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 addition for weak
(03-31-78)	externals; and points not tested SYMPL control statement option. Appendix F contains a glossary.
F	This revision documents SYMPL 1.4. New features include SYMPL text, CID interface, a new
(07-20-79)	diagnostic system, and a description of system-independent arrays. Appendix G contains coding
	conventions. The manual has been partially reorganized.
F	This revision documents SYMPL 1.4. It incorporates various technical corrections and
(01-31-80)	editorial improvements. This revision supersedes all previous revisions.
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	F
iii/iv	F
v/vi	F
vii	F
viii	F
1-1 thru 1-5	F
1-6	E
1-7	F
1-8	E
2-1 thru 2-6	F
2-7	E
2-8	F
2-9	F
2-10 thru 2-12	E
3-1 thru 3-6	F
4-1	E
4-2	F
4-3	E
4-4 thru 4-8	F
5-1	F
5-2	F
5-3	E
5-4	F
5-5	E
5-6 thru 5-9	F
6-1 thru 6-3	E
6-4 thru 6-6	F
A-1	E
A-2	E
B-1 thru B-17	F
C-1	E
C-2	E
D-1 thru D-25	E
E-1	E
E-2	E

Page	Revision
F-1	E
F-2	E
G-1 thru G-4	E
H-1	E
H-2	E
Index-1	F
Index-2	F
Index-3	E
Comment Sheet	F
Mailer	–
Back Cover	–

Page	Revision

PREFACE

SYMPL Version 1.4, which is a systems programming language, operates under control of the following operating systems:

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

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

SCOPE 2 for the CONTROL DATA® CYBER 170 Model 176, CYBER 70 Model 76, and 7600 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 are listed below.

Publication	Publication Number
CYBER Interactive Debug Version 1 Reference Manual	60481400
NOS Version 1 Operating System Reference Manual, Volume 1 of 2	60435400
NOS Version 1 Operating System Reference Manual, Volume 2 of 2	60445300
NOS/BE Version 1 Operating System Reference Manual	60493800
SCOPE Version 2 Reference Manual	60342600

CDC manuals can be ordered from Control Data Corporation, Literature and Distribution Services, 308 North Dale Street, St. Paul, Minnesota 55103.

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

INDEX

FIGURES

TABLES

The SYMPL compiler is designed for use by system programmers writing compilers and system software. Consequently, SYMPL omits many facilities typically found in high-level languages, particularly features intended for scientific and commercial applications programming. At the same time, SYMPL places minimal restrictions on allowable optimizations.

SYMPL is a procedure-oriented language similar to JOVIAL, which was derived from ALGOL-58 (the 1958 version of the International Algorithmic Language, as described in the December 1958 issue of the Communications of the ACM).

SYMPL is a readable and concise programming language that uses self-explanatory English words and the familiar notations of algebra and logic. In addition, SYMPL has no format restrictions: comments can be intermixed among the symbols of a program to document it.

Coding conventions for SYMPL are less restrictive than for most languages. The source program is considered to be a stream of characters; card or line boundaries are ignored. Significant columns of a card image are 1 through 72; columns beyond 72 are not interpreted. For purposes of source information, the compiler assumes column 72 of one card image is adjacent to column 1 of the next card. All SYMPL names, constants, operators, or symbols can be broken across card images without any special continuation marks.

SYMPL CHARACTER SET

The SYMPL character set has 55 characters:

A through Z and $

0 through 9

* / + - = < > ( ) [ ] " # . , : ; blank

Some of the characters are not represented on the keys of all keypunches or terminals and they might not appear on all line printers. The characters shown above are from the American Standard Character Set for Information Interchange (ASCII) and are used throughout the body of this manual. For instance, the characters " and # appear in the ASCII character set, but appendix A shows that ≠ and ≡ might appear with another character set. A programmer should use whatever is appropriate to achieve a display code value of 64 to represent the constant delimiter " and a display code value of 60 to represent the comment delimiter #.

Characters that are not part of the SYMPL character set can be used in a program only within a character constant or a comment.

SYMPL characters classified as marks serve as delimiting characters. In the correct circumstances, any of the marks can delimit an identifier. The character blank (also known as a space) is an element within the set of marks. Consequently, its use is significant.

Whenever one blank is required as a delimiter, any number of blanks is allowed. Whenever a nonblank delimiter is required, any number of surrounding blanks is allowed.

Table 1-1 shows the SYMPL marks and their uses.

TABLE 1-1. SYMPL MARKS

Mark	Use
+	Arithmetic operator for addition.
-	Arithmetic operator for subtraction.
*	Arithmetic operator for multiplication.
/	Arithmetic operator for division.
**	Arithmetic operator for exponentiation.
=	Assignment operator for replacing value of left side with value of right side.
==	Assignment operator for exchange of values on right and left sides.
,	Separator, for expressions, list elements, etc.
.	Decimal point in real constant.
:	Delimit labels; separate bounds of array dimension; separate status indicator from value.
;	Terminate statements and declarations.
blank	Delimit identifiers; introduce readability.
(and)	Delimit argument lists; group expression elements; denote call-by-value argument; etc.
[and]	Delimit subscripts and array bounds.
< and >	Delimit arguments for intrinsic functions B, C, and P, and system-independent array specifications.
"	Delimit character constants, status constant values, octal constant values, and hexadecimal constant values.
#	Delimit both limits of comment and DEF body.

COMMENTS

A comment is an arbitrary string of characters enclosed within pound signs; as in:

#string#

The only characters that cannot appear within a comment are the semicolon and the pound sign. A semicolon terminates the comment and causes a diagnostic.

Comments can appear between or within SYMPL statements. Anyplace a blank can appear, a comment can appear, with the following exceptions. A comment cannot appear:

- Within a comment
- After the name in a DEF declaration since the body of DEF is also delimited by #.

IDENTIFIERS

An identifier is a string of 1 through 12 letters, digits, or $ beginning with a letter ($ is considered to be a letter). The two types of identifiers are reserved words and programmer-defined words.

Reserved words have predefined meanings to the SYMPL compiler. They can be used only in the contexts described in this manual. Table 1-2 lists all of the reserved words and the roles they play in SYMPL programs; context-defined descriptors also appear in this table although they are not reserved words.

Programmer-defined identifiers name entities, such as constants or variables, within the program. They cannot be the same as a reserved word.

The remainder of this manual uses the term identifier to indicate a programmer-defined entity. Reserved words are indicated in text in capital letters.

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. Numerically, FALSE is zero and TRUE is nonzero.

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 and result of integer multiplication and division, must 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 type of integer constant 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 plus or minus sign. The string can contain 1 through 18 digits; it cannot contain blanks. The absolute value for a decimal integer must 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 expressed in 59 bits. If 60 significant bits are written, the leftmost bit is used as a sign in one'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 one'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.

TABLE 1-2. SYMPL RESERVED WORDS AND DESCRIPTORS

Word or Descriptor	Used To
A†	Denote aligned allocation for system-independent array.
ABS	Intrinsic function that returns an absolute value.
AND	Boolean operator.
ARRAY	Declare dimensioned entity.
B†	Denote Boolean data type. Also, intrinsic function that accesses bits of a data item.
BASED	Declare an array that has a structure but no storage.
BEGIN	With END, delimit a compound statement.
C†	Denote character data type. Also, intrinsic function that accesses characters of a data item.
COMMON	Declare data to reside in loader common blocks.
CONTROL	Introduce a compiler directive
DEF	Declare a macro.
DO	Introduce clause of FOR statement.
ELSE	Mark statement to be executed on the FALSE evaluation of the Boolean expression in an IF statement.
END	With BEGIN, delimit a compound statement.
ENTRY	Declare alternate entry point for procedure or function.
EQ	Relational operator.
FALSE	Boolean constant having the integer value 0.
FOR	Introduce FOR statement.
FPRC	Declare formal procedure name used as a parameter or declare a formal function name that is used as a parameter.
FUNC	Introduce a function subprogram or declare a formal function name that is used as a parameter.
GOTO	Introduce unconditional branch in program flow.
GQ	Relational operator.
GR	Relational operator.
I†	Declare signed integer data type.
IF	Introduce conditional IF statement.
ITEM	Declare variable data item.
LABEL	Declare label.
LAN	Arithmetic operator.
LIM	Arithmetic operator.
LNO	Unary arithmetic operator.
LOC	Intrinsic function that returns address.
LOR	Arithmetic operator.
LQ	Relational operator.
LQV	Arithmetic operator.
LS	Relational operator.
LXR	Arithmetic operator.
NOT	Boolean operator.
NQ	Relational operator.
O†	Prefix octal constant.
OR	Boolean operator.
P†	Denote parallel array storage. Also, intrinsic function that allows reference to based array pointer.
PRGM	Introduce program, rather than a subprogram, module.
PROC	Introduce a procedure subprogram.
R†	Denote real data type.
RETURN	Exit from subprogram or program.
S†	Denote status data type. Also, denote serial array storage.
STATUS	Introduce status declaration.
STEP	Introduce clause of FOR statement.
STOP	Return control to operating system.
SWITCH	Declare a vector of labels.
TERM	Terminate module compilation.
TEST	Change flow of FOR statement execution.
THEN	Introduce clause of IF statement.

TABLE 1-2. SYMPL RESERVED WORDS AND DESCRIPTORS (Contd)

Word or Descriptor	Used To	Word or Descriptor	Used To
TRUE	Boolean constant having the value 1.	XDEF	Generate loader entry point.
U†	Denote unsigned integer data type or unaligned allocation for system-independent array.	XREF	Generate loader external reference.
UNTIL	Introduce clause of FOR statement.	$BEGIN	With $END, delimit statements to be compiled at compiler call parameter option.
WHILE	Introduce clause of FOR statement.	$END	With $BEGIN, delimit statements to be compiled at compiler call parameter option.
X†	Prefix hexadecimal constant.		

†Context descriptor that is not a reserved word.

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 precision. A diagnostic message is given when a number falls outside of the hardware limits.

STATUS FUNCTIONS AND CONSTANTS

Status functions and status constants represent small integer values the compiler has associated with the identifiers in a status list. They can be used anywhere integer constants can be used.

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:

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

Use of single-character status list names that are the same as the context descriptors O, S, and X can cause conflicts. For example:

STATUS X A,B,C;

The use of X"A" in the program is interpreted as the hexadecimal representation of the decimal value 10, and not the status item.

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-by-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 words) and their meanings for the different types of operators. Tables 1-4 and 1-5 show truth tables for the logical and masking operators.

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	F	F	T	T
b2	F	T	F	T
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
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

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

Part-word array items appearing in an expression are lengthened to 60 bits. They are right-justified and zero-filled, except for character items, which are left-justified and blank-filled, and signed integer items, which are right-justified and sign-extended.

Furthermore, 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 from 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 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. An exception is exponentiation, in which neither operand is converted. The three operand types that exist for conversion purposes are as follows, listed in order from highest to lowest:

Real

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 character to 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 or more character operand is set, the converted integer is a negative value, and 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 integer to real occurs by floating the integer, as provided by hardware instructions. The resulting real value is expressed in single precision format.

Additional rules for specific operators are:

- Division by zero is undefined.
- Division of an integer by an integer results in truncation of any remainder: 11/3=3, for example.

Rules for conversion of operands are shown in table 1-6.

Figure 1-1 presents additional examples of the evaluation of expressions.

Masking Expressions

A masking expression contains operands of any type and the logical masking operators LNO, LAN, LOR, LXR, LIM, and LQV.

The logical operators perform bit manipulations. As with numeric arithmetic operators, a hierarchy of operators exists, as shown in the list above.

No conversion of operand types occurs with masking operators. Character data is restricted to one word, however. Any character operand less than 10 characters is left-justified and blank-filled before being used in a masking expression. Any character operand greater than 10 characters is truncated to 10.

BOOLEAN EXPRESSIONS

A Boolean expression yields a Boolean value TRUE or FALSE. The two types of expressions that yield Boolean results are:

- Relational expressions that compare values of arithmetic expressions.

A. LNO(A+B*(C-D*E-(-F+G)/3)) is functionally equivalent to:

```
D * E   → I1
C - I1  → I2
-F      → I3
I3 + G  → I4
I4 / 3  → I5
I2 - I5 → I6
B * I6  → I7
A + I7  → I8
LNO I8  → Result
```

B. A**B/C+D*E*F-G is functionally equivalent to:

```
A ** B → I1
I1 / C → I2
D * E  → I3
I3 * F → I4
I2 + I4 → I5
I5 - G → Result
```

C. A**B/(C+D)*(E*F-G) is functionally equivalent to:

```
A ** B → I1
C + D  → I2
I1 / I2 → I3
E * F  → I4
I4 - G → I5
I3 * I5 → Result
```

D. A*(B+((C/D)-E)) is functionally equivalent to:

```
C / D  → I1
I1 - E → I2
B + I2 → I3
A * I3 → Result
```

E. (A*(SIN(X)+1.)-Z)/(C*(D-(E+F))) is equivalent to:

```
SIN(X)  → I1
I1 + 1. → I2
A * I2  → I3
I3 - Z  → I4
E + F   → I5
D - I5  → I6
C * I6  → I7
I4 / I7 → Result
```

Figure 1-1. Examples of Arithmetic Expression Evaluation

- Logical expressions that involve only Boolean operands and operators.

Relational Expressions

A relational expression compares the value of two arithmetic expressions or character operands. A relation is TRUE if the operands satisfy the relation specified by the operator; otherwise, the relation is FALSE.

The operands for a relational expression must be arithmetic expressions or character operands. Any arithmetic expression is evaluated before the relational expression is evaluated, with the arithmetic evaluation following the hierarchy of operators and order of operands described above. Character operands are left-justified and blank-filled before being evaluated. Two character operands are compared algebraically by their display code values, and trailing blanks are not significant.

The relational operators are:

EQ	Equal to
GR	Greater than
LS	Less than
GQ	Greater than or equal to
LQ	Less than or equal to
NQ	Not equal to

Evaluation is as follows: If bits are all zero, the result is FALSE; otherwise the result is TRUE. The exception is that -0 and +0 in a full word are considered equal and FALSE. (SYMPL does not follow the conventions of FORTRAN which uses the sign bit for testing TRUE and FALSE).

There is no precedence for relational operators. Evaluation is left to right.

Logical Expressions

A logical expression contains only Boolean operators and Boolean operands. The result of expression evaluation is TRUE or FALSE.

The Boolean operands include scalars of type B, functions of type B, and relational expressions.

The Boolean operators are listed in order of highest to lowest precedence in evaluation:

NOT	Logical negation
AND	Logical conjunction
OR	Logical disjunction

Assuming L1 and L2 are logical expressions, the logical operators are defined as:

NOT L1	FALSE only if L1 is TRUE
L1 AND L2	TRUE only if L1 and L2 both are TRUE
L1 OR L2	FALSE only if L1 and L2 both are FALSE.

TABLE 1-6. OPERAND CONVERSION DURING EXPONENTIATION

Base/Exponent	Integer	Real	Character
Integer	No conversion Result: integer.	Base converted to real, exponent converted to integer. Result: real.	Exponent not converted but interpreted as integer. Result: integer.
Real	No conversion. Result: real	Exponent converted to integer. Result: real	Exponent converted to integer. Result: real.
Character	Base not converted, but interpreted as integer. Result: integer.	Base converted to real. Exponent converted to integer. Result: real.	Both operands converted to integer. Result: integer.

The result of a Boolean expression is always 0 or 1, even if an operand is a Boolean array item which includes several smaller items. Such array items are tested for zero versus nonzero, for example:

```
ARRAY;
    ITEM B1 B(0,0,2),
         B2 B(0,0,1);
```

Setting B1=TRUE does not set B2 to TRUE.

Evaluation of a Boolean expression terminates as soon as evaluation of any part of the expression determines the result. For example, in the logical expression L1 AND L2 AND L3 evaluation stops as soon as L1 is found to be FALSE, since the expression is FALSE once any FALSE value is discovered.

The expression A OR B AND NOT C is evaluated as if it were written:

(A OR (B AND (NOT C)))

Data in a SYMPL program is either a scalar or an array item.

Scalars are declared with ITEM declarations. Scalars occupy at least one full word of storage.

Arrays are declared with ARRAY or BASED ARRAY declarations followed by simple or compound ITEM declarations describing items in the array. An item in an array need not occupy a full word of storage.

Both scalar and array items are named entities that represent values that are preset when a program is loaded or gain values by arithmetic replacement. An exception is a filler item in a system-independent array, which has no name and whose value is therefore unavailable to the user.

Each data item must be declared before it is referenced.

ITEM DECLARATION

An ITEM declaration defines a scalar that occupies a full word in storage or, in the case of character data, that occupies as many words as necessary to hold the number of characters specified. Six types of data can be defined, each having a particular storage format associated with it:

Type	Format
Boolean	Boolean data has the value TRUE or FALSE.
Character	Character data is represented in display code, with 6 bits for each character and 10 characters per word. Characters are left-justified with blank fill.
Integer	Integer data is represented in binary integer format in which the leftmost bit represents the sign bit and the remaining bits represent the value.
Unsigned integer	Unsigned integer data is represented in binary integer format with all 60 bits being used for the value.
Real	Real data is represented in single-precision floating-point format. Restrictions on the maximum values of operands in expressions and results of expression evaluation are those common to the hardware.
Status	Status data is represented by unsigned integer format. It differs from unsigned integer format only in its use in a program and the way it can assume values. See STATUS declaration below.

The format of an ITEM declaration is:

ITEM name type=preset, name type=preset, . . . ;

name — Identifier of 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 a procedure.

type — Type of item:

B	Boolean
C(lgth)	Character, with lgth specifying number of characters. Length cannot exceed 240.
I	Signed integer; default
U	Unsigned integer
R	Real
S:stlist	Status, with stlist specifying the name of the status list from which the item is to assume values.

When an item is assigned a value, the value is converted to the type specified by the ITEM declaration.

preset — Optional; value to which item is to be initialized at load time, expressed as a constant. Any specified constant is set in the item without regard for whether the constant matches the item type. When preset is omitted, the equal sign is also omitted.

The characters B, C, I, U, R, and S are not reserved words; they can be used elsewhere in a program as variable names.

Examples of ITEM declarations are:

- Define item X as type real:

```
ITEM X R;
```

- Define item Y as integer, and define Z as a character item having 10 characters:

```
ITEM Y,Z C(10);
```

- Define NBIT as integer with a value of 6:

```
ITEM NBIT=6; or ITEM NBIT I=6;
```

- Define ERR as characters ERROR NUMB in one word and ER left-justified in next word:

```
ITEM ERR C(12) = "ERROR NUMBER";
```

- Define OFF as Boolean value FALSE:

```
ITEM OFF B = FALSE;
```

- Define status item BIRD with the same value as CANARY has in status list ORDER:

```
ITEM BIRD S:ORDER=S"CANARY";
```

STATUS DECLARATION

A STATUS declaration defines a list of items that the compiler is to associate with small unsigned integer values. The purpose of the declaration is to allow mnemonic references to certain variables of small integer value. The compiler assigns the value 0 to the first identifier of the list, the value 1 to the second identifier of the list, and so forth.

The format of a STATUS declaration is:

```
STATUS stlist identifier, identifier,...;
```

stlist	Name by which list is to be known. Identifier of 1 through 12 letters, digits, or $ that does not begin with a digit and does not duplicate the name of a reserved word. Status list names S, X, and O cause ambiguities.
identifier	Identifier to be associated with status list stlist. Need not be unique with a program since the status list with which it is associated can always be determined from the context. Can duplicate reserved word. An identifier cannot be duplicated in the list.

Identifiers in the status list are called status values. They are used in the form of a status function, a status constant, or a status switch, as described elsewhere.

Examples of STATUS declarations and references to items so declared are:

- Preset VAL to the status list value of TERM:

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

This causes VAL to be set to the unsigned integer 2.

- Set X to the status list value of BLUE:

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

This causes X to be set to 3.

- 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 a multiple branch statement.

Two types of 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 reference to an out of range switch value produces a run-time diagnostic.

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 label name can be omitted for that value. A comma is still required to mark the position of the label, unless it is the last label in the list. A label name in a switch list cannot duplicate a switch name.

