHANNEL ERROR TEST; SET FLAG IF ERROR
LOGICAL		
ANA	65	AND TO A
ANB	25	AND TO B
ORA	67	OR TO A

*A slash (/) indicates that either instruction code can be used to perform the same operation.

Mnemonic	Instruction Code	Function
LOGICAL (continued)		
ORB	27	OR TO B
EOA	66	EXCLUSIVE OR TO A
EOB	26	EXCLUSIVE OR TO B
MAA	75	MEMORY AND A TO MEMORY
MAB	35	MEMORY AND B TO MEMORY
COMPARISON		
COA	45	COMPARE ONES WITH A
COB	05	COMPARE ONES WITH B
CMA	47	COMPARE MAGNITUDE OF M WITH A
CMB	07	COMPARE MAGNITUDE OF M WITH B
CEA	46	COMPARE M EQUAL TO A
CEB	06	COMPARE M EQUAL TO B
BRANCH		
BRU	73	BRANCH UNCONDITIONALLY
BRC	32	BRANCH, CLEAR INTERRUPT, AND LOAD FLAG
BRL	33	BRANCH AND LOAD FLAG
BFF	31	BRANCH ON FLAG FALSE
BFT	71	BRANCH ON FLAG TRUE
BDA	70	BRANCH ON DECREMENTING A
BAX	30	BRANCH AND EXCHANGE A AND B
BRM	77	BRANCH AND MARK PLACE
BMC	37	BRANCH, MARK PLACE, AND CLEAR FLAG
SHIFT		
CYA	42	CYCLE A
CYB	02	CYCLE B
CFA	43	CYCLE FLAG AND A
CFB	03	CYCLE FLAG AND B
CYD	02/42*	CYCLE DOUBLE
CFD	43	CYCLE FLAG AND DOUBLE
CFI	03	CYCLE FLAG AND DOUBLE INVERSE

*A slash (/) indicates that either instruction code can be used to perform the same operation.

Mnemonic	Instruction Code	Function
INPUT/OUTPUT		
WIN	15	WORD IN
RIN	55	RECORD IN
WOT	11	WORD OUT
ROT	51	RECORD OUT
PIN	14	PARALLEL INPUT
POT	10	PARALLEL OUTPUT
BPI	54	BLOCK PARALLEL INPUT
BPO	50	BLOCK PARALLEL OUTPUT
EOM	00(40*)	ENERGIZE OUTPUT M
SES	01(41*)	SENSE EXTERNAL SIGNAL

*Codes EOM 40 and SES 41 are reserved for use in special system applications.

APPENDIX F. SPECIAL INSTRUCTIONS - SDS 900 SERIES/SDS 9300

A. SDS 9300 Register Change Instruction (040)

This instruction has three main functions:

1. Interchange and/or modify information between selected bytes of A and B.
2. Interchange and/or modify information among selected bytes of A, B, and the index registers.
3. Load the address portion of a selected index register from the address portion of the instruction.

In modes 1 and 2, the address portion of the instruction serves to extend the operation code; each of the address bits has a particular significance during instruction decoding and execution. In mode 3, however, the interpretation of the address portion is the conventional one in which the 15-bit value defines an operand. Therefore, in mode 3, the instruction is programmed by following the mnemonic, AXB, by an expression in the operand field. The assembler inserts the value of the expression in the instruction's 15-bit address portion.

When programmed in Mode 1 or 2, the instruction may be given one of two mnemonics: RCH or COPY. The assembler processes the operand field of RCH in the conventional manner, inserting the evaluated operand field expression into the instruction's 15-bit address portion. In general, the expression is an octal number representing the bit pattern that specifies the function to be performed. This implies a detailed knowledge of the instruction on the programmer's part.

The operand field of COPY, on the other hand, is interpreted differently. The field consists of a byte selection "mask" followed by one or more grouped expression lists that describe the desired operation(s). The programmer need be concerned only with operational legitimacy and not with its specification via bit patterns.

EXAMPLES:

Label	Operation	Operand	Effect
	COPY	(0,(A,B))	Clear A and B
	COPY	(A,B)	Copy A into B
	COPY	(A,B), (B,A)	Exchange A and B
	COPY	077, (A,B,B)	Merge the low order six bits of A and B in B.

Unless a merge is specified, the assembler automatically sets the "clear" bit. Thus, the second line causes the generation of 0 40 37703.

Format:

Label	Operation	Operand
LABEL	COPY	E, (E11, . . . , E1N), (E21, E2N), . . . , (EM1, . . . , EMN)

Since parenthetical notation is used in the operand field, parentheses have not been used to denote "optional". As usual, the label is optional and may or may not be external. The first operand and all successive operand lists are also optional.

