60496400

CONTROL DATA CORPORATION

SYMPL VERSION 1 REFERENCE MANUAL

CDC® OPERATING SYSTEMS:
NOS 1
NOS/BE 1
SCOPE 2

60496400

CONTROL DATA CORPORATION

SYMPL VERSION 1 REFERENCE MANUAL

CDC® OPERATING SYSTEMS:
NOS 1
NOS/BE 1
SCOPE 2

REVISION RECORD	
REVISION	**DESCRIPTION**
A	Original printing.
(11-1-75)	
B	This revision documents SYMPL 1.2, PSR level 439. New features include CONTROL statement
(12-06-76)	additions for trace and optimization. See list of effective pages.
C	This revision documents SYMPL 1.2, PSR level 446. It reflects SYMPL support of the CYBER 170
(03-01-77)	Model 176. See list of effective pages.
D	This revision documents SYMPL 1.3. New features include CONTROL statement
(03-31-78)	addition for weak externals; and points not tested SYMPL control statement option. Appendix F
	contains a glossary.
Publication No. 60496400	

REVISION LETTERS I, O, Q AND X ARE NOT USED

Address comments concerning this manual to:

CONTROL DATA CORPORATION
Publications and Graphics Division
215 MOFFETT PARK DRIVE
SUNNYVALE, CALIFORNIA 94086

or use Comment Sheet in the back of this manual

LIST OF EFFECTIVE PAGES

New features, as well as changes, deletions, and additions to information in this manual are indicated by bars in the margins or by a dot near the page number if the entire page is affected. A bar by the page number indicates pagination rather than content has changed.

Page	Revision
Cover	–
Title Page	–
ii	D
iii/iv	D
v/vi	D
vii, viii	D
1-1, 1-2	B
1-3	C
1-4, 1-5	B
1-6 thru 1-8	D
1-9	B
1-10	C
2-1	B
2-2, 2-3	C
2-4 thru 2-6	D
2-7	C
2-8, 2-9	B
2-10 thru 2-12	D
3-1, 3-2	B
3-3	D
3-4 thru 3-7	B
4-1	B
4-2 thru 4-4	C
4-5	B
4-6 thru 4-8	C
4-9	D
5-1, 5-2	B
5-3	C
5-4	D
5-5	B
5-6, 5-7	C
5-8, 5-9	D
6-1	C
6-2, 6-3	D
6-4	C
6-5, 6-6	D
A-1	A

Page	Revision
A-2	D
B-1	D
B-2, B-3	B
B-4	D
C-1	B
C-2, C-3	D
C-4	B
D-1 thru D-3	A
D-4	B
D-5 thru D-7	A
D-8	B
D-9 thru D-15	A
D-16	B
D-17, D-18	A
D-19, D-20	B
D-21	A
D-22	B
D-23 thru D-25	D
E-1, E-2	B
F-1, F-2	D
Index-1 thru -3	D
Comment Sheet	D
Mailer	–
Back Cover	–

Page	Revision

PREFACE

SYMPL version 1.3, which is a systems programming language, operates under control of the following operating systems:

SCOPE 2 for the CONTROL DATA® CYBER 170 Model 176, CYBER 70 Model 76, and 7600 Computer Systems

NOS/BE 1 for the CDC® CYBER 170 Series, CYBER 70 Models 71, 72, 73, 74 and 6000 Series Computer Systems

NOS 1 for the CONTROL DATA CYBER 170 Models 171, 172, 173, 174, 175, CYBER 70 Models 71, 72, 73, 74, and 6000 Series Computer Systems

This reference manual presents the semantics and rules for writing programs in the SYMPL language. It includes sufficient information to prepare, compile, and execute such programs. An appendix presents the syntax of the language in metalinguistic form.

The reader of this manual is assumed to have knowledge of the operating system and computer system under which SYMPL will be used.

Other publications of interest:

Publication	Publication Number
NOS 1 Operating System Reference Manual, Volume 1	60435300
NOS 1 Operating System Reference Manual, Volume 2	60445300
NOS/BE 1 Operating System Reference Manual	60493800
SCOPE 2 Reference Manual	60342600

CDC manuals can be ordered from Control Data Literature and Distribution Services, 8001 East Bloomington Freeway, Minneapolis, MN 55420

This product is intended for use only as described in this document. Control Data cannot be responsible for the proper functioning of undescribed features or parameters.

CONTENTS

APPENDIXES

FIGURES

TABLES

CONSTANTS

SYMPL has five types of constants. Each is a sequence of characters which defines its own value. The constant types are: Boolean, character, integer, real, and status.

BOOLEAN CONSTANTS

Boolean constants represent the two elements of Boolean algebra. They are specified by the reserved words TRUE and FALSE.

CHARACTER CONSTANTS

Character constants represent alphanumeric data. A character constant has the format:

"string"

string String of 1 through 240 characters of the computer character set shown in appendix A. If the character " is to appear in the string, it must be specified by two consecutive " marks.

For example:

"TAPE01" "ERROR %%"

"QUOTES" "A" " "

INTEGER CONSTANTS

Integer constants represent numeric values. The three types of integer constants are: decimal, octal, and hexadecimal.

During execution, the maximum allowable value for an integer constant depends on the use of the constant. The value of an integer to be converted to a real value and the value of an integer operand for, and the result of, integer multiplication and division must be able to be expressed in 47 bits. High-order bits are lost when a larger value exists, but no diagnostic informs the programmer of such a condition.

Each of the types of integer constants is specified in a different way. Also, each appears in storage in a format appropriate to its type, as described with ITEM declarations for data types.

Decimal Integer Constant

A decimal constant is a string of decimal digits 0 through 9 with an optional preceding + or - sign. The string can contain 1 through 18 digits; it cannot contain blanks. The absolute value for a decimal integer must be able to be expressed in 59 bits.

For example:

+15 -1 4096

Hexadecimal Constant

A hexadecimal constant represents 4 bits in storage for each hexadecimal digit in the constant. The absolute value for a hexadecimal constant must be able to be expressed in 59 bits. If 60 significant bits are written, the leftmost bit is used as a sign in two's complement; and if the constant is stored in a signed integer format of n bits, the nth bit from the right is used as the sign bit.

A hexadecimal constant has the format:

X"string"

string String of 1 through 15 hexadecimal digits 0 through 9 and A through F. Embedded blanks are ignored.

For example:

X"7FFF" X"9"

Octal Constant

An octal constant represents 3 bits in storage for each octal digit in the constant. If 60 significant bits are written, the leftmost bit is used as a sign in two's complement; and if the constant is stored in a signed integer format of n bits, the nth bit from the right is used as the sign bit.

An octal constant has the format:

O"string"

string String of 1 through 20 octal digits 0 through 7. Embedded blanks are ignored.

For example:

O"777" O"33"

REAL CONSTANTS

Real constants represent numeric values in standard single-precision normalized floating point format. A real constant is a string of decimal digits that includes a decimal point and can include a leading sign. Optionally, it can include an exponent representing multiplication by a power of 10. The exponent is specified as either of the semantically equivalent letters D or E followed by an optional plus or minus sign and a decimal integer. A real constant cannot be represented by a string containing an embedded blank.

For example:

3.14E2 -24. 37.E-3

The magnitude limits of a real constant are approximately 10^{-293} to 10^{+322} with up to 15 digits of accuracy. A diagnostic message is given when a number falls outside of the hardware limits.

STATUS FUNCTIONS AND CONSTANTS

Status functions and constants represent small integer values the compiler has associated with the identifiers in a status list. They can be used to preset scalar and array items and can be used in expressions.

Both status constants and status functions require a preceding STATUS declaration to define a status list and identifiers associated with the status list, as described in section 2.

A status function has the format:

stlist"stvalue"

Use of a status function accesses the integer associated with stvalue in status list stlist.

A status constant is a shorthand method of writing a status function. The format of a status constant is:

S"stvalue"

Since a status constant does not indicate which status list it belongs to, it must be used only in a context where the status constant is directly attributable to a particular status list. Such contexts are:

Presetting a scalar or array item of type S.

Joining a status variable by an operator such as:

OPCODE=S"NOP"; IF OPCODE NE S"NOP" . . .

OPERATORS

Operators are used in arithmetic expressions and Boolean expressions. The operators are of type arithmetic, relational, and logical.

Arithmetic operators are of two types:

Numeric operators perform arithmetic operations to yield a numeric result.

Masking operators perform bit-bit-bit operations to yield a numeric result.

Relational operators work with arithmetic operands to produce a Boolean result.

Logical operators work with Boolean values and yield a Boolean result.

Table 1-3 shows the SYMPL symbols (reserved word) and their meanings for the different types of operators. Tables 1-4 and 1-5 show truth tables for the logical and masking operators.

EXPRESSIONS

An expression is a rule for computing a value. During evaluation of an expression the values of the operands in the expression are combined according to the language rules to form a single value.

Each of the following is an expression:

Constant

Scalar

Subscripted array item

Function reference, except the P function

TABLE 1-3. SYMPL OPERATORS

Symbol	Meaning
	Numeric Operators
+	Addition; unary plus.
-	Subtraction; unary minus.
*	Multiplication.
/	Division.
**	Exponentiation.
	Masking Operators
LNO	Logical NOT (bit-by-bit NOT).
LAN	Logical AND (bit-by-bit AND).
LOR	Logical OR (bit-by-bit OR).
LXR	Logical exclusive OR.
LIM	Logical imply.
LQV	Logical equivalent.
	Relational Operators
EQ	Is equal to.
GR	Is greater than.
GQ	Is greater than or equal to.
LQ	Is less than or equal to.
LS	Is less than.
NQ	Is not equal to.
	Logical Operators
NOT	Negation.
AND	Conjunction.
OR	Union.

TABLE 1-4. TRUTH TABLE FOR LOGICAL OPERATORS

b1	False	False	True	True
b2	False	True	False	True
	Logical			
NOT b1	T	T	F	F
b1 AND b2	F	F	F	T
b1 OR b2	F	T	T	T

TABLE 1-5. TRUTH TABLE FOR MASKING OPERATORS

a	0	0	1	1
b	0	1	0	1
	Masking			
LNO a	1	1	0	0
a LAN b	0	0	0	1
a LOR b	0	1	1	1
a LXR b	0	1	1	0
a LIM b	1	1	0	1
a LQV b	1	0	0	1

Further, any of the above entities combined with a unary operator or binary operator also produces an expression.

The two types of expressions are:

Arithmetic expressions that yield numeric values.

Boolean expressions that yield Boolean values of TRUE or FALSE.

Boolean operands and Boolean expressions differ in nature from arithmetic operands and expressions; they cannot be involved with numeric arithmetic expressions. No numeric arithmetic operator applies to any Boolean operand and vice versa.

Evaluation of an expression begins with evaluation of operators with higher precedence and continues with evaluation of operators with lower precedence; otherwise, evaluation proceeds left to right. A different order of evaluation can be specified by the programmer through the use of parentheses: expressions within parentheses are evaluated before the result is combined with other operands.

ARITHMETIC EXPRESSIONS

Arithmetic expressions yield a numeric value. The two types of arithmetic expressions are:

Numeric arithmetic expressions that involve operands of any type except Boolean. Operands are treated as a single value in these expressions.

Logical masking arithmetic expressions that involve operands of any type except Boolean, Operands are treated on a bit-by-bit level in these expressions.

For both types of expressions operators have implicit ranking, with evaluation of the expression preceeding from operators with higher precedence to operators with lower precedence.

Arithmetic operators are as follows. They are listed in order of highest to lowest precedence:

( )	Parentheses, beginning with innermost pair
**	Exponentiation
* /	Multiplication and division, from left to right
+ -	Unary plus and minus
+ -	Addition and subtraction, from left to right
LNO	Logical NOT (complement)
LAN	Logical AND
LOR	Logical inclusive OR
LXR	Logical exclusive OR
LIM	Logical imply
LQV	Logical equivalence

SYMPL has no implicit multiplication in which algebraic multiplication can be indicated by X(Y) or (X)(Y).

Numeric Arithmetic Expressions

A numeric arithmetic expression contains only numeric operands and numeric arithmetic operators. The numeric operators are: **, *, /, +, and -. The numeric operands include constants, scalars, subscripted array items, and function references; the type of any numeric operand must not be Boolean.

When operands of different types are used in a single expression, the compiler converts the type of one operand such that the common type of both operands is the higher type. The four operand types that exist for conversion purposes are as follows, listed in order from highest to lowest:

Real

Signed integer

Unsigned integer

Character.

For example, given integer item I and real item R, the expression (I + R) is evaluated in floating point arithmetic after the value of I is converted to type real. Similarly, the expression ((I + 2) * R) is computed by:

Adding I and 2 in integer mode

Converting the result to floating point format

Multiplying the result by R in floating point format.

Character operands are lowest in the conversion hierarchy. Conversion of type character to type integer is affected by the number of characters declared in the character operand. (The length of a scalar or array item is specified in its declaration; the length of a character constant is the number of characters in the string; the length of a C function is the number of characters indicated in the function.) If bit 59 of a 10 character operand is set, the converted integer is a negative value. If the operand has more than 10 characters, only the first 10 characters are used in an expression evaluation. For operands less than 10 characters, the characters are shifted right to normal integer position and zero filled.