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];
```

The D option on the SYMPL control statement or the CONTROL PACK statement can be used to pack switches two to a word, thus saving space at the expense of execution time.

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. All of the status values need not be associated with a label, and 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 a rectangular arrangement of elements. The elements are called entries; each entry is composed of a number of items. The items in an entry have the same format for each entry in an array. The number of words in the array 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.

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 exists in the entry, the ITEM declaration should be a compound statement.

An array is either system-dependent or system-independent. A system-dependent array is one in which the exact location (word number and bit number) is specified by the user. A system-independent array is one in which the user specifies only the length of each item, and whether or not it overlaps other items. System-independent arrays are recommended for all new applications.

SYSTEM-INDEPENDENT ARRAYS

A system-independent array is one in which the exact location of array items and the entry (position) of the array are not specified by the user, but are calculated by SYMPL. Use of system-independent arrays is recommended for new applications because such programs are more easily transportable between systems with different characteristics.

A system-independent array is a serial array with the attribute INERT (section 5). It differs from a system-dependent array in that the location of array items relative to each other is undefined unless the user explicitly specifies that they are to overlap or be contiguous.

Like system-dependent arrays, system-independent arrays must be declared in a declaration containing two parts: an array declaration header, followed by item declarations.

Array Declaration Header

The format of an array declaration header for a system-independent array is:

```
ARRAY   name[low:up,low:up,...] algnt;
```

name	Identifier specifying the name of the array. It can be omitted unless the array appears in a BASED ARRAY, XDEF, or XREF declaration, or is used as an actual or formal parameter.
low	Lower bound of a dimension of the array. Maximum absolute value is $2^{17}-1$. Can be positive, negative, or zero. If low and the following colon are omitted, zero is assumed.
up	Upper bound of a dimension of the array. Can be positive, negative, or zero. Must be greater than or equal to low.
algnt	Arrangement of items in memory:
	A Aligned. Each item begins at the left side of a new word. Character items can occupy more than one word. This arrangement promotes speed of access at the expense of memory space.
	U Unaligned. Each item begins at the next bit position that enables the item to be contained within one word. An exception is character items, which begin at the next character position, and can be split across word boundaries. This arrangement promotes saving memory at the expense of execution speed.

An array can have up to seven dimensions. Each pair of bounds in the declaration defines one dimension. If the bounds list is omitted, [0:0] is assumed.

Array Item Declarations

The system-independent array declaration header is followed by the declarations for the items contained in the array. If more than one item declaration is used, they should be grouped as a compound bracketed by BEGIN and END. The format of the ITEM declaration for system-independent arrays is:

ITEM name type<size,pos>=[preset],

name type <size,pos>=[preset]...;

name — Identifier specifying the name of the entry item. Must be unique within the procedure. If the pos parameter is present, the name can be omitted, defining a filler item.

type — Type of array item:

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

size — Item length, expressed as an unsigned integer constant. Length is in characters for character data, and in bits for all other data. R type items must be 60 bits long. Only C type items can cross word boundaries; 60 bits is the maximum length for all other items. Size defaults as shown in table 2-1.

pos — Indicates the position of the item relative to other items in the same array. If pos (and the preceding comma) are omitted, the position of the item relative to other items is undefined, except that they will not overlap. If pos is an item name, the first bit position of the current item is the same as the first bit position of the named item. If pos is a +, the current item begins at the next available bit position after the last defined item.

preset — Value or set of values to initialize the item (or multiple occurrences of the same item). See Presetting Arrays.

The pos parameter allows subfields of an item to be defined, each with its own name. All the subfield item declarations that overlap a given item must be grouped together, but they need not occur immediately after the declaration for the overlapped item. The first declaration in the group names the item being overlapped in the pos parameter, and the remaining items use the plus sign to indicate the subsequent subfields. Any item with a plus sign as the pos parameter must be preceded by an item with a non-null pos parameter. Dummy items can be used to space over unused subfields. For example, to declare an array with one-word entries, each containing two 20-bit subfields at each end of the word and a dummy subfield in the middle:

```
ARRAY TABLE[1:100]A;
BEGIN
ITEM ALLOFIT U<60>;
ITEM FIELD1 U<20,ALLOFIT>;
ITEM DUMMY U<20,+> ;
ITEM FIELD2 U<20,+> ;
END
```

TABLE 2-1. ARRAY ITEM DESCRIPTOR LIMITS

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

A subfield must not be defined to exceed the boundaries of the more inclusive field.

Sub-subfields can be defined, and are subject to the same options and restrictions as subfields. The name can be omitted from a subfield (an item with a non-null pos parameter); such items are called filler items. In the preceding example, the item declaration for item DUMMY could be replaced by:

ITEM<20,+>;

provided that the item is not referenced in the program. The type is the same as that of the field being overlapped.

All items in a group of subfield items must be of compatible types. They must all fall within the same class; the item of which the subfields form a part must also be in the same class. The classes are as follows:

Class 1	Character
Class 2	Real
Class 3	Signed integer, unsigned integer, status, and Boolean

Real items are always 60 bits long. If any other length is specified, 60 is substituted and a trivial diagnostic is issued.

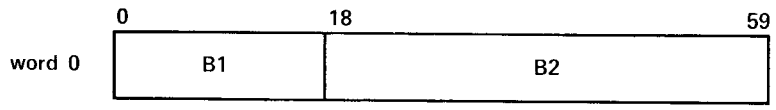
Some examples of system-independent array declarations are shown in figure 2-1.

Array declaration

Arrangement of items in storage

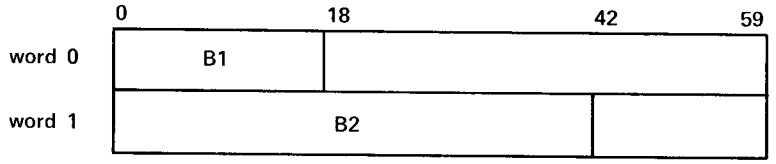
Array whose entries each occupy one 60-bit word:

```
ARRAY B U;
    BEGIN
    ITEM B1 I<18>;
    ITEM B2 C<7>;
    END
```



Array whose entries each occupy two 60-bit words:

```
ARRAY B A;
    BEGIN
    ITEM B1 I<18>;
    ITEM B2 C<7>;
    END
```



Array using a filler item and sub-subfields (Y and Z):

```
ARRAY ABC [1:144] A;
    BEGIN
    ITEM W<18>;
    ITEM <6,W>;
    ITEM X<12,+>;
    ITEM Y<5,X>;
    ITEM Z<7,+>;
    END
```

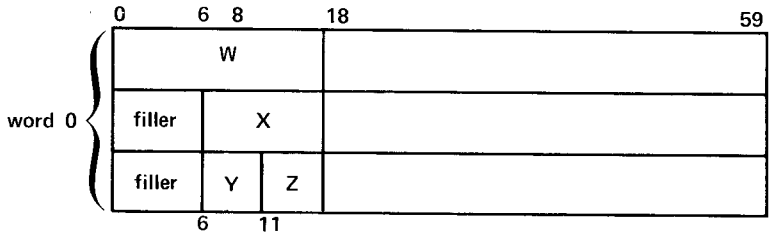


Figure 2-1. System-Independent Array Examples

The following are examples of invalid declarations:

- Subfield XC is larger than XA; declaration could be rewritten to make XA a subfield of XC:

```
ARRAY X [1:1001] U;
    BEGIN
    ITEM XA I<7>;
    ITEM XB B<1>;
    ITEM XC U<8,XA>;
    END
```

- Field with plus sign as pos parameter follows field with null pos parameter; could be made legal by reversing W2 and W3, or making W2 a subfield of W:

```
ARRAY XX[1:667,23:29]U;
    BEGIN
    ITEM W      I<18>;
    ITEM W1     I<6,W>;
    ITEM W2     I<4>;
    ITEM W3     I<7,+>;
    END
```

Incompatible type of field and subfield:

```
ARRAY Q [1:2] U;
    BEGIN
    ITEM QA U<18>;
    ITEM QB U<6,QA>;
    ITEM QC C<2,+>;
    END
```

SYSTEM-DEPENDENT ARRAYS

A system-dependent array is one in which the user specifies the location of each item in the array entry. Both the word and bit positions in which the item begins are specified. System-dependent arrays are not recommended for new applications.

Array Declaration Header

The format of a system-dependent 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, or is used as an actual or formal parameter.

low — Integer constant indicating lower bound of a dimension of the array. Maximum absolute value is $2^{17}-1$. Can be signed positive or negative. If low and its following colon are omitted, 0 is assumed.

up — Integer constant indicating upper bound of a dimension of the array. Maximum absolute value is $2^{17}-1$. Can be signed positive or negative. Must be equal to 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. The declaration ARRAY S(n) declares a parallel array named S, not an unnamed serial array.

Differences between serial and parallel allocation are shown in figure 2-2. 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(2,0,10);
    END
```

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 of 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

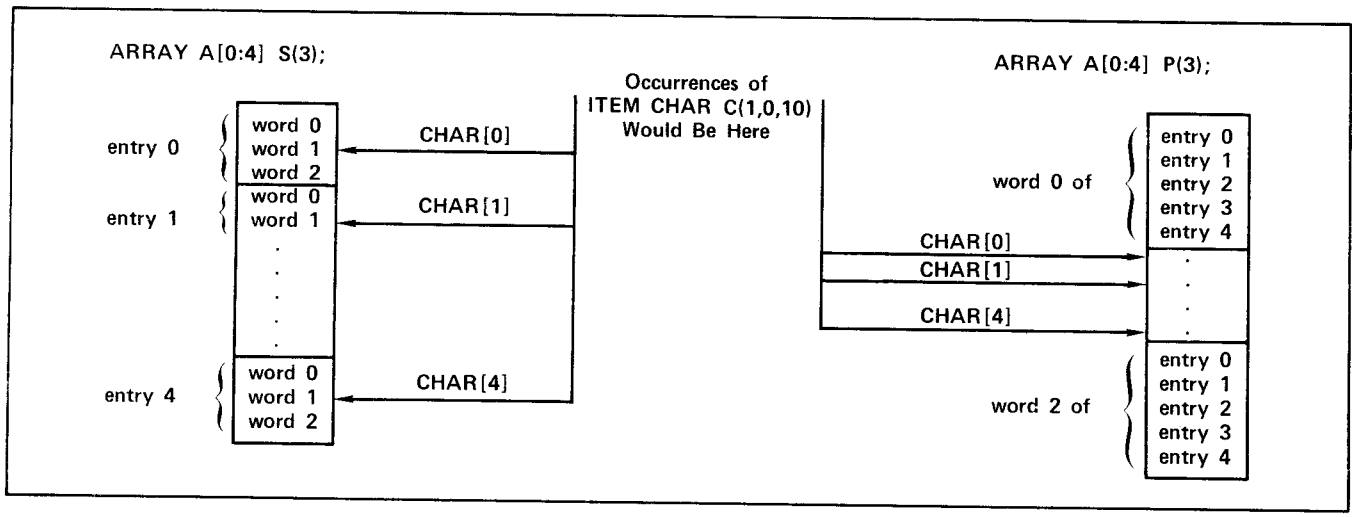


Figure 2-2. Differences in Serial and Parallel Allocation

ep — Entry position. Word number in which the integer or character item starts, starting from 0; expressed as an unsigned integer constant. ep can be less than esize; the user is responsible for the validity of any results that ensue.

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 is a bit number and must be divisible by six.

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.

All items are right-justified and zero-filled, except character items, which are left-justified and blank-filled. The constant is not converted. No conversion of the preset is performed regardless of the type of item being preset.

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.

Items can overlap; the user is responsible for the consequences if the same field is declared with two different types.

Serial and Parallel Arrays

When a system-dependent array has only one entry, or its entries occupy no more than one word, the distinction between serial and parallel arrays is meaningless. For an array with more than one entry, or for an array with one entry having more than one word, the time required to access any given item is affected. The distinction becomes critical when array items are declared with word positions beyond the entry size.

Figure 2-3 shows an example of serial array storage based on a declaration of:

```
ARRAY SAR[0:10]S(2);
    BEGIN
    ITEM AA(0,0,60);
    ITEM AB(1,0,60);
    END
```

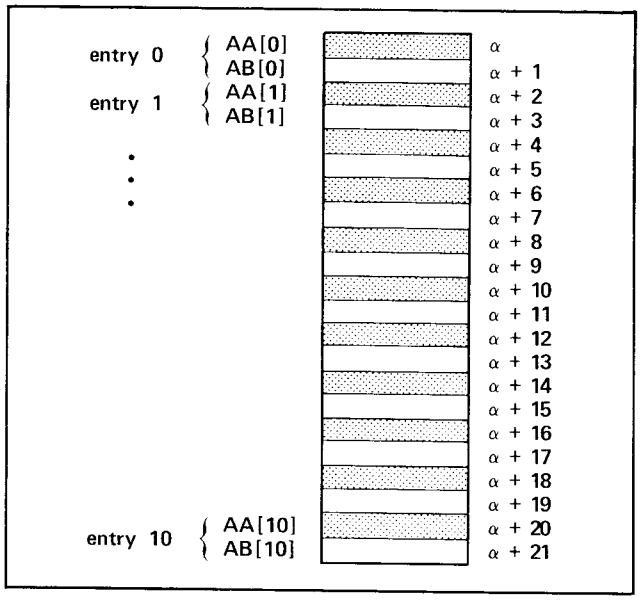


Figure 2-3. Serial Array Allocation

Each entry in the array is two words long. The first item of the entry is AA; the second item is AB. All array items with a 0 value for ep are stored at location SAR+x where x is even, while all array items with a 1 value for ep are stored at location SAR+x where x is odd. (Shaded areas indicate entries with ep=0.)

If, in figure 2-3, a third item AC (2,0,60) is added, the entry is said to be over-addressed, since AC[0] is the same location as AA [1]. Such a practice should be used cautiously because of optimization considerations discussed in appendix C.

Figure 2-4 shows the same array as figure 2-3 with parallel, rather than serial, storage allocation, based on a declaration of:

```
ARRAY PAR[0:10]P(2);
    BEGIN
    ITEM AA(0,0,60);
    ITEM AB(1,0,60);
    END
```

Each entry in the array is two words in length, as it is with serial allocation. The first item in the entry is AA; the second is AB.

Occurrences of word 0 of the array (item AA) are stored contiguously, and occurrences of word 1 (item AB) are stored contiguously after those with ep=0.

If item AA is over-addressed, AA[11] is the same as AB[0]. Such a practice should be avoided because of optimization considerations discussed in appendix C.

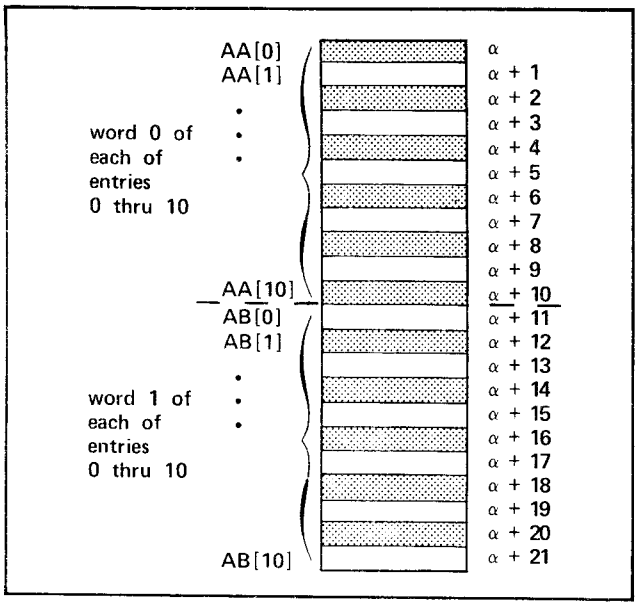


Figure 2-4. Parallel Array Allocation

Figure 2-5 shows the implications of serial and parallel allocation for arrays in which array items occupy more or less than a full word in the entry.

The array illustrated in figure 2-5A is declared by:

```
ARRAY NENT[0:3] S(4);
    ITEM A1    I(0,0,15);
    ITEM B1    U(0,15,15);
    ITEM C1    U(0,30,30);
    ITEM D1    C(1,0,20);
    ITEM E1    R(3,0,60);
    END
```

Notice that in serial allocation all four words of each entry are contiguous, but that the occurrence of one item is not contiguous with other occurrences of the same item.

In contrast with figure 2-5A, figure 2-5B shows the same four-word entry in parallel allocation. The array entry description is the same, but the array header is:

```
ARRAY NENT[0:3] P(4);
```

Notice that in parallel allocation all four words of each entry are not contiguous, but that the occurrence of one item is contiguous with other occurrences of that same item. That is, all occurrences of array item A1[n] are contiguous.

For an item that crosses word boundaries in a parallel array, the same parallel structure is maintained, as shown by array item D1 in figure 2-5B. The first word of array item D1[n] is stored together with all other occurrences of the first word of D1[n], and the second word of array item D1[n] is stored together with all other occurrences of the second word of array item D1[n].

The physical split of a multiword item does not affect the logical operation of a bead function that specifies characters to be extracted from an item split between words: C<5,9>F, for example, accesses correctly the 9 characters beginning with character 5 of item F even though the words are not contiguous in memory.

Parallel and serial arrays offer contrasting advantages. Parallel arrays are more efficient when items do not exceed one word and the sum of the lengths of the items does not exceed the entry size. Serial arrays are more efficient for multiword items and for references to items that exceed the bounds of the entry. Serial arrays are also more efficient if the size of the array is to be increased at execution time (which cannot be done directly through SYMPL).

ARRAY REFERENCES

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

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

To reference the 40th occurrence of REFITEM, which in this example is the 40th word of the array, the reference is:

```
REFITEM[39]
```

The subscript for the item reference must be an arithmetic expression. If the type of the arithmetic expression is other than integer, the result of the expression is converted to integer. Only the lower 18 bits of the value are used.

If the array being referenced has more than one dimension, the subscript must have as many arithmetic expressions as there are array dimensions. For instance:

```
ARRAY[0:1,0:2,0:3];
    ITEM B I;
```

All of the following are possible references:

```
B[1,1,1]

B[X+Y,1,Z]

B[B[1,B[0,0,0],X],B[1,B[X,1,Y],Y],Z]
```

If an array entry occurs only once (that is, dimensions are specified as [0:0],[10:10], and so forth), it can be referenced without a subscript.

PRESETTING ARRAYS

Elements of an array are initialized by an ITEM declaration that has a list of values associated with the array item name. For instance, an array with one dimension is initialized:

```
ARRAY SIGMA[2:6];
ITEM CHI C(0,54,1)=["A","D","G","K","N"];
```

The resulting structure and values are:

A	CHI[2]
D	CHI[3]
G	CHI[4]
K	CHI[5]
N	CHI[6]

The first word of this parallel array is CHI[2].

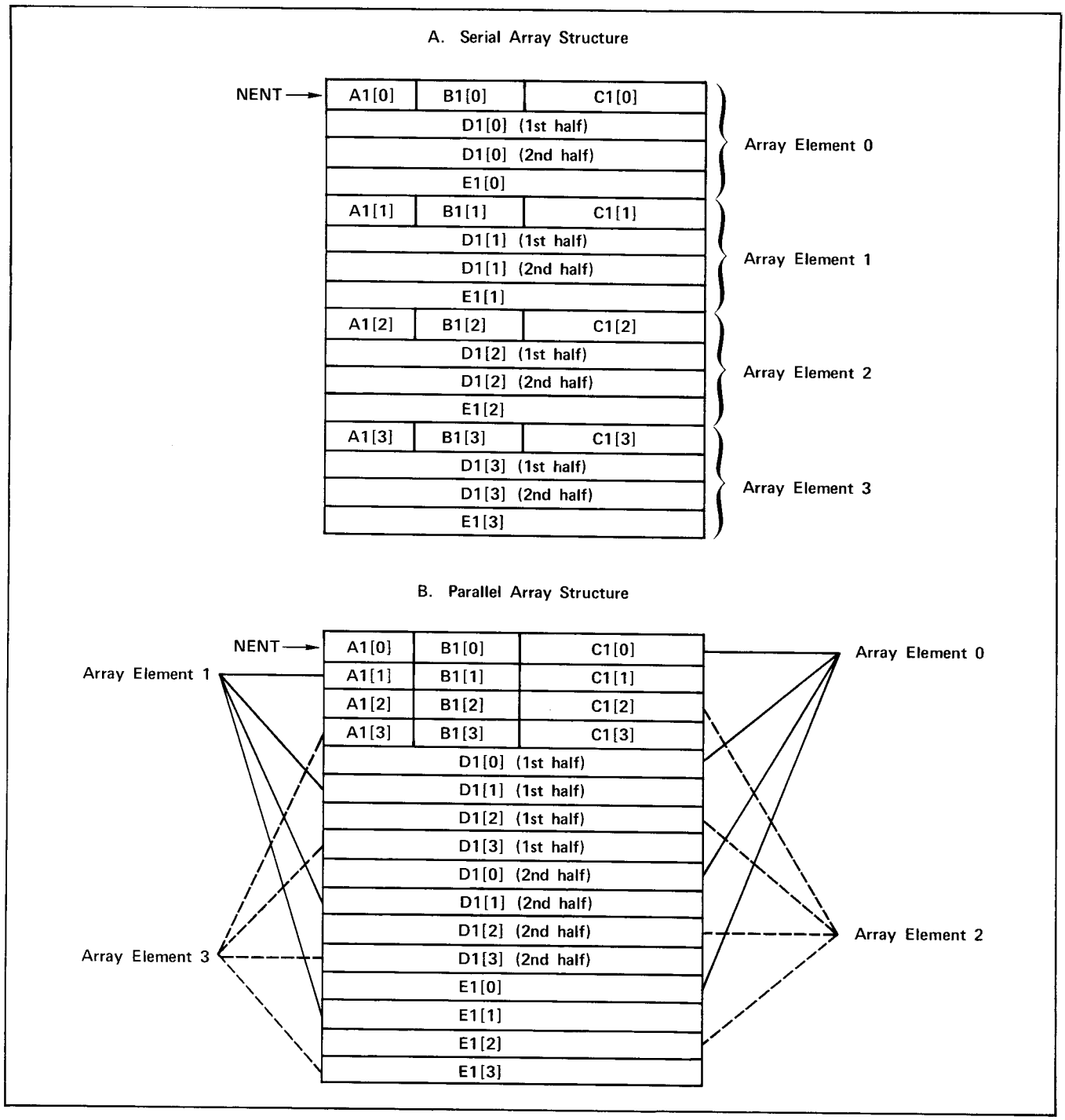


Figure 2-5. Serial and Parallel Arrays with Multiword Items

The list of constant values for array item elements need not specify an initial value for every element. The values given are used to set elements starting with the first element. Any element that is not to be set is indicated by a null value:

- Null values are indicated by adjacent commas.
- Trailing null values can be omitted.
- To initialize the second and fourth element of array SIGMA above, for instance:

```
ITEM CHI C(0,54,1)=["D",,"K"];
```

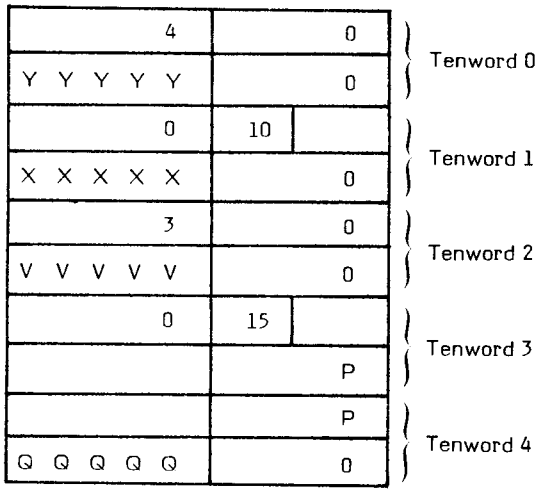
For entries with more than one item, the preset values are specified by the item with which they are associated, not the word in memory in which they might appear. The programmer specifies the value for each occurrence of an item; the compiler constructs words as necessary to produce an array with the given specifications.

Any item not preset is set to zero if any other item in the same word is preset. Otherwise, unpreset items are initialized to the current loader default.

Example:

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

Resulting structure and values are:



P indicates a loader preset.

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.

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 be 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.

If overlapping fields are preset, the last specified preset applies to the bits shared by the items.

ARRAY STORAGE AND ADDRESSING

Given the array header:

```
ARRAY [b1:u1,b2:u2,...]
    alloc(esize);
```

the number of entries in the array is:

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

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_i;

$[s_1,s_2,\ldots,s_n]$ is computed from:

$$e_i=s_i-b_i$$

$$size=u_i-b_i-1$$

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

where esize is entry size.

For parallel allocation, the location of an element is computed from:

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 $[b_1,\ldots,b_n]$.

For a three-dimensional 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

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

In a two-dimensional array, memory locations are:

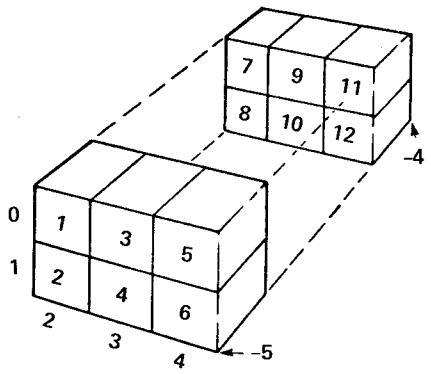
```
ARRAY PSI[1:3,0:3] alloc(2);
    BEGIN
    ITEM X(0,0,60)
    ITEM Y(1,0,60)
    END
```

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[3,3]
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, an array declaration might be:

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

The resulting structure of array RHO is shown in figure 2-6.



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 2-6. 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.

Based arrays can be used when the size or location of an array is not known at compile time, if the array might be moved during execution, or if the same array definition is needed in several locations.

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. Array name is required; dimensions are optional.

With two exceptions, references to based arrays are the same as references to any other array, with the same result. The exceptions are:

1. The P function, which requires a based array as its argument.

2. Actual parameters; if a based array is an actual parameter to a procedure, the pointer variable, not the array name, must be passed as the actual parameter.

Based arrays cannot be preset.

Statements, as opposed to declarations, are executable. They can be grouped into two types depending on their function:

- Value assignment statements cause a value to be assigned to a scalar or an array item element. These statements are:

  Exchange statement

  Replacement statement

- Flow-of-control statements control the order of statement execution. These statements are:

  GOTO statement

  IF statement

  FOR statement and its associated TEST statement

  STOP statement

  RETURN statement

  Procedure call statement

All executable statements can be labeled.