RULES:

1. The byte selection mask, if present, is the first expression to appear in the operand field. It is not enclosed within parentheses. In the absence of this expression, the assembler assumes the mask 077777777 to be implicitly specified. Actually, the assembler cannot insert the mask directly into the byte-selection position of the instruction, since the 24-bit value must be mapped into three or eight bits. However, it is convenient to think of the mask in this manner. Since the mask may be an expression, it need not always be written as an octal number.

EXAMPLES:

Label	Operation	Operand	Effect
EXP	EQU	0777	
HI3	EQU	070000000	
	COPY	EXP, (B,1), (0,B)	$(B)_{15-23} \rightarrow X1_{15-23}$, $0 \rightarrow (B)_{15-23}$
	COPY	HI3 , (A,B)	$(A)_{0-2} \rightarrow B_{0-2}$

Unless the programmer indicates that the specified index register be cleared (in a Mode 2 register change), the assembler automatically sets one of the bits 12, 13, or 14 to prevent the register from being cleared.

2. Following the mask, one or more parenthetical expression lists appear, separated by commas. Within a list, two or more expressions (or expression groups) appear. The first of these specify the source of information flow, and the last specifies the destination. In the case of three or more successive expressions, an OR is implied. Thus, COPY operations are specified by ordered groupings of values. The following definitions relate the value of an expression to the 24-bit source value/register or destination register. Where actual registers are not involved (0 and -1) it is convenient to imagine the existence of two fictitious registers always containing all zeros and all ones, respectively.

Value		Meaning*
-5	-(A)	The negative (2s complement) of (A)
-4	$\overline{(A)}$	The inverse (1s complement) of (A)
-3	$\overline{(B)}$	The inverse (1s complement) of (B)
-1	-1	All 1's
0	0	
1	(X1)	
2	(X2)	
3	(X3)	
4	(B)	
5	(A)	

*( ) denote "the contents of".

Therefore to refer to the registers mnemonically, the programmer must precede his program by equality directives such as:

A	EQU	5
B	EQU	4
X2	EQU	2
IA	EQU	-4
IB	EQU	-3
ONES	EQU	-1

EXAMPLES:

Mnemonic Notation	Absolute	Interpretation
COPY (A,B), (B,A)	COPY (5,4), (4,5)	Exchange A and B
COPY (IA, B), (0,A) COPY (1-A,B), (0,A)	COPY (-4,4), (0,5)	Copy inverse of A into B and clear A
COPY 070, (ONES, B) COPY 070, (-1,B)	COPY 070, (-1,4)	Form mask in B_{18-21}

Thus, the programmer can specify any legitimate register change without having to write the necessary bit pattern explicitly and without being restricted to a pre-selected set of mnemonic op-codes. Also, the assembler diagnoses the variable field for legitimacy.

B. SDS 920/930 REGISTER CHANGE INSTRUCTION (046)

The SDS 920/930 Register Change instruction has some, but not all, of the capabilities of its 9300 counterpart. The differences are:

1. The SDS 920/930 RCH does not provide for byte selection except for selecting the low-order nine bits.
2. The SDS 920/930 Computers include only one index register.
3. There is no capability for copying (or merging) the one's complement of one register into another.

FORMAT:

Label	Operation	Operand
LABEL	COPY or COPYE	(E11, . . . , E1N), (E21, . . . E2N), . . .,(EM1, . . . , EMN)

As before, the label is optional and may or may not be external. All expression lists are optional. The mnemonic COPY implies that operands are whole-word registers; the mnemonic COPYE causes the exponent portion (the low-order nine bits) only to be affected.

COPY(E) operations are specified by ordered groupings of values. The following definitions relate the value of an expression to the 24-bit source value/register or destination register.

Value		Meaning
-5	-(A)	The negative (2s complement) of A
0	0	A register containing all 0s
2	(X)	The index register
4	(B)	The contents of B
5	(A)	The contents of A

EXAMPLES:

Mnemonic Notation	Absolute	Interpretation
COPY (A,B), (B,A)	COPY (5,4), (4,5)	Exchange A and B
COPYE (B,X), (0,B)	COPYE (4,2),(0,4)	$B_{15-23} \longrightarrow X_{15-23}$ $B_{15} \longrightarrow X_{0-14}$ $0 \longrightarrow B_{15-23}$
COPY (A,B,X)	COPY (5,4,2)	Merge A and B to X

APPENDIX G. INPUT/OUTPUT - DEVICE EOMs (SKSs)