Character-to-real conversion occurs by conversion to integer followed by conversion of the integer to a floating point format.

Conversion from type integer to type real occurs by floating the integer, as provided by hardware instructions. The resulting real value is expressed in single precision format.

Preset VAL to the unsigned integer value 2:

```
STATUS WORDS BEGIN, END, TERM;
ITEM VAL S:WORDS=S"TERM";
```

Set X to 3:

```
STATUS COLOR RED, OR, YEL, BLUE;
X=COLOR"BLUE";
```

Test LETTER for the display code value equivalent to Q:

```
STATUS ALPHA A,B, . . . X,Y,Z;
IF LETTER EQ S"Q" THEN. . .
```

SWITCH DECLARATION

A SWITCH declaration defines a list of label names that the compiler is to associate with small unsigned integer values. The purpose of the declaration is to allow mnemonic references to label names in a GOTO statement.

Two types of switches, and two SWITCH declaration formats, exist. The first is a straightforward list of label names; the second combines STATUS capabilities into the SWITCH declaration.

When a switch is referenced in a GOTO statement, the value of the switch subscript expression must be within the range of defined switches. If the program is compiled with the C parameter (range checking) on the compiler call, an execution-time check is made to determine whether the value is within the range of valid values. When range checking is selected, any value out of range produces a diagnostic and program abort. If range checking is not selected, any reference to an out of range switch value produces an undefined result.

ORDINARY SWITCH

In the simpler form of a switch, the compiler assigns a value to each label named. The first label in the list is assigned a value 0, the second label is assigned the value 1, and so forth.

The format of a SWITCH declaration specifying only label names is:

```
SWITCH swname label, label, . . . ;
```

swname	Name by which switch is known. Identifier of 1 through 12 letters, digits, or $ that does not begin with a digit and does not duplicate a reserved word.
label	Label name to be associated with swname. If the switch is never accessed by a particular value, a null parameter (two consecutive commas) can appear in the list for that value.

An example of the declaration and use of an ordinary switch AAA that transfers control to label LAB3 when the value of I is 2 is:

```
SWITCH AAA LAB1, LAB2, LAB3;
GOTO AAA[I];
```

STATUS SWITCH

A status switch references a previously declared STATUS declaration. The SWITCH declaration associates the switch name with a status list; each label name in the switch list is then paired with one of the identifiers from the status list as specified by the SWITCH declaration parameters.

The format of a SWITCH declaration specifying a status list is:

```
SWITCH swname:stlist label:stvalue, label:
  stvalue, . . . ;
```

swname	Name by which switch is known. Identifier of 1 through 12 letters, digits, or $ that does not begin with a digit and does not duplicate a reserved word.

stlist	Name by which status list is known, as declared by a previous STATUS declaration.
label	Label name to receive the same value as the status value following the colon.
stvalue	Status value from list stlist to be associated with the preceding label name.

The status values can appear in a switch list in an order other than that of their status list. Also, all of the status values need not be associated with a label. The same label can be associated with more than one status value. A status value, however, can only appear once in a switch list.

An example of a declaration of a status switch WHICHONE and its use to transfer control to LABZ when the value of the GOTO statement argument is 3 is:

```
STATUS COLOR RED, ORG, YEL, GRN;
SWITCH WHICHONE:COLOR LABX:YEL,
   LABZ:GRN;
.
.
.
GOTO WHICHONE[COLOR"GRN"];
```

ARRAY DECLARATION

An ARRAY declaration defines an arrangement of item-like elements. An array can be viewed as a rectangular assortment of entries, each composed of one particular occurrence of each item comprising the entry. The number of entries must be less than 65535.

In storage an array entry occupies an integral number of whole words. Items within the entry can be as small as one bit or as large as 24 words of character data; only type character items can cross the boundary of a word in the array, however.

An array is declared by an ARRAY declaration header followed by an ITEM declaration. If no items exist in the entry, a null declaration (blank followed by a semicolon) should follow the ARRAY declaration. If more than one item (field) exists in the entry, the ITEM declaration should be a compound statement.

The format of an ARRAY declaration header is:

```
ARRAY name [low:up, low:up, . . .]
   alloc (esize),
```

name	Identifier specifying the name of the array. It can be omitted unless the ARRAY declaration appears in a BASED ARRAY, XDEF, or XREF declaration.
low	Lower bound of a dimension of the array, expressed as an integer with modulo 2^{18}. Can be signed positive or negative. If low and its following colon are omitted, 0 is assumed.
up	Upper bound of a dimension of the array, expressed as an integer with a modulo 2^{18}. Can be signed positive or negative. Must be equal or greater than the preceding low with which it is paired.
alloc	Allocation of the entries in the array in storage. P Parallel allocation in which the first words of each entry are allocated contiguously, followed by the second words of each entry, and so forth. S Serial allocation in which all the words of one entry are allocated contiguously. If alloc is omitted, P is assumed.
esize	Entry size. Number of words in an array entry, expressed as an unsigned integer. Esize must be less than 2048 words. If esize and its enclosing parentheses are omitted, 1 is assumed.

An array can have up to seven dimensions. Each low:up pair in the ARRAY declaration defines a dimension of the array. (Dimensions specify the coordinates that identify an element of the array.) If the bounds list is omitted, [0:0] is assumed.

Differences between serial and parallel allocation are in figure 2-1. In this figure, array A has one dimension, a three word entry that occurs five times. CHAR[1] is the reference that accesses the second occurrence of item CHAR defined to occupy word 1 of the entry. A full declaration for this array might be:

```
ARRAY A[0:4] S(3);
    BEGIN
    ITEM HDR I(0,0,60);
    ITEM CHAR C(1,0,10);
    ITEM TRFR C(2,0,20);
    END
```

Parallel allocation offers execution advantages and should be used when possible.

The format of the ITEM declaration of an array is as follows. If more than one array item is being declared, all declarations should appear between BEGIN and END. The declaration is similar, but not identical, to the ITEM declaration for scalars.

```
ITEM name type(ep,fbit,size)=[preset],
   name type(ep,fbit,size)=[preset], . . . ;
```

name — Identifier specifying the name of the entry item, expressed as 1 through 12 letters, digits, or $ that does not begin with a digit and does not duplicate the name of a reserved word. Must be unique within procedure.

type — Type of array item:

B	Boolean
C	Character
I	Signed integer; default
U	Unsigned integer
R	Real
S:stlist	Status associated with list stlist

ep — Entry position. Word number in which the high-order bit of the item occurs, starting from 0; expressed as an unsigned integer constant.

fbit — Bit position at which item begins, starting on the left and counting from 0 through 59; expressed as an unsigned integer constant.

For a character item, fbit must be divisible by six.

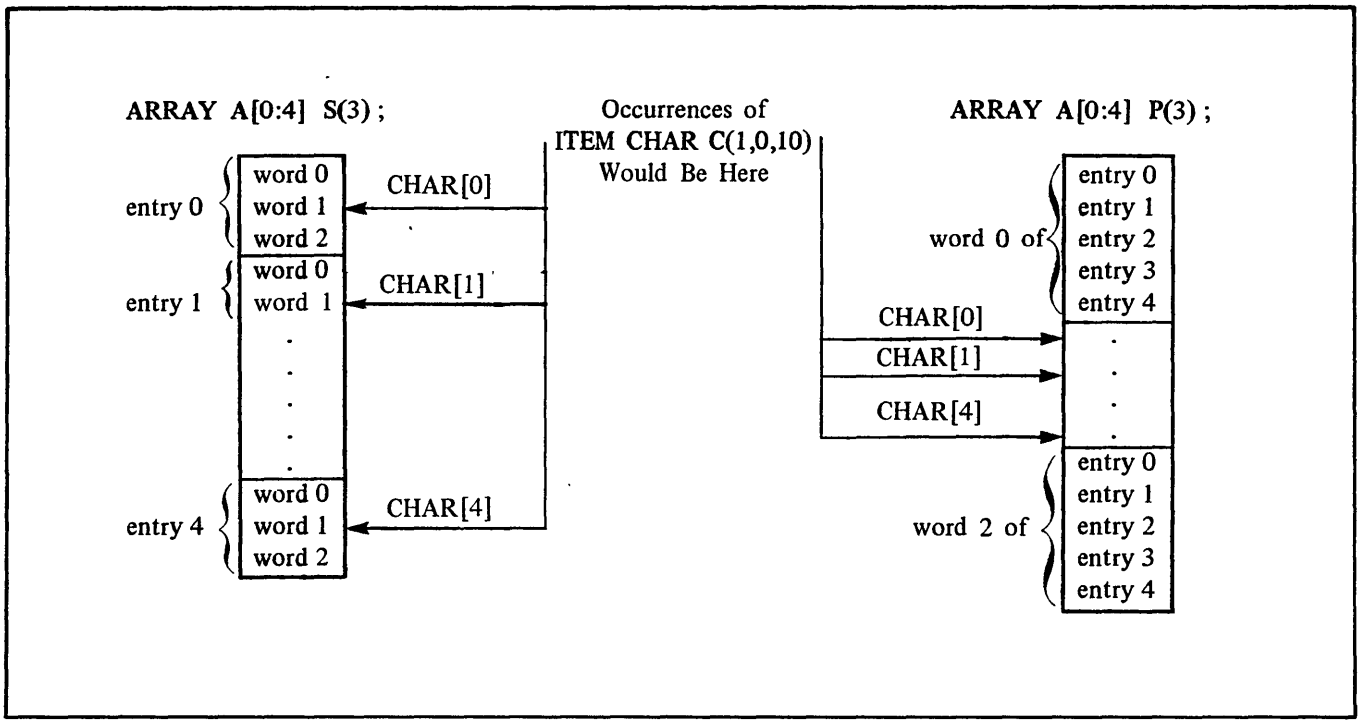


Figure 2-1. Differences in Serial and Parallel Allocation

size

Item length, expressed as an unsigned integer constant appropriate to the type, as shown in table 2-1. Only C type data can cross word boundaries.

R type data must have a size of 60.

preset

For a single occurrence array entry item, value to which item is to be initialized at load time, expressed as a constant.

For a multiple occurrence array entry item, a set of values arranged in a list in the same order as the allocation order of different instances of the items in storage.

Any constant specified is set in the item, aligned appropriately in the field, without regard to other fields in the word.

If the entire field descriptor (ep,fbit,size) is omitted, ep and fbit default to 0 and size defaults as shown in table 2-1. One parameter within the parentheses is assumed to be ep, with fbit=0 and size as in the table; two parameters are assumed to be ep and fbit.

ARRAY REFERENCES

A particular instance of an array item is known as an element. To reference a particular element, a subscript enclosed in brackets is appended to the array item name. For instance:

```
ARRAY REF[0:99];
ITEM REFITEM;
```

To reference the 40th element, which in this example is the 40th word, the reference is:

```
REFITEM[39]
```

The subscript for the array item must be an arithmetic expression. If the type of the arithmetic expression is other than integer, the result of the expression will be converted to integer mode of modulo 2^{17}.

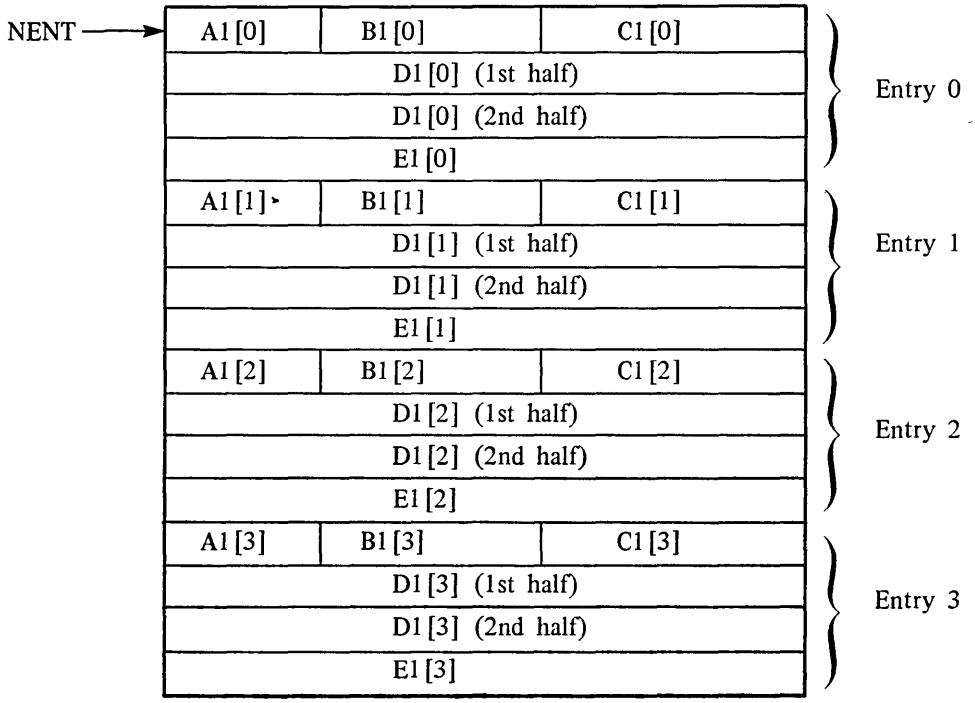
If the array being referenced has more than one dimension, the subscript must have as many arithmetic