LABELS

A label is an identifier used to locate a statement.

The format of a statement label is:

name:

name Identifier of 1 through 12 letters, digits, or $ that does not begin with a digit and does not duplicate another identifier in the subprogram or a reserved word.

No blanks or comments are permitted between the last character of the name and the terminating colon.

Since a labeled statement is itself a statement, two labels in sequence are synonymous.

A label can appear at any point in the program where it is legal for a statement to appear. If a declaration or subprogram is labeled, the label locates the next executable statement.

The scope of a label is the procedure in which it appears, plus any procedures nested within that procedure. Grouping of statements into compound statements by means of BEGIN and END does not affect the scope of a label.

A given label name can only appear once in a procedure. However, the name can be duplicated in a nested procedure. If a branch to the name is encountered, the choice of which label to branch to is governed by the following rules:

- If a statement labeled with the name has already been encountered in the same procedure or an outer procedure, the branch is to that statement.
- If no statement yet encountered has been labeled with the name, the branch is to the next statement labeled with that name at the same level or an outer level.
- If the name appears in a LABEL declaration in a procedure, then the branch is to the next statement labeled with that name at the same level or an outer level, regardless whether that label has been encountered yet or not.

A LABEL declaration has the format:

LABEL name, name,...;

name Label that is to be subsequently declared.

For example:

- Procedure NAME1 is nested within procedure NAME. The compiler links a label L1 reference within NAME1 to the label #PREV# L1 previously encountered during compilation.

```
          PROC NAME;
          BEGIN
#PREV#      L1:...
            PROC NAME1;
            BEGIN
            GOTO L1;
            .
            .
            .
#INNR#      L1:...
              END #NAME1#
        END #NAME#
```

- The same procedures with a LABEL declaration within the nested NAME1 transfers control to L1 #INNR# when GOTO L1 executes:

```
          PROC NAME;
          BEGIN
#PREV#      L1:...
              PROC NAME1;
              BEGIN
              LABEL L1;
            GOTO L1;
            .
            .
            .
  #INNR#      L1:...
              END
      END
```

REPLACEMENT STATEMENT

The replacement statement assigns a value to a scalar or subscripted array item. When the statement executes, the value of the expression on the right side replaces the current value of the entity on the left side of the statement.

The format of the replacement statement is:

```
v = exp;
```

v — One of the following entities whose value is to be replaced:

- Scalar
- Subscripted array item
- P function
- Bead function
- Function name, if statement is within a function of the same name

exp — Arithmetic or relational expression.

The value of the right side is adjusted to the size of the left side operand. If necessary, integers are truncated on the left; character data is truncated on the right. Expansion occurs with leading zeros for integers; characters are left-justified and blank-filled.

If one side of the statement is Boolean, the other must also be Boolean. No conversions occur to or from Boolean.

If the left side of a replacement statement is a bead function (C or B described in section 4), only the specified bits are replaced. The remainder of the referenced item is not affected.

The expression on the right side of the statement is converted to the type of the left side, if necessary, before the value is assigned to the left side. Conversions occur as follows:

- Integers are converted to character operands by the left-justification of the rightmost 6 bits of the integer. Remaining positions in the character field are blank-filled.
- Integers are converted to real operands by floating them, as provided by hardware instructions. The resulting real values are expressed in single-precision format.
- Real values are converted to integers by truncating any fractions in the real values. Significance is preserved if the integer can be expressed in 48 bits.
- Real values are converted to character operands by first converting to integer, then converting the integers to characters.

Character operands are converted to integer by right-justifying and zero-filling them in a single word. If the operand is more than ten characters long, only the first ten characters are used.

Character operands are converted to real by converting them to integer and then floating the result.

Conversion does not occur between integer and unsigned integer; they are treated identically. If a 60-bit unsigned integer has a 1 in the sign bit, it is treated as a negative number; all other unsigned integers are treated as positive numbers.

Table 3-1 summarizes conversions performed by the replacment statement.

Examples of replacement statements:

- Assign the value of TEMP to ITEMA:

```
ITEMA = TEMP;
```

- Assign B the value TRUE or FALSE as a result of the logical conjunction of the value of E and the relational value TRUE or FALSE obtained from the evaluation of C NQ D:

```
B = C NQ D AND E;
```

EXCHANGE STATEMENT

The exchange statement causes the exchange of values of the left and right sides of the exchange operator ==. 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.

TABLE 3-1. REPLACEMENT STATEMENT CONVERSIONS

Expression Type/ Variable Type	Real	Integer	Character
Real	---	Truncate fractional part of number.	First convert to integer; then convert that integer to character.
Integer	Float number.	---	Left-justify lower 6 bits of number with blanks.
Character	Convert to integer, then float.	Right-justify and zero-fill all or first ten characters of value; discard the remainder.	---

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

SYMPL guarantees that subscript or bead function components of expressions which must be evaluated to compute the address of v1 or v2 are computed once, before either replacement. 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 DO statement

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
```

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.

In the form:

```
FOR i = aexp1 STEP aexp2 UNTIL aexp3 DO
statement
```

if the first character of aexp2 is a minus sign, then the loop is executed until i is not greater than aexp3. For example, in the following statement the loop is executed three times, with I equal to 1, -6, and -13 (assuming that it is a slowloop):

```
FOR I = 1 STEP -7 UNTIL -17 DO...
```

However, the following loop is never executed if it is a slowloop, and executed once if it is a fastloop:

```
FOR I = 1 STEP (-7) UNTIL -17 DO...
```

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 use data of any type except Boolean. The results of the expressions are converted to the mode of the induction variable. The Boolean expression in the WHILE clause can be arbitrarily complex.

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.

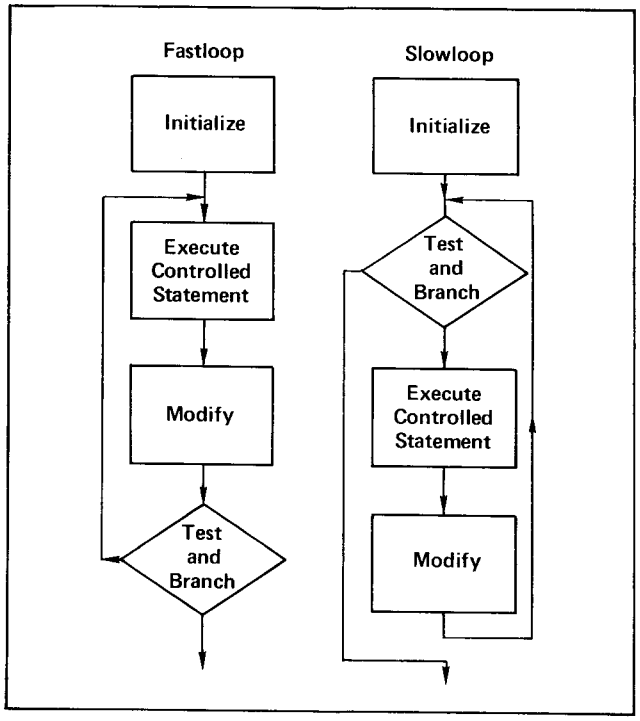


Figure 3-1. Generalized Fastloop and Slowloop Flowcharts

Fastloops always execute at least once 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 induction variable, the STEP nor the UNTIL expression can be modified within the loop since SYMPL might evaluate these expressions before the start of the loop. The WHILE clause can be changed within 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.

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. The induction variable of a fastloop is not preserved when control is transferred outside the loop. If the controlled statement is entered by a GOTO statement from outside the FOR statement, the value of the induction variable is undefined.

Table 3-2 shows the different types of FOR statements and the logic of their generated code.

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

TABLE 3-2. SLOWLOOP AND FASTLOOP EXPANSION COMPARED

Statement	Slowloop	Fastloop

`FOR I=X1 DO A=0;`	`I=X1;`
	`L: A=0;`
	`GOTO L;`
		`I=X1;`
		`L: A=0;`
		`GOTO L;`

`FOR I=X1 STEP X2+2 DO A=0;`	`I=X1;`
	`L: A=0;`
	`I=I+X2+2; GOTO L;`
		`temp1=X2+2;`
		`I=X1;`
		`L: A=0;`
		`I=I+temp1; GOTO L;`

`FOR I=X1 STEP X2+2 UNTIL X3+3 DO A=0;`	`I=X1;`
	`L: IF I LQ X3+3 THEN`
	`BEGIN`
	`A=0;`
	`I=I+X2+2; GOTO L;`
	`END`
		`temp1=X2+2;`
		`temp2=X3+3;`
		`I=X1;`
		`L: A=0;`
		`I=I+temp1;`
		`IF I LO temp2 THEN GOTO L;`

`FOR I=X1 STEP -X2+2 UNTIL X3+3 DO A=0;`	`I=X1;`
	`L: IF I GQ X3+3 THEN`
	`BEGIN`
	`A=0;`
	`I=I-X2+2; GOTO L;`
	`END`
		`temp1 X2+2;`
		`temp2=X3+3;`
		`I=X1;`
		`L: A=0;`
		`I=I+temp1;`
		`IF I GQ temp2 THEN GOTO L;`

`FOR I=X1 WHILE BX DO A=0;`	`I=X1;`
	`L: IF BX THEN`
	`BEGIN`
	`A=0; GOTO L;`
	`END`
		`I=X1;`
		`L: A=0;`
		`IF BX THEN`
		`GOTO L;`

`FOR I=X1 STEP X2+2 WHILE BX DO A=0;`	`I=X1;`
	`L: IF BX THEN`
	`BEGIN`
	`A=0;`
	`I=I+X2+2; GOTO L;`
	`END`
		`temp1=X2+2;`
		`I=X1;`
		`L: A=0;`
		`I=I+temp1;`
		`IF BX THEN GOTO L;`

`FOR I=X1 STEP 27 WHILE A LQ B AND C GR D DO A=0;`	`I=X1;`
	`L: IF A LQ B AND C GR D`
	`THEN`
	`BEGIN`
	`A=0;`
	`I=I+27;`
	`GOTO L;`
	`END`
		`I=X1`
		`L: A=0;`
		`I=I+27;`
		`IF A LQ B AND C GR D`
		`THEN GOTO L;`

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 structured method of branching to the modify-test-branch step; it is meaningful only within the controlled statement of a FOR statement.

The format of the TEST statement is:

```
TEST name;
```

name — Name of the item used as induction variable in a loop containing the TEST statement. If omitted, control transfers to the modify-test-branch sequence of the innermost loop.

When TEST is executed, control transfers to the modify-test-branch for the specified induction variable. Consequently, the other inner index modify-test-branch steps could be skipped and those induction variables would not be incremented for the next iterations. If TEST transfers control outside the innermost level, the inner levels are terminated and can be reentered only through their FOR statement beginning. Meanwhile, the induction variables have the values they had when the jump outside the loop occurred, except for fastloops.

Examples of TEST use within FOR:

- Bypass all source statements between the TEST statement and END if the Boolean expression of an IF statement is TRUE.

```
FOR A=0 STEP 1 UNTIL 52
DO
    BEGIN
    IF DEMAND[TODY] GR
        DEMAND[TOMRW]
        THEN TEST;
    .
    .
    .
    END
```

- Bypass innermost nested statements within a FOR loop.

```
FOR A = 0 STEP 1
 UNTIL 100
DO
 BEGIN #A#
 .
 .
 .
 FOR B=99 STEP -1
    UNTIL 0
 DO
    BEGIN #B#
    IF INCOME[B] GR 1000
     OR CREDIT[B] EQ S"GOOD"
    THEN
     BEGIN
     TEST A;
     END
    IF END INCOME[B] LS 5000
     AND AGE[B] LS 18
    THEN
     BEGIN
     TEST B;
     END
    .
    .
    .
    END #B#
END #A#
```

If the conditions in the first IF statement are satisfied, control passes to the modify-test-branch for the outer loop, index A. If the first TEST statement had not specified A, control would have passed to the innermost test for B. If both conditions in the first IF statement are FALSE, execution bypasses the first TEST statement; and if the conditions of the second statement are satisfied, TEST B is executed, which passes control to the test for index B. Only when the above conditions are FALSE are the source statements following TEST B executed.

GOTO STATEMENT

The GOTO statement unconditionally transfers control to a statement designated by a label name or a subscripted switch list name.

The format of a GOTO statement specifying a label name is:

```
GOTO label;
```

label — Name of a label within the program, a formal label, or an XREF label.

The format of a GOTO statement specifying a subscripted switch list name is:

```
GOTO swname[exp];
```

swname — Name of a switch list previously declared in a SWITCH declaration.

exp — Arithmetic expression whose value is one of the small integers the compiler assigns to switch list entities.

The following are examples of GOTO statements:

- Unconditionally transfer control to label JAIL:

```
GOTO JAIL;
```

- Unconditionally transfer control to the label whose value corresponds to the value of the expression A+B-C:

```
SWITCH ASW ZERO, ONE, TWO, THR, FOUR;
.
.
.
GOTO ASW[A+B-C];
```

IF STATEMENT

The IF statement causes a conditional transfer of control depending on the value of a Boolean expression within the statement. The format of the IF statement is:

```
    IF bexp THEN statement1
or
    IF bexp THEN statement1 ELSE statement2
```

bexp	Boolean expression; can be arbitrarily complex.
statement1	Statement to be executed when the value of the Boolean expression is TRUE.
statement2	Statement to be executed when the value of the Boolean expression is FALSE.

If ELSE statement2 is omitted, and the value of the Boolean expression is FALSE, the next statement after the IF statement is executed.

Each statement within the IF statement follows the syntax rules for statements; each can be a simple or compound statement; each statement must be terminated with a semicolon.

When an IF statement is nested within another IF statement, ELSE clauses are always associated with the inner nested incomplete IF statements, as shown in the examples below.

The following are examples of IF statements.

- Set PSI to 6 when RHO is less than or equal to 1 or XI is 1; otherwise set PSI to 9:

```
IF RHO LQ 1
    OR XI EQ 1
THEN
    PSI = 6;
ELSE
    PSI = 9;
```

- Execute procedure VOTER when AGE is at least 18; otherwise execute procedure MINOR:

```
IF AGE GQ 18
THEN
    VOTER;
ELSE
    MINOR;
```

- Change a TRUE value of BOOL to FALSE:

```
IF BOOL
THEN
    BEGIN
    BOOL = BOOL AND BOOL2;
    END
```

- Set NUMOPDS to the number of operands in the binary expression indicated by LOPD and ROPD:

```
IF LOPD EQ S "NULL"
THEN
    NUMOPDS = 0;
ELSE
    BEGIN
    IF ROPD EQ S "NULL"
    THEN
        NUMOPDS = 1;
    ELSE
        NUMOPDS = 2;
    END
```

RETURN STATEMENT

The RETURN statement returns control to the calling routine.

The format of the RETURN statement is:

```
    RETURN;
```

The compiler generates a RETURN statement after the last statement in the body of a procedure or function.

A RETURN statement in a main program is equivalent to a STOP statement.

STOP STATEMENT

The STOP statement halts program execution and returns control to the operating system. A STOP statement is generated automatically by the compiler after the last statement of the main program.

The format of the STOP statement is:

```
    STOP;
```

A SYMPL program can be a main program, a procedure subprogram, or a function subprogram. Function and procedure subprograms can be nested within a main program or another subprogram.

SCOPE OF VARIABLES

Names declared within a subprogram body are recognized only within that subprogram. And, since subprograms can be nested, any name declared within a subprogram body is therefore recognized within any subprogram nested within it.

The scope of a declaration is the subprogram within which it occurs. When nested subprograms contain declarations for the same name, the innermost declaration has precedence.

In figure 4-1, item AA can be referenced from any of procedures A, B, C, or D. Item CC, however, can be referenced from within C, but not from within procedure D. Within procedure D, a reference to item DD uses the value indicated by #LOCAL#; in any other procedure, a reference to DD uses the value indicated by #GLOBAL#.

```
PROC A;
BEGIN #A#
ITEM AA;
ITEM DD; #GLOBAL#
    .
    .
    .
    PROC B;
    BEGIN #B#
    ITEM BB;
        .
        .
        .
        PROC C;
        BEGIN #C#
        ITEM CC;
        .
        .
        .
        END #C#

        PROC D;
        BEGIN #D#
        ITEM DD; #LOCAL#
        .
        .
        .
        END #D#
    END #B#
END #A#
```

Figure 4-1. Scope of Declarations

When a name is used, the last declaration encountered is the declaration in effect, even if it occurs at an outer level. For labels and procedures only, a name can be used before it is declared. In this case, if the same name has already been declared at an outer level, the definition at the outer level is in effect. Under these circumstances, the only way the definition in the inner level can be used is to provide an FPRC declaration for the procedure or a LABEL declaration for the label. The FPRC and LABEL declaration occurs before the use of the procedure or label, and signals the compiler that the actual procedure or label occurs later.

An example of a LABEL declaration and use is shown in section 3. An FPRC declaration is required under similar circumstances. The FPRC declaration is:

FPRC name;

name Procedure name to be declared subsequently.

The outermost subprogram name of a compilation unit is used by the loader as the name of the unit. Such a unit can be referenced as an external subprogram. External subprograms, and programs that reference external subprograms, are subject to the XDEF and XREF rules noted below. Communication between separately compiled programs and subprograms can be performed by COMMON declarations, by passing parameters to the subprograms, or by the XDEF/XREF mechanism.

MAIN PROGRAM

A main program consists of a program header followed by a series of declarations and statements and ended by a TERM statement. The TERM statement is explained in section 5.

The format of a main program header is:

PRGM name;

name Name by which program is known. Identifier of 1 through 12 letters, digits, or $ that does not begin with a digit or $ and does not duplicate a reserved word. For loader purposes, the name is truncated to seven characters.

PROCEDURES

A procedure is subprogram that executes when its name, or one of its alternative entry points, is called. Parameters can be passed to the procedure as part of its call.

When a procedure is called, execution begins at the first executable statement following the procedure name or entry point, depending how the procedure was referenced. Execution continues until a RETURN statement is encountered, a GOTO statement in the procedure transfers control outside the procedure, or the procedure ends. Termination of the procedure through its last statement or a RETURN statement returns control to the statement following the one that called the procedure. A procedure cannot call itself.

A procedure must be defined by a procedure declaration. The declaration can appear anywhere in a program that any other declaration can appear. The flow of control during execution of a main program or subprogram is not affected by the nesting of any subprogram.

A procedure declaration consists of the following elements in this order:

- Procedure header
- Optional series of declarations
- Procedure body consisting of a single elementary or compound statement. The compound statement can include declarations, alternative entry points, or other elementary or compound statements.

The format of a procedure header is:

PROC name (param, param, . . .);

name Identifier of 1 through 12 letters, digits, or $ that does not begin with a digit and does not duplicate a reserved word.

param Optional formal parameter used within the procedure for which an actual parameter is to be substituted at the time the procedure executes. A null parameter is invalid.

Any of the following can be specified by name:

Scalar	Array
Label	Based array
Procedure	Function

A scalar name can be enclosed in parentheses to indicate to the compiler that the actual parameter is to be passed by value rather than by address.

A call-by-value parameter should be used whenever possible because it produces more efficient object code. Such a parameter cannot be used, however, if its value must be returned to the calling program. Calls-by-value reference a copy of an actual parameter, not the parameter itself. Therefore, changes made to the parameter within the subprogram are not reflected in the calling program.

An example of a procedure declaration that sets a 100 word array to real values 0.0 is:

```
PROC CLEAR(X,(N));
    BEGIN
    ARRAY X[99];
          ITEM XX R(0,0,60);
    ITEM N;
    ITEM I;
    FOR I=0 STEP 1 UNTIL N DO
          XX[I] = 0.0;
    END
```

Procedure CLEAR contains a single compound statement. Declarations for parameters X and N appear within the compound statement, as does the declaration for the induction variable of the FOR loop. I could also be a global declaration, rather than a local declaration within CLEAR.

FORMAL PARAMETERS

The procedure body must contain a declaration for each of its formal parameters, except for a label name used as a parameter. (Any formal parameter that does not have an associated declaration is assumed to be a label.) Formal parameters, and other entities used within the procedure, must be declared before they are referenced.

The declarations for scalars, arrays, based arrays, and subscripted array items are the same within a procedure body as they would be elsewhere.

The declarations for procedures and functions that are formal parameters require a declaration in the following formats:

- For each procedure name used as a formal parameter:

  FPRC name;

  name Formal name of procedure.

- For each function name used as a formal parameter:

  FUNC name type;

  name Formal name of function.

  type Type of function: B, C(lgth), U, I, R, S.

A LABEL declaration for a formal parameter label is permitted but not required. If it is present, it declares the formal label, rather than indicating that a label with the same name appears later in the procedure. If a label with the same name appears in the procedure, the label can only be branched to backwards; a branch appearing before the label references the parameter label.

A declaration for a formal parameter is not recognized if it occurs in a nested inner procedure. For example, in the following, the declaration ITEM A produces a scalar local to procedure Y, not an association with parameter A of procedure X.

```
PROC X(A);
BEGIN
    PROC Y(B);
    BEGIN
    ITEM A;
    .
    .
    .
    END
.
.
.
END
```

ACTUAL PARAMETERS

Actual parameters are those arguments passed to the procedure when the procedure is called during execution. Any of the following can be used as an actual parameter:

Arithmetic expression	Scalar name
Boolean expression	Label name
Array name	Procedure name
Subscripted array name	Function name
P function	Subscripted array item name

Actual parameters should correspond to formal parameters as follows:

Actual Parameter	Formal Parameter
Subscripted array item name	Scalar
Scalar Expression	Scalar
P function	Based array
Subscripted array name	Array (not based array)
Other	Same as actual

Expressions are evaluated before subprogram execution and the addresses of temporary locations containing the resulting values are passed to the procedure. Other parameters are passed to the procedure as addresses. When a function name without a parameter list is an actual parameter, the formal parameter is assumed to be a function name. A parameter list is required if the function value is to be passed to the called procedure. When a based array is a formal parameter, only the address is passed as an actual parameter.

A single array item reference is considered an expression and evaluated accordingly. The resulting value of the array item is passed. Subscripted array names are passed as the address of the first word with that subscript, thus permitting the formal array to be offset from the actual array. For example, B[0] overlays A[5] with these declarations and references:

```
ARRAY A[1:10];;
P(A[5]);
PROC P(B); ARRAY B[0:5];
```

When a subprogram name or scalar name is enclosed in parentheses in an actual parameter list, the name is evaluated before the call and passed to the procedure as a temporary variable. For example, calling FUNNY with parameter J results in procedure execution and return to the calling program.