FORMAT:

Label	Operation	Operand
[[$] LABEL]	OP	E1[, E2[, E3]]

The format for device (I/O peripheral unit) EOMs (SKSs) is different from that of the standard instruction; for a definition of the format, refer to the appropriate SDS reference manual. The expressions E1, E2, and E3 have the following meaning:

E1:C[†] : Channel (Buffer), nominally, 0

E2:U : Unit Number

E3:CC: Character Transmission Mode (1-4)
(Paper Tape Channel for PSC;
Number of lines to be spaced for PSP)

Mnemonic		Instruction Code	Function
TYPEWRITER, U=1, 2,3			
RKB	C, U, CC	EOM 002X0X	Read Typewriter Keyboard
TYP	C, U, CC	EOM 002X4X	Type
PAPER TAPE, U=1,2			
RPT	C, U, CC	EOM 002X0X	Read Paper Tape
PPT	C, U, CC	EOM 002X4X	Punch Paper Tape
PTL	C, U, CC	EOM 000X4X	Punch Paper Tape with Leader
CARDS, U=1,2			
CRT	C, U	SKS 01200X	Card Reader Test
CFT	C, U	SKS 01100X	Card End-of-file Test
FCT	C, U	SKS 01400X	First Column Test
RCD	C, U, CC	EOM 002X0X	Read Cards Decimal
RCB	C, U, CC	EOM 003X0X	Read Cards Binary
SRC	C, U	EOM 01200X	Skip Remainder of Card
CPT	C, U	SKS 01404X	Card Punch Test
PBT	C, U	SKS 01204X	Punch Buffer Test
PCD	C, U, CC	EOM 002X4X	Punch Cards Decimal
PCB	C, U, CC	EOM 003X4X	Punch Cards Binary

[†]For the SDS 92 the channel designator (E1) is absent.

Mnemonic		Instruction Code	Function
MAGNETIC TAPE, U=0, 1,. . . , 7			
TRT	C, U	SKS 01041X	Tape Ready Test
FPT	C, U	SKS 01401X	File Protected Test
BTT	C, U	SKS 01201X	Beginning of Tape Test
ETT	C, U	SKS 01101X	End of Tape Test
WTD	C, U, CC	EOM 002X5X	Write Tape Decimal
WTB	C, U, CC	EOM 003X5X	Write Tape Binary
EFT	C, U, CC	EOM 003X7X	Erase Forward Tape
ERT	C, U, CC	EOM 007X7X	Erase Reverse Tape
RTD	C, U, CC	EOM 002X1X	Read Tape Decimal
RTB	C, U, CC	EOM 003X1X	Read Tape Binary
SFD	C, U, CC	EOM 002X3X	Scan Forward Decimal
SFB	C, U, CC	EOM 003X3X	Scan Forward Binary
SRD	C, U, CC	EOM 006X3X	Scan Reverse Decimal
SRB	C, U, CC	EOM 007X3X	Scan Reverse Binary
REW	C, U, CC	EOM 01401X	Rewind
RTS	C	EOM 014000	Convert Read to Scan
MAGNETIC TAPE (41.7KC and 96KC only), U=0, 1,. . . , 7 (META-SYMBOL only)			
DT2	C, U	SKS 01621X	Density Test (200 BPI)
DT5	C, U	SKS 01661X	Density Test (500 BPI)
DT8	C, U	SKS 01721X	Density Test (800 BPI)
TFT	C	SKS 013610	Tape File Test
TGT	C	SKS 012610	Tape Gap Test
SRR	C	EOM 013610	Skip Remainder of Record
PRINTER, U=1, 2 (These mnemonics appear in META-SYMBOL only.)			
PLP	C, U, CC	EOM 002X6X	Print Line Printer
PSC	C, U, CC	EOM 01X46X	Printer Skip to Channel
PSP	C, U, CC	EOM 01X66X	Printer Up Space
EPT	C, U	SKS 01406X	End of Page Test
PFT	C, U	SKS 01106X	Printer Fault Test
POL	C, U	EOM 01206X	Printer Off-line
PRT	C, U	SKS 01206X	Printer Ready Test

APPENDIX H. INPUT/OUTPUT - CHANNEL OPERATIONS (SDS 925/930/9300)

The initiation of an I/O channel operation consists of alerting the channel (generally with a device EOM), executing an interlace control EOM, and issuing (via POT) an interlace (I/O) control word (IOCW). An IOCW can accommodate a 14-bit address and a 10-bit word count. Whenever the count exceeds 10 bits or the address is 15 bits (930/9300 only) the extra high-order bits are required in the EOM. To simplify the programming of input/output, special I/O command PROCs have been incorporated in the standard META-SYMBOL system PROCs. The use of these PROCs is described below.