TABLE 2-1. ARRAY ITEM DESCRIPTOR LIMITS

Type	fbit Alignment	Maximum Length	Default Length	May Cross Words
I	bit	60 bits	60	no
U	bit	60 bits	60	no
R	bit 0	60 bits	60	no
B	bit	60 bits	1	no
C	byte	240 bytes	1	yes
S	bit	60 bits	60	no

Table 2-1. Array Item Descriptor Limits

A. Serial Array Structure



B. Parallel Array Structure

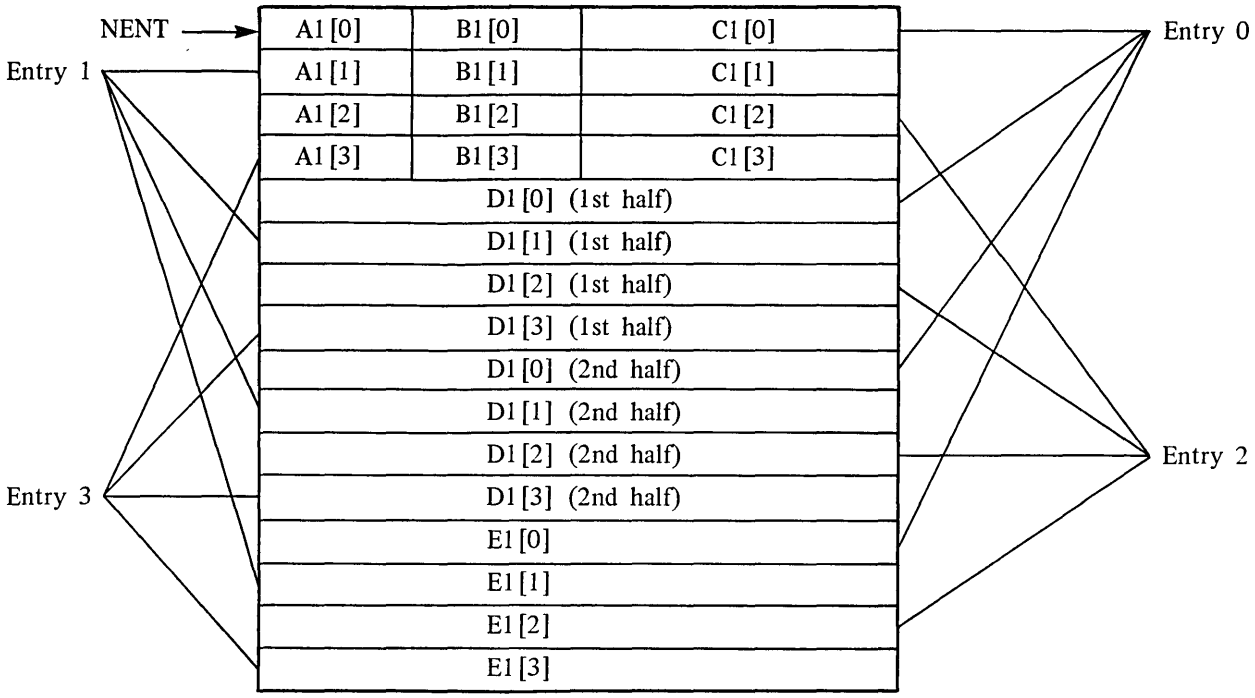


Figure 2-4. Serial and Parallel Arrays with Multiword Items

```
ARRAY TENWORD [0:4] S(2);
BEGIN ITEM A I(0,0,30)=[4, ,3, ,6];
      ITEM B I(0,0,45)=[ , 10, , 15];
      ITEM C C(1,0,5)=["YYYYY","XXXXX",
        "VVVVV","RRRRR","QQQQQ"];
END
```

Resulting structure and values are:

4	
Y Y Y Y Y	
	10
X X X X X	
3	
V V V V V	
	15
R R R R R	
6	
Q Q Q Q Q	

C[0]

C[1]

C[2]

C[3]

C[4]

Multidimensional arrays are preset using nested brackets. Brackets should be nested to the level of the number of subscripts. The leftmost subscript varies most rapidly, as it does in FORTRAN Extended.

Basically, the preset list for a declaration is a set of constant values, with the same order as the allocation order of the elements. This list is presented in sections enclosed in square brackets, and nested to a depth of the number of dimensions in the array. An N dimensional array at the first level of nesting has as many sections as the Nth dimension of the array. Each of these sections has as many sections as the N-1st dimension, and so forth. At the deepest level, each section has as many values as the first dimension of the array. Each section at the first level contains values for the instances of the array item with the same rightmost subscript; the subscript associated with each section varying from the lower bound at the left to the upper bound at the right. Each section of the second level contains values for those instances with the same rightmost two subscripts, and so forth. The outermost section is appended to the array item declaration with an equals sign.

Repetition of values can be indicated by bracketing a list of values with a parentheses and a count. For example:

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

and

2(2(0,2))is equivalent to 0,2,0,2,0,2,0,2

A two-dimensional parallel array, for example, is initialized by:

```
ARRAY OMEGA[0:1,0:2];
ITEM MU I(0,0)=[[1,2] [3,4] [5,6]];
```

This presetting is equivalent to:

```
ARRAY OMEGA[0:1,0:2];
ITEM MU I(0,0);
MU [0,0]=1;
MU [1,0]=2;
MU [0,1]=3;
MU [1,1]=4;
MU [0,2]=5;
MU [1,2]=6;
```

As with single-dimension arrays, not all elements of a multidimensional array need to initialized. Elements that are not to be initialized can be represented by null brackets as well as by brackets containing null values. For instance:

```
[[[, ,2] [, 1,]] [[, ,] [3,4,5]] [[, ,] [, ,]]]
```

is equivalent to

```
[[[, ,2] [,1]] [[ ] [3,4,5]] [ ]]
```

Repetition of bracketed sections is indicated by placing a count outside the bracket. For instance:

```
2[[1,3] [2(2)]]
```

is equivalent to

```
[[1,3] [2,2]] [[1,3] [2,2]]
```

Only the first 6000 words of an array can have preset values.

ARRAY STORAGE AND ADDRESSING

Given the array header:

ARRAY [b_1:u_1, b_2:u_2, . . .] alloc(esize);

the number of entries in the array is:

$$(u_1-b_1+1)(u_2-b_2+1)\ .\ .\ .(u_n-b_n+1)$$

At compilation time, an array is allocated the following amount of storage:

(number of entries)(esize)

The allocation of an element with respect to the location of its array name is affected by whether storage allocation is serial or parallel.

For serial allocation, the location of element $[s_1,s_2,\ .\ .\ .,s_n]$ is computed from:

$$e_i=s_i-b_i$$

$$\text{address}+ep+e_1(esize)+e_2(size_1+esize)+\ .\ .\ .$$
$$+e_n(size_1*\ .\ .\ .*size_{n-1}*esize)$$

where $size_i$ is u_i-b_i-1 and esize is entry size.

For parallel allocation:

$$\text{address}+ep*size_1*\ .\ .\ .*size_{n-1}+e_1+e_2*size_1$$
$$+\ .\ .\ .e_n*size_1*\ .\ .\ .*size_{n-1}$$

where address is the address of element $[b_1,\ .\ .\ .b_n]$.

For a three-dimension array, the relative location of A[i,j,k] with respect to $A[b_1,b_2,b_3]$ is given by:

location (A[i,j,k])=

location $(A[b_1,b_2,b_3])+(x+L(y+M(z)))$ (esize)

where $x=i-b_1$
$y=k-b_2$
$z=k-b_3$
$L=u_1-b_1+1$
$M=u_2-b_2+1$

A three-dimension array can be initialized, for example, by:

```
ARRAY XYZ[0:2,3:5,-4:-2];
ITEM PI(0,0,60)=[3[3(4)]];
```

Each element of an array resides in a particular row or column. For example:

	column 0	1	2	3
0	4	0	7	-8
row 1	23	-9	11	6
2	-7	14	-2	77

In this array, the value 77 resides in row 2, column 3. Because there are three rows and four columns, this array has the dimensions 3 by 4.

Array items are allocated in column order: that is, the leftmost subscript varies most rapidly.

In a two-dimensional array, memory locations are:

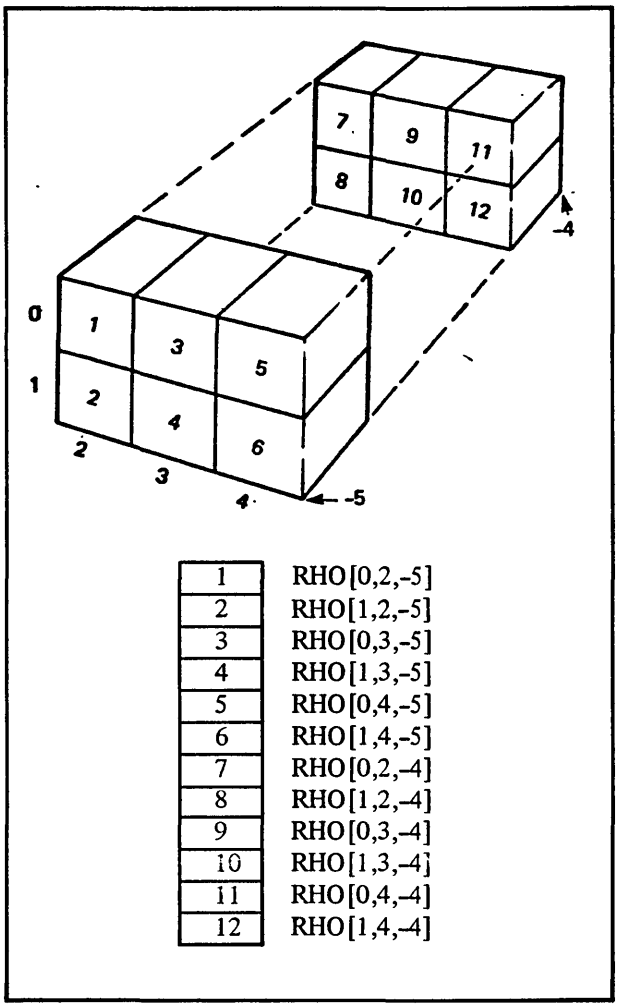
```
ARRAY PSI[1:3,0:3]alloc(2);
ITEM X,Y(1);
```

Parallel	Serial
X[1,0]	X[1,0]
X[2,0]	Y[1,0]
X[3,0]	X[2,0]
X[1,1]	Y[2,0]
X[2,1]	X[3,0]
X[3,1]	Y[3,0]
X[1,2]	X[1,1]
X[2,2]	Y[1,1]
X[3,2]	X[2,1]
X[1,3]	Y[2,1]
X[2,3]	X[3,1]
X[3,3]	Y[3,1]
Y[1,0]	X[1,2]
Y[2,0]	Y[1,2]
Y[3,0]	X[2,2]
Y[1,1]	Y[2,2]
Y[2,1]	X[3,2]
Y[3,1]	Y[3,2]
Y[1,2]	X[1,3]
Y[2,2]	Y[1,3]
Y[3,2]	X[2,3]
Y[1,3]	Y[2,3]
Y[2,3]	X[3,3]
Y[3,3]	Y[3,3]

For a three-dimensional array, the concept and memory locations are:

ARRAY RHO[0:1,2:4,-5:-4]P(1);

Resultant structure of array RHO is shown in figure 2-12.



1	RHO[0,2,-5]
2	RHO[1,2,-5]
3	RHO[0,3,-5]
4	RHO[1,3,-5]
5	RHO[0,4,-5]
6	RHO[1,4,-5]
7	RHO[0,2,-4]
8	RHO[1,2,-4]
9	RHO[0,3,-4]
10	RHO[1,3,-4]
11	RHO[0,4,-4]
12	RHO[1,4,-4]

Figure 2-5. Structure of Array RHO

BASED ARRAY DECLARATION

A based array is an array for which the compiler does not allocate storage; rather the compiler creates a specific pointer variable compiled with an undefined value. All references to a based array are compiled in relation to the pointer variable. From a logical standpoint, a based array provides a template that can be superimposed over any area of memory during execution.

A program using the based array has the responsibility to set the pointer variable through the intrinsic function P. The P function and its use with based arrays is described in section 4.

The based array name is declared in a BASED ARRAY declaration. The array items are declared as they are for normal arrays for which storage is allocated.

The format of the BASED ARRAY header is:

BASED array-dec;

or

BASED BEGIN array-dec, array-dec . . . END

array-dec — Full array declaration including the ARRAY declaration for a header and a simple or compound ITEM declaration for the entry in the array.

Based arrays should be used when the programmer does not know prior to execution time where the array is to be located. Based arrays are used, for instance, with a memory manager such as CMM when the position of an array is not known at load time.

References are made to based arrays just as if they were normal arrays, once the pointer variable is set.

EXCHANGE STATEMENT

The exchange statement causes the exchange of values of the left-hand and right-hand sides of the statement. Appropriate type conversion occurs during the exchange if necessary: in A==B, B is converted as if A=B appeared, with A converted as if B=A appeared.