```
ITEM J;
    PROC FUNNY(FACE);
    BEGIN
    ITEM FACE;
    ITEM A, B;
    A=FACE;
    J=3;
    B=FACE;
EAR:    IF A EQ B THEN
            GOTO EAR;
    END

J=4;
FUNNY(J);
```

If FUNNY had been called with FUNNY((J)), however, an endless loop would exist.

Enclosing an actual scalar name in parentheses results in a protection of the value with which the procedure was called.

Actual arguments to subprograms are either call-by-reference or call-by-value. Call-by-reference means that the address of the argument is passed to the subprogram. The value of the argument can be changed by the called subprogram, unless the argument is a procedure, function, or label; the changed value is effective in both the called subprogram and the calling subprogram. Call-by-value means that the value of the argument is computed and stored in a temporary variable; then the address of the temporary variable is passed. Therefore, any change in value of the argument is only effective in the called subprogram.

The following types of arguments are passed by call-by-reference:

- Scalar items
- Arrays
- Labels
- Procedure names and function names with no argument list.

The following types of arguments are passed by call-by-value:

- Subscripted array items
- Constants
- Expressions other than item names
- LOC function references
- The form:

  (item name)

If a P function reference is passed as an actual argument, what is passed is the address of a temporary variable containing the pointer to the based array.

SYMPL does not check to see that corresponding formal and actual parameters are compatible in kind. However, the results can be invalid if they are not. The types of correspondence that produce valid results are shown in table 4-1.

TABLE 4-1. ACTUAL/FORMAL PARAMETER CORRESPONDENCE

Kinds of Formal Parameters	Valid Kinds of Actual Parameters
Scalar (call-by-reference)	Item name or call-by-value scalar
Scalar (call-by-value)	Expression item, or subscripted array item
Array	Array name (based or non-based)†
Based array	P function or address expression
Label	Label name
Procedure	Procedure name
Function	Function name

†A based array name used as an actual parameter requires an array as a formal parameter. The formal parameter must be a based array only if the called program must reposition the array for the calling program.

FUNCTIONS

A function is a subprogram whose name is associated with a specific value. It executes, and thereby determines the value of the name, when the function name appears in an expression. The function name value then is used in evaluation of the expression.

Two types of functions exist:

- Intrinsic functions that can be referenced without a corresponding declaration within the program.
- Programmer-supplied functions that must be declared before they can be referenced.

PROGRAMMER-SUPPLIED FUNCTIONS

A function must be declared by a function declaration. The declaration can appear anywhere in a program that any other declaration can appear. The flow of control during execution of a main program or subprogram is not affected by the nesting of any subprogram.

Unlike a procedure declaration, a function declaration or an XDEF declaration of the function name must appear before the function is referenced in an expression.

A function declaration consists of the following elements in this order:

- Function header
- Optional series of declarations
- Function body consisting of a single elementary or compound statement. The compound statement can include declarations, alternate entry points, or other elementary or compound statements.

The format of a function header is:

FUNC name (param, param, . . .) type;

name Identifier of 1 through 12 letters, digits, or $ that does not begin with a digit and does not duplicate a reserved word.

param Optional formal parameter used within the function for which an actual parameter is to be substituted at the time the function is referenced in an expression.

Any of the following can be specified by name:

Scalar	Array
Label	Based array
Procedure	Function

If a scalar name is specified, it can be enclosed in parentheses to indicate to the compiler that the actual parameter is to be passed to the function by value rather than by reference, as discussed for procedure parameters.

type Type of the function's result:

B	Boolean
I	Integer; default
U	Unsigned integer
R	Real
C(lgth)	Character of length lgth
S:stlist	Status

The function body must set the function name to a value before the end of the body statement or before any RETURN within the function. Otherwise, the value returned by the function is undefined. The function name must not appear on the right side of an assignment statement or as an actual argument within the function body, however, since functions cannot be recursive.

The function body must contain a declaration for each of its formal parameters except a label. Information pertinent to parameter declarations of procedures is also pertinent to functions.

INTRINSIC FUNCTIONS

Five intrinsic functions exist:

Function	Purpose
ABS	Obtain absolute value
B	Reference bit string from a field
C	Reference character string from a field
LOC	Reference address of data
P	Set pointer to based array

The B function and the C function are called bead functions since their purpose is to access consecutive bits or characters of a string.

ABS Function

The ABS function returns the absolute value of the function argument. Its format is:

ABS(exp)

exp — Expression whose absolute value is to be returned.

The type of the argument determines the type returned:

Argument Type	Type Returned
Real	Real
Integer or unsigned integer	Unsigned integer
Other	Same argument specified in call

B Function

The B function accesses one or more bits from a specified item, thereby creating an unsigned integer value. The format of the B function is:

B first,lgth iname

first — Arithmetic expression specifying the first bit of iname to be accessed, numbering from 0 on the left of iname. (Numbering is not from the left of the word in which iname appears.)

lgth — Arithmetic expression specifying the number of consecutive bits to be accessed. If omitted, 1 is assumed. A constant length of zero is not allowed. If an expression is used whose value at execution time is zero, the results are undefined.

The maximum value for lgth is affected by both the type of iname and its position in a memory word:

For type I, U, or R, lgth is limited to 60 and word boundaries cannot be crossed.

For type C, lgth is also limited to 60, but word boundaries can be crossed.

No check is made to verify that the bead size is less than or equal to the item size.

iname — Name of a scalar or subscripted array item from which bits are to be accessed.

When a B function is the object of a replacement statement, only the bits specified are affected by execution of the statement. The bead function is of the type unsigned integer; therefore, the right side of the replacement statement is converted to unsigned integer before the assignment takes place. The conversion rules are given in the description of the replacement statement in section 3.

When a B function is used other than as the object of a replacement statement, the specified bits are extracted and are right-justified and zero-filled to convert them to unsigned integer. The type of the item from which the bits were extracted does not affect the conversion process.

The number of bits that can be accessed is limited by the length of the item being referenced. The bead size is not checked to insure that it is less than or equal to the size of the item being accessed. The results of overaddressing may not be what the user intended.

An example of the B function use is the following, which counts the number of 0 bits in FLAGS:

```
ITEM COUNT;
ITEM FLAGS;
COUNT=0;
FOR I=0 STEP 1 UNTIL 59 DO
    IF B<I,1>FLAGS EQ 0
    THEN
    COUNT=COUNT + 1;
```

C Function

The C function accesses one or more 6-bit characters from a specified item, thereby creating character value. The format of the C function is:

C<first,lgth> iname

first — Arithmetic expression specifying the first character of iname to be accessed, numbering from 0 on the left of iname. (Numbering is not from the word in which iname appears.)

lgth — Arithmetic expression specifying the number of consecutive characters to be accessed. If omitted, 1 is assumed. A constant length of zero is not allowed. If an expression is used whose value at execution time is zero, the results are undefined.

The maximum value for lgth is affected by both the type of iname and its position within a memory word:

For type, I, U, or R, lgth is limited to 10 and word boundaries cannot be crossed.

No check is made to verify that the bead size is less than or equal to the item size.

For type C, lgth is limited to 240 and word boundaries can be crossed.

iname Name of a scalar or subscripted array item from which characters are to be accessed.

When a C function is the object of a replacement statement, only the characters specified are affected by execution of the statement. The bead function is of the type character; therefore, the right side of the replacement statement is converted to character before the assignment takes place. The conversion rules are given in the description of the replacement statement in section 3.

When a C function is used other than as the object of a replacement statement, the specified characters are extracted and converted to character. They are left-justified and blank-filled. The type of the item from which the characters were extracted does not affect the conversion process.

The number of characters that can be accessed is limited by the length of the item being referenced. The bead size is not checked to insure that it is less than or equal to the size of the item being accessed. The results of overaddressing may not be what the user intended.

An example of the use of the C function is:

```
ITEM BOAT C(10)="SERIAL0ABC";
    C<7,3>BOAT="XYZ";
```

Results in BOAT having a value SERIAL0XYZ.

Type conversion occurs as necessary when a C function is used in a replacement statement. A reference to an item containing an address cannot, for example, extract that address with a reference C<7,3>FIRST, since such a reference converts the integer address to type character.

LOC Function

The LOC function returns an address of a data structure during program execution. The format of the LOC function is:

LOC(argument)

argument Name of any of the following:

- Scalar
- Subscripted array item
- Procedure name
- Function name
- Label name
- Switch name
- Array name with optional subscript
- P function

In this context an array name with subscripts results in the address of the array entry with that subscript. For example:

```
P<BASE> = LOC(ARRAY1[I]);
```

An array item name returns the address of the word in which the item begins.

When a P function is used as an argument for LOC, the address of the pointer word to the based array is obtained.

The first word address of an array or based array is available via the LOC function.

P Function

The P function references the pointer variable for a based array. The format of the P function is:

P<barray>

barray Name of a based array.

By setting the P function to an integer value, the contents of memory at that location can be referenced. For example, the contents of RA+1 within a program field length can be accessed as RA 1 by:

```
BASED ARRAY MEM[99]; ITEM RA;
P<MEM>=0;
```

Other examples of the P function and based arrays are:

- Chain down a list structure and pass back the address of the desired list element in MATCH:

```
BASED ARRAY LIST  [0:0] S(1);
    BEGIN
    ITEM THING      I(0, 0,42);
    ITEM NEXTLIST   I(0, 42,18);
    END
ITEM LISTHEAD  I;
ITEM TARGET    I;
ITEM MATCH     I;
.
.
.
P<LIST> = LISTHEAD;
FOR MATCH=0
    WHILE MATCH EQ 0
DO
    BEGIN
    IF THING EQ TARGET
    THEN
        MATCH = P<LIST>;
    ELSE
        P<LIST> = NEXTLIST;
    END
```

- Preset a based array BLOCK to 0. In such an instance LOC(A) might have been passed to the procedure containing the following:

```
BASED ARRAY BLOCK [99];
    ITEM WORD I(0,0,60);
.
.
.
P<BLOCK> = LOC(A);
FOR I=0 STEP 1 UNTIL 99 DO
    WORD I  = 0;
```

ALTERNATE ENTRY POINTS

Alternate entry points can be declared for both procedures and functions with the ENTRY declaration.

The format of the ENTRY declaration for a procedure and a function, respectively, are:

```
ENTRY PROC name (param, param, . . . );

ENTRY FUNC name (param, param, . . . ) type;
```

parameters and type are optional, depending on the needs of the subprogram. The default type is integer.

The ENTRY declaration need not duplicate parameters associated with the subprogram name if such parameters are not used when the subprogram is called by the alternate entry point. Values for parameters not associated with the particular entry are undefined.

The type of an alternate entry point need not be the same type as the name of the principal entry point. At execution time, the value returned is of the same type as the entry point through which the function is entered.

A character item longer than ten characters cannot be returned by any function with alternate entry points. If the function has no alternate entry points, arbitrarily long character items can be returned. Only one entry point to a procedure can be active at one time.

An example of a procedure with an alternate entry point follows. The procedure searches an array until an element matching the actual parameter TARGET is found. If the procedure is entered through the principal entry point RESEARCH, the parameter START specifies where in the array to begin searching. If the procedure is entered through the alternate entry point SEARCH, searching starts from the beginning of the array.

```
    PROC RESEARCH ((START), MATCH, LIST,
      (TARGET));
    BEGIN
    ARRAY LIST [0:100] S(1);
        BEGIN
        ITEM THING      I(0, 0,42);
        END
    ITEM MATCH          I;
    ITEM TARGET         I;
    ITEM START          I;
    ITEM I              I;

    GOTO SRCH;

    ENTRY PROC SEARCH (MATCH, LIST, (TARGET));

    START = 0;
SRCH:
MATCH = 0;
    FOR I=START STEP 1
        WHILE MATCH EQ 0
        AND I LQ 100
    DO
        BEGIN
        IF THING EQ TARGET
        THEN
            MATCH = I;
        END

    END
    TERM
```

INTERPROGRAM COMMUNICATION

Three SYMPL declarations allow communication between subprograms that are compiled separately: COMMON, XREF, and XDEF. SYMPL truncates all names to be passed to the loader to the first seven characters, although all the characters retain significance internally. Such names must not begin with $.

COMMON DECLARATION

The COMMON declaration provides up to 509 blocks of storage that can be referenced by more than one subprogram. Variables in these blocks are assigned storage when the program is loaded.

The COMMON declaration must appear in the outermost level of a compilation.

The format of the COMMON declaration is:

```
COMMON name; datadec

or

COMMON name; BEGIN datadec datadec . . .END
```

name — Name of common block. If omitted, blank common is used.

datadec — Declaration for a scalar, array, or based array as described in section 2.

Preset values can be included in the data declaration if the common block is named. (Blank common cannot be initialized at load time.) The subprogram containing this declaration must be compiled with the P parameter of the compiler call in order for preset values to be compiled; alternatively, the CONTROL PRESET compiler-directing statement can be included in the subprogram when preset values are to be initialized in named common.

When a based array is declared to be in common, only the pointer to the array is passed in common.

Variables are stored in a common block in the order they are declared. Relative locations for all items should be the same in all subprograms referencing a particular block.

XDEF DECLARATION

The XDEF declaration generates an entry point such that the loader can link the specified names to those declared by XREF in separately compiled modules. It also allocates storage for any variables.

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. XDEF data can be preset.

The xdec for a procedure, function or label is:

```
PROC name;

FUNC name type;

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

XDEF declarations for procedure and function names enable nested procedures and functions to be referenced from separately compiled modules. The XDEF declaration must appear in a containing block before or after the procedure or function being specified in the XDEF declaration. The XDEF declaration cannot appear in the procedure or function that is being declared with the XDEF.

The name of the outermost procedure or function in a compilation unit cannot be declared with the XDEF statement. An entry point is automatically generated for the outermost procedure or function.

An example of the use of XDEF declarations is:

```
PROC A;
BEGIN #A#
  XDEF PROC B;
  PROC B;
  BEGIN #B#
  .
  .
  .
  END #B#
.
.
.
END #A#
TERM
PROC C;
BEGIN #C#
  XREF PROC B;
  XREF PROC A;
  .
  .
  .
  A;
  B;
END #C#
TERM
```

This example shows two separate compilation units. The first compilation unit contains two procedures, A and B. Procedure A is the outermost block and procedure B is nested in procedure A. The second compilation unit contains one procedure, C, which references both procedures in the first compilation unit.

Since procedure A is the outermost block of a compilation unit, an entry point is automatically generated for it. Therefore, it can be called from other compilation units, but it cannot be declared in an XDEF declaration.

The XDEF declaration that appears before procedure B causes an entry point to be generated for procedure B so that it can be called from other compilation units. The XDEF declaration can appear before or after procedure B, but not in procedure B.

Procedure C is in a separate compilation unit. It references both procedure A and procedure B. Procedures A and B must be declared in an XREF declaration.

An example of use of the XDEF and XREF declarations for data items is:

- 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 external in a separately compiled program which could be written in another language.

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

Four types of SYMPL statements are compiler-directing statements rather than executable statements. These are:

- $BEGIN and $END statements which allow enclosed source statements to be compiled only in the presence of the E parameter of the SYMPL control statement. They specify SYMPL debugging features.
- DEF statement which allows a mnemonic reference to a string of up to 240 characters. This is similar to a macro facility.
- CONTROL statement which affects the compilation, depending on words used in the statement.
- TERM statement which terminates compilation units.

$BEGIN/$END DEBUGGING FACILITY

Source language statements that are being compiled only for debugging purposes should be preceded by a $BEGIN statement and followed by an $END statement. This causes those statements to be compiled only when the E parameter appears on the SYMPL control statement. If the E parameter is not specified, statements between the $BEGIN and $END statements are not compiled. See section 6 for a description of the E parameter.

$BEGIN and $END are syntactically equivalent to BEGIN and END and can be used to delimit compound statements. However, such use can lead to unanticipated problems when the program is compiled without the E parameter: specifically, use of $BEGIN/$END to delimit the compound controlled statement of IF, FOR, ELSE, or PROC statement induces the compiler to use the succeeding statement as the controlled statement during a compilation without the E parameter, which probably is a logic error. Correct syntax should be maintained for a program whether or not it is compiled in debugging mode: BEGIN followed by $BEGIN is valid and useful.

A TERM statement must not appear between $BEGIN and $END. Furthermore, $END must not be produced by a DEF expansion.

DEF FACILITY

DEF is a compiler-directing statement that associates a character string with an identifier name. During compilation, each reference to the DEF identifier name is replaced by the character string of the DEF body; the resulting statement is then compiled in the normal manner. No calculation or evaluation occurs during replacement: only character string substitution occurs. The compiler does not print the expanded DEF.

DEF adds to the maintainability of a program by allowing constants or part of a statement to be referenced mnemonically. It also allows generation of in-line code for short functions. For example:

- Specify upper array bound for a table:

```
DEF TABSIZE #32#
ARRAY [TABSIZE];
```

- Define T and F corresponding to TRUE and FALSE

```
DEF T #TRUE#;
DEF F #FALSE#;
```

- Define a Boolean expression for use in IF or FOR statements:

```
DEF BOOL2 #A GR B AND B NQ 0#;
IF BOOL2 THEN . . .
```

The DEF identifier name can be defined to cause one of two types of character string substitutions:

- When the DEF statement does not include parameters, the DEF body is substituted in exactly the form in which it is declared.
- When the DEF statement includes parameters, the DEF body is modified according to parameters accompanying the DEF identifier name reference.

A DEF statement with parameters provides a simple macro capability for the SYMPL language.

The DEF statement can appear anywhere in a program that a declaration or executable statement can appear, except within a common block or a XDEF or XREF declaration. The declaration is subject to normal rules for declarations:

- The DEF statement must appear before the defined name is referenced.
- The DEF statement has no effect outside the subprogram in which it occurs.

A name defined by a DEF statement is defined from that point through the end of the subprogram. The name can be redefined through another DEF statement, which will generate a trivial diagnostic. A DEF cannot be undefined. No language facility exists for returning an identifier to the usage it had before its first DEF declaration.

BASIC DEF USAGE

In order to use the DEF facility, a DEF must be declared. The DEF can then be referenced, or called, in a SYMPL program.

DEF Name Declarations

The format of the DEF statement is:

DEF name (param,param, . . .) #character string#;

or

DEF name

name — Name by which the character string is subsequently to be referenced. Must be identifier of 1 through 12 letters, digits, or $ beginning with a letter.

param — Formal parameter which is to be replaced by an actual parameter when the DEF is expanded. Must be identifier of 1 through 12 letters, digits, or $ that does not begin with a digit. Parameters are optional.

The formal parameter names need only be unique within each parameter list. They can duplicate names defined elsewhere in the program.

character string — DEF body that is to replace references to the DEF name. Any character string can appear, including a null string. The character # in the string must be represented by ##. As many as 240 characters can appear in the string.

The body of a DEF is a character string. It has no meaning until it is expanded by a DEF reference. The DEF body can reference another DEF or data which is undefined at DEF declaration time. However, everything referenced in the DEF body must be defined when the DEF is referenced.

Although DEF declarations can be nested, they cannot be circular.

A legal nesting is:

```
DEF BOOL #A AND B#;
DEF A #C EQ 3#;

A reference to IF BOOL THEN X=1;
expands as IF C EQ 3 AND B THEN X=1;
```

An illegal circular definition is:

```
DEF TWO # BEGIN ONE END #;
DEF ONE # TWO #;
```

The formal parameters in a DEF declaration are recognized within the DEF body except that if the parameters appear within a comment or within a string delimited by quote marks, they are not modified during expansion. Otherwise, each occurrence of the parameter within the DEF body is replaced by an actual parameter accompanying the DEF name reference. The characters B, C, E, I, O, P, R, S, U, and X are not replaced by an actual parameter when they appear as a syntax-defining descriptor.

The programmer is responsible for the syntactic correctness of the statements that result from DEF substitution. For instance, this example is incorrect:

- DEF SIZE #1000#; DEF HALF #SIZE/2#; with reference ITEM A = HALF; produces a syntax error because substitution results in ITEM A = 1000/2; and items must be preset by constant, not expression, values.

DEF Name References

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

- The DEF name appears within a comment.
- The DEF name appears within a constant or string.
- The DEF name or the DEF parameter name appears as the identifier being defined by an ITEM, ARRAY or COMMON declaration.
- The DEF name is one of the following and is used in the applicable syntax-defining context:

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

  Array layout specifiers, P, S, A, U.

  Constant prefixes O, S, X.

  Intrinsic function B, C, P.

  Real number specifier 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. A null parameter is specified by consecutive commas.

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.

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.

ADVANCED DEF USAGE

The following paragraphs discuss three advanced topics that involve use of the DEF facility. These are:

- Parameters
- DEF expansion
- Comments

Parameters

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

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

All DEF and parameter substitution is strictly character string substitution.

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 Expansion

When a DEF reference is encountered by the SYMPL compiler, the DEF expansion is performed. In order to expand the DEF, the compiler repeats the following steps:

1. The actual parameters of the DEF reference are internally defined as DEFs and are substituted for the formal parameters.

   For example, the sequence:

   ```
   DEF A (B) #B + C#;
   Y = A(2);
   ```

   generates the equivalent of the declaration:

   ```
   DEF B #2#;
   ```

2. The DEF body is scanned from the left and all DEF names encountered are expanded.

These two steps are repeated until no DEF names remain in the sequence.

Because actual parameters are converted into DEFs, this procedure can produce results other than those the user intends. For example:

```
DEF TWO (X,Y)   #X = 1; Y = 2;#;
DEF ONE (Z)     #TWO(Z)#;
```

The DEF name reference ONE(#A,B#) is expanded in the following steps:

1. The actual parameter #A,B# is declared as a DEF(Z) (the # characters bracketing A, B are discarded):

   ```
   DEF Z #A,B#;
   ```

2. The text for ONE replaces the call to ONE:

   ONE(A#,B#) becomes TWO(Z)

3. The string is searched again from the left. This time, the first DEF name encountered is TWO.

4. The actual parameters for the reference to TWO are declared as DEF bodies equivalent to the formal parameter. Since Z has not been expanded yet, there is only one actual parameter. It replaces X and a null string replaces Y:

```
DEF X#Z#;
DEF Y ##;
```

5. The text for TWO replaces the call to TWO:

```
TWO(Z) becomes X = 1; Y =2;
```

6. The string is searched again from the left; this time, the first DEF name encountered is X. The DEF name is replaced by the DEF body. This process is repeated; the remaining DEF names encountered are X, Z, and Y, in that order. The successive stages of the expansion are as follows:

```
X = 1; Y = 2;
Z = 1; Y = 2;
A,B = 1; Y = 2;
A,B = 1; = 2;
```

The final result is not what the user intended since it is not valid syntax.

The body of a DEF with parameters and its actual parameters cannot contain the character ↓. This character is allowed in DEFs which have no parameters.

Comments

Comments are allowed within the parentheses delimiting any parameters. Comments also are allowed within the DEF body as long as they are delimited by ##, since the DEF body itself is delimited by #. Comments are not allowed between the DEF name and the left parenthesis and they are not allowed between the right parenthesis and the # which delimits the character string. Since the DEF body is retained in memory during compilation, excessive use of unneeded comments within the body causes larger compilation field length.

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, OBJLIST.
- Conditional compilation control words are IFxx, FI, ENDIF.
- Compilation option selections are PACK, PRESET, FTNCALL.
- FOR statement loop specifications are FASTLOOP, SLOWLOOP.
- Memory 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 compilation unit. 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
OBJLIST	List object code

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

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

- When the relationship defined in the CONTROL IFxx statement tests TRUE, the following source statements are compiled.
- When the relationship defined in the CONTROL IFxx statement tests FALSE, the following source statements are skipped through a matching CONTROL FI or CONTROL ENDIF statement.

The CONTROL IFxx statement is particularly useful because the constants to be tested by the relationship can be DEF names or parameters.