1. Load Channel with Remote Command

   The mnemonic LCH (Load Channel) is written at the point of execution. Its operand field specifies the location of a remote I/O command. The valid (remote) I/O commands are:

IORD	Input/Output Record and Disconnect
IORP	Input/Output Record and Proceed
IOSD	Input/Output until Signal and Disconnect
IOSP	Input/Output until Signal and Proceed
IOCT	Input/Output under Count and Terminate (Non-terminal-function interlace operation)

   EXAMPLE:

```
              Source Code                            Generated Code (expressed symbolically)
              LCH      ALPHA                                    EXU        ALPHA
               .                                                POT        ALPHA + 1
               .                                                 .
               .                                                 .
ALPHA         IORD     [*] ADDR,COUNT,ICD           ALPHA       EOM/EOD
                                                                IOCW       ADDR,COUNT
```

   The asterisk causes an EOD to be generated instead of an EOM.

   ADDR points to the beginning of the buffer area.

   COUNT specifies the number of words to be input/output.

   ICD is an interrupt control digit (0, 1, 2, or 3).

   If ALPHA is tagged, the tag is generated in both the EXU and the POT.

   "Overflow" bits for the address and count are automatically inserted into the EOM.

2. Load Channel with Proximate Command

   These mnemonics cause the generation of the entire I/O packet (EOM, POT, IOCW) and are, therefore, more economical of space in those cases where the programmer does not desire multiple references to an IORD.

The five mnemonics are:

LCRD	Load Channel for I/O Record, Disconnect Mode
LCRP	Load Channel for I/O Record, Proceed Mode
LCSD	Load Channel for I/O until Signal, Disconnect Mode
LCSP	Load Channel for I/O until Signal, Proceed Mode
LCCT	Load Channel for I/O under Count, Terminate Mode

These mnemonics are written at the point of execution. Their operand fields are identical to those of the remote I/O commands (e.g., IORD). The assembler generates an EOM-POT combination and inserts the IOCW in the literal table. The extra high-order address and count bits are inserted into the EOM by the assembler. Note that for 930 or 9300 target machines it is possible for a relocatable buffer area to be loaded such that it can be referenced only by a 15-bit address. In this case, the separation of the EOM from the IOCW precludes the possibility of the loader inserting the high-order address bit into the EOM. The assembler flags such potential difficulties with an 'R'.

EXAMPLE:

	Source Code	Generated Code (expressed symbolically)		
LCRD	[*] ADDR,COUNT,ICD		EOM	
			POT	LTE
			⋮	
		LTE	IOCW	

The symbol LTE is used to denote a literal table entry.

3. "Hand-Coded" I/O

The Interlace Control EOM may always be written by specifying the EOM's address portion as an octal number in the operand field. However, the programmer must then know at what location the I/O block begins, since the EOM contains the high order address and count bits. Naturally, this is not always possible, especially in the case of relocatable programs. In fact, for relocatable buffer areas (on a 930 or 9300), the programmer should always prefer the first method since only then does the Loader know where the EOM is relative to the IOCW.

The system provides the following mnemonics to simplify the coding of the Interlace Control EOM. Their operand fields are identical in format to those of the IOXX and LCXX Command PROCs.

ICRD	I/O Record and Disconnect EOM
ICRP	I/O Record and Proceed EOM
ICSD	I/O until Signal and Disconnect EOM
ICSP	I/O until Signal and Proceed EOM
ICCT	I/O under Count and Terminate EOM

Detecting one of these mnemonics, the assembler generates the appropriate EOM (or EOD), inserting the terminal function bits and the high order address and count bits. The restriction on the use of relocatable buffer areas which applies to the LCXX PROCs also applies to these.

EXAMPLE:

```
        ICRD        [*] ADDR,COUNT,ICD
        POT             ALPHA
          .
          .
ALPHA   IOCW            ADDR, COUNT
```

APPENDIX I. META-SYMBOL/FORTRAN INTERFACE (SDS 9300 COMPUTERS ONLY)

As indicated in the introduction, the merits of any programming language depend strongly upon its application. While some applications demand a mathematically oriented language, such as FORTRAN, others require the close contact with the machine that the programmer can gain only through "machine language" programming.

Frequently, the optimal solution to the programming problem consists of "marrying" two or more languages, and coding different sections of the program in the languages most appropriate. However, this cannot be accomplished without providing for a common interface, and the burden for the interface is generally placed upon the language having the least restrictive syntax.