The format of the exchange statement is:

v1 = = v2

vi — Entities whose values are to be exchanged. Any of the following can appear:

- Scalar
- Subscripted array item
- P-function
- Bead function

The two characters == must appear consecutively without an intervening blank.

SYMPL guarantees that subscript or bead function components of expressions which must be evaluated to compute the address of v1 or v2 are computed only once. The order of expansion as to which variable is stored first is not guaranteed, however. The exchange process refers to the expression values by referring to temporary variables. For example, the exchange statement A==B occurs as if it were written:

```
temp=A;
A=B;
B=temp;
```

Temporary variables are used for storage of component and subscript expressions, so that the old values are always used. The expansion of I==J[I] is:

```
temp1=I;
temp2=I;
I=J[I];
J[temp1]=temp2;
```

The subscript expression J[I] is the old value until the statement is complete.

FOR STATEMENT

The FOR statement is a generalized looping control statement. A simple or compound statement following the DO clause of FOR executes repetitively as long as the condition established by the FOR statement is TRUE.

The format of the FOR statement has several forms:

FOR i=aexp1 STEP aexp2 DO statement

FOR i=aexp1 STEP aexp2 UNTIL aexp3 DO statement

FOR i=aexp1 WHILE bexp DO statement

FOR i=aexp1 STEP aexp2 WHILE bexp DO statement

FOR i=aexp1 DO statement

i — Counter for the loop called the induction variable. Must be a scalar of any type except B or C.

aexp1 — Arithmetic expression indicating the initial value of the induction variable.

aexp2 — Arithmetic expression indicating a value to be added to the induction variable for each execution of the loop.

aexp3 — Arithmetic expression indicating the last value for the induction variable for which loop repetition is to occur.

statement — Simple or compound statement to be executed repetitively. This statement is called the controlled statement.

bexp — Boolean expression that must be TRUE for repetitive loop execution.

Since the form FOR i=aexp DO statement produces an infinite loop, the programmer-supplied statement must provide for an exit jump.

The expressions used in the STEP and UNTIL clauses can utilize data of any type. The result of the expression is converted to the mode of the induction variable.

Two types of loops, known as fastloops and slowloops, can be generated by the compiler, depending on the appearance of the compiler-directing CONTROL statement. Figure 3-1 compares the two types of loops.

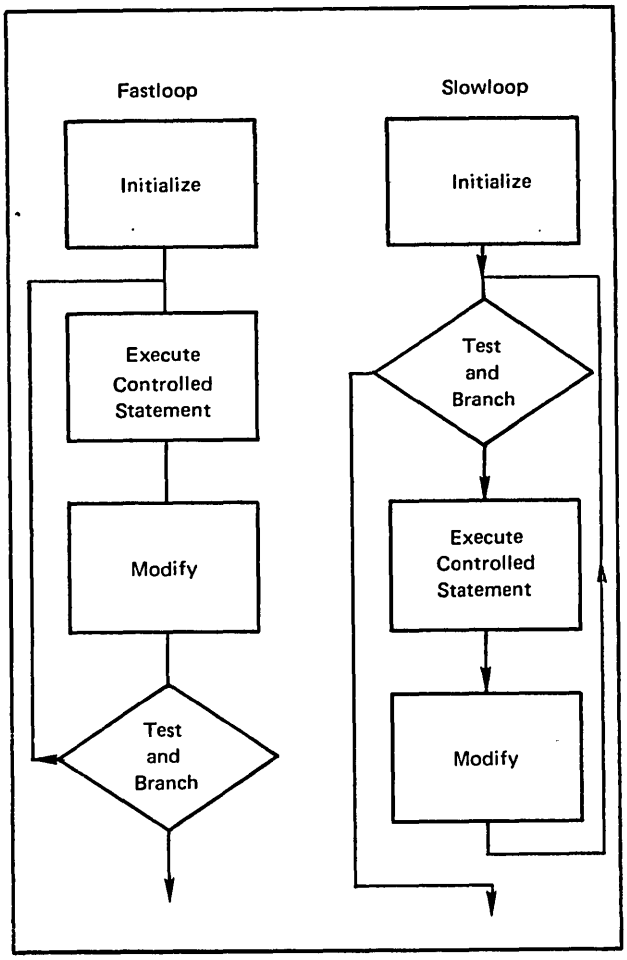


Figure 3-1. Generalized Fastloop and Slowloop Flowcharts

Fastloops always execute at least once (similarly to American National Standard X3.9-1966, FORTRAN DO loops) since the test for the condition is at the end of the loop. To produce predictable results, the elements of the FOR statement are restricted as follows:

> The induction variable must be integer type. It can be signed. The absolute value of the induction variable must be able to be contained within 17 bits.
>
> Neither the step nor the test expression can be modified within the loop. SYMPL might evaluate these expressions before the start of the loop.

Slowloops need not execute at least once since the test for the condition is at the beginning of the loop. The restrictions of fastloops do not hold for slowloops.

Fastloops are preferable since they can be optimized by the compiler.

The default is slowloop, but it can be overridden for following FOR statements: a CONTROL FASTLOOP statement affects all FOR statements begun before a later CONTROL SLOWLOOP statement. A loop control statement within a FOR statement can affect a nested loop, but not the loop in process. See section 5 for an example of loop control.

For both types of loops, the value of the induction variable is undefined after the loop is complete. For slowloops, however, the current value of the induction variable is preserved if the controlled statement causes a jump out of the loop. Moreover, if the controlled statement is entered by a GOTO statement from outside the FOR statement, the value of the induction variable might be undefined.

Figure 3-2 shows the different types of FOR statements and the logic of their generated code. For slowloops, the object code has a direct correspondence with the SYMPL statements shown; this is not the case with fastloops.

The step value and final value shown in figure 3-2 in temporary locations are not guaranteed: if variables involved in these expressions are modified within the loop, results are not predictable.

TEST STATEMENT WITHIN A FOR STATEMENT

In a FOR statement, the compiler automatically supplies the modification, test, and branching steps of a loop. The TEST statement provides a means of branching to the modify-test-branch step; it is meaningful only within the controlled statement of a FOR statement.

The format of the XDEF declaration is:

```
XDEF xdec
or
XDEF BEGIN xdec xdec . . . END
```

xdec — Name of any procedure, function or label that is to be referenced in an externally compiled program; or a full data declaration for a scalar, array, switch, or based array.

The xdec for a procedure, function or label is:

```
PROC name;

FUNC name type;

LABEL name, name, . . . ;
```

XDEF declarations for procedure and function names can occur either before or after the declarations of the procedure or function.

An example of use of the XDEF and XREF declarations is as follows:

Procedure A is compiled with:

```
XREF ITEM COUNT I;
```

Procedure B is compiled with:

```
XDEF ITEM COUNT I;
```

Any reference to COUNT from within procedure A accesses the storage reserved for the item within procedure B, assuming both A and B are available at load time.

XREF DECLARATION

The XREF declaration generates external references to the specified names. It is assumed that storage for variables is allocated and appropriately declared with XDEF in a separately compiled program.

The format of the XREF declaration is:

```
XREF xdec
or
XREF BEGIN xdec xdec . . . END
```

xdec — Any of the following whose storage is declared with XDEF:

Data declaration for a scalar without preset.

Data declaration for an array without presets.

Data declaration for a based array.

```
PROC name;

FUNC name type;

LABEL name, name, . . . ;

SWITCH name, name, . . . ;
```

XREF itself is not terminated by a semicolon, but each declaration within the XREF statement requires a terminating semicolon.

Examples of XREF statements are:

```
XREF BASED ARRAY AA; ITEM XX;

XREF BEGIN
       SWITCH JUMVEC;
       FUNC LINEUP R;
       ARRAY[0:9,0:9]S(5);
          BEGIN
          ITEM ZZ C(0,0,40);
          ITEM YY R(4,0,60);
          END
       END
```

Each parameter in the actual parameter list is delimited by the final parenthesis or a comma. A parameter consists of all the characters between successive parameter delimiters.

Any character can appear as part of the actual parameter string, but characters with syntax-defining meaning might require special coding:

Any parameter string that contains a semicolon must be bounded by #. The bounding # are removed prior to substitution.

Any parameter string that contains # must specify ## to produce a single # substitution.

Any parameter string that contains incorrectly unbalanced or nested ( ), < > , or [ ] must be bounded by #. The bounding # are removed prior to substitution.

Any comma within a parameter string is not recognized as a parameter delimiter when that comma is contained within a balanced set of ( ), < >, or [ ].

All actual parameters valid for a procedure or function call are valid as DEF parameter strings. No restriction limits the DEF name reference parameter strings to items or expressions, however.

For example:

- Define BYTE and reference it by BYTE(C,5,2**J):

```
DEF BYTE(B,J,K) # B<J>A[K] #;
```

Expansion produces:

```
C<5>A[2**J]
```

- Define CHECK with two parameters and a body that uses the BYTE specified above:

```
DEF CHECK(X,ERROR) # IF BYTE(B,1,X)
  EQ 1 THEN GOTO OK; ERROR#;
```

Reference:

```
CHECK(CALL(3,B),#ERROR=37;
  GOTO FAIL#);
```

Expansion:

```
IF B<1>A[CALL(3,B)] EQ 1 THEN GOTO
  OK; ERROR=37; GOTO FAIL;
```

- Another definition of CHECK with the same parameters produces the following expansion, given the same reference:

```
DEF CHECK(X,ERROR)#IF BYTE
  (B,1,##X##) EQ 1 THEN GOTO OK;
  ERROR#;
```

Expansion:

```
IF B<1>A[X] EQ 1 THEN GOTO OK;
  ERROR=37; GOTO FAIL;
```

DEF NAME REFERENCES

Once a DEF name has been defined, subsequent references to that name are replaced by the characters in DEF body. No substitution occurs in the following circumstances, however:

The DEF name appears within a comment.

The DEF name appears within a constant.

The DEF name or the DEF parameter name appears as the identifier being defined by an ITEM, ARRAY or COMMON declaration.

The DEF name corresponds to one of the following and the name appears in a syntax-defining context:

Type descriptor abbreviations B, C, I, R, S, U.

Array layout specifiers P, S.

Constant prefixes O, S, X.

Intrinsic function B, C, P.

Real number specifiers D, E.

When the DEF declaration does not include parameters, compilation simply replaces the DEF name with the DEF body.

When the DEF declaration includes parameters, each reference to the DEF name must be followed by an actual parameter list. The format of the DEF name reference with parameters is:

```
name(param,param, . . . )
```

name — Name defined in a prior DEF declaration within this subprogram.

param — String of characters to replace a formal parameter.

No comment can appear between the DEF name and the left parenthesis of the actual parameter list.

A one-to-one correspondence exists between the positions of parameters in each list. The first actual parameter replaces all occurrences of the first formal parameter within the DEF body; the second actual parameter replaces all occurrences of the second parameter; and so forth. The number of actual parameters must not exceed the number of formal parameters: such a condition is detected as a fatal error and DEF name substitution is suppressed.

The number of actual parameters can be fewer than the number of formal parameters, however. Any formal parameter without a corresponding actual parameter is replaced by a null character string. This allows the expansion of a DEF name with a variable number of actual parameters.

CONTROL STATEMENT

The CONTROL statement directs the compiler to take immediate action. Several different types of control words in the statement cause different types of actions:

Output listing control specifications are EJECT, LIST, NOLIST, OBJLST.

Conditional compilation control words are IF, FI, ENDIF.

Compilation option selections are PACK, PRESET, FTNCALL.

FOR statement loop specifications are FASTLOOP, SLOWLOOP.

Core residence selections are LEVEL1, LEVEL2, LEVEL3.

Variable attribute specifications are DISJOINT, OVERLAP, REACTIVE, INERT.

Weak external specification is WEAK.

Traceback selection is TRACEBACK.

Each of the different functions is described separately below.

A CONTROL statement can appear anywhere in a program that a statement can appear. It can also appear within BEGIN and END enclosing a list of array items, based arrays, external declarations, or common declarations.

The effect of a CONTROL statement can be reflected in an entire module. The end of a procedure or function does not cancel the statement; only TERM cancels a CONTROL statement.

LISTING CONTROL

Four forms of the CONTROL statement affect output listings. The general format is:

CONTROL control-word;

Control-word — One of the following:

EJECT	Skip to new page of listing
LIST	Resume normal listing of source statements
NOLIST	Suspend normal listing of source statements
OBJLST	List object code

EJECT, LIST, and NOLIST cause the compiler to take action at the time the statement is encountered among the source statements.

OBJLST applies to the entire module. Its appearance anywhere within the module affects the entire module.

The H parameter of the SYMPL compiler call overrides CONTROL NOLIST.

CONDITIONAL COMPILATION

The CONTROL statement can be used to determine whether source statements following the CONTROL statement are to be compiled:

When the relationship defined in the CONTROL statement tests TRUE, the following source statements are compiled.

ATTRIBUTES OF VARIABLES SPECIFICATION

The SYMPL compiler attempts to produce efficient executable code. Because the compiler cannot predict the precise use of a variable in subsequent source statements, it must forego many efficiencies that would produce inaccurate code by particular variable references. The programmer, however, can be aware of data use and, through the CONTROL statement, can inform the compiler of usage characteristics. By classifying variables and array items as separate or potentially overlapping, the programmer provides the information that the compiler needs to decide optimizations.