The form of the CONTROL IFxx statement for conditional compilation is:

CONTROL IFxx const1, const2;

IFxx — Relationship of const1 and const2 that is to be tested:

IFEQ	Const1 equal to const2
IFLS	Const1 less than const2
IFLQ	Const1 less than or equal to const2
IFGR	Const1 greater than const2
IFGQ	Const1 greater than or equal to const2
IFNQ	Const1 not equal const2

const1, const2 — Constants to be tested by the condition-word relationship. If const2 and its preceding comma is omitted, 0 is assumed.

Both constants must be the same type; they can be type integer, real, Boolean, or character.

Character constants can be compared only by IFEQ and IFNQ. Leading and trailing blanks are significant for the comparison such that A is not equal to A followed by a blank. Character strings may be compared only for equality and inequality.

Conditional source statements must be bracketed between the CONTROL IFxx statement defining the relationship to be tested and either one of the following CONTROL statements:

CONTROL FI;
CONTROL ENDIF;

Conditional statements can be nested.

If conditional statements are suppressed, syntax and semantic checks are not performed; DEF names are not expanded; and comment strings are not examined for CONTROL FI or ENDIF. In this situation, a semicolon does not terminate a comment string.

Since DEF is not expanded when conditional statements are suppressed, DEF cannot be used to generate the CONTROL ENDIF for an outer CONTROL IFxx statement.

COMPILATION OPTION SELECTION

These forms of the CONTROL statement can affect the options under which a module is compiled. The format is:

CONTROL control-word;

control-word — One of the following:

PACK	Select D option such that switches are packed two entries to a 60-bit word. This option requires less memory but more execution time.
PRESET	Select P option such that items declared in named common blocks are initialized. If the P option is not selected by the compiler call or the CONTROL statement, presets for common block items are ignored.
FTNCALL	Select F option such that procedure calling sequences are compatible with FORTRAN; that is, they have a word of all zeros terminating the parameter list.

The appearance of a CONTROL PACK or CONTROL FTNCALL statement anywhere within a module affects the entire module.

CONTROL PRESET must appear before the common block declarations.

FOR LOOP CONTROL

The code generated by FOR loops can be controlled by a CONTROL statement in the format:

CONTROL looptype;

looptype — Type of loop to be generated:

FASTLOOP	Test and branch within loop, so loop must execute at least once. The FOR statement containing such a loop is restricted in several ways, as discussed in section 3.
SLOWLOOP	Test and branch occurs at beginning of loop, so loop need not execute at all. SLOWLOOP is assumed in the absence of FASTLOOP.

CONTROL FASTLOOP and CONTROL SLOWLOOP can appear anywhere a statement can appear. The statement remains in effect until another CONTROL statement for loop control is encountered, whether that other statement is in the same subprogram or not.

Loop control statements can be nested. A FOR statement generates a fastloop or slowloop depending on where it appears: a nested statement affects the inner loop, but not the loop in which it appears. For example, loop I is a fastloop and loop L is a slowloop when the following appears:

```
CONTROL FASTLOOP;
FOR I=1 STEP J UNTIL K DO
    BEGIN
    CONTROL SLOWLOOP;
    FOR L=M STEP 1 UNTIL END DO
        X=X+L;
    END #I LOOP#
```

MEMORY RESIDENCE SELECTION

Common blocks and based arrays can be allocated in either of the two types of memory. The format of the CONTROL statement to select residence is:

CONTROL LEVELn name, name, . . .;

n — Memory in which specified common blocks or based arrays are to reside. The meaning of the level indicated is affected by the hardware available.

For CYBER 70 Model 76 and 7600 and CYBER 170 Model 176 computers:

- 1 — Small central memory (SCM) residence.
- 2 — Large central memory (LCM) residence accessed directly.
- 3 — Large central memory (LCM) residence accessed by block transfer to SCM. Items at this level can only be passed as parameters.

For CYBER 70 Models 71, 72, 73, and 74, CYBER 170, Models 171, 172, 173, 174, 175, 720, 730, 750, 760, and 6000 series systems:

- 1,2 — Central memory residence
- 3 — Extended core storage residence accessed by block transfer to central memory. Items at this level can only be passed as parameters.

name — Name of a common block or based array. If name is the name of a level 3 scalar or non-based array, then name or LOC(name) can appear as an actual parameter, if name is the name of a level 3 based array, then P<name> or LOC(P<name>) can appear as an actual parameter. Level 3 names cannot be used in any other context.

If the name of a common block is declared in a CONTROL LEVEL statement, all items in the block are at the specified level. If a based array is declared within a common block, its pointer is at the specified level of the common block; the array itself can be at a different level.

If the name of a based array is declared in a CONTROL LEVEL statement, the array is assumed by the compiler to be at the specified level. It is the user's responsibility to ensure that the array is in fact at the specified level; otherwise, the results are undefined. The pointer to the array need not be at the same level.

ATTRIBUTES OF VARIABLES SPECIFICATION

The SYMPL compiler attempts to produce efficient executable code. Because the compiler cannot always determine the precise use of a variable throughout a program, it must forego many efficiencies that might result in incorrect code in unusual circumstances. 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 conflicting, 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 explanation 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 entry word in a single array is accessed by more than one entry in the same array using different entry positions. See explanation 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 entry word in a single array is not accessed by more than one entry in the same array using different entry positions. INERT is the opposite of REACTIVE.

var — Variable with the attribute specified. It can be an array name, but not an array item.

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

If no CONTROL statements specifying variable attributes appear in a module, the compiler assumes that variables are used in a non-optimal way in the module. The treatment of variable references in this case is not precisely the same as when any of the variable attributes are specified. Specifically, the compiler assumes that any of the following can occur:

- Formal parameters can destroy each other.
- Formal parameters can destroy global variables and vice versa. A based array can destroy any other based or fixed array, but a fixed array does not destroy any other array.
- All arrays are considered reactive.
- A procedure call can destroy all common, XDEF and XREF variables.
- Variables do not interfere with each other in any other way.

If any CONTROL statement specifying a variable attribute appears in a module, then compilation of the module proceeds as though the module were preceded by a CONTROL REACTIVE statement and a CONTROL DISJOINT statement. The specified statements take effect when they are encountered in the source program. Use of the CONTROL statement to classify variables is recommended because future versions of the compiler might require it.

Overlapping

Overlapping takes place when a single word in memory is referenced as more than one entity in a conflicting way. Overlapping takes place when all of the following are true:

- A single word in memory is referenced by more than one name. The two names could be two scalars, an array and a scalar, or two arrays.
- All the references occur in the same procedure.
- At least one of the references is a store.

If the compiler knows that two variables are disjoint, it can perform optimizations such as elimination of redundant code. For example, in the following sequence:

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

If A and B are both declared to be disjoint, the same value can be assigned to both A and Y, thus eliminating the need to fetch the value of A before assigning it to Y. On the other hand, if procedure P is called by the following call:

```
P (V,V);
```

then A and B are different names for the same memory location. In this case, the value of A must be fetched before the store to Y, since the value is changed by the statement B = 4. If either A or B were declared disjoint, incorrect code would result; both must be declared OVERLAP.

There are three cases in which overlapping references can occur. These are:

- When a procedure with two or more formal parameters is called with the same variable appearing twice as an actual parameter. The example above shows this case.
- When a based array points to a fixed array that is used in a conflicting way in the same code. Example:

```
ARRAY A;
ITEM AI;
BASED ARRAY B;
ITEM BI;
P<B> = LOC(A);
X = AI[2];
BI[2] = 3;
Y = AI[2];
```

  Since A is based on B, the value of AI 2 must be fetched twice, since it is changed by the statement BI 2 = 3.

- When a procedure uses a global variable, and the procedure is called with that variable as an actual parameter. Example:

```
ITEM A. . .
.
.
.
B (A);
.
.
.
PROC B(C);
ITEM C, D I;
BEGIN #B#
C = 2;
A = 4;
D = C;
END #B#
```

In this example, the procedure B is called with A as an actual parameter. Since A is assigned a value in the statement A = 4, the value of C is also changed. Therefore, when the statement D = C is executed, the value of C must be fetched; D cannot be simply set to 2. When overlapping is used, all affected variables must be declared OVERLAP. In the last example, both A and C must be declared OVERLAP.

Reactive Arrays

An array is reactive if the same word in the array is referred to by more than one item name in a conflicting way. For an array to be reactive, all of the following must be true:

- A single word in the array must be referenced by more than one item declared in the same array.
- The items referencing the word must have different entry positions.
- All the references must be in one procedure.
- At least one of the references must be a store.

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. This is useful when generating capsules for use by the Fast Dynamic Loader. See the Loader Reference Manual for more information.

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. There is no way for an executing program to tell if the external has actually been loaded. If it has not been loaded an arithmetic mode error results.

When a weak external is satisfied, it is linked as if it were a normal external. If it is not satisfied, 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. To complete FORTRAN compatibility, the F parameter of the SYMPL compiler call must also be specified. Code generated by a SYMPL calling program is not compatible with FORTRAN 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, modulo 4096.

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

SYMPL TEXTS

If the same declarations apply to data in more than one SYMPL compilation unit, the declarations can be compiled once as a text and referenced from each of the units. Thus, instead of compiling the declarations once for each compilation unit using them, the user can compile them one time only. Another advantage of this feature is that it ensures that data that is identical in different modules is declared in exactly the same way in each module. If there are many such declarations, placing them in texts saves compilation time.

SYMPL texts are created by the CONTROL STEXT statement and used by the USETEXT statement.

Text Creation

The format of the CONTROL statement specifying text creation is:

CONTROL STEXT;

The statement can only appear in a program or a procedure with no formal parameters. It cannot appear in a function. Its appearance within a compilation unit results in creation of a text including declarations occurring anywhere within the unit. The name of the text is the program or procedure name, truncated to seven characters if necessary.

The compiled text is written as an overlay to the file specified by the B option of the SYMPL control statement. Any executable statements occurring in the compilation unit are not compiled. The presence of executable code suppresses text generation. Nevertheless, the program or procedure must be syntactically complete; that is, it must include a null executable statement.

Embedded procedures and functions are not allowed in a text-generating compilation unit, nor are XDEF declarations. LABEL and SWITCH declarations are diagnosed but are not included in the text record. Presets included in declarations have no effect.

The CONTROL statement options FASTLOOP, SLOWLOOP, PRESET, PACK, FTNCALL, and TRACEBACK have no effect on a text module. Variable attribute specifications (CONTROL REACTIVE, INERT, OVERLAP, DISJOINT) apply only to the text module, and not to other modules in the compilation. Entities that can be validly declared in a text module include scalars, arrays, based arrays, array items, status lists, and common blocks. XREF declarations are valid, but XDEF declarations are not allowed. For a status item, the associated status declaration must occur before the item declaration or else in a SYMPL text used in compilation of the current text. All references to DEF names in a text module result in DEF body replacement at text generation time. Redefinition in a module using the text module has no effect. For example:

```
DEF UP #12#;
ARRAY A[1:UP];
```

Array A is always length 12 even if UP is redefined in a module that uses the text module being defined.

Presets are ignored in a text unit. Previously compiled texts can be used to generate new text overlays. See the discussion of USETEXT.

Text Usage

To specify SYMPL texts to be used in compiling a module, the following statement is used:

USETEXT $text_1, \ldots, text_n$

text Name of a SYMPL text on a file specified by the Y control statement option or a library specified by the Z control statement option (section 6).

The USETEXT statement must begin in column 1 and cannot be continued onto another source line. Multiple USETEXT statements are allowed. One or more blanks are required between USETEXT and the first text name, and commands are required between text names. The USETEXT statement must appear before the first statement of the compilation unit (PRGM, PROC, or FUNC statement) but after any OVERLAY statement. A module can contain more than one USETEXT statement; a maximum of 64 texts can be specified for any module. If no USETEXT directive appears, no SYMPL text is available during compilation.

When the module containing the USETEXT statement is compiled, the declarations in the text module behave as though they were physically present immediately after the PRGM, PROC, or FUNC statement.

A text module can be compiled using a previously compiled text module (that is, both the USETEXT statement and the CONTROL STEXT statement can appear in the same module). In this case, however, only the declarations from the source input module, not those from the previously compiled text module, appear in the new SYMPL text.

When a text contains a status item associated with a status list in another text, the name of the text containing the status list must precede the name of the text containing the status item in the USETEXT statements.

If the compiler encounters portions of the same common block in more than one SYMPL text, a fatal diagnostic is issued.

For a SYMPL text on a library (specified by the Z control statement option), the compiler obtains 20000_8 words of central memory before calling the loader to load the overlay. After the load, any unused field length is relinquished. If 20000_8 words is not sufficient, the user must increase the field length through an RFL control statement. SYMPL does not reduce field length below the amount specified by the RFL control statement.

TERM STATEMENT

The TERM statement signals the end of a compilation unit. It must be the last statement of a program (or subprogram being compiled separately).

The format of the TERM statement is:

TERM

No semicolon follows TERM.

Once the compiler encounters TERM, all further statements on the card are skipped.

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 image encountered at the beginning of a compilation unit contains the characters OVERLAY in columns 1 through 7, followed by (1,1, origin), the module is treated as if an LCC OVERLAY statement appeared in a COMPASS program.

The name of 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.

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.

DB CYBER INTERACTIVE DEBUG

omitted	Do not generate code to allow CYBER Interactive Debug interface, unless job is in debug mode (DEBUG, ON control statement has been executed).
DB	Generate code to allow CYBER Interactive Debug interface.
DB=0	Do not generate code to allow CYBER Interactive Debug interface, even if job is in debug mode.

Selection of this option automatically selects the W control statement option and sets CONTROL TRACEBACK for the entire 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.

EL ERROR LEVEL

All errors of the specified level, and errors of higher levels, are listed on the output listing. The levels are listed here in ascending order.

omitted	If ET is omitted or ET=W, F, or C, list diagnostics of level W and higher. If ET=D, list diagnostics of level D and higher. IF ET=T, list diagnostics of level T and higher.
EL	Same as EL=F
EL=D	List system-dependent diagnostics and T, W, F, and C diagnostics. System-dependent diagnostics are produced for usages that, while syntactically correct, might not produce correct results on all machines, or might not be supported by future versions of the compiler.
EL=T	List trivial diagnostics and W, F, and C diagnostics. Trivial diagnostics result from usages that are syntactically correct but questionable.

EL=W — List warning diagnostics and F and C diagnostics. Warning diagnostics result from usages that are syntactically incorrect, but from which the compiler has been able to recover by making an assumption about what was intended.

EL=F — List all fatal diagnostics and C diagnostics. Fatal diagnostics result when the compiler cannot resolve a syntactic or semantic error. No code is generated for the statement causing the error; compilation resumes with the next statement.

EL=C — List only catastrophic errors. Catastrophic errors are those which cause immediate termination of compilation, and an abort of the job step.

ET ERROR TERMINATION

If the ET option indicates error termination for a certain level of error, and an error of that level or higher level occurs, the job step aborts to an EXIT(S) control statement (under NOS/BE) or an EXIT control statement (under NOS) when compilation finishes. For an explanation of the error levels, see the EL parameter. The ET=T option replaces the A option.

omitted — Same as ET=C.

ET — Same as ET=F.

ET=D — Abort job step if errors of level D or higher occur.

ET=T — Abort job step if errors of level T or higher occur.

ET=W — Abort job step if errors of level W or higher occur.

ET=F — Abort job step if errors of level F or higher occur.

EF=C — Abort job step if errors of level C occur.

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 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 code.

K — Generate an RJ to the points-not-tested interface routine SYMCK$ after every label and conditional jump. This will enable the user to find all paths in the executable code and determine which of the paths are exercised. Also, generate traceback code. The user must supply the routine SYMCK$.

L LISTING FILE

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

omitted — If O, R, or X are not specified, write source statement listing and diagnostics to file OUTPUT. If O, R, or X are specified, suppress source listing.

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.

For SYMPL text generation, N is selected by default.

When a SYMPL text is used, unreferenced variables declared in the text are not listed, regardless of the N option.

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. Because the compiler might reorder code for optimization, the user should specify a range of lines slightly larger than that actually desired.

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. A maximum of seven libraries can be specified.

T SYNTAX CHECK

omitted — Check syntax and generate binary code.

T — Check syntax, but do not generate binary code. Nullifies any B, O, R, or X parameters.

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 SYMPL TEXT INPUT FILE

omitted — Do not read any SYMPL text from a file.

Y=0 — Do not read any SYMPL text from a file.

Y — Read SYMPL texts named in the USETEXT statement from the file SYMTEXT.

Y=lfn — Read SYMPL texts named in the USETEXT statement from the file lfn.

The Y parameter can be specified up to seven times. While processing the Y parameter, the compiler reads the specified file and makes a table of all texts encountered and their position on the file. While processing the USETEXT statement, texts are accessed randomly.

Files specified by the USETEXT statement are searched for first among the files specified by Y parameters, from left to right, and then in the library specified by the Z parameter. The files specified by the Y parameter must be sequential files residing on random access devices. A maximum of 64 texts can reside on all the files specified by Y parameters.

Z SYMPL TEXT INPUT LIBRARY

omitted	Do not read any SYMPL text from a library.
Z=0	Do not read any SYMPL text from a library.
Z	Read SYMPL texts named in the USETEXT statement from the job's current global library set.
Z=lib	Read SYMPL texts named in the USETEXT statement from the library named lib. The library can be either a user library or a system library in the job's current global library set.

The Z parameter cannot be specified more than once. The search order for SYMPL texts is explained under the Y parameter. The Z parameter is not valid under NOS 1.

OUTPUT LISTINGS

Figure 6-1 shows a SYMPL procedure that adds a new entry to a linked list. It is a separate compilation and is linked into the proper context by the loader.

```
FUNC ADDITEM (INVLIST, (LISTPTR), (PARTNUM), DESCR);
BEGIN  #ADDITEM#
#-------------------------------------------------------------------------#
#                                                                         #
#      F U N C    A D D I T E M                                           #
#                                                                         #
#      PURPOSE -- ADD A NEW PART TO THE INVENTORY LIST.  THE PART IS      #
#                 ADDED IN PART-NUMBER ORDER, AND THE BASIC               #
#                 INFORMATION FOR THE PART IS FILLED IN THE NEW ENTRY.    #
#      INPUT   -- THE PARAMETERS GIVE THE ADDRESS OF THE INVENTORY LIST,  #
#                 THE START OF THE NUMERIC LIST, AND A DESCRIPTION OF     #
#                 THE NEW ENTRY.                                          #
#      OUTPUT  -- THE INVENTORY LIST IS UPDATED WITH A NEW ENTRY.         #
#                 FUNCTION RESULT IS A POINTER TO THE NEW ENTRY.          #
#                                                                         #
#-------------------------------------------------------------------------#

      ITEM DESCR         C(20);      # DESCRIPTION OF THE NEW PART       #
      ITEM INDX          I;          # ARRAY POINTER, USED FOR SEARCH    #
      ITEM LISTPTR       I;          # POINTER TO START OF INVENTORY LIST#
      ITEM NEWENTRY      I;          # POINTER TO NEW LIST ENTRY         #
      ITEM PARTNUM       I;          # PART-NUMBER OF NEW PART           #

      ARRAY INVLIST [0:4000] S(3);  # INVENTORY LIST ARRAY              #
        BEGIN
        ITEM INV$PARTNUM   I( 0, 0,24);  # INVENTORY PART-NUMBER        #
        ITEM INV$LINK      I( 0,42,18);  # PTR TO NEXT SEQUENTIAL ENTRY #
        ITEM INV$DESCR     C( 1, 0,20);  # DESCRIPTION OF THE PART      #
        END
      XREF FUNC GETSPACE;           # RETURNS WORDS IN LISTSPACE        #


      FOR INDX = LISTPTR            # START SEARCHING THE LIST          #
        WHILE INV$LINK[INDX] LS PARTNUM  # FIND NUMERIC POSITION        #
        AND INV$LINK[INDX] NQ 0     # ...OR END OF LIST                 #
      DO
        BEGIN
        INDX = INV$LINK[INDX];      # GET NEXT SEQUENTIAL ENTRY         #
        END

      NEWENTRY = GETSPACE(4);       # GET 4 WORDS FOR NEW LIST ENTRY    #
      INV$LINK[NEWENTRY] = INV$LINK[INDX];  # LINK NEW ENTRY IN LIST    #
      INV$LINK[INDX] = NEWENTRY;    # LINK OLD ENTRY TO NEW ONE         #
      INV$PARTNUM[NEWENTRY] = PARTNUM;  # NOW FILL IN NEW INFORMATION   #
      INV$DESCR[NEWENTRY] = DESCR;

      ADDITEM = NEWENTRY;           # RETURN ADDR OF THE NEW ENTRY      #

END  #ADDITEM#
TERM
```

Figure 6-1. Sample Source Program

A job deck for syntax analysis for the function would appear as:

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

Output from a compilation normally includes the source statement listing and a diagnostic summary. Only the diagnostics specified with the EL parameter on the SYMPL control statement are printed. The compiler also issues a list of page numbers that contain diagnostic messages. Only diagnostics that are actually printed are included in this list.

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 of two or more parts. Both must be examined since references to items can appear in both parts.

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 subprogram ADDITEM. 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.ARRAY	Based array
ARRAY	Array
PROGRM	Program

③ M — Mode of data representation

B	Boolean
C	Character
I	Integer
P	Parallel (arrays only)
R	Real
S	Status (Serial if type is array or based array)
U	Unsigned integer
X	External (if type is PROC)
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 is ARYITM, LOC refers to first occurrence of item. If followed by an X, the name is an external. If followed by a Y, the name is a weak external.

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

⑥ NUM — Number of bits; if MODE is 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 ADDITEM. 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.

```
ADDITEM   FUNCTION          * STORAGE MAP *          SYMPL 1.4 (79267)   79/10/24. 09.41.14.   PAGE 2
   ①      ②    ③ ④          ⑤  ⑥
NAME·C(10) TYPE   M LOC      FBIT NUM   NAME:C(10) TYPE   M LOC      FBIT NUM   NAME:C(10) TYPE   M LOC      FBIT NUM

ADDITEM    FUNC   I     11     0  60   DESCR      ITEM   C     0     0  20   GETSPACE   FUNC   I     0X    0  60
INDX       ITEM   I      1     0  60   INVLIST    ARRAY  S     5               INV$DESCR  ARYITM C     1     0  20
INV$LINK   ARYITM I      0    42  18   INV$PARTNU ARYITM I     0     0  24   LISTPTR    ITEM   I     2     0  60
NEWENTRY   ITEM   I      3     0  60   PARTNUM    ITEM   I     4     0  60
```

Figure 6-2. Storage Map

```
   ADDITEM   FUNCTION              * CROSS REFERENCE *          SYMPL 1.4 (79267)   79/10/24. 09.41.14.     PAGE 3
      ①       ②      ③   ④          ⑤          ⑥
NAME:C(10) TYPE     M DEFINED     SCOPE      SET/USED/ATTRIBUTE - *=USED,A=ATTRIBUTE

ADDITEM    FUNC     I       1     (GLOBAL)      48
DESCR      ITEM     C      19     ADDITEM       46*
GETSPACE   FUNC     I      31 X   ADDITEM       42*
INDX       ITEM     I      20     ADDITEM       35*   36*   39*   43*   44*   34    39
INV$DESCR  ARYITM   C      29     ADDITEM       46
INV$LINK   ARYITM   I      28     ADDITEM       35*   36*   39*   43*   43    44
INV$PARTNU ARYITM   I      27     ADDITEM       45
LISTPTR    ITEM     I      21     ADDITEM       34*
NEWENTRY   ITEM     I      22     ADDITEM       43*   44*   45*   46*   48*   42
PARTNUM    ITEM     I      23     ADDITEM       35*   45*
```

Figure 6-3. Cross-Reference Map

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.ARRAY	Based array
STSCON	Status constant
DEFINE	DEF
STSLST	Status list
PROGRM	Program
ARRAY	Array

③ M — Mode of data representation

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

④ DEFINED — Line number in source listing or name of SYMPL text where declaration is defined; if followed by C, declaration is in common block. If followed by X, the name is an external. If followed by Y, it is a weak external.