Such interface allows the execution of META-SYMBOL programs in conjunction with programs written in the SDS FORTRAN IV language.

META-SYMBOL recognizes the following directives. Note that, as META-SYMBOL directives, they are subject to some restrictions (noted below) not present for the analogous FORTRAN statements.

1. LOGICAL v1, v2, . . . , vn
2. INTEGER v1, v2, . . . , vn
3. REAL v1, v2, . . . , vn
4. COMPLEX v1, v2, . . . , vn
5. DOUBLEPRECISION v1, v2, . . . , vn

(Note that DOUBLEPRECISION is one word.)

Each vn represents a variable name. The assembler ignores redundant declarations; it flags conflicting declarations as errors.

6. COMMON V1, . . . , Vn/B1/V11, . . . , V1n . . . /Bm/Vm1, . . . , Vmn

   Each V represents a variable name or an array name followed by its dimensions in parentheses: e.g., "A(3, 4, 5)".

   Each B represents a COMMON block name. If no block name appears, META-SYMBOL assumes blank common. At the beginning of each COMMON statement, it assumes blank common.

   Since COMMON statements are cumulative over the program, no variable may meaningfully appear in COMMON twice. The assembler recognizes this error.

   No symbol that can be used in the operation field of a META-SYMBOL program may appear as a COMMON block name.

Dimension information, legal in type statements in FORTRAN IV, may not be used in META-SYMBOL type directives. Such dimensions must appear in a COMMON statement. For example,

```
REAL      A(9)
COMMON    A
```

is illegal in META-SYMBOL. The correct form is

```
REAL      A
COMMON    A(9)
```

It is mandatory that each variable used in a COMMON statement be previously defined in a type directive (REAL, etc.).

COMMON allocation is in SDS mode (integer variables are allocated one word; real, two words; etc.).

Generalized array bounds as permitted in SDS Extended FORTRAN IV must be translated either to an integer quantity or to an expression resulting in the correct integer quantity at assembly time. For example, FORTRAN allows

```
REAL      A
COMMON    A(-3 : 3)
```

META-SYMBOL must have

```
REAL      A
COMMON    A(E)
```

where E has the value 7 at assembly time.

No continuation is permitted in type directives; however, any type directive may be used more than once.

SYNTAX

Columns 1 to 6 must be blank. One or more blanks must appear between the directive name and the list. No blanks may appear within words or within variable lists. (Blank common is indicated by two successive slashes.)

SEMANTICS

1. Common variables are assigned relative locations within the appropriate block in order of appearance in the program. The assembler computes the size of each named COMMON block by summing the sizes of the variables named.

2. The type directives (LOGICAL, INTEGER, etc.) specify to the assembler the size of each COMMON variable and array element. The assembler keeps a table of the space required for each type.

It is essential in a program in which a named COMMON variable is referred to that the COMMON and type directives give the assembler enough information to compute the size of the block and the relative location of each variable referred to. It is mandatory to list all variables named in the COMMON block, to give the dimensions of all arrays in COMMON directives, and to list each variable in a type directive.

APPENDIX J. COMPATIBILITY WITH SDS SYMBOL 4 AND SYMBOL 8 (900 SERIES ONLY)

In 1963, SDS announced two assemblers for the 900 Series Computers: SYMBOL 4 and SYMBOL 8. Patterned after other familiar assemblers, SYMBOL 4 proved popular with users; literals and macros were added in SYMBOL 8.

It can be seen from this manual that SYMBOL and META-SYMBOL offer still an additional level of capability to the SYMBOL 4/SYMBOL 8 user. In some cases, however, the additional generality of the new assemblers has created some incompatibilities with respect to the 1963 assemblers. To assist users in converting to the new assemblers, these incompatibilities have been resolved in all but exceptional cases.

Compatibility has been provided in two ways:

SYMBOL

The assembler accepts programs written either in the SYMBOL or in the SYMBOL 4 language.

META-SYMBOL

The assembler consists of an Encoder and a Translator. The Translator accepts only encoded META-SYMBOL language. The conversion from SYMBOL 4/8 to META-SYMBOL is accomplished by the Encoder, which has a special Compatibility Mode. Since the Translator offers optional recovery of the source language, SYMBOL 4/8 programs can be easily converted, if desired, to META-SYMBOL source language.

The compatibility features are described in greater detail below:

A. Label Field

Both assemblers allow symbols to begin with a numeric character. Symbols are not allowed to contain special characters. The symbol must begin in column 1.