The format of the CONTROL statement for specifying attributes of variables is:

```
CONTROL attribute var, var, . . . ;
or
CONTROL attribute;
```

attribute — Attribute of variables in the statement list:

- OVERLAP — Variables might be referenced by more than one name, as shown in examples below. OVERLAP is the opposite of DISJOINT.
- DISJOINT — Variables are referenced by a single name only. DISJOINT is the opposite of OVERLAP.
- REACTIVE — A given word in a single array might contain two items, or parts of items, being referenced together although the two items are not declared to overlay each other. See examples below. REACTIVE is the opposite of INERT.

  Items with declarations that show one field overlaying another field are detected by the compiler, so that REACTIVE need not be declared.
- INERT — A given word in a single array does not contain items, or parts of items, referenced together. INERT is the opposite of REACTIVE.

var — Variable with the attribute specified.

If the list of variables is omitted, the CONTROL statement becomes a global switch that affects all subsequently declared variables not otherwise referenced by a contrary individual specification.

If neither the global switch format nor the individual specification format of the CONTROL statement appears, the module is compiled as described in appendix C, Possible Optimizations. If any CONTROL statement specifying an attribute appears in the module, the global switch format CONTROL REACTIVE and CONTROL OVERLAP is assumed at the beginning of a module. Use of the CONTROL statement to classify variables is encouraged because future versions of the compiler might require such classification.

The definitions of overlap and disjoint refer only to variables in separate arrays; for overlapping items within a single array, the distinction between reactive and inert must instead be drawn.

Overlapped Variables

One program might refer to the same variable by two names when formal parameters or based arrays are referenced. For example:

```
PROC P(A,B);
.
.
.
A=2;
B=4;
Y=A;
```

A call to procedure P in the form P(V,V) represents two occurrences of the same actual parameter: during compiler optimization the store of the value of Y must not use the value of A from the A=2 statement.

Similarly, with a based array B based on A:

```
PROC P(A,B);
X=A[2];
B[2]=3;
Y=A[2];
```

Since A and B refer to the same array, the compiler must not store Y such that it refers back to the first A[I].

Variable names that interfere with each other as illustrated above are called overlapped variable names. If such interference does not occur, the variables are said to be disjoint.

To determine whether variables should be specified as OVERLAP or DISJOINT, the programmer must examine the entire module, not simply a given subprogram. The compiler reserves the right to inspect all procedures and functions in a given module for use of variables and it considers that normal nonexternal variables are not destroyed by calls to global subprograms whether external or not. But if local procedures are called which have access to the names of local variables, the compiler detects all the variables such a procedure explicitly stores.

Variables known through COMMON, XDEF, and XREF declarations are considered destroyed by calls to an external subprogram. Overlapped behavior exists when an external subprogram destroys nonexternal variables.

Reactive Arrays

Two items in one array can interfere with optimization when references to items do not match the declarations of these items. For example:

```
ARRAY [0:100] S (1);
  ITEM A (0), B (1);
.
.
.
B[I]=A[J]*2;
Q=A[J];
```

Item B is outside the bounds of one array entry and it interfers with the next entry. If the array is always indexed by 2, B does not interfere with A. However, if I is set to J-1, the A(J) is destroyed by a store to B(I).

Array items that interfere with each other as in this illustration are said to be reactive items. If such interference does not occur, the items are said to be inert. An array is reactive if it has two items A and B such that for A[i] and B[j] with i not equal to j at some time during execution, any part of A[i] is in the same word as any part of B[j]. It is not necessary for the fields to overlap: reactive arrays occur when both items are in the same word.

To determine whether an array item should be classified as REACTIVE or INERT, the programmer must examine an entire module, including all variables affected by other procedures it might call.

WEAK EXTERNALS

When a compiled program is loaded before execution, the loader searches for a matching entry point for all externals and loads the subprogram in which they occur. Under some circumstances this can result in the loading of subprograms not required for current execution. Through using a CONTROL statement to declare an external weak, the programmer can specify that the external is not necessarily to be satisfied.

A weak external does not cause a search for the matching entry point. If the program that contains the entry point is loaded for some other reason, however, that weak external is linked.

When a weak external is satisfied, it is linked as if it were a normal external. If it is not satisifed, no error message is produced.

The format of the CONTROL statement specifying a weak external is:

```
CONTROL WEAK name, name, . . . ;
```

name — Name of array, based array, function, item, label, procedure, or switch.

Name must have been previously declared as external by using XREF.

TRACEBACK FACILITY

SYMPL uses standard calling sequences for transferring control to a procedure or subroutine of another language. In this sequence, register A1 contains the address of a parameter list and each parameter to be passed occupies one word of the list. Execution of an RJ instruction to the entry point links the programs. For debugging purposes, SYMPL provides an option for traceback.

The format of the CONTROL statement for tracing purposes is:

```
CONTROL TRACEBACK;
```

The appearance of this statement anywhere within the module selects the option for the entire module. Traceback code is generated automatically when the K parameter (points-not-tested) of the SYMPL compiler call is used.

The traceback code generated for procedures and functions is compatible with traceback of FORTRAN Extended. To complete FORTRAN Extended compatibility, the F parameter of the SYMPL compiler call must also be specified. Code generated by a SYMPL calling program is never compatible with FORTRAN Extended traceback, however.

Traceback code generated is as follows:

If the procedure of function has a single entry, the generated constant word is:

```
VFD 42/0Hname,18/ept
```

name Subprogram name left-justified and blank filled or truncated to seven characters.

ept Address of subprogram entry point.

If the procedure or function has multiple entries, the generated constant word is:

```
VFD 42/0Hname,18/temp
```

name Subprogram primary entry point.

temp Address of a copy of the return information taken from the most recent entry point.

The return jump instruction for the subprogram call is forced upper. The lower 30 bits of the instruction contain:

```
VFD 12/line,18/trace
```

line Approximate source line number of call.

trace Address of the constant word described above for the innermost subprogram containing the call statement.

COMPILER CALL AND OUTPUT LISTINGS 6

COMPILER CALL

The SYMPL compiler is called with a control statement that conforms to operating system syntax. The control statement cannot be continued.

More than one program or subprogram can be compiled by a single call to the compiler as long as they follow each other on the source file without any file boundaries between them. The compiler recognizes a TERM statement as the end of a module and ignores any further statements on the same card or card image. Compilation resumes with the next card, which is assumed to be the start of another program or subprogram. A comment can precede a program or subprogram header.

If the first card or card image encountered at the beginning of a loader module contains the character OVERLAY in columns 1 through 7, the remainder of the module is treated as if an LCC statement appeared in a COMPASS program.

The name on the compiler call statement is SYMPL. If all default parameters are selected, the compiler call appears as:

```
SYMPL.
```

A variety of compilation options can be specified in a parameter list following the compiler call name. If the name of the source input file is NEWONE, for example, the compiler call appears as:

```
SYMPL,I=NEWONE.
```

All compilation parameters are optional and can appear in any order. Parameters are listed below in alphabetical order.

A ABORT JOB AFTER ERRORS

omitted	Execute next control statement whether or not any errors are diagnosed during compilation.
A	Execute control statement after an EXIT(S) control statement if errors are found at the end of compilation.

B BINARY CODE FILE

omitted	Write binary output from compilation to file LGO.
B	Write binary output from compilation to file LGO.
B=0	Suppress generation of binary code.
B=lfn	Write binary output from compilation to file lfn, where lfn is one through seven letters or digits beginning with a letter.

C CHECK SWITCH RANGE

omitted	Do not generate code to check range of switch references. Any reference to an undefined switch value produces either an endless loop, a mode error, or a wild jump.
C	Generate code to check range of switch references. During execution any reference to an out-of-range switch or an unspecified switch value produces a diagnostic and a program abort.

D PACK SWITCHES

omitted	Generate one word for each switch.
D	Generate one word with two switch points, reducing the size of generated code but increasing execution time. Produces the same result as CONTROL PACK within a program.

E COMPILE $BEGIN/$END STATEMENTS

omitted — Do not compile source statements bracketed between $BEGIN and $END.

E — Compile source statements bracketed between $BEGIN and $END.

F FORTRAN CALLING SEQUENCE

omitted — Do not compile a word of all zeros at the end of a parameter list.

F — Compile a word of all zeros at the end of each parameter list as required by the FORTRAN Extended calling sequence. Produces the same result as a CONTROL FTNCALL statement within a program.

H LIST ALL SOURCE STATEMENTS

omitted — List source statements according to CONTROL NOLIST and CONTROL LIST statements within the program.

H — List all source statements, regardless of CONTROL NOLIST statements within the program.

I SOURCE INPUT FILE

omitted — Compile card images from file INPUT.

I — Compile card images from file COMPILE.

I=lfn — Compile card images from file lfn.

K POINTS-NOT-TESTED

omitted — Do not generate points-not-tested interface code.

K — Generate an RJ to the points-not-tested interface routine after every label and conditional jump. Find all paths in the executable code and determine which of the paths are exercised by the test base. Also, generate traceback code.

L LISTING FILE

Any O, R, or X parameter must be concatenated with any L parameter, as in: LXOR=PRINTIT.

omitted — Write source statement listing and diagnostics to file OUTPUT.

L — Write source statement listing and diagnostics to file OUTPUT.

L=1 — Write summary of resources used to file OUTPUT.

L=0 — Suppress all listing output, including that selected by O, R, and X; list only diagnostics.

L=lfn — Write source statement listing and diagnostics to file lfn, with lfn being one through seven letters or digits beginning with a letter.

N CROSS REFERENCE UNREFERENCED ITEMS

omitted — List only referenced items on the cross reference map selected by the R parameter.

N — List referenced and unreferenced data items on the cross reference map selected by the R parameter.

O LIST OBJECT CODE

Any L, R, or X parameter must be concatenated with any O parameter, as in: OL=LIST/35/45.

omitted — Do not list binary object code.

O=st/end — List binary object code generated by range of source statements indicated:

- st — Number of first source statement whose object code is to be listed. Default is 0.
- end — Number of last source statement whose object code is to be listed. Default is last statement in program.

If only one number appears after =, it is presumed to be end. The line numbers appear to the left of the source images on the listing.

O=lfn/st/end — List binary object code from specified source statements on file lfn, where lfn is one through seven letters or digits beginning with a letter. st and end are as above.

P PRESET COMMON

omitted — Data items in common blocks are not to be initialized.

P — Initialize data items in common blocks according to the preset values in the data declarations. Produces the same result as a CONTROL PRESET statement within a program.

R LIST CROSS-REFERENCE MAP

Any L, O, or X parameter must be concatenated with any R parameter, as in: RX=SHOW.

omitted — Do not list cross reference table and common blocks.

R — List cross reference table and common blocks on file OUTPUT.

R=lfn — List cross reference table and common blocks on file lfn, where lfn is one through seven letters or digits beginning with a letter.

S EXECUTION LIBRARY

omitted — Compile LDSET tables with references to these libraries:

SYMLIB/FORTRAN for NOS and NOS/BE operating systems

SYMIO/FORTRAN for SCOPE 2 operating system

S=0 — Suppress LDSET table generation.

S=lib — Generate LDSET tables with references to library lib. Multiple libraries can be specified with slashes between library names, as in: S=AAA/MMM/TTT.

T SYNTAX CHECK

omitted — Check syntax and generate binary code.

T — Check syntax, but do not generate binary code.

W SINGLE STATEMENT CODE GENERATION

omitted — Generate object code with multiple source statement intermixed.

W — Generate object code that maintains a close correspondence with its source statement. While the resulting object code might be less efficient, it is useful for debugging.

X LIST STORAGE MAP

Any L, R, or O parameter must be concatenated with any X parameter, as in: RX=OUTPUT.

omitted — Do not list storage map or common blocks.

X — List storage map and common blocks on file OUTPUT.

X=lfn — List storage map and common blocks on file lfn, where lfn is one through seven letters or digits beginning with a letter.

Y SUPPRESS DIAGNOSTIC 136

omitted — List diagnostic 136 (Semi ends comment) as required.

Y — Suppress diagnostic 136 listing, but take normal corrective action.

OUTPUT LISTINGS

Figure 6-1 shows a SYMPL main program SORT100 that can be used to sort 100 items. It calls procedure SORTER which was compiled separate from SORT100 since TERM appeared at the end of SORT100. SORT-100 consequently contains an XREF statement that declared SORTER to be an external program.