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

⑥ SET/USED/ATTRIBUTE — Source listing line numbers of references to NAME, * indicates use as other than left side of the replacement statement. An A indicates appearance in attribute specification, such as XDEF.

STANDARD CHARACTER SETS

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 is specified when the operating system is installed. The standard character sets are shown in table A-1.

Depending on another installation option, the system assumes an input deck has been punched either in 026 or 029 mode, regardless of the character set in use. 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. 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.

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 alternate mode remains in effect throughout the job unless reset by another alternate mode specification.

Graphic character representation on a terminal or printer depends on the installation character set and the device type. CDC graphic characters in table A-1 are applicable to BCD terminals. ASCII subset graphic characters are applicable to ASCII-CRT and ASCII-TTY terminals.

TABLE A-1. STANDARD CHARACTER SETS

SYMPL	Display Code (octal)	CDC			ASCII		
		Graphic	Hollerith Punch (026)	External BCD Code	Graphic Subset	Punch (029)	Code (octal)
: (colon)	00[†]	: (colon)[††]	8-2	00	: (colon)[††]	8-2	072
A	01	A	12-1	61	A	12-1	101
B	02	B	12-2	62	B	12-2	102
C	03	C	12-3	63	C	12-3	103
D	04	D	12-4	64	D	12-4	104
E	05	E	12-5	65	E	12-5	105
F	06	F	12-6	66	F	12-6	106
G	07	G	12-7	67	G	12-7	107
H	10	H	12-8	70	H	12-8	110
I	11	I	12-9	71	I	12-9	111
J	12	J	11-1	41	J	11-1	112
K	13	K	11-2	42	K	11-2	113
L	14	L	11-3	43	L	11-3	114
M	15	M	11-4	44	M	11-4	115
N	16	N	11-5	45	N	11-5	116
O	17	O	11-6	46	O	11-6	117
P	20	P	11-7	47	P	11-7	120
Q	21	Q	11-8	50	Q	11-8	121
R	22	R	11-9	51	R	11-9	122
S	23	S	0-2	22	S	0-2	123
T	24	T	0-3	23	T	0-3	124
U	25	U	0-4	24	U	0-4	125
V	26	V	0-5	25	V	0-5	126
W	27	W	0-6	26	W	0-6	127
X	30	X	0-7	27	X	0-7	130
Y	31	Y	0-8	30	Y	0-8	131
Z	32	Z	0-9	31	Z	0-9	132
0	33	0	0	12	0	0	060
1	34	1	1	01	1	1	061
2	35	2	2	02	2	2	062
3	36	3	3	03	3	3	063
4	37	4	4	04	4	4	064
5	40	5	5	05	5	5	065
6	41	6	6	06	6	6	066
7	42	7	7	07	7	7	067
8	43	8	8	10	8	8	070
9	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
blank	55	blank	no punch	20	blank	no punch	040
, (comma)	56	, (comma)	0-8-3	33	, (comma)	0-8-3	054
. (period)	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
" (quote)	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
; (semicolon)	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 by some driver software.

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. A list of page numbers that contain diagnostic messages is printed at the end of the error summary.

The compiler aborts when compilation cannot be continued for one of the following reasons:

- Error in the SYMPL control statement. The following dayfile message is issued:

  PARAMETER n IN ERROR

- A large number of syntax errors has been detected.
- A user syntax error occurs that is discovered too late for the compiler to recover, for example, a zero value length for a B function.
- Internal compiler error.

When the compiler aborts, a C level error message in the following form is issued to the file specified by the L control statement parameter:

SYMPL COMPILER ERROR nnnn -- error text

where nnnn is the error number. Compiler abort messages start at 800.

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- cccccccccc COMPILED cp secs
-SYMPL- ERRORS IN TEXT GENERATION
-SYMPL- SYMPL TEXT NOT WRITTEN
-SYMPL- SYMPL TEXT NOT FOUND
-SYMPL- MORE THAN 7 Y PARAMETERS
-SYMPL- MORE THAN 1 Z PARAMETER
-SYMPL- MORE THAN 64 SYMPL TEXTS ON Y FILES
-SYMPL- MORE THAN 64 TEXTS ON USE TEXT
-SYMPL- Y FILE NOT ON RANDOM DEVICE
-SYMPL- SOURCE ERRORS--ABORT REQUESTED

Table B-1 lists the message number and text of the compilation diagnostics. The error levels listed with the diagnostics are explained under the EL parameter in section 6.

TABLE B-1. COMPILER ERROR MESSAGES

Error Number	Error Level	Message	Significance	Action
1	W	LONG IDENTIFIER - FIRST 12 CHARACTERS USED	Identifier was truncated to 12 characters. It may duplicate another identifier.	Verify that the identifier is unique within the first 12 characters.
2	F	SYMPL TEXT CONTAINS EMBEDDED -PROC- OR -FUNC-	Self-explanatory.	Correct error and recompile.
3	F	UNDECLARED IDENTIFIER USE DELETED	An identifier was referenced but not declared.	Check for misspelled identifier or supply declaration for the diagnosed identifier. Check for errors in declaration if error 21 was issued.
4	F	ILLEGAL OCTAL OR HEX CONSTANT	Octal constants can contain digits 0-7; hexadecimal constants can contain digits 0-9, A-F.	Correct constant and recompile.
5	F	-TERM- MISSING	A TERM statement is required to terminate compilation. It must appear after the final END statement.	Supply TERM statement and recompile.

TABLE B-1. COMPILER ERROR MESSAGES (Contd)

Error Number	Error Level	Message	Significance	Action
6	F	BAD STATUS CONSTANT USE - 0 USED	Self-explanatory.	Correct error and recompile.
7	F	BAD NESTING OF PARENTHESES OR BRACKETS	Self-explanatory.	Correct error and recompile.
8	W	ILLEGAL CHARACTER IN INPUT - IGNORED	A character was used that is not part of the SYMPL character set.	Replace illegal character and recompile.
9	W	CHARACTER STRING LONGER THAN 240 BYTES - 240 USED	Character string was truncated to 240 characters.	Verify that the truncation does not affect the logic of the program.
10	F	-PROC- OR -FUNC- NAME CANNOT BE A RESERVED WORD	A reserved word was used as a procedure or function name.	Change the procedure or function name and recompile.
11	F	SYMPL TEXT CONTAINS XDEF	Self-explanatory.	Correct error and recompile.
12	F	ILLEGAL ARRAY IDENTIFIER USE DELETED	Self-explanatory.	Change array identifier and recompile.
13	F	ILLEGAL STATUS LIST IDENTIFIER USE DELETED	Self-explanatory.	Change status list identifier and recompile.
14	F	ILLEGAL COMMON IDENTIFIER USE DELETED	Self-explanatory.	Change common identifier and recompile.
15	F	SEMICOLON MISSING AFTER ARRAY DECLARATION	Self-explanatory.	Supply semicolon and recompile.
16	F	CRUD AT START OF STATEMENT DELETED	The extraneous characters that preceded the statement were removed.	Check the previous statement for syntax errors. Correct errors and recompile.
17	F	ILLEGAL KEYWORD USE DELETED	A reserved word was used improperly.	Check for programmer-defined name that duplicates a reserved word. Correct error and recompile.
18	F	ARRAY ITEM DECLARATION LIST LACKS -END-	An array item list must start with a BEGIN statement and terminate with an END statement.	Supply the missing END statement and recompile.
19	W	DUPLICATE DECLARATION OVERRIDES	Two or more declarations for the same name were found. The last declaration encountered was used.	Verify that the last declaration encountered is the one that was intended.
20	F	ITEM DECLARATION IDENTIFIER ERROR	The identifier in an ITEM declaration was invalid or missing.	Correct the identifier or supply the missing identifier. Recompile.
21	F	DECLARATION DISCARDED - SCAN RESUMES AT SEMICOLON	Syntax errors were detected in a declaration. Uses of the declared name were also diagnosed.	Correct declaration and recompile.
22	W	ITEM DECLARATION TYPE ERROR - I ASSUMED	An invalid data type was specified. The item was assumed to be an integer.	Verify that the item was intended to be an integer.

TABLE B-1. COMPILER ERROR MESSAGES (Contd)

Error Number	Error Level	Message	Significance	Action
23	F	SYMPL TEXT CONTAINS EXECUTABLE STATEMENTS	Self-explanatory.	Correct error and recompile.
24	W	SIGNED PRESET ILLEGAL FOR THIS TYPE - IGNORED	This data type cannot have a sign. The sign in the preset specification was ignored.	Verify that the proper data type and preset were specified.
25	F	SCAN RESUMES AT -BEGIN-	Previous syntax errors were detected causing the diagnostic scan to be temporarily terminated. Scan was resumed at the BEGIN statement.	Correct previous errors and recompile. Some errors may not have been diagnosed.
26	F	BAD STATEMENT OR SEMICOLON MISSING - SEMICOLON ASSUMED	Self-explanatory.	Correct error and recompile.
27	F	ITEM PRESET ERROR	Self-explanatory.	Correct error and recompile.
28	F	SYMPL TEXT CONTAINS LABEL	Self-explanatory.	Correct error and recompile.
29	F	BASED, XDEF OR XREF ARRAYS NEED IDENTIFIER	An identifier is not optional in this type of declaration.	Supply identifier and recompile.
30	F	ARRAY ITEM DECLARATION SYNTAX ERROR	Self-explanatory.	Correct error and recompile.
31	F	ARRAY ITEM DECLARATION TYPE ERROR	Self-explanatory.	Correct error and recompile.
32	W	BAD ARRAY BOUND VALUES - ASSUMED [0:0]	Invalid array bounds were specified. The lower bound must be less than or equal to the upper bound. A one-entry array was assumed.	Verify that a one-entry array was intended.
33	F	ARRAY BOUND SYNTAX ERROR	Self-explanatory.	Correct error and recompile.
34	W	PARTWORD SPECIFICATION ERROR IN ARRAY ITEM DECLARATION - DEFAULT TAKEN	Self-explanatory.	Verify that the default does not affect the logic of the program.
35	W	FIRST BIT ALIGNMENT WRONG IN ARRAY ITEM DECLARATION - 0 USED	Self-explanatory.	Verify that the default does not affect the logic of the program.
36	W	ILLEGAL ARRAY ITEM BOUNDARY - DEFAULT TAKEN	Self-explanatory.	Verify that the default does not affect the logic of the program.
37	F	MAXIMUM ARRAY SIZE EXCEEDED	Self-explanatory.	Correct error and recompile.
38	F	TOO MANY PRESET GROUPS	Self-explanatory.	Correct error and recompile.

TABLE B-1. COMPILER ERROR MESSAGES (Contd)

Error Number	Error Level	Message	Significance	Action
39	F	ARRAY PRESET SYNTAX ERROR	Self-explanatory.	Correct error and recompile.
40	F	-COMMON- MUST BE AT OUTER SCOPE ONLY	Self-explanatory.	Correct error and recompile.
41	F	BAD COMMON DECLARATION IGNORED	Self-explanatory.	Correct common declaration and recompile.
42	F	BAD XREF OR XDEF IGNORED	A syntax error was detected in an XDEF or XREF declaration. References to the declared identifier were also diagnosed.	Correct declaration and recompile.
43	F	BAD BASED DECLARATION IGNORED	An error was detected in a based array declaration. References to the declared name were also diagnosed.	Correct error and recompile.
44	F	XDEF OR XREF LIST CRUD DELETED	The extraneous characters that appeared in an XDEF or XREF list were removed.	Correct error and recompile.
45	F	SYMPL TEXT CONTAINS SWITCH DECLARATION	Self-explanatory.	Correct error and recompile.
46	F	COMMON LIST SCAN RESUMES AT NEXT -ARRAY- OR -ITEM-	Previous syntax errors in the common list caused the diagnostic scan to be temporarily terminated. Scan was resumed at an ARRAY or ITEM declaration.	Correct previous syntax errors and recompile. Some errors may not have been diagnosed.
47	F	SYMPL TEXT CONTAINS INNER SCOPE VARIABLE	Self-explanatory.	Correct error and recompile.
48	F	-END- ENDS BAD COMMON LIST	The diagnosed END statement was used to terminate a common list that contains syntax errors.	Correct syntax errors in common list and check for possible BEGIN - END mismatches. Recompile.
49	F	-DEF- DECLARATION SYNTAX ERROR	Self-explanatory.	Correct error and recompile.
50	F	BAD FORMAL PARAMETER DECLARATION	Self-explanatory.	Correct error and recompile.
51	F	PROGRAM BEGINS BADLY	A program must begin with a PRGM, PROC, or FUNC statement.	Correct error and recompile.
52	F	-PRGM- DECLARATION LACKS IDENTIFIER	An identifier is required as a program name.	Supply program name and recompile.
53	F	-PRGM- DECLARATION ERROR - CRUD PRECEDES SEMICOLON	Self-explanatory.	Correct error and recompile.
54	F	XDEF OR XREF LIST SCAN RESUMES AT LEGAL ENTRY	Previous syntax errors caused the diagnostic scan to be temporarily terminated. Scan resumed at a legal entry in the declaration.	Correct previous syntax errors and recompile. Some errors may not have been diagnosed.

TABLE B-1. COMPILER ERROR MESSAGES (Contd)

Error Number	Error Level	Message	Significance	Action
55	F	DUPLICATE COMMON BLOCK DECLARATION IN SYMPL TEXT	Self-explanatory.	Correct error and recompile.
56	F	-END- ENDS BAD XDEF OR XREF LIST	The diagnosed END statement was used to terminate an XDEF or XREF declaration that contains syntax errors.	Correct syntax errors in declaration and check for possible BEGIN - END mismatches. Recompile.
57	W	SYMPL TEXT DUPLICATE DECLARATION OVERRIDES	Two or more declarations were found for the same name. The declaration appearing in the SYMPL text was used.	Verify that the proper declaration was used.
58	F	SYMPL TEXT CONTAINS FORMAL PARAMETER	Self-explanatory.	Correct error and recompile.
59	W	-FUNC- DECLARATION TYPE ERROR - I ASSUMED	An illegal type was specified for a function. The function was assumed to return an integer value.	Verify that the function was intended to be an integer function.
60	W	CONSTANT TOO LARGE, SIGNIFICANCE LOST	The high-order bits were truncated.	Verify that the truncation does not affect the logic of the program.
61	F	SCAN RESUMES AT SEMICOLON	Previous syntax errors caused the diagnostic scan to be temporarily terminated. Scan was resumed at a semicolon.	Correct the previous syntax errors and recompile. Some errors may not have been diagnosed.
62	F	DUPLICATE FORMAL PARAMETER IN LIST	An identifier cannot appear more than once in a formal parameter list.	Change the duplicate identifiers and recompile.
63	F	DUPLICATE PARAMETER - PRIOR DECLARATION THIS SCOPE	Self-explanatory.	Correct error and recompile.
64	F	PARAMETER LIST SYNTAX ERROR	Self-explanatory.	Correct error and recompile.
65	F	-PROC- DECLARATION LACKS IDENTIFIER	An identifier must be supplied as a procedure name.	Supply procedure name and recompile.
66	F	-PROC- DECLARATION SYNTAX ERROR	Self-explanatory.	Correct error and recompile.
67	F	UNDECLARED LABEL OR PROCEDURE IDENTIFIER	A label or procedure name was referenced but not declared.	Check for misspelled label or procedure reference, or supply declaration for label or procedure name. Recompile.
68	F	FORMAL IDENTIFIER LACKS DECLARATION	An identifier was referenced but not declared.	Check for misspelled identifier reference, supply declaration for identifier, or correct error in declaration. Recompile.

TABLE B-1. COMPILER ERROR MESSAGES (Contd)

Error Number	Error Level	Message	Significance	Action
69	W	PARAMETER NOT DEFINED IN THIS SCOPE, AND NOT USED	A formal parameter was not declared. Undeclared parameters are assumed to be labels.	If the diagnosed parameter is not intended to be a label, declare it to be a formal parameter and recompile. Otherwise, verify that the label is used in the scope.
70	F	ILLEGAL -DEF- IDENTIFIER - NO EXPANSION	The syntax error suppressed expansion of the DEF.	Correct the error in the DEF identifier and recompile.
71	F	ENTRY -PROC- MAY NOT CALL ITSELF	Recursive routines are not allowed in SYMPL.	Remove recursive calls and recompile.
72	F	END OF RECORD SEEN BEFORE -TERM- FOUND	A compilation unit must be contained on one record and must end with a TERM statement.	Supply TERM statement and recompile.
73	F	TOO MANY SUBSCRIPTS FOR ARRAY OR ARRAY ITEM REFERENCE	Arrays and array references can have up to 7 dimensions.	Restructure array and recompile.
74	F	TOO MANY SUBSCRIPTS FOR SWITCH REFERENCE	Self-explanatory.	Correct error and recompile.
75	F	NOT ENOUGH SUBSCRIPTS FOR ARRAY OR ARRAY ITEM REFERENCE	An array was declared without subscripts or an array reference contains fewer subscripts than specified in the array declaration.	Supply missing subscripts and recompile.
76	F	BAD SUBSCRIPT LIST	Self-explanatory.	Correct error and recompile.
77	F	ILLEGAL LABEL OR PROCEDURE IDENTIFIER USE DELETED	Self-explanatory.	Correct error and recompile.
78	F	STATUS SWITCH DECLARATION LACKS STATUS LIST IDENTIFIER	Self-explanatory.	Supply status list identifier and recompile.
79	F	BAD LABEL USE IN STATUS SWITCH	Self-explanatory.	Correct error and recompile.
80	F	STATUS SWITCH ERROR - VALUE TOO LARGE	Self-explanatory.	Correct error and recompile.
81	F	STATUS SWITCH ERROR - DUPLICATE STATUS CONSTANT VALUES	Self-explanatory.	Correct error and recompile.
82	F	STATUS SWITCH ERROR - MISSING STATUS CONSTANT	Self-explanatory.	Supply status constant and recompile.
83	F	BEGIN/END MISMATCH, PROBABLE DISASTER	The number of BEGIN and END statements differ.	Check all BEGIN - END pairs for mismatches. Correct errors and recompile.
84	F	-IF- EXPRESSION NOT BOOLEAN	Self-explanatory.	Correct error and recompile.
85	F	-WHILE- EXPRESSION NOT BOOLEAN	Self-explanatory.	Correct error and recompile.

TABLE B-1. COMPILER ERROR MESSAGES (Contd)

Error Number	Error Level	Message	Significance	Action
86	F	CRUD AFTER FINAL END IGNORED	More characters were found after the final END statement.	Check for possible BEGIN - END mismatches. Correct error and recompile.
87	F	-DEF- IDENTIFIER EXPANSION NEST TOO DEEP - IDENTIFIER DELETED	Self-explanatory.	Correct error and recompile.
88	F	MISSING -DO- HAS BEEN FOUND	Self-explanatory.	Correct error and recompile.
89	F	MISSING -THEN- HAS BEEN FOUND	Self-explanatory.	Correct error and recompile.
90	F	MISSING -DO-	A DO statement was expected.	Check program logic.
91	F	MISSING -THEN-	A THEN clause was expected.	Check program logic.
92	F	ERROR IN INITIAL VALUE EXPRESSION OF INDUCTION VARIABLE	Self-explanatory.	Correct error and recompile.
93	F	-STEP- EXPRESSION ERROR	Self-explanatory.	Correct error and recompile.
94	F	-UNTIL- EXPRESSION ERROR	Self-explanatory.	Correct error and recompile.
95	F	-WHILE- EXPRESSION ERROR	Self-explanatory.	Correct error and recompile.
96	F	BAD -GOTO- DELETED	Self-explanatory.	Correct error and recompile.
97	F	BAD REPLACEMENT STATEMENT DELETED	Self-explanatory.	Correct error and recompile.
98	W	PARTWORD SPECIFICATIONS AFTER FIRST 3 IGNORED	Self-explanatory.	Verify that the error does not affect the logic of the program.
99	F	ITEM DISCARDED - SCAN RESUMES AT COMMA	Excessive errors were detected in the ITEM declaration. The declaration was ignored.	Correct syntax errors in the declaration and recompile.
100	F	HANGING -IF- CLAUSE	Self-explanatory.	Correct error and recompile.
101	F	HANGING -FOR- CLAUSE	Self-explanatory.	Correct error and recompile.
102	F	HANGING -ELSE-	Self-explanatory.	Correct error and recompile.
103	F	EXTRA -END- FOUND, -BEGIN- FOR SUBPROGRAM ASSUMED	A BEGIN statement was assumed at the beginning of the subprogram, and was associated with the extra END statement. This may have caused other diagnostics to be generated.	Check all BEGIN - END pairs to insure proper structure. Correct all errors and recompile.

TABLE B-1. COMPILER ERROR MESSAGES (Contd)

Error Number	Error Level	Message	Significance	Action
104	F	ILLEGAL UNDECLARED PARAMETER USE DELETED	Self-explanatory.	Correct error and recompile.
105	F	-FOR- STATEMENT INDUCTION VARIABLE ERROR	Self-explanatory.	Correct error and recompile.
106	F	-IF- EXPRESSION ERROR	Self-explanatory.	Correct error and recompile.
107	F	DUPLICATE XDEF OR XREF DECLARATIONS FOR IDENTIFIER	Self-explanatory.	Remove duplicate declarations and recompile.
108	F	XDEF -PROC- OR -FUNC- NOT FULLY DECLARED	Self-explanatory.	Correct error and recompile.
109	F	BAD FORMAL PARAMETER DECLARATION	Self-explanatory.	Correct error and recompile.
110	F	REDUNDANT FORMAL PARAMETER DECLARATION	Self-explanatory.	Correct error and recompile.
111	F	BAD PARAMETER LIST	Self-explanatory.	Correct error and recompile.
112	F	BOOLEAN ILLEGAL IN ARITHMETIC CONTEXT	Self-explanatory.	Correct error and recompile.
113	F	COMMON LIST LACKS -END-	Common lists must be surrounded by BEGIN and END statements.	Supply missing END statement and recompile.
114	F	BASED LIST LACKS -END-	Based array lists must be surrounded by BEGIN and END statements.	Supply missing END statement and recompile.
115	F	XDEF OR XREF LIST LACKS -END-	XDEF and XREF lists must be surrounded by BEGIN and END statements.	Supply missing END statement and recompile.
116	F	BAD COMMON LIST DECLARATION DELETED	Self-explanatory.	Correct error and recompile.
117	F	BAD BASED ARRAY DECLARATION DELETED	Self-explanatory.	Correct error and recompile.
118	F	BASED LIST SCAN RESUMES WITH -ARRAY-	Previous syntax errors caused the diagnostic scan to be temporarily terminated. Scan was resumed at the ARRAY declaration.	Correct previous errors and recompile. Some errors may not have been diagnosed.
119	F	-END- ENDS BAD BASED ARRAY LIST	The diagnosed END statement was used to terminate a based array list that contains syntax errors.	Correct syntax errors and check for possible BEGIN - END mismatches. Recompile.
120	T	ZERO LENGTH -DEF- STRING IGNORED	Self-explanatory.	Check program logic.
121	W	CHARACTER ITEM LENGTH OMITTED - 1 ASSUMED	Self-explanatory.	Verify that a character length of 1 was intended.
122	F	BAD ARRAY ENTRY SIZE	Self-explanatory.	Correct error and recompile.