B. Operation Field

1. Instruction mnemonics:

The following EOM/SKS mnemonics are included in addition to those listed in Appendix B and Appendix C.

TOPW/TOPY
DISW/DISY
PTLW/PTLY
PPTW/PPTY
RPTW/RPTY
TYPW/TYPY
RKBW/RKBY
RCBW/RCBY
RCDW/RCDY
RTDW/RTDY
RTBW/RTBY
WTDW/WTDY
WTBW/WTBY
ETW
SFBW
SRBW
REWW

Programmed operator mnemonics are not recognized and are treated as indicated in Appendix A.*

Directives:

Recognized	Ignored*
ORG	FORT
BORG	BLK
BSS	LIL
OCT	TCD
DEC	LIST
BCI	UNLIST
BOOL	**REL
**VFD	**BES
**MACRO	**IDEN
	**LOAD
	**LTAB
	**TITLE
	**DETAIL

*Only when a 900 Series target machine is specified.

**Illegal in SYMBOL

a. The FORTRAN interface (FORT, BLK) is solved by the use of external definitions.

b. All programs are relocatable unless preceded by AORG.

c. Bootstrap loaders are available separately.

d. List suppression may be specified when the assembler is loaded.

3. Indirect Addressing:

Indirect addressing is allowed to be indicated by an asterisk following an instruction mnemonic.

C. Operand Field

1. Location counter:

An asterisk is allowed to denote the location counter. In cases where an expression which includes the symbol * is to be indirectly addressed, the syntax of SYMBOL and of SYMBOL 4 cannot be mixed. Thus, either

```
        LDA*    *+5
or      LDA     *$+5
```

is permissible, but

```
        LDA     **+5
```

is not.

2. Octal/Decimal Interpretation:

Octal interpretation of the operand field is forced in the case of SYMBOL and META-SYMBOL for the operations EOM, SKS, RCH and OPD. Decimal interpretation is never forced. Therefore, the instruction

```
        RSH     010
```

would cause a right shift of 8 places.

3. Literals:

a. Leading O is converted to zero

b. Leading H is converted to surrounding quotes.

c. Internal B and E (binary and decimal scale factors) are converted to the operator notation using */ and *+, respectively.

4. The VFD line is translated to a list which is then handled by a system PROC.

5. Macros:

   MACROs are translated to PROCs.

6. Spaces:

   Spaces are converted to 060 whenever they occur in a TEXT line or within a literal character string. The BCI directive is translated to BCD, and the word count is multiplied by four in order to agree with the BCD syntax.

7. "Fill" operand:

   Whenever the expression ** occurs as an operand, a diagnostic flag will result.

8. Breakpoint Test (BPT)

   Whenever multiple breakpoints are tested with the BPT pseudo operation, incorrect assembly will result. This is because the BPT mn of SYMBOL 4/8 must be written BPT m, n for SYMBOL/META-SYMBOL.

APPENDIX K. SDS STANDARD BINARY LANGUAGE

The following description specifies a standard binary language for the SDS 900 Series and 9300 Computers. The intention has been that this language be both computer-independent and medium-independent. Thus, there is provision for handling Programmed Operator definitions and references even though the 9300 does not have this hardware feature; similarly, there is a provision for relocation relative to blank COMMON, even though this requirement is not present in SDS 900 Series FORTRAN II.

In the following, a file is the total binary output from the assembly/compilation of one program or subprogram. A file is both a physical and a logical entity since it can be subdivided physically into unit records and logically into information blocks. While a unit record (in the case of cards) may contain more than one record, a logical record may not overflow from one unit record to another.

1. CONTROL WORD - 1st word in each type of record

Type (T)		Word Count (C)	Mode (binary)	Folded Checksum (FC)	Field
			101		contents
0 2	3	4 8	9 11	12 23	bit number

T	RECORD TYPE
000	Data Record (text)
001	External References & Definitions, Block & Program Lengths
010	Programmed Operator References and Definitions
011	End Record (Program or Subroutine end)
100 thru 111	Not Assigned

C = total number of words in record, including Control Word

Note that the first word contains sufficient information for handling these records by routines other than the loader (that is, tape or card duplicate routines.) The format is also medium-independent, but preserves the MODE indicator positions desirable for off-line card-handling.