A job deck for syntax analysis compilation both the main program and subprogram would appear as:

```
jobcard.
any accounting statement.
SYMPL,T.
7/8/9
all SYMPL source statement
6/7/8/9
```

Output from a compilation normally includes the source statement listing, and a diagnostic summary.

```
PRGM SORT100  ;
BEGIN
BASED ARRAY AA[99]  ;
      ITEM X  ;
ITEM NOREFERENC:
XREF PROC SORTER;
ARRAY TOBESORTED [99]  ;
      ITEM T:
P<AA> = LOC(TOBESORTED)  ;
SORTER (P<AA>);
END
TERM
      PROC SORTER(SORT);
      BEGIN
      ARRAY SORT[99]:
          ITEM VALUE;
      ITEM I;
      ITEM FLAG I=0  ;
L1:   FOR I=0 STEP 1 UNTIL 98 DO
                IF VALUE[I+1] GR VALUE[I] THEN
                BEGIN
                VALUE[I+1] == VALUE[I]:
                FLAG = 1;
                END
      IF FLAG EQ 0 THEN
                RETURN;
      FLAG = 0 ;
      GOTO L1;
      END ESORTERE
      TERM
```

Figure 6-1. Sample Source Program

Any storage map, cross-reference map, or object listing follows on a separate page of the listing. The last information shown summarizes the number of words of memory and the time required for compilation. The parameters of the compiler call used for compilation, whether selected explicitly or implicitly, are also shown.

A large map might appear on the output listing in two parts. Both should be examined.

STORAGE MAP

The storage map is a dictionary of all programmer-created declarations in the source program. It is selected by the **X** parameter of the compiler call. Figure 6-2 shows the storage map from the SORT100 main program of figure 6-1. Information appearing on the map includes:

1 NAME First ten characters only of declarations are printed.

2 TYPE Defines the name as one of the following types:

ARYITM	Array item
COMMON	Common block
ITEM	Item
FUNC	Function
PROC	Procedure
LABEL	Label
B.ARRY	Based array
ARRAY	Array
PROGRAM	Program

3 M Mode of data representation

B	Boolean
C	Character
I	Integer
P	Parallel (arrays only)
S	Status (Serial if type is array or based array)
U	Unsigned integer
X	External
Y	Weak external

4 LOC Octal address relative to start of routine; if followed by C, LOC is relative to start of common block. If type = ARYITM, LOC refers to first occurrence of item.

5 FBIT First bit, numbered from 0 to 59, left to right.

6 NUM Number of bits; if MODE = C, number of bytes.

CROSS-REFERENCE MAP

The cross-reference map lists the properties of each declaration and shows the source line number at which the entity was declared or referenced. It is selected by the R parameter of the compiler call.

Figure 6-3 shows the cross-reference map from subprogram SORT100 of figure 6-1. Since the subprogram was compiled with the N parameter of the SYMPL compiler call, items that were declared, but not referenced, also appear on the map. Information appearing on the map includes:

1 NAME First ten characters only of declarations are printed.

2 TYPE Defines the name as one of the following types:

ARYITM	Array item
COMMON	Common block
ITEM	Item
FUNC	Function
PROC	Procedure
LABEL	Label
B.ARRY	Based array
STSCON	Status constant
DEFINE	DEF
STSLST	Status list
PROGRM	Program
ARRAY	Array

```
      SORT100   PROGRAM              * STORAGE MAP *

   (1)   (2)  (3)(4)    (5) (6)

NAME:0(10) TYPE   M LOC     FBIT NUM    NAME:C(10) TYPE    M LOC     FBIT NUM    NAME:C(10) TYPE    M LOC     FBIT NUM

AA         B.ARRY P       0             I          ARYITM I       2     0  60    NOREFERENC ITEM    I       1     0  60
SORTER     PROC   X       0             SORT100    PROGRM       151             SYS=       PROC    X       0
TOBESCRTED ARRAY  P       2             X          ARYITM I       0     0  60
```

Figure 6-2. Storage Map

3 M Mode of data representation

B	Boolean
C	Character
I	Integer
P	Parallel (arrays only)
S	Status (serial if type = array)
U	Unsigned integer
X	External
Y	Weak external

4 DEF Line number in source listing where declaration is defined; if followed by C, declaration is in common block.

5 SCOPE Name of outermost procedure within which declaration occurs; if type = STSCON, SCOPE is the name of the status list of which the item is a member.

6 SET/USED Source listing line numbers of references to NAME, * indicates use as other than left-hand side of the replacement statement.

```
SORT100   PROGRAM          * CROSS REFERENCE *
   (1)          (2)     (3)  (4)   (5)      (6)
NAME:C(10)  TYPE     M   DEF SCOPE  SET/USED/ATTRIBUTE-*=USED,A=ATTRIBUTE

AA          R.ARRY   I    3  SORT100      9      10
NOREFERENC  ITEM     I    5  SORT100
SORTER      PROC     X    6  SORT100     10*
T           ARYITM   I    8  SORT100
TOBESORTED  ARRAY    P    7  SORT100      9*
X           ARYITM   I    4  SORT100
```

Figure 6-3. Cross-Reference Map

STANDARD CHARACTER SETS A

CONTROL DATA operating systems offer the following variations of a basic character set:

CDC 64-character set

CDC 63-character set

ASCII 64-character set

ASCII 63-character set

The set in use at a particular installation was specified when the operating system was installed.

Depending on another installation option, the system assumes an input deck has been punched either in 026 or in 029 mode (regardless of the character set in use). Under NOS/BE the alternate mode can be specified by a 26 or 29 punched in columns 79 and 80 of the job statement or any 7/8/9 card. The specified mode remains in effect through the end of the job unless it is reset by specification of the alternate mode on a subsequent 7/8/9 card.

Under NOS, the alternate mode can be specified by a 26 or 29 punched in columns 79 and 80 of any 6/7/9 card, as described above for a 7/8/9 card. In addition, 026 mode can be specified by a card with 5/7/9 multipunched in column 1, and 029 mode can be specified by a card with 5/7/9 multipunched in column 1 and a 9 punched in column 2.

Graphic character representation appearing at a terminal or printer depends on the installation character set and the terminal type. Characters shown in the CDC Graphic column of the standard character set table are applicable to BCD terminals; ASCII graphic characters are applicable to ASCII-CRT and ASCII-TTY terminals.

Display Code (octal)	CDC			ASCII		
	Graphic	Hollerith Punch (026)	External BCD Code	Graphic Subset	Punch (029)	Code (octal)
00†	: (colon)††	8-2	00	: (colon) ††	8-2	072
01	A	12-1	61	A	12-1	101
02	B	12-2	62	B	12-2	102
03	C	12-3	63	C	12-3	103
04	D	12-4	64	D	12-4	104
05	E	12-5	65	E	12-5	105
06	F	12-6	66	F	12-6	106
07	G	12-7	67	G	12-7	107
10	H	12-8	70	H	12-8	110
11	I	12-9	71	I	12-9	111
12	J	11-1	41	J	11-1	112
13	K	11-2	42	K	11-2	113
14	L	11-3	43	L	11-3	114
15	M	11-4	44	M	11-4	115
16	N	11-5	45	N	11-5	116
17	O	11-6	46	O	11-6	117
20	P	11-7	47	P	11-7	120
21	Q	11-8	50	Q	11-8	121
22	R	11-9	51	R	11-9	122
23	S	0-2	22	S	0-2	123
24	T	0-3	23	T	0-3	124
25	U	0-4	24	U	0-4	125
26	V	0-5	25	V	0-5	126
27	W	0-6	26	W	0-6	127
30	X	0-7	27	X	0-7	130
31	Y	0-8	30	Y	0-8	131
32	Z	0-9	31	Z	0-9	132
33	0	0	12	0	0	060
34	1	1	01	1	1	061
35	2	2	02	2	2	062
36	3	3	03	3	3	063
37	4	4	04	4	4	064
40	5	5	05	5	5	065
41	6	6	06	6	6	066
42	7	7	07	7	7	067
43	8	8	10	8	8	070
44	9	9	11	9	9	071
45	+	12	60	+	12-8-6	053
46	-	11	40	-	11	055
47	*	11-8-4	54	*	11-8-4	052
50	/	0-1	21	/	0-1	057
51	(	0-8-4	34	(	12-8-5	050
52	)	12-8-4	74	)	11-8-5	051
53	$	11-8-3	53	$	11-8-3	044
54	=	8-3	13	=	8-6	075
55	blank	no punch	20	blank	no punch	040
56	, (comma)	0-8-3	33	, (comma)	0-8-3	054
57	. (period)	12-8-3	73	. (period)	12-8-3	056
60	≡	0-8-6	36	#	8-3	043
61	[	8-7	17	[	12-8-2	133
62	]	0-8-2	32	]	11-8-2	135
63	% ††	8-6	16	% ††	0-8-4	045
64	≠	8-4	14	" (quote)	8-7	042
65	←	0-8-5	35	_ (underline)	0-8-5	137
66	∨	11-0 or 11-8-2†††	52	!	12-8-7 or 11-0†††	041
67	∧	0-8-7	37	&	12	046
70	↑	11-8-5	55	' (apostrophe)	8-5	047
71	↓	11-8-6	56	?	0-8-7	077
72	<	12-0 or 12-8-2†††	72	<	12-8-4 or 12-0†††	074
73	>	11-8-7	57	>	0-8-6	076
74	≤	8-5	15	@	8-4	100
75	≥	12-8-5	75	\	0-8-2	134
76	¬	12-8-6	76	~ (circumflex)	11-8-7	136
77	; (semicolon)	12-8-7	77	; (semicolon)	11-8-6	073

†Twelve zero bits at the end of a 60-bit word in a zero byte record are an end of record mark rather than two colons.

††In installations using a 63-graphic set, display code 00 has no associated graphic or card code; display code 63 is the colon (8-2 punch). The % graphic and related card codes do not exist and translations yield a blank (55_8).

†††The alternate Hollerith (026) and ASCII (029) punches are accepted for input only.

DIAGNOSTICS B

The SYMPL compiler recognizes errors in SYMPL syntax. An applicable diagnostic message is printed on OUTPUT immediately preceding the line on which the error was detected. In addition, the total number of diagnostic messages is printed along with a detailed listing of each message number and the condition that caused the error.

The compiler aborts under several conditions:

Error in the compiler call. A dayfile message PARAMETER n IN ERROR is generated.

An attempt is made to compile some types of incorrect programs. An internal diagnostic message accompanies such an abort.

Other dayfile messages that might be produced include:

–SYMPL– INSUFFICIENT FL
–SYMPL– INSUFFICIENT SCM FL
–SYMPL– INSUFFICIENT LCM FL
–SYMPL– EMPTY INPUT FILE
–SYMPL– COMPILER ABORT
–SYMPL– BAD EXP CALL TO FTN
–SYMPL– BAD LOADER CALL
–SYMPL– ccccccccc COMPILED cp secs

Table B-1 lists the message number and text of the compilation diagnostics. Abbreviations used in these messages are:

Abbreviation	Description
BOOL	Boolean
CHAR	Character
CHARS	Characters
CONS	Constant
DECL	Declaration
DUP	Duplicate
ERR	Error
EXPR	Expression
FUNC	Function
HEX	Hexadecimal
ID	Identifier
IFXX	Conditional compilation computation word
ILL	Illegal
PARAM	Parameter
PARENS	Parenthesis
PROC	Procedure
PROG	Program
REF	Reference
REFS	References
REPL	Replacement
SEMI	Semicolon
SPECS	Specifications
STMT	Statement
STRG	String
UNDECL	Undeclared
XDEF	External definition
XREF	External reference
/	Or

TABLE B-1. COMPILER ERROR MESSAGES

Message Number	Condition Causing Message
001	LONG ID–FIRST 12 CHARS USED
002	DUP DECL–NEW ONE OVERRIDES
003	UNDECL ID DELETED
004	ILL OCTAL/HEX CONS

TABLE B-1. COMPILER ERROR MESSAGES (cont.)