TABLE B-1. COMPILER ERROR MESSAGES (Contd)

Error Number	Error Level	Message	Significance	Action
123	F	BRACKET NEST TOO DEEP	Self-explanatory.	Correct error and recompile.
124	F	ILLEGAL EXPRESSION TYPE ON LEFT SIDE	Self-explanatory.	Correct error and recompile.
125	F	BAD BEAD FUNCTION	Self-explanatory.	Correct error and recompile.
126	F	OPERATOR OR OPERAND MISSING IN EXPRESSION	Self-explanatory.	Correct error and recompile.
127	W	LONG CHARACTER STRING - 240 BYTES USED	The first 240 characters were used.	Verify that the error does not affect the logic of the program.
128	F	BAD -LOC- FUNCTION	The LOC function was used incorrectly.	Correct error and recompile.
129	F	BAD -ABS- FUNCTION	The ABS function was used incorrectly.	Correct error and recompile.
130	F	BAD INDUCTION VARIABLE TYPE	Self-explanatory.	Correct error and recompile.
131	F	VARIABLE IN -TEST- IS NOT AN INDUCTION VARIABLE	The variable specified in the TEST statement must be the induction variable of the loop being tested.	Check for misspelled variable name. Correct error and recompile.
132	F	-TEST- ILLEGAL OUTSIDE LOOP	The TEST statement must appear inside the loop for which the induction variable is being tested.	Move TEST statement inside the appropriate loop and recompile.
133	F	SCAN RESUMES AT -BEGIN-, -ITEM- OR SEMICOLON	Previous syntax errors caused the diagnostic scan to be temporarily terminated. Scan resumed at the next BEGIN, ITEM, or semicolon.	Correct previous syntax errors and recompile. Some errors may not have been diagnosed.
134	F	BEAD FUNCTION NEEDS IDENTIFIER	Self-explanatory.	Supply identifier and recompile.
135	F	DUPLICATE STATUS IDENTIFIER	Self-explanatory.	Correct error and recompile.
136	W	SEMICOLON ENDS COMMENT	A comment cannot contain a semicolon or a pound sign.	Check for a missing comment delimiter, or remove invalid character from the comment string.
137	F	-CONTROL- STATEMENT SYNTAX ERROR	Self-explanatory.	Correct error and recompile.
138	F	CHARACTER IN REAL CONSTANT IS NOT D OR E	Self-explanatory.	Change character in real constant to a D or E and recompile.
139	W	FORMAL PARAMETER PRESET IS IGNORED	A formal parameter cannot be preset.	Check program logic.
140	F	-XREF- PRESET IS ILLEGAL	A name that is defined in an externally-compiled procedure cannot be preset in this compilation unit.	Remove preset spcification and recompile.

TABLE B-1. COMPILER ERROR MESSAGES (Contd)

Error Number	Error Level	Message	Significance	Action
141	F	BLANK COMMON PRESET IS ILLEGAL	Blank common cannot be preset.	Remove preset specification and recompile.
142	F	BASED ARRAY ITEM PRESET IS ILLEGAL	Based arrays cannot be preset.	Remove preset specification and recompile.
143	F	BAD P FUNCTION	The P function was used incorrectly.	Correct error and recompile.
144	W	LENGTH OF CHARACTER ITEM IS GREATER THAN 240 BYTES - 240 USED	The first 240 characters were used.	Verify that the truncation does not affect the logic of the program.
145	W	NO SUBSCRIPT FOR ARRAY ITEM - LOWER ARRAY BOUND USED	The subscript was missing from the array reference. The compiler assumed the first element of the array was intended.	Verify that the first element is intended.
146	F	CIRCULAR DEF NAME EXPANSION - EXPANSION IGNORED	A DEF cannot directly or indirectly reference itself.	Rewrite the DEFs to remove the recursive references. Recompile.
147	F	ENTRY -PROC- OR -FUNC- NOT ALLOWED IN A -PRGM-	Self-explanatory.	Remove ENTRY and recompile.
148	F	ILLEGAL CHARACTER IN PARAMETERIZED -DEF- TEXT	Self-explanatory.	Correct error and recompile.
149	F	ILLEGAL COMPARISON IN -CONTROL IF-	Self-explanatory.	Correct error and recompile.
150	F	TOO MANY DEF PARAMETERS	Self-explanatory.	Correct error and recompile.
151	W	ILLEGAL CONDITIONAL DIRECTIVE IGNORED	Self-explanatory.	Verify that the error does not affect the logic of the program.
152	F	LABEL IS ILLEGAL AS A VALUE PARAMETER	Self-explanatory.	Correct error and recompile.
153	F	ARRAY IS ILLEGAL AS A VALUE PARAMETER	Self-explanatory.	Correct error and recompile.
154	F	-PROC- OR -FUNC- IS ILLEGAL AS A VALUE PARAMETER	Self-explanatory.	Correct error and recompile.
155	F	COMMON BASED ARRAY DECLARATION ERROR	Self-explanatory.	Correct error and recompile.
156	F	SYMPL TEXT IS NOT A -PRGM- OR -PROC-	Self-explanatory.	Correct error and recompile.
157	F	XREF SWITCH ERROR	Self-explanatory.	Correct error and recompile.
158	F	UNMATCHED -CONTROL IF-	A CONTROL IF statement must have a corresponding CONTROL FI or CONTROL ENDIF statement.	Supply a CONTROL FI or CONTROL ENDIF statement to delimit conditional block.
159	F	-DEF- PARAMETER ERROR	Self-explanatory.	Correct error and recompile.

TABLE B-1. COMPILER ERROR MESSAGES (Contd)

Error Number	Error Level	Message	Significance	Action
160	F	([OR < NESTING TOO DEEP)	Self-explanatory.	Correct error and recompile.
161	F	([OR < NEST MISMATCH)	Self-explanatory.	Correct error and recompile.
162	F	-DEF- PARAMETER TOO LONG	Self-explanatory.	Correct error and recompile.
163	F	-DEF- PARAMETER COUNT ERROR	Self-explanatory.	Correct error and recompile.
164	F	RECOVERY AT SEMICOLON	Previous syntax errors caused the diagnostic scan to be temporarily terminated. Scan was resumed at the semicolon.	Correct previous syntax errors and recompile. Some errors may not have been diagnosed.
165	F	BAD -DEF- ACTUAL PARAMETER	Self-explanatory.	Correct error and recompile.
166	F	BAD UNDECLARED PROC OR LABEL LIST	Self-explanatory.	Correct error and recompile.
167	F	SYMPL TEXT CONTAINS -LABEL- OR INNER -PROC-	Self-explanatory.	Correct error and recompile.
168	F	-TERM- ENCOUNTERED PREMATURELY - NEXT LINE BEGINS A NEW SUBPROGRAM	Compilation was terminated by a TERM statement before the end of the program was reached. This may have caused errors in the next compilation unit.	Check for a misplaced TERM statement. Check BEGIN - END matching in first compilation unit. Correct errors and recompile.
169	F	ATTRIBUTE SPECIFIED TO UNKNOWN VARIABLE	Self-explanatory.	Correct error and recompile.
170	F	SCALAR ITEMS MAY NOT BE INERT OR REACTIVE	Self-explanatory.	Correct error and recompile.
171	F	ONLY ITEMS AND ARRAYS HAVE ATTRIBUTES	Self-explanatory.	Correct error and recompile.
172	F	BAD ATTRIBUTE OR LEVEL SPECIFICATION LIST	Self-explanatory.	Correct error and recompile.
173	F	FAST FOR LOOP INDUCTION VARIABLE ERROR	Self-explanatory.	Correct error and recompile.
174	F	BAD GLOBAL ATTRIBUTE SPECIFICATION	Self-explanatory.	Correct error and recompile.
175	F	LEVEL ONLY APPLIES TO COMMON AND BASED ARRAYS	Self-explanatory.	Correct error and recompile.
176	F	BAD USE OF LEVEL 3 VARIABLE	The level 3 data appeared in one of the prohibited forms.	Correct error and recompile.
177	F	INDUCTION VARIABLE MUST BE SCM RESIDENT	The induction variable must be located in small central memory.	Correct error and recompile.
178	F	-CONTROL WEAK- ONLY APPLIES TO EXTERNAL SYMBOLS	Self-explanatory.	Correct error and recompile.

TABLE B-1. COMPILER ERROR MESSAGES (Contd)

Error Number	Error Level	Message	Significance	Action
179	F	ARRAY ENTRY-SIZE TOO LARGE	Self-explanatory.	Correct error and recompile.
180	F	TOO MANY ARRAY DIMENSIONS	Only 7 dimensions are allowed for an array.	Reduce number of array dimensions and recompile.
181	F	RECURSIVE -PROC- OR -FUNC- CALL NOT ALLOWED	A procedure or function cannot call itself, and cannot call other sub-programs that result in a recursive call.	Restructure subprogram calls to avoid recursion. Recompile.
182	F	ERROR IN REAL CONSTANT	Self-explanatory.	Correct error and recompile.
183	D	XREF OR XDEF LABELS WILL NOT BE SUPPORTED IN FUTURE VERSIONS	Self-explanatory.	None.
184	D	XREF OR XDEF SWITCH WILL NOT BE SUPPORTED IN FUTURE VERSIONS	Self-explanatory.	None.
185	D	XDEF -PROC- WILL NOT BE SUPPORTED IN FUTURE VERSIONS	Self-explanatory.	None.
186	D	XDEF -FUNC- WILL NOT BE SUPPORTED IN FUTURE VERSIONS	Self-explanatory.	None.
187	D	-CONTROL FASTLOOP- WILL NOT BE SUPPORTED IN FUTURE VERSIONS	Self-explanatory.	None.
188	D	-CONTROL SLOWLOOP- WILL NOT BE SUPPORTED IN FUTURE VERSIONS	Self-explanatory.	None.
189	D	LOC OF -LABEL- IS MACHINE DEPENDENT	Self-explanatory.	None.
190	D	LOC OF -SWITCH- IS MACHINE DEPENDENT	Self-explanatory.	None.
191	D	LOC OF -PROC- IS MACHINE DEPENDENT	Self-explanatory.	None.
192	D	LOC OF -FUNC- IS MACHINE DEPENDENT	Self-explanatory.	None.
193	D	-BIT- BEAD FUNCTION ON A CHARACTER ITEM IS MACHINE DEPENDENT	Self-explanatory.	None.
194	D	-BYTE- BEAD FUNCTION ON A NON-CHARACTER ITEM IS MACHINE DEPENDENT	Self-explanatory.	None.
195	D	PRESET TYPE DIFFERENT FROM ITEM TYPE IS MACHINE DEPENDENT	Self-explanatory.	None.
196	D	P FUNCTION SET TO A CONSTANT IS MACHINE DEPENDENT	Self-explanatory.	None.
197	D	OUT OF SCOPE GOTO IS MACHINE DEPENDENT	Self-explanatory.	None.

TABLE B-1. COMPILER ERROR MESSAGES (Contd)

Error Number	Error Level	Message	Significance	Action
198	D	EXPLICIT CONSTANTS 6, 10, 18, 42, 60 IN EXECUTABLE CODE MAY BE MACHINE DEPENDENT	Self-explanatory.	None.
199	D	EXCEEDING ARRAY ENTRY SIZE IS MACHINE DEPENDENT	Self-explanatory.	None.
200	D	-MODULE- AND -FASTFOR- WILL BE RESERVED IN FUTURE VERSIONS	Self-explanatory.	None.
201	F	ARRAY ITEM OVERLAY CANNOT BE A CONSTANT	Self-explanatory.	Correct error and recompile.
202	F	TOO MANY SPECIFICATIONS IN ITEM DECLARATION	Self-explanatory.	Correct error and recompile.
203	F	ARRAY ITEM LENGTH MUST BE AN UNSIGNED CONSTANT	Self-explanatory.	Correct error and recompile.
204	F	ERROR IN ARRAY ITEM SPECIFICATION, SECOND PART	An error was found in the overlay part of the declaration.	Correct error and recompile.
205	F	CLOSING ANGLE BRACKET EXPECTED IN ARRAY ITEM DECLARATION	Self-explanatory.	Correct error and recompile.
206	F	FILLER ITEM VALID ONLY IN OVERLAY-TYPE ARRAY ITEM DECLARATIONS	Self-explanatory.	Correct error and recompile.
207	W	SPECIFICATION OF ARRAY ENTRY SIZE ILLEGAL FOR A OR U ARRAYS - IGNORED	Self-explanatory.	Check program logic.
208	F	ITEM SPECIFIED AS OVERLAY IS NOT DEFINED IN THIS ARRAY DECLARATION	Self-explanatory.	Correct error and recompile.
209	F	ARRAY ITEM OVERLAY CLASSES ARE INCOMPATIBLE	Self-explanatory.	Correct error and recompile.
210	W	LENGTH OF REAL ITEM FORCED TO A FULL WORD	A real item must occupy a full word.	Verify that the item length used does not affect the logic of the program.
211	F	LENGTH OF SUBFIELD EXCEEDS LENGTH OF ORIGINAL FIELD - DECLARATION DISCARDED	Self-explanatory.	Correct error and recompile.
212	F	PREVIOUS ARRAY ITEM DECLARATION DID NOT SPECIFY PLUS OR OVERLAY	Self-explanatory.	Correct error and recompile.
213	W	SPECIFIED ITEM LENGTH EXCEEDS MAXIMUM ALLOWABLE - DEFAULT TAKEN	The item was truncated to the default length.	Verify that the item length used does not affect the logic of the program.
214	F	LEFT ANGLE BRACKET EXPECTED FOR ARRAY ITEM - DECLARATION DISCARDED	Self-explanatory.	Correct error and recompile.
215	F	FILLER ITEMS CANNOT BE PRESET	Self-explanatory.	Remove preset specification and recompile.

TABLE B-1. COMPILER ERROR MESSAGES (Contd)

Error Number	Error Level	Message	Significance	Action
216	F	LEFT PARENTHESIS EXPECTED IN ARRAY ITEM DECLARATION - DECLARATION DISCARDED	Self-explanatory.	Correct error and recompile.
217	F	STATUS LIST NOT DEFINED IN CURRENT OR PREVIOUS TEXTS	Self-explanatory.	Correct error and recompile.
218	D	DEFAULT VARIABLE BEHAVIOR WILL CHANGE IN FUTURE VERSIONS	Self-explanatory.	None.
224	W	FUNCTION NAME THE SAME AS FORMAL PARAMETER NAME	Self-explanatory.	Change function name or formal parameter name.
225	F	CODE GEN SUPPRESSED	The compiler stopped generating object code because of previous syntax errors. The compiler continued to scan the source code for errors.	Correct all syntax errors and recompile.
800	C	INSUFFICIENT FIELD LENGTH (MORESPC IN CONTROL)	Compilation aborted.	
801	C	SCM FREE SPACE BLOCK INSUFFICIENT SIZE (TSPACE IN SRCH)	Compilation aborted.	
802	C	FIND LOOP (FIND IN SRCH)	Compilation aborted.	
803	C	ARRAY PRESET EXCEEDS MAXIMUM SIZE (PSET IN ALOCTR)	Compilation aborted.	
804	C	ARRAY ITEM PRESET LARGER THAN BUFFER (GETSP IN ALOCTR)	Compilation aborted.	
805	C	COND REPL OUTSIDE OF RULE BODY, BAD GENESIS TABLES (ANZS)	Compilation aborted.	
806	C	SAVE CONTROL TABLE OVERFLOW (CSAVE IN CODE)	Compilation aborted.	
807	C	WHATS DOING TABLE OVERFLOW (CSAVE IN CODE)	Compilation aborted.	
808	C	BAD ENDSAVE REQ. NO SAVE REQ EXTANT (ENDSAVE IN CODE)	Compilation aborted.	
809	C	RESTR/FORGET ON NON-EXTANT SAVE BUFFER (ZAP IN CODE)	Compilation aborted.	
810	C	RESTR/FORGET ON CURRENT SAVE BUFFER (ZAP IN CODE)	Compilation aborted.	
811	C	OSAVE ON CURRENT SAVE BUFFER (OSAVE IN CODE)	Compilation aborted.	
812	C	OSAVE ON NON-EXTANT SAVE BUFFER (OSAVE IN CODE)	Compilation aborted.	
813	C	WHATS DOING TABLE OVERFLOW (OSAVE IN CODE)	Compilation aborted.	

TABLE B-1. COMPILER ERROR MESSAGES (Contd)

Error Number	Error Level	Message	Significance	Action
814	C	IL FAT OVERFLOW (HATCHK)	Compilation aborted.	
815	C	> 100 PROGRAM SCOPES (SCPIN IN PF1SUB)	Compilation aborted.	
816	C	NO PRAGMATIC FUNC (PF11)	Compilation aborted.	
817	C	BAD UNDEF ID (PF11)	Compilation aborted.	
818	C	BRACKET/PAREN NEST TOO DEEP (PF11)	Compilation aborted.	
819	C	NO PRAGMATIC FUNC (PF12)	Compilation aborted.	
820	C	C-TYPE ITM ERR (PF12)	Compilation aborted.	
821	C	COMPOUND STATEMENT COLLECTION ERR (PF12)	Compilation aborted.	
822	C	NO PRAGMATIC FUNC (PF13)	Compilation aborted.	
823	C	PARAM LIST TABLE OVERFLOW (PF13)	Compilation aborted.	
824	C	SYNTAX ERROR (PF13)	Compilation aborted.	
825	C	LOOP CONTROL TABLE OVERFLOW (PF14 IN PF13)	Compilation aborted.	
826	C	FALL THROUGH (FILTRO) LINE nnnn	Compilation aborted.	
827	C	STACK UNDERFLOW (PHASBS) LINE nnnn	Compilation aborted.	
828	C	UNEXPECTED END-OF-FILE (PHASBS) LINE nnnn	Compilation aborted.	
829	C	ILLEGAL IL SEEN (PHASBS) LINE nnnn	Compilation aborted.	
830	C	STACK OVERFLOW (PHASBS) LINE nnnn	Compilation aborted.	
831	C	TRIAD TABLE OVERFLOW (GETRD) LINE nnnn	Compilation aborted.	
832	C	STACK NON-EMPTY AT BREAKPOINT (PHASBS) LINE nnnn	Compilation aborted.	
833	C	BAD SWITCH STATE INDEX (BRKPT) LINE nnnn	Compilation aborted.	
834	C	ZERO-DIVIDE ATTEMPT (EXPGEN) LINE nnnn	Compilation aborted.	
835	C	NON-EXISTENT VALU DELETION ATTEMPT (DHASH) LINE nnnn	Compilation aborted.	
836	C	NON-EXISTENT VALU RETRIEVAL ATTEMPT (EXPGEN) LINE nnnn	Compilation aborted.	
837	C	LOOP STACK OVERFLOW (BRKPT) LINE nnnn	Compilation aborted.	

TABLE B-1. COMPILER ERROR MESSAGES (Contd)

Error Number	Error Level	Message	Significance	Action
838	C	LOOP STACK UNDERFLOW (BRKPT) LINE nnnn	Compilation aborted.	
839	C	ILLEGAL LEFT SIDE (REPRC) LINE nnnn	Compilation aborted.	
840	C	VALB OF LOCALLY SAVD VALUE (BRKPT) LINE nnnn	Compilation aborted.	
841	C	STACK HISTORY OVERFLOW (PHASBS) LINE nnnn	Compilation aborted.	
842	C	INFINITE OPERAND (CXIOP IN CONSAM) LINE nnnn	Compilation aborted.	
843	C	INDEFINITE OPERAND (CXIOP IN CONSAM) LINE nnnn	Compilation aborted.	
844	C	PARM STACK NON-EMPTY (IADCON IN ADCON) LINE nnnn	Compilation aborted.	
845	C	PARM STACK OVERFLOW (SADCON IN ADCON) LINE nnnn	Compilation aborted.	
846	C	PARM STACK UNDERFLOW (FADCON IN ADCON) LINE nnnn	Compilation aborted.	
847	C	PARM LIST OVERFLOW (ADCON) LINE nnnn	Compilation aborted.	
848	C	ILLEGAL FUNI BYTE (PHASBS) LINE nnnn	Compilation aborted.	
849	C	UNEXPECTED END-OF-FILE, NO PTPM (GICFB IN CODGJ1) LINE nnnn	Compilation aborted.	
850	C	CANT FIND TEMP (PTEMP IN CODGJ1) LINE nnnn	Compilation aborted.	
851	C	ICFT OVERFLOW (ICOVRFL IN CODGJ1) LINE nnnn	Compilation aborted.	
852	C	UNEXPECTED TYPE (PSTOU IN CODGJ2) LINE nnnn	Compilation aborted.	
853	C	BADLY FORMED ICF (POS IN CODGJ2) LINE nnnn	Compilation aborted.	
854	C	UNEXPECTED OPERAND FOR STORE (PSTOS IN CODGJ2) LINE nnnn	Compilation aborted.	
855	C	ILLEGAL OP CODE (CODGJ3) LINE nnnn	Compilation aborted.	
856	C	ILLEGAL SEQ. TERMINATING OP CODE (CODGJ3) LINE nnnn	Compilation aborted.	
857	C	BADLY FORMED ICF (CODGJ3) LINE nnnn	Compilation aborted.	
858	C	BAD BPSB OPERATOR OPERAND (CODGK1) LINE nnnn		

TABLE B-1. COMPILER ERROR MESSAGES (Contd)

Error Number	Error Level	Message	Significance	Action
859	C	ILLEGAL BPSB OPERATOR OPERAND (CODGK1) LINE nnnn	Compilation aborted.	
860	C	CANT INSERT LOAD (ISLOO IN CODGK1S) LINE nnnn	Compilation aborted.	
861	C	CIRCULAR ENTRY IN READY SET (RPC4O IN CODGK3S) LINE nnnn	Compilation aborted.	
862	C	INVALID ICFT INDEX (RPCOO IN CODGK3S) LINE nnnn	Compilation aborted.	
863	C	TOO MANY COMMON BLOCKS (EDITOR)	Compilation aborted.	
864	C	VALB OF LOCALLY SAVED VALU (EXPGEN) LINE nnnn	Compilation aborted.	
865	C	UNINTERPRETABLE SOURCE, EXPRESSION TOO LONG (SPRECG)	Compilation aborted.	
866	C	MODULE MUST BEGIN WITH PROC/FUNC/PRGM (ANZS)	Compilation aborted.	
867	C	BEADFUNC WITH LENGTH = 0 ILL (PHASBS)	Compilation aborted.	
868	C	BAD CLASS IN TEXT (USETEXT)	Compilation aborted.	
869	C	MORE THAN 64 TEXTS ON USETEXT (INIT15)	Compilation aborted.	
870	C	Y FILE NOT ON RANDOM DEVICE (INIT15)	Compilation aborted.	
871	C	MORE THAN 64 SYMPL TEXTS ON Y FILES (INIT15)	Compilation aborted.	
872	C	SYMPL TEXT NOT FOUND (INIT15)	Compilation aborted.	
873	C	INSUFFICIENT SCM FL (INIT15)	Compilation aborted.	
874	C	INSUFFICIENT LCM FL (INIT15)	Compilation aborted.	