An exclusive OR checksum is used. If the symbol -- is used to denote exclusive OR, and W_i denotes the i-th word in the record, $1 \leq i \leq C$, then

$$FC = (W_1)_{0-11} \text{ -- } (C)_{0-11} \text{ -- } (C)_{12-23} \text{ -- } 07777$$

where

$$C = W_2 \text{ -- } W_3 \text{ -- } \dots \text{ -- } W_c$$

2. DATA RECORD FORMAT (T=0)

Control Word — Word 1

Record Type (T)		$3 \leq C \leq 30$	Mode (binary)	Folded Checksum
000	0		101	
0–2	3	4–8	9–11	12–23

Load Address Word — Word 2

	Data Word Modifiers (M)	Load Address Modifiers (A)	Load Address (Relative or Absolute)
0			
0	1–4	5–8	9–23

Data Word 1 — Word 3

Instruction or Constant
0–23

Words 3 thru n+2 contain instructions or constants, (where $1 \leq n \leq 24$)

Load Relocation — Word n+3

Load address relocation word (present iff $(M) \cap 1=1$)
0–23

Common Relocation — Word n+4

Blank common relocation word (present iff $(M) \cap 2=2$)
0–23

Programmed Operator Relocation — Word n+5

Programmed operator relocation word (present iff $(M) \cap 4=4$)
0–23

Special I/O Relocation — Word n+6

Special Input/Output operation relocation (present iff $(M) \cap 8=8$)
0–23

Words n+3 thru n+6 are modifier words. Each bit in each of these words corresponds to a data word (bits 0 thru 23 correspond to words 3 thru n+2, respectively). A bit set to one (1) indicates that the specified data word required modification by the loader. There are four (4) types of modification (and hence four possible modifier words) which are indicated in data records. Presence of a modifier word is indicated by the M (data word modifier) field in the load address word.

The load address is subject to modification as indicated by the "A" field of the load address word as follows ((A) = 0 means absolute):

$(A) \cap 1 = 1$, current load relocation bias is added to load address

3. EXTERNAL REFERENCES AND DEFINITIONS, BLOCK AND PROGRAM LENGTHS

(T=1) (Includes labeled common, blank common and program lengths)

Control Word — Word 1

Record Type (T)		$4 \leq C \leq 31$*	Mode (binary)	Folded Checksum
001	0		101	
0–2	3	4–8	9–11	12–23

* From 1 to 10 items per record

Common or Program Length Item

1 to 8 char. LABEL

C1	C2	C3	C4
C5	C6	C7	C8

Length Word

Item Type		B	C	Length of Program or Common Block (L)
00	00000			
0–1	2–6	7	8	9–23

External Reference Item

Label

C1	C2	C3	C4
C5	C6	C7	C8

Chain Word

Item Type		Address Modifiers (A)**	Address of Last Reference
01	000		
0–1	2–4	5–8	9–23

External Definition Item

Label

	C1	C2	C3	C4
	C5	C6	C7	C8

Value Word

Item Type		Address Modifiers (A)**	Absolute or Relocatable Value
10	000		
0 1	2 4	5 8	9 23

External symbolic definitions include subroutine "identification" as a subset and require no special treatment of subroutines with multiple names. B = 1 if (L) is program length, C = 1 if (L) is length of a labeled common block.

** See data record, load address word, for interpretation

External Reference with Addend Item*

Label

	C1	C2	C3	C4
	C5	C6	C7	C8

Chain Word

Item Type		Address Modifiers	Address of Last Reference
11	000		
0 1	2 4	5 8	9 23

Addend Word

not used	Value of Addend
000000000	
0 8	9 23

4. PROGRAMMED OPERATOR REFERENCES AND DEFINITIONS (T=2)

Control Word — Word 1

Record Type (T)		$4 \le C \le 31$**	Mode (Binary)	Folded Checksum
010	0		101	
0 2	3	4 8	9 11	12 23

*One of these items for each unique reference; e.g., each of the following references is represented by a separate item:

A+5, B+5, B+6, C+2, C+5

** From 1 to 10 items per record

Internal Programmed Operator

C1	C2	C3	C4
C5	C6	C7	C8

1 to 8 Char. Mnemonic

Item Type	Sequence No.	R	Origin of Programmed Operator Routine
00		1	
0 1	2 7	8	9 23

Programmed Operator Reference

C1	C2	C3	C4
C5	C6	C7	C8

1 to 8 Char. Mnemonic

Item Type	Sequence No.	R	Not used
01		0	000000000000000
0 1	2 7	8	9 23

External Programmed Operator Definition

C1	C2	C3	C4
C5	C6	C7	C8

1 to 8 Char. Mnemonic

Item Type	Sequence No.	R	Origin of Programmed Operator Routine
10		1	
0 1	2 7	8	9 23

R=1 iff origin of Programmed Operator Routine is relocatable.
The sequence No. indicates the order in which the definitions or reference occurred in the source program.