Message Number	Condition Causing Message
005	TERM MISSING
006	BAD STATUS CONS USE
007	BAD NESTING OF PARENS/ BRACKETS
008	CRUD CHAR IN INPUT
009	CHAR STRG>240 BYTES–240 USED
010	ILL ARRAY ITEM ID USE DELETED
011	ILL SWITCH ID USE DELETED
012	ILL ARRAY ID USE DELETED
013	ILL STATUS LIST ID USE DELETED
014	ILL COMMON ID USE DELETED
015	SEMI MISSING AFTER ARRAY DECL
016	CRUD AT START OF STMT DELETED
017	ILL KEYWORD USE DELETED
018	ARRAY ITEM DECL LIST LACKS END
019	DUP DECL OVERRIDES
020	ITEM DECL ID ERR
021	DECL DISCARDED–SCAN RESUMES AT SEMI
022	ITEM DECL TYPE ERR–I ASSUMED
023	ILL ITEM LENGTH–1 BYTE USED
024	SIGNED PRESET ILL FOR THIS TYPE–IGNORED
025	SCAN RESUMES AT -BEGIN-
026	MISSING SEMI
027	ITEM PRESET ERR
028	SEMI ACCEPTED AS NULL STMT
029	BASED/XDEF/XREF ARRAYS NEED ID
030	ARRAY ITEM DECL SYNTAX ERR
031	ARRAY ITEM DECL TYPE ERR
032	BAD ARRAY BOUND VALUES– ASSUMED [0:0]
033	ARRAY BOUND SYNTAX ERR
034	ARRAY ITEM DECL PARTWORD SPECS ERR–DEFAULT TAKEN
035	ARRAY ITEM DECL 1ST BIT ALIGNMENT WRONG–0 USED
036	ILL ARRAY ITEM BOUNDARY– DEFAULT TAKEN
037	TOO MANY ARRAY ENTRIES
038	TOO MANY PRESET GROUPS
039	ARRAY PRESET SYNTAX ERR
040	COMMON/XDEF/XREF–AT OUTER SCOPE ONLY
041	BAD COMMON DECL IGNORED
042	BAD XREF/XDEF IGNORED
043	BAD BASED DECL IGNORED
044	XDEF/XREF LIST CRUD DELETED
045	SWITCH DECL SYNTAX ERR
046	COMMON LIST SCAN RESUMES AT -ARRAY-/-ITEM-
047	STATUS DECL SYNTAX ERR
048	-END- ENDS BAD COMMON LIST
049	DEF DECL SYNTAX ERR
050	BAD FORMAL PARAM DECL
051	PROGRAM BEGINS BADLY
052	PROG DECL LACKS ID
053	PROG DECL ERR–CRUD PRECEDES SEMI
054	XDEF/XREF LIST SCAN RESUMES AT LEGAL ENTRY
055	FORMAL LABEL DECL SYNTAX ERR
056	-END- ENDS BAD XDEF/XREF LIST
057	FORMAL PROC DECL SYNTAX ERR
058	FUNC DECL LASKS ID
059	FUNC DECL TYPE ERR– I ASSUMED
060	FUNC DECL LACKS SEMI
061	SCAN RESUMES AT SEMI
062	DUP FORMAL PARAM ID IN LIST
063	DUP PARAM ID–PRIOR DECL THIS SCOPE
064	PARAM LIST SYNTAX ERR
065	PROC DECL LACKS ID
066	PROC DECL SYNTAX ERR
067	UNDECL LABEL/PROC ID
068	FORMAL ID LACKS DECL
069	PARAM NOT USED IN THIS SCOPE
070	ILL DEF ID–NO EXPANSION
071	ENTRY PROC MAY NOT CALL ITSELF
073	TOO MANY PARAM/ARRAY/ARRAY ITEM REFS
074	TOO MANY SUBSCRIPTS:SWITCH REF
075	NOT ENOUGH SUBSCRIPTS FOR ARRAY/ARRAY ITEM REFS
076	BAD SUBSCRIPT LIST
077	ILL LABEL/PROC ID USE DELETED
078	STATUS SWITCH DECL LACKS STATUS LIST ID
079	BAD LABEL USE IN STATUS SWITCH

TABLE B-1. COMPILER ERROR MESSAGES (cont.)

Message Number	Condition Causing Message
080	STATUS SWITCH–VALUE TOO LARGE
081	STATUS SWITCH–DUP CONSTANT VALUES
082	STATUS SWITCH–MISSING CONSTANT
083	BEGIN/END MISMATCH. PROBABLE DISASTER
084	IF EXPR NOT BOOL
085	WHILE EXPR NOT BOOL
086	CRUD AFTER FINAL END IGNORED
087	-DEF- ID EXPANSION NEST TOO DEEP-ID DELETED
088	YOUR -DO- HAS BEEN FOUND
089	THE -THEN- HAS BEEN FOUND
090	MISSING -DO-
091	MISSING -THEN-
092	INITIAL VALUE EXPR ERR
093	-STEP- EXPR ERR
094	-UNTIL- EXPR ERR
095	-WHILE- EXPR ERR
096	BAD -GOTO- DELETED
097	BAD REPL STMT DELETED
098	PARTWORD VALUES AFTER FIRST 3 IGNORED
099	ITEM DISCARDED–SCAN RESUMES AT COMMA
100	HANGING -IF- CLAUSE
101	HANGING -FOR- CLAUSE
102	HANGING -ELSE-
103	EXTRA END–OMITTED BEGIN FOR SUBPROGRAM ASSUMED
104	ILL UNDECL PARAM USE DELETED
105	FOR STMT: INDUCTION ID ERR
106	-IF- EXPR ERR
107	DUP XDEF/XREF DECLS FOR ID
108	XDEF PROC/FUNC: NOT FULLY DECL
109	BAD FORMAL DECL
110	REDUNDANT FORMAL DECL
111	BAD PARAM LIST
112	BOOL ILL IN ARITH CONTEXT
113	COMMON LIST LACKS -END-
114	BASED LIST LACKS -END-
115	XDEF/XREF LIST LACKS -END-
116	COMMON LIST CRUD DELETED
117	BASED LIST CRUD DELETED
118	BASED LIST SCAN RESUMES WITH -ARRAY-
119	-END- ENDS BAD BASED LIST
120	0 LENGTH -DEF- STRING IGNORED
121	CHAR LENGTH OMITTED–1 ASSUMED
122	BAD ARRAY ENTRY SIZE
123	BRACKET NEST TOO DEEP
124	ILL EXPR TYPE THIS LEFT SIDE
125	BAD READ FUNC
126	EXPR OP CONCATENATION ERR
127	LONG CHAR STRG–240 BYTES USED
128	BAD -LOC- FUNC
129	BAD -ABS- FUNC
130	BAD INDUCTION ID TYPE
131	NON INDUCTION ID IN -TEST-
132	-TEST- ILL OUTSIDE LOOP
133	SCAN RESUMES AT -BEGIN-/ -ITEM-/SEMI
134	READ FUNC NEEDS ID
135	DUP STATUS ID
136	SEMI ENDS COMMENT
137	CONTROL STMT SYNTAX ERR
138	CHAR NOT D/F IN REAL OR COUBLE CONSTANT
139	FORMAL PARAM PRESET ILL
140	XREF PRESET ILL
141	BLANK COMMON PRESET ILL
142	BASED ARRAY ITEM PRESET ILL
143	BAD P-FUNC
144	CHARACTER ITEM>240 BYTES - 240 USED
145	NO SUBSCRIPT FOR ARRAY ITEM - 0 USED
146	CIRCULAR DEF NAME EXPANSION - EXPANSION IGNORED
147	NO MAIN PROC FOR ENTRY PROC
148	ILLEGAL CHAR IN MACRO DEF
149	ILLEGAL IFXX COMPARE
150	TOO MANY DEF PARAMS
151	ILLEGAL CONDIT DIRECTIVE IGNORED
152	ILLEGAL VALUE PARAM-LABEL
153	ILLEGAL VALUE PARAM-ARRAY
154	ILLEGAL VALUE PARAM-PROC
155	COMMON BASED ARRAY DECL ERROR

TABLE B-1. COMPILER ERROR MESSAGES (cont.)

Message Number	Condition Causing Message
156	LABEL DECL ERROR
157	XREF SWITCH ERROR
158	UNMATCHED IFXX
159	DEF PARAM ERROR
160	([OR < NESTING TOO DEEP
161	([OR < NEST MISMATCH
162	PARAMETER TOO LONG
163	PARAMETER COUNT ERROR
164	RECOVERY AT ;
165	BAD DEF ACTUAL PARAMETER
166	BAD UNDCL PROC/LABEL LIST
167	ILL DEF PARAM USAGE
168	SORRY BUT IFXX MUST HAVE 2 PARAMS--FOR THE TIME BEING
169	ATTRIBUTE SPECIFIED TO UNKNOWN VARIABLE
170	SIMPLE ITEMS MAY NOT BE INERT/REACTIVE

Message Number	Condition Causing Message
171	ONLY ITEMS AND ARRAYS HAVE ATTRIBUTES
172	BAD ATTRIBUTE/LEVEL SPECIFICATION LIST
173	FAST FOR LOOP INDUCTION VARIABLE ERROR
174	BAD GLOBAL ATTRIBUTE SPEC
175	LEVEL ONLY APPLIES TO COMMON AND BASED ARRAYS
176	BAD USE OF LEVEL 3 VARIABLE
177	INDUCTION VARIABLES MUST BE SCM RESIDENT
178	WEAK ONLY APPLIES TO EXTERNAL SYMBOLS
179	ARRAY ENTRY-SIZE TOO LARGE
180	ARRAY DIMENSION TOO LARGE
181	RECURSIVE PROC/FUNC CALL NOT ALLOWED
182	ERROR IN REAL CONSTANT

PROGRAMMING SUGGESTIONS C

COMPILER

Space required for compilation is proportional to the number of symbols in the source program. Approximately five words of core are dedicated to each name in the program, in the form of a symbol table entry.

Time required for compilation is proportional to the size of the object program, in terms of the amount of syntax to be scanned. Although data declarations do not generate code, they use significant amounts of compiler time and field length, especially data presets.

Compilation time can be further reduced by judicious use of the compiler options such as suppression of object code and cross reference listings.

DEF declarations can increase readability of SYMPL source programs and facilitate changes to them. However, DEF declarations and expansions increase compilation time and field length, accordingly.

OBJECT CODE

SUBSCRIPTS

Code produced by referencing subscripted variables can be affected by the means of expressing the subscript. For example, an integer constant can be partially evaluated at compile time so that one instruction is required to access an array item (given the item is a full word); but a scalar integer variable requires four instructions to access the item. Thus, a reference to A[3] requires one instruction; but A[I], where I=3, requires four instructions to retrieve the same item.

ARRAYS

Parallel arrays are accessed more efficiently than serial arrays when an array entry exceeds one word. For arrays with one-word entries, no difference in object code speed or space is apparent. Parallel arrays, rather than serial arrays, should be used when possible. Fixed arrays are accessed more efficiently than based arrays, which require a level of indirectness to access an entry. Whenever possible, fixed arrays should be used.

COST OF ACCESSING DATA TYPES

If an array item is a full 60-bit word, access does not depend upon its type. For items which are not 60-bit words, however, type and bit position assignment affect the code required to access them, as follows:

Signed integers are accessed more efficiently than unsigned integers. If the item is 18 bits long, the SXi instruction is used to access signed integers. Signed integer items are accessed more efficiently if they are the leftmost bits of a word. Unsigned integer items are accessed more efficiently if they occupy the rightmost bits of a word rather than the middle or leftmost bits. Boolean items are most efficiently accessed by allocating the whole word or the leftmost bit of a word rather than one bit elsewhere. Otherwise, they are accessed as unsigned integers are accessed.

FOR LOOPS

The break-even point in code generated for in-line and FOR loop code is 3-4 iterations. Of the following sequences, the second generates fewer instructions and runs faster.

```
FOR I=0 STEP 1 UNTIL 2 DO
  PWORD[I] = 0;
```

```
PWORD[0] = 0;
PWORD[1] = 0;
PWORD[2] = 0;
```

If four or more items were being set by the above sequence, the loop would have required less code but required more time.

In general, the less source code in the FOR statement, the faster it will run. Of the following code sequences, the second is faster since the loop limit is computed and the value stored only once.

```
FOR I = 0 STEP 1 UNTIL B/C DO
    PWORD[I] = K**J;
```

```
A = B/C;
D = K**J;
FOR I = 0 STEP 1 UNTIL A DO
    PWORD[I] = D;
```

One execption is that FOR loop execution time can be reduced with more source code as in the following example where the second sequence would be faster even though more code would be generated.

```
FOR I = 0 STEP 1 UNTIL 89 DO
    PWORD[I] = 0;
```

```
FOR I = 0 STEP 3 UNTIL 89 DO
    BEGIN
      PWORD[I] = 0;
      PWORD[I+1] = 0;
      PWORD[I+2] = 0;
    END
```

DATA CONVERSION

Integer-to-character conversion is byte-oriented; the character-to-integer conversion is word-oriented. When an integer item is converted to character mode, the rightmost 6-bit byte is left-justified and blank filled in the character field; yet, character-to-integer conversion is performed by right-justifying the right end of the last word of the character item and zero filling it on the left. Character field definitions can cross word boundaries. Arithmetic operations with character data, including masking, makes the code machine dependent because it reduces the string to one word.

The conversions can be circumvented by the use of bit bead functions. For example, B<0,60>FLTINGPT =INTEGER; would cause the integer to be stored in the floating point item without conversion. B<0,60> CHARACTER=INTEGER also would cause the full word to be stored in CHARACTER, not just the low-order six bits.

PROC SUBPROGRAMS

Formal parameters should be called by value whenever possible. If a procedure must reference its formal call by address parameter more than once, a local variable should be declared, set to the value of the formal parameter, and subsequently referenced instead of the formal parameter. Actual call-by-name parameters are referenced indirectly in the generated code; this level of indirectness can be overcome by evaluating the parameter once and making it local to the procedure by storing the parameter's value in a local variable.

FUNC SUBPROGRAMS

The statements under the heading PROC subprograms are true for FUNC subprograms also. When the subprogram must return a result, a function should be used rather than a procedure that returns a value. Use of the function saves two instructions. For example: a routine is needed to convert from integer to display code, with the result to be stored in one of three arrays, depending upon the section of code where the call originates. If a function is used (as in ARRAYWORD[I] = FUNCTION[INT] rather than a procedure (as in PROCED (INT); ARRAYWORD[I] = INTT), two SAi k instructions are saved per call. The saving is realized since functions return their result in register X6 rather than in a memory location.

CODING HINTS

Based array references are candidates for scratch variable storage if referenced more than once in a sequence of source code, since based array references are indirect.

When storing into many items of the same data structure (array) clustered together, those that refer to the same word of storage should be described in the same order in which they occur.

POSSIBLE OPTIMIZATIONS

The SYMPL language permits the compiler to move code to achieve optimization. SYMPL 1.2 and later versions, at the present time, do not perform global flow analysis. They do, however, perform many local optimizations including: compile-time computation of constant expressions, conversion of many multiplies to

shift-and-add, and elimination of many redundant loads and stores. Therefore, if the program has any OVERLAP or REACTIVE variables, they should be declared to assure correct compilation on SYMPL 1.2 and later versions of the compiler.

In SYMPL 1.2 and later versions, if no CONTROL statements with INERT, REACTIVE, DISJOINT, or OVERLAP appear, the program is called unbehaved and is considered to adhere to SYMPL 1.1 rules, which are:

Formal parameters can destroy global variables and vice versa.

A based array can destroy all other based and fixed arrays, but a fixed array does not destroy any other arrays.

All arrays are considered reactive.

An external call can destroy all COMMON, XDEF and XREF variables.

Formal parameters can destroy each other.

There are no other interferences between variables.

These definitions are retained in SYMPL 1.2 and later versions to accommodate existing programs until correct behavior statements are inserted.

OPTIMIZATIONS POSSIBLE UNDER GLOBAL OPTIMIZATION

The compiler is permitted all the optimizations listed below.

Constant Subsumation: If a constant is assigned to a variable, replace the variable with the constant up until a point where its value may be destroyed.

Common-Expression removal: If the same expression occurs twice and none of the variables are destroyed in between, save the result of the first computation, eliminate the code for the second computation, and reference the saved value.

Removal of identities: Remove statements such as I=I; and through constant subsumation and the mechanisms of common-expression removal, the optimizer might determine that a statement is in fact an identify though this is not apparent in the source.

Code removal from loops: Recognize program flow which is a loop, whether it is a formal FOR-loop or not, and optimize any loop which is not spoiled by a branch entering from outside. Code which is invariant during the loop is moved in front of the loop.

Strength reduction: In a fastloop, certain multiply operations on the induction variable are converted to additions to a temporary variable, and certain exponentiations are similarly converted to multiplications.

This SYMPL language definition permits analysis of program flow to discover loops (including nested loops) and to determine which expressions are invalidated by forward branches. It may also analyze all procedures and functions within a module to determine which variables they use and which ones they destroy. This enables the optimizer to optimize over many function or procedure calls. Since it is possible for code to be removed over long distances in the program, the programmer must inspect the entire module to determine OVERLAP or REACTIVE behavior.

The compiler never moves code from one procedure to another. Suppose PROC Q stores B(I) and PROC P references A(J) and B(I) is based on A(J). If P calls Q, there is danger of the A(J) reference being moved past the call to Q; this is overlapped behavior and the CONTROL OVERLAP statement is required to prevent such optimization. But if the program is restructured so that P and Q are parallel (neither one calls the other), then this is not overlapped behavior. For example:

```
PROC MAIN;
    BEGIN
    ARRAY A[10]; ITEM AA(0);
    BASED ARRAY B[10]; ITEM BB(0);
      :
      PROC INIT;
      BEGIN
      AA(1) = 31;
      END #INIT#
      :
    P<B> = LOC (A);

L1:
    INIT;
    X = BB[I];
      :
    IF BOOLE THEN GOTO L1;
    END #MAIN#
```

Here the compiler might remove BB[I] from the loop, causing an error that might be difficult to locate. The statement

```
CONTROL OVERLAP A,B;
```

solves this problem. However, if the code between the INIT call and the IF BOOLE statement is converted to a procedure, the problem will not arise and no CONTROL statement is required.

Such a problem occurs frequently in programs having a separate initialization section: the program can remain well-behaved if both the initialization and the body are made into separate procedures.

Another common problem is the local based array whose pointer is manipulated by an external procedure. (The Common Memory Manager is a case in point.) Such based arrays must be declared overlapped. For example:

```
XREF PROC GETSPC;
BASED ARRAY X[100]; ITEM XX (0);
   :
GETSPC(X, 100);
Q=P<X>;
GETSPC(Y, 50);
R=P<X>;
```

Suppose the routine GETSPC is external and manages dynamic storage, and suppose that at the GETSPC(Y, 50) call, it moves block X. Now if the optimizer removes the expression P<X> and sets R to the old P<X> from the statement Q=P<X>, the result will be wrong.

The compiler can assume that GETSPC(Y) does not destroy X because X is a local, and theoretically GETSPC cannot get at X unless X is a parameter. This assumption is not of course fully correct; however, we define the language to consider this to be overlapped behavior and require the statement:

```
CONTROL OVERLAP X;
```

TREATMENT OF EXTERNALS AND COMMON

All badly-behaved and all external variables (XDEF, XREF, and COMMON) are considered destroyed by an external call. Any global flow analysis analyzes all possible flow of control resulting from an XDEF label, and considers that all variables are destroyed by entry at such a label.

condition word := { ifeq | ifne | ifls | iflq | ifgq | ifgr }

control word := { eject | list | nolist | objlst | pack | preset | fi | traceback | ftncall | fastloop | slowloop }

attribute := { level ∧ lev list | inert ∧ var list | reactive ∧ var list | disjoint ∧ var list | overlap ∧ var list | weak ∧ weak list }

lev list := { lev descr | lev list ∨, ∨ lev descr }

lev descr := { common name | based array name | ϕ }

var list := { var descr | var list ∨, ∨ var descr }

var descr := { array name | based array name | item name | ϕ }

weak list := { weak descr | weak list ∨ , ∨ weak descr }

weak descr := { array name | based array name | function name | item name | label name | proc name | switch name }

ifeq := mark ⌋ IFEQ ⌊ mark

ifne := mark ⌋ IFNE ⌊ mark

ifls := mark ⌋ IFLS ⌊ mark

iflq := mark ⌋ IFLQ ⌊ mark

ifgq := mark ⌋ IFGQ ⌊ mark

ifgr := mark ⌋ IFGR ⌊ mark

eject := mark ⌋ EJECT ⌊ mark

list := mark ⌋ LIST ⌊ mark

nolist := mark ⌋ NOLIST ⌊ mark

objlst := mark ⌋ OBJLST ⌊ mark

pack := mark ⌋ PACK ⌊ mark

preset := mark ⌋ PRESET ⌊ mark

fi := { mark ⌋ FI ⌊ mark | mark ⌋ ENDIF ⌊ mark }

traceback := mark ⌋ TRACEBACK ⌊ mark

ftncall := mark ⌋ FTNCALL ⌊ mark

fastloop := mark ⌋ FASTLOOP ⌊ mark

slowloop := mark ⌋ SLOWLOOP ⌊ mark

level := mark ⌋ LEVEL { 1 | 2 | 3 } ⌊ mark

inert := mark ⌋ INERT ⌊ mark

reactive : = ⌋ mark REACTIVE ⌊ mark

disjoint : = ⌋ mark DISJOINT ⌊ mark

overlap : = ⌋ mark OVERLAP ⌊ mark

weak : = ⌋ mark WEAK ⌊ mark

declaration :=
- array dec
- based dec
- common dec
- def dec
- entry dec
- func dec
- item dec
- label dec
- proc dec
- status dec
- switch dec
- xdef dec
- xref dec
- formal array dec
- formal based dec
- formal func dec
- formal item dec
- formal label dec
- formal proc dec

statement :=
- compound statement
- exchange statement
- for statement
- goto statement
- if statement
- labeled statement
- proc call statement
- replacement statement
- return statement
- stop statement
- test statement

GLOSSARY F

ARITHMETIC EXPRESSION – An expression that yields a numeric value.

BASED ARRAY – A structure that can be superimposed over any area of memory during program execution. No storage is allocated for a based array during compilation; rather the compiler creates a specific pointer variable compiled with an undefined value. Based arrays are used when the position of an array is not known at load time.

BEAD FUNCTION – A function that accesses consecutive bits or characters of an item.

BOOLEAN EXPRESSION – An expression that yields a Boolean value of TRUE or FALSE.

DELIMITER – A character that is used to separate and organize data items or statements. SYMPL-characters classified as marks serve as delimiting characters.

ENTRY POINT – A location within a procedure or function that can be referenced from a calling program. Each entry point has a name with which it is associated.

EXCHANGE STATEMENT – A statement that causes the exchange of values of the left-hand and right-hand sides of the statement.

EXPRESSION – A sequence of identifiers, constants, or function calls separated by operators and parentheses. The evaluation of an expression yields a value.

EXTERNAL REFERENCE – A reference in one module to an entry point in another module. Throughout the loading process, the loader matches externals to the correct entry points. External references are specified by the XREF statement.

EXTERNAL SUBPROGRAM – A subprogram that is compiled as a separate module.

FASTLOOP – A type of FOR statement where the test and branch is at the end of the loop. Fastloops always execute at least once. Contrast with slowloop.

FUNCTION – A subprogram used within an expression. It returns a value through its name. The text of a function must contain an assignment statement that assigns a value to the function name. A function can also return values through its parameters. Contrast with procedure and main program.

IDENTIFIER – A string of 1 through 12 letters, digits, or $ beginning with a letter ($ is considered to be a letter). This manual uses the term identifier to indicate a programmer-defined entity. Contrast with reserved words.

INDUCTION VARIABLE – A scalar that is used as the counter for the loop in a FOR statement.

LOGICAL OPERATOR – An operator that works with Boolean values and yields a Boolean result. Contrast with masking operator, numeric operator, and relational operator.

MAIN PROGRAM – A module that consists of a main program header followed by a series of declarations and one statement (usually compound) and ended by a TERM statement. Contrast with function, procedure, and subprogram.

MASKING OPERATOR – An operator that performs bit-by-bit operations that yield numeric results. Contrast with logical operator, numeric operator, and relational operator.

MODULE – A separately compiled main program or subprogram. Compilation of a module is terminated whenever a TERM statement is encountered.

NUMERIC OPERATOR – An operator that performs arithmetic operations to yield numeric results. Contrast with logical operator, masking operator, and relational operator.

PARALLEL ALLOCATION – The first words of each array entry are allocated contiguously, followed by the second words of each entry, and so forth. Contrast with serial allocation.

P-FUNCTION – A function that references the pointer variable for a based array.

POINTER VARIABLE – The variable created by the compiler for a based array. The pointer variable is set by the P-function.

PROCEDURE – A subprogram that can, but need not, return values through any of its parameters. It is called when its name or one of its alternative entry points is referenced. Contrast with function and main program.

RELATIONAL OPERATOR – An operator that works with arithmetic or character operands to produce a Boolean result. Contrast with logical operator, masking operator, and numeric operator.

REPLACEMENT STATEMENT – A statement that assigns a value to a scalar, subscripted array item, P-function, bead function, or function name.

RESERVED WORDS – Identifiers that have predefined meaning to the SYMPL compiler.

SCALAR – An item that is not in an array. An ITEM declaration outside an array defines a scalar.

SCOPE OR VARIABLE – The set of statements in which the declaration of the variable is valid.

SERIAL ALLOCATION – All the words of one array entry are allocated contiguously. Contrast with parallel allocation.

SLOWLOOP – A type of statement where the test and branch is at the beginning of the loop. Slowloops need not execute at all. Contrast with fastloop.

SUBPROGRAM – A function or procedure. Subprograms can be compiled as separate modules. Contrast with main program.

TYPE – The representation of data. Data can be type integer, unsigned integer, real, character, Boolean, or status.

WEAK EXTERNAL – An external reference that is ignored by the loader during library searching and cannot cause any other program to be loaded. A weak external is linked, however, if the corresponding entry point is loaded for any other reason.

XDEF DECLARATION – A declaration that generates an entry point that can be used by the loader. It is used in the declaring program to define an identifier as external. Storage is allocated for the identifier. Contrast with XREF declaration.

XREF DECLARATION – A declaration that generates an external reference to the specified identifier. It is used in the referencing program. Use of XREF implies that the identifier has been declared to be external in another program. No storage is allocated for the identifier. Contrast with XDEF declaration.

INDEX

COMMENT SHEET

CONTROL DATA CORPORATION

TITLE: SYMPL Version 1 Reference Manual

PUBLICATION NO. 60496400 REVISION D

This form is not intended to be used as an order blank. Control Data Corporation solicits your comments about this manual with a view to improving its usefulness in later editions.

Applications for which you use this manual.

Do you find it adequate for your purpose?

What improvements to this manual do you recommend to better serve your purpose?

Note specific errors discovered (please include page number reference).

CUT ON THIS LINE

General comments:

FROM NAME: _____ POSITION: _____

COMPANY NAME: _____

ADDRESS: _____

NO POSTAGE STAMP NECESSARY IF MAILED IN U.S.A.

FOLD ON DOTTED LINES AND STAPLE

STAPLE

STAPLE

FOLD

FOLD

FIRST CLASS
PERMIT NO. 8241

MINNEAPOLIS, MINN.

BUSINESS REPLY MAIL
NO POSTAGE STAMP NECESSARY IF MAILED IN U.S.A.

POSTAGE WILL BE PAID BY

CONTROL DATA CORPORATION
Publications and Graphics Division
215 Moffett Park Drive
Sunnyvale, California 94086

CUT ON THIS LINE

FOLD

FOLD

STAPLE

STAPLE