5. END RECORD (T=3)

Control Word (Word 1)

Record Type (T)		$2\leq C\leq 4$	Mode (Binary)	Folded Checksum
011	0		101	
0 2	3	4 8	9 11	12 23

Length of Program (Word 2)

	Transfer Word Modifiers (M)*		1+Maximum Value of Location Counter
0		0000	
0	1 4	5 8	9 23

Transfer Word (Word 3)

	"BRU"	Transfer Address
000	000001	
0 2	3 8	9 23

This may be followed by modifier words as described in Section 2.

*See data record description for interpretation

SDS 701 South Aviation Blvd./El Segundo, California 90245

EXECUTIVE OFFICES
AND OPERATING DIVISIONS
701 South Aviation Blvd.
El Segundo, Calif. 90245
(213) 679-4511

EASTERN SYSTEMS DEPT.
12150 Parklawn Drive
Rockville, Maryland 20852
(301) 933-5900

PRINTED CIRCUITS DEPT.
600 East Bonita Avenue
Pomona, Calif. 91767
(714) 628-7371

TECHNICAL TRAINING
5250 West Century Blvd.
Los Angeles, California 90045
(213) 679-4511

SALES OFFICES

Eastern

Maryland Engineering Center
12150 Parklawn Drive
Rockville, Maryland 20852
(301) 933-5900

69 Hickory Drive
Waltham, Mass. 02154
(617) 899-4700

Brearley Office Building
190 Moore Street
Hackensack, New Jersey 07601
(201) 489-0100

1301 Avenue of the Americas
New York City, N.Y. 10019
(212) 765-1230

673 Panorama Trail West
Rochester, New York 14625
(716) 586-1500

P.O. Box 168
1260 Virginia Drive
Fort Washington Industrial Park
Fort Washington, Pa. 19034
(215) 643-2130

Southern

Suite 620
State National Bank Bldg.
200 W. Court Square
Huntsville, Alabama 35801
(205) 539-5131

Orlando Executive Center
1080 Woodcock Road
Orlando, Florida 32803
(305) 841-6371

2964 Peachtree Road, N.W.
Suite 350
Atlanta, Georgia 30305
(404) 261-5323

Suite 311-B
First National Bank Office Bldg.
7809 Airline Highway
Metairie, Louisiana 70003
(504) 721-9172

8383 Stemmons Freeway
Suite 233
Dallas, Texas 75247
(214) 637-4340

3411 Richmond Avenue
Suite 202
Houston, Texas 77027
(713) 621-0220

Midwest

2720 Des Plaines Avenue
Des Plaines, Illinois 60018
(312) 824-8147

17500 W. Eight Mile Road
Southfield, Michigan 48076
(313) 353-7360

4367 Woodson Road
St. Louis, Missouri 63134
(314) 423-6200

Seven Parkway Center
Suite 238
Pittsburgh, Pa. 15220
(412) 921-3640

Western

1360 So. Anaheim Blvd.
Anaheim, Calif. 92805
(213) 865-5293

5250 West Century Blvd.
Los Angeles, California 90045
(213) 679-4511

505 W. Olive Avenue
Suite 300
Sunnyvale, Calif. 94086
(408) 736-9193

World Savings Bldg.
Suite 401
1111 So. Colorado Blvd.
Denver, Colo. 80222
(303) 756-3683

Fountain Professional Bldg.
9004 Menaul Blvd., N.E.
Albuquerque, N.M. 87112
(505) 298-7683

Dravo Bldg., Suite 501
225 108th Street, N.E.
Bellevue, Wash. 98004
(206) 434-3991

Canada

864 Lady Ellen Place
Ottawa 3, Ontario
(613) 722-8387

England

London Branch
I.L.I. House
Olympic Way
Wembley Park
Middlesex

INTERNATIONAL
REPRESENTATIVES

France

Compagnie Internationale
pour l'Informatique, C.I.I.

EXECUTIVE AND
SALES OFFICES
66-68 Route de Versailles
78 - Louveciennes
951 86 00 (Paris area)

MANUFACTURING
AND ENGINEERING
Rue Jean Jaures
Les Clayes-sous-Bois
Seine et Oise
950 94 00 (Paris area)

Israel

Elbit Computers Ltd.
Subsidiary of Elron
Electronic Industries Ltd.
88 Hagiborim Street
Haifa
6 4613

Japan

F. Kanematsu & Co. Inc.
Central P.O. Box 141
New Kaijo Building
Marunouchi, Chiyoda-ku
Tokyo

90 05 06G