GLOSSARY C

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.

Compilation Unit -
A separately compiled main program or subprogram terminated by a TERM statement or end-of-section.

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 point has a name with which it is associated.

Exchange Statement -
A statement that causes the exchange of values of the left and right 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.

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 of 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 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.

METALANGUAGE D

The mechanics for defining the syntactic forms of SYMPL are accomplished through an elementary descriptive language, capable of defining any phrase-structured language.

SYMPL is described in a metalanguage by a set of statements called productions, each of which describes one form belonging to SYMPL. The forms of a language are its syntactic entities, such as the sentence or adverbial phrase (from English), or arithmetic expressions (from FORTRAN, for example).

Every form of SYMPL is described by one metalinguistic production.

Format of a production is as follows:

form name := context ⌋ form definition ⌊ context

form name — Underscored name of the form defined by this production. In the metalanguage every underscored sequence is a form name.

:= — Production symbol, which may be read: has the form.

form definition — Structure of the form defined by this production (whose name is given as the form name of the production). The definition of a form specifies the set of character sequences (utterances) that it represents; form definitions specify a sequence of the following entities:

Characters of the SYMPL character set, which represent themselves.

Names of SYMPL forms, which represent sequences of characters of the SYMPL character set, as specified by the productions which describe the form names.

Sets of entities like the above, from which any one may be chosen. Such a set is enclosed within braces to indicate alternatives. The use of such alternative sets may be recursively defined; thus the form definition

$$\underline{X} \left\{ \begin{array}{ll} P & \left\{ \begin{array}{l} \underline{R} \\ S \end{array} \right\} \\ \underline{Q} & \end{array} \right\}$$

is equivalent to a choice of one of the following alternative sequences:

$\underline{X}$ P $\underline{R}$

$\underline{X}$ P S

$\underline{X}$ $\underline{Q}$

The null form ϕ represents zero characters of SYMPL. Typically, ϕ is used as one member of an alternative set if no member of the set must be chosen.

context⌋ and ⌊context — Optional constraints upon applicability of the production. If a production contains either or both context sequences, the specified form name only represents the sequence of SYMPL characters defined by form definition when it occurs in the given context. A context sequence is formed similarly to a form definition sequence.

Thus, the production pair

$$\underline{X} := \left\{ \begin{array}{l} A \\ \underline{X}\ A \end{array} \right\}$$

$$\underline{Y} := B \rfloor\ \underline{X}\ \lfloor B$$

describes sequences of the character A as the form name Y only when they are delimited by occurrences of the character B.

To summarize, seven symbols are peculiar to the metalanguage:

Underscore line	_____
Production symbol	:=
Null symbol	ϕ

Braces	{	and	}
Context delimiters	⌋	and	⌊

All other printed characters in metalinguistic productions are either form names (underscored) or self-representative members of the SYMPL character set.

BASIC NOTATION AND ELEMENTS

CHARACTER SET

SYMPL programs are composed of 55 characters, as follows:

Letters

letter : = { A B C D E F G H I J K L M N O P Q R S T U V W X Y Z $ }

Digits

digit := { 0 1 2 3 4 5 6 7 8 9 }

Marks

mark := { * / + - ( ) [ ] < > # " = . , ; : ƀ }

ƀ represents a blank space.

BLANK SPACES AND COMMENTS

space := { ƀ | comment }

△ := { space | △ space }

$$\underline{\vee} := \left\{ \begin{array}{l} \underline{\wedge} \\ \phi \end{array} \right\}$$

$$\underline{\text{comment}} := \# \ \underline{\text{comment string}} \ \#$$

$$\underline{\text{comment string}} := \left\{ \begin{array}{l} \psi \\ \underline{\text{comment string}} \ \psi \\ \phi \end{array} \right\}$$

ψ represents any key punch character except semicolon (;) and pound sign (#), either of which will terminate a comment.

The forms $\underline{\wedge}$ and $\underline{\vee}$ are used throughout the metalinguistic description to represent one or more blanks and zero or more blanks, respectively.

IDENTIFIERS

$$\underline{\text{ident}} := \underline{\text{mark}} \ \rfloor \ \underline{\text{ident part}} \ \lfloor \ \underline{\text{mark}}$$

$$\underline{\text{ident part}} := \left\{ \begin{array}{l} \underline{\text{letter}} \\ \underline{\text{ident part}} \quad \left\{ \begin{array}{l} \underline{\text{letter}} \\ \underline{\text{digit}} \end{array} \right\} \end{array} \right\}$$

RESERVED SYMBOLS

The 52 SYMPL words are represented as follows:

$$\underline{\text{abs}} := \underline{\text{mark}} \ \rfloor \ \text{ABS} \ \lfloor \ \underline{\text{mark}}$$

$$\underline{\text{and}} := \underline{\text{mark}} \ \rfloor \ \text{AND} \ \lfloor \ \underline{\text{mark}}$$

$$\underline{\text{array}} := \underline{\text{mark}} \ \rfloor \ \text{ARRAY} \ \lfloor \ \underline{\text{mark}}$$

$$\underline{\text{based}} := \underline{\text{mark}} \ \rfloor \ \text{BASED} \ \lfloor \ \underline{\text{mark}}$$

$$\underline{\text{begin}} := \underline{\text{mark}} \ \rfloor \ \text{BEGIN} \ \lfloor \ \underline{\text{mark}}$$

$$\underline{\text{common}} := \underline{\text{mark}} \ \rfloor \ \text{COMMON} \ \lfloor \ \underline{\text{mark}}$$

$$\underline{\text{control}} := \underline{\text{mark}} \ \rfloor \ \text{CONTROL} \ \lfloor \ \underline{\text{mark}}$$

$$\underline{\text{def}} := \underline{\text{mark}} \ \rfloor \ \text{DEF} \ \lfloor \ \underline{\text{mark}}$$

$$\underline{\text{do}} := \underline{\text{mark}} \ \rfloor \ \text{DO} \ \lfloor \ \underline{\text{mark}}$$

```
else    := mark ] ELSE [ mark
end     := mark ] END [ mark
entry   := mark ] ENTRY [ mark
eq      := mark ] EQ [ mark
false   := mark ] FALSE [ mark
for     := mark ] FOR [ mark
fprc    := mark ] FPRC [ mark
func    := mark ] FUNC [ mark
goto    := mark ] GOTO [ mark
gq      := mark ] GQ [ mark
gr      := mark ] GR [ mark
if      := mark ] IF [ mark
item    := mark ] ITEM [ mark
label   := mark ] LABEL [ mark
lan     := mark ] LAN [ mark
lim     := mark ] LIM [ mark
loc     := mark ] LOC [ mark
lor     := mark ] LOR [ mark
lno     := mark ] LNO [ mark
lq      := mark ] LQ [ mark
lqv     := mark ] LQV [ mark
ls      := mark ] LS [ mark
lxr     := mark ] LXR [ mark
not     := mark ] NOT [ mark
nq      := mark ] NQ [ mark
or      := mark ] OR [ mark
prgm    := mark ] PRGM [ mark
proc    := mark ] PROC [ mark
return  := mark ] RETURN [ mark
status  := mark ] STATUS [ mark
```

step	:=	mark	⌋	STEP	⌊	mark
stop	:=	mark	⌋	STOP	⌊	mark
switch	:=	mark	⌋	SWITCH	⌊	mark
term	:=	mark	⌋	TERM		mark
test	:=	mark	⌋	TEST	⌊	mark
then	:=	mark	⌋	THEN	⌊	mark
true	:=	mark	⌋	TRUE	⌊	mark
until	:=	mark	⌋	UNTIL	⌊	mark
while	:=	mark	⌋	WHILE	⌊	mark
xdef	:=	mark	⌋	XDEF	⌊	mark
xref	:=	mark	⌋	XREF	⌊	mark
spbegin	:=	mark	⌋	$BEGIN	⌊	mark
spend	:=	mark	⌋	$END	⌊	mark

The action of $BEGIN and $END depends on the presence of option E on the SYMPL control statement.

SPECIAL IDENTIFIERS

array item name	:=	ident
array name	:=	ident
based array name	:=	ident
common name	:=	ident
def name	:=	ident
formal array name	:=	ident
formal based name	:=	ident
formal func name	:=	ident
formal item name	:=	ident
formal proc name	:=	ident
func name	:=	ident
item name	:=	ident
label name	:=	ident

proc name := ident

program name := ident

status list name := ident

switch name := ident

DEF DECLARATIONS

DEF Specification

def head := def ∧ ident

defmac head := def head opt space (∨ def params ∨)

def dec := def head opt space #DEF # ∨ ;

defmac dec := defmac head opt space #DEF #

opt space := { ϕ | opt space ƀ }

DEF string := { ψ | DEF string ψ }

def params := { ident | def params ∨ , ∨ ident }

ψ represents any keypunch character for DEF declarations with no parameters.

DEF Expansion

defmac expansion := defmac name ∨ (def parlist)

def par list := { def par | def par list ∨ , ∨ def par }

def par := { Any character sequence that meets the limitations discussed in section 4 regarding balanced brackets, or the characters ; # ? }

EXPRESSIONS

Arithmetic Expressions

arith exp := { unary op ∨ / ϕ } infix stuff

infix stuff := { arith thing / infix stuff ∨ binary op ∨ arith thing }

arith thing := { item name / array reference / func call / const / (∨ arith exp ∨) }

unary op := { + / - / lno }

binary op := { ** / * / / / + / - / lan / lor / lxr / lim / lqv }

Boolean Expressions

Boolean exp := { Boolean thing / Boolean exp ∨ Boolean op ∨ Boolean thing }

Boolean thing := { array reference / item name / relation / Boolean const / not ∨ Boolean thing / func call / (∨ Boolean exp ∨) }

An item must be declared type B for use as a Boolean operand.

Boolean op := { and / or }

relation := arith exp ∨ relational op ∨ arith exp

relational op := { eq / gr / ls / gq / lq / nq }

CONSTANTS

const := { Boolean const / char const / integer const / real const / status const }

Integer Constants

integer const := { dec integer / octal const / hex const / status func }

The status func is a special form of integer constant defined under status declarations.

dec integer := { dec integer / ϕ } digit

octal const := O " octal stuff "

octal stuff := { octal stuff / ϕ } { octal digit / △ }

octal digit := { 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 }

Hexadecimal Constants

hex const := X " hex stuff "

hex stuff := { hex stuff | ϕ } { hex digit | ^ }

hex digit := { digit | A | B | C | D | E | F }

Boolean Constants

Boolean const := { true | false }

Character Constants

char const := " char string "

char string := { char string ψ | ϕ }

ψ represents any keypunch character.

Status Constants

status const := S " v status const string v "

status const string := ident

Real Constants

real const := { integer part / ϕ } . { fraction part / ϕ } { exponent part / ϕ }

integer part := dec integer

fraction part := dec integer

exponent part := { D / E } { + / - / ϕ } dec integer

ITEMS

ITEM Declaration

item dec := item ∧ item descr list ∨ ;

item descr list := { item descr / item descr list ∨ , ∨ item descr }

item descr := item name { ∧ type / ϕ } ∨ item preset

type := { U / I / R / B / C ∨ (∨ length ∨) / S ∨ : ✓ status list name }

U = unsigned integer type

I = integer type

R = real type

B = Boolean type

S = status type

C = character type length is a size subfield in characters or bytes

length := integer const

Item Presets

Optionally, the item may be assigned an initial value:

item preset := { = [v] { + [v] / - [v] / ϕ } const / ϕ }

STATUS DECLARATIONS

Specification

status dec := status ^ status list name ^ status name list v ;

status name list := { status value / status name list v , v status value }

status value := { status const string / ϕ }

Status Function

status func := status list name " v status const string v "

ARRAYS

Array Declarations

array dec := array { ^ array name / ∅ } v array descr v ; v item part

array descr := { [v array bounds list v] / ∅ } v { layout / ∅ } v { entry size / ∅ }

array bounds list := { bound pair / array bounds list v , v bound pair }

bound pair := { low bound v : v / ∅ } high bound

low bound := { + / - / ∅ } integer const

high bound := { + ∨ / - ∨ / ∅ } integer const

layout := { P / S }

entry size := (∨ integer const ∨)

Array Item Declarations

item part := { begin ∧ array item dec list ∨ end / array item dec / ; }

array item dec list := { array item dec / array item dec list ∨ array item dec }

array item dec := item ∧ array item descr list ∨ ;

array item descr list := { array item descr / array item descr list ∨ , ∨ array item descr }

array item descr := array item name ∨ array item specs ∨ array preset

array item specs := { U / I / R / B / C / S ∨ : ∨ status list name / ∅ } ∨ { (∨ ep ∨ { , ∨ fbit ∨ { , ∨ size ∨ / ∅ } / ∅ }) }

ep := integer const

fbit := integer const

size := integer const

Array Presets

array preset := { ∅ / = ∨ value set }

set sequence := { ϕ | set sequence value set | set sequence integer const value set }

value set := { [∨ value list ∨] | [∨ set sequence ∨] }

value list := { value | integer const ∨ (∨ value list ∨) | value list ∨ , ∨ value }

value := { ϕ | { + ∨ | - ∨ | ϕ } const }

Array References: Subscripts

array reference := array item name ∨ subscriptor

subscriptor := [∨ subscript list ∨]

subscript list := { subscript | subscript list ∨ , ∨ subscript }

subscript := arith exp

Based Arrays and the P-Function

based dec := based ∧ { array dec | begin ∧ array dec list ∨ end }

array dec list := { array dec | array dec list ∨ array dec }

p func := P < ∨ based array name ∨ >

FUNCTIONS

Function Calls

func call := { func name | bead func | loc func | p func | abs func } { arguments | ϕ }

arguments := (v actual par list v)

actual par list := { actual par | actual par list v , v actual par }

Bead Function

bead func := { B | C } < v arith exp v { , v arith exp v | ϕ } > v data

data := { item name | array reference }

Intrinsic LOC Function

loc func := loc v (v { item name | array reference | proc name | func name | switch name | label name | array name { v subscriptor | ϕ } | p func } v)

Intrinsic ABS Function

abs func := abs v (v arith exp v)

VALUE ASSIGNMENT

replacement statement := { sink / func name } v = v source v ;

exchange statement := sink v = = v sink v ;

sink := { item name / array reference / p func / bead func }

source := { arith exp / Boolean exp }

FLOW OF CONTROL

Label Declaration

label dec := label name:

labeled statement := label dec { v statement / ϕ }

SWITCH Declaration

switch dec := switch ^ switch name v switch specs v ;

switch specs := { switch list / : v status list name v switch order }

switch list := { switch point / switch list v , v switch point }

switch point := { label name / ϕ }

switch order := { order pair / switch order v , v order pair }

order pair := label name v : v status const string

GOTO Statement

goto statement := goto ∧ { label name / switch name v [v arith exp v] } { v } ;

IF Statement

if statement := if clause v statement { v else part / ϕ }

if clause := if v Boolean exp v then

else part := else v statement

FOR Statement

for statement := for clause v statement

for clause := for ∧ item name v = v loop control v do

loop control := initial value { v step part / v while part / ϕ } { v while part / v until part / ϕ }

initial value := arith exp

step part := step v arith exp

until part := until v arith exp

while part := while v Boolean exp

TEST Statement

test statement := test { ∧ / ϕ } item name { v } ;

PROCEDURES

Procedure Call Statement

proc call statement := proc name { v arguments / ϕ } { v } ;

RETURN Statement

return statement := return ∨ ;

STOP STATEMENT

stop statement := stop ∨ ;

SUBPROGRAM DECLARATIONS

subprogram dec := { proc dec | func dec }

proc dec := proc dec clause ∨ dec list ∨ statement

func dec := func dec clause ∨ dec list ∨ statement

proc dec clause := proc ∧ proc name { ∨ | (∨ formal par list ∨) | φ } ∨ ;

formal par list := { formal par | formal par list ∨ , ∨ formal par }

func dec clause := func ∧ func name { { { ∨ | (∨ formal par list ∨) } { ∧ type | φ } } { ∨ type | φ } } ∨ ;

dec list := { declaration | dec list ∨ declaration | φ }

LABELS AND PARAMETERS

Formal Label Declarations

formal label dec := label ∧ label name list ∨ ;

label name list := { label name | label name list ∨ , ∨ label name }

Formal Parameters

formal based dec := based dec

formal item dec := item dec

formal array dec := array dec

formal proc dec := fprc ∧ formal proc name ∨ ;

formal func dec := func ∧ formal func name { ∧ type / ϕ } ∨ ;

value par := (∨ formal item name ∨)

formal par := { formal based name / formal item name / formal array name / formal proc name / formal func name / label name / value par }

Actual Parameters

actual par := { item name / array name { ∨ subscriptor / ϕ } / proc name / func name / label name / arith exp / Boolean exp / p func }

ENTRIES

entry dec := entry ∧ { proc dec clause / func dec clause } ;

COMMON STATEMENT

common dec := common { ∧ common name / ϕ } ∨ ; ∨ { data dec / begin ∨ data dec list ∨ end }

data dec list := { data dec / data dec list ∨ data dec }

data dec := { item dec / array dec }

EXTERNALS

XREF (External Reference) Declarations

xref dec := xref ∧ xdec part

xdec part := { begin ∧ xdec list ∨ end / xdec }

xdec list := { xdec / xdec list ∨ xdec }

xdec := { item dec / array dec / proc heading / func heading / formal label dec / switch dec / formal switch dec / based dec }

formal label dec := label ∧ label name list ∨ ;

label name list := { label name / label name list ∨ , ∨ label name }

formal switch dec := switch ∧ switch name list ∨ ;

switch name list := { switch name | switch name list ∨ , ∨ switch name }

proc heading := proc ∧ proc name ∨ ;

func heading := func ∧ proc name { ∧ / φ } type ∨ ;

XDEF (External Definition) Declarations

xdef dec := xdef ∧ xdec part

PROGRAMS

Program Structure

program := { program head | subprogram dec } ∨ term

program head := { prgm dec | program head ∨ declaration | program head ∨ statement }

prgm dec := prgm ∧ program name ;

Compound Statements

compound statement := { compound head ∨ end | compound head ∨ spend }

compound head := { begin | spbegin | compound head ∨ statement | compound head ∨ declaration }

CONTROL Statement

control statement := { control ∧ control word ∨ ; | control ∧ conditional phrase ∨ ; | control ∧ attribute ∨ }

conditional phrase := condition word ∧ condition params

condition params := { constant | constant ∨ , ∨ constant }

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

SYMPL contains four procedures that are links to the FORTRAN execution-time PRINT routines. Linkage is the SYMPL library SYMIO.

To use these routines, a FORTRAN main program must call the SYMPL program. The PROGRAM statement of the main program must name the file used for output.

Within the SYMPL program, the procedures PRINT, PRINTFL, LIST, and ENDL must be declared as external references through the XREF statement. These procedures can also be declared and referenced by the names PRINT$, PRINTFL$, LIST$, and ENDL$ if necessary to avoid conflicts with programmer supplied procedures.

PRINT and PRINTFL are used to specify the format of items to be printed, using the specification of the FORMAT statement of FORTRAN. (PRINT differs from PRINTFL only in that PRINT is used to write to the file OUTPUT, while PRINTFL can be used to write to a file with a different logical file name.) The variables or arrays that would be presented in a WRITE or PRINT statement in FORTRAN are specified individually in SYMPL as LIST statement parameters. ENDL, which has no analog in FORTRAN, must terminate each output sequence begun by PRINT or PRINTFL.

Any errors in the format specification or output list are detected during execution by the FORTRAN routines. Any error messages generated are in the FORTRAN reference manual.

PROCEDURE PRINT

PRINT is used to format information to be written to the file OUTPUT. It initiates an output sequence. The call to procedure PRINT is:

```
PRINT (format string);
```

format string — Character string duplicating the specifications of a FORTRAN format specification, including the parentheses of the FORTRAN specification. Hollerith constants, variables, array, carriage control, spacing, or any other legal FORTRAN format specification can be included in the literal string.

PROCEDURE PRINTFL

PRINTFL is used to format information to be written to a file with a logical file name other than OUTPUT. It initiates an output sequence that must be terminated by ENDL. The call to procedure PRINTFL is:

```
PRINTFL (format string, file);
```

format string — Literal duplicating the specifications of a FORTRAN format specification, including the parentheses of the FORTRAN specification. Hollerith constants, variables, array, carriage control, spacing, or any other legal FORTRAN format specification can be included in the literal string.

file — File on which the information is to be written, expressed in terms of the file information table (FIT) for the file as shown in example 2 at the end of this appendix.

PROCEDURE LIST

LIST is used to specify an item or array to be printed. It must be preceded by a PRINT or PRINTFL procedure call and must be followed by an ENDL call. Any number of LIST calls can appear between PRINT and ENDL. Only one item or array can be specified in a single LIST call. A one-to-one correspondence should exist between the format specification in the preceding PRINT or PRINTFL call and the order in which items or arrays are named in following LIST calls. The call to procedure LIST is:

```
LIST (argument);
```

argument — Item expression subscripted array item or similar entity to be output.

PROCEDURE ENDL

ENDL terminates an output sequence begun by PRINT or PRINTFL. It must appear after the last LIST call for a given sequence; or, if no LIST calls exist in the sequence, ENDL must appear after the PRINT or PRINTFL call.

The call to procedure ENDL is:

```
ENDL;
```

EXAMPLES

1. Print VALUE OF I=nnn where nnn is the value of I in integer format. Information is to appear on a new line on the file OUTPUT, then the file is to be positioned to the next line:

```
XREF PROC PRINT;
XREF PROC LIST;
XREF PROC ENDL;
PRINT ("(1X,*VALUE OF I=*,I3,/)");
LIST (I);
ENDL;
```

The SYMPL code is equivalent to the FOTRAN statements:

```
PRINT 99,I

99 FORMAT(1X,*VALUE OF I=*,I3,/)
```

2. Print THIS IS TAPE3 on file TAPE3. Assume that the FORTRAN main program contains PROGRAM FT (OUTPUT, TAPE3):

```
XREF PROC PRINTFL;
XREF PROC ENDL$;
BASED ARRAY B; ITEM BB(0,42,18);
XREF ITEM TAPE3$;
P<B>=LOC(TAPE3$);
PRINTFL("(*THIS IS TAPE3*)",B)
ENDL$;
```

Accompanied by the COMPASS statements:

```
TAPE3$ VFD     60/=XTAPE3≡
       ENTRY   TAPE3$
```

Where TAPE3≡ is the FORTRAN convention for the name of FIT for TAPE3.

3. Output an array.

```
XREF PROC PRINT;
XREF PROC LIST;
XREF PROC ENDL;
.
.
.
CNTR=LOC(ADDR);
PRINT("(1X,O6/4O30)");
ITEM I;
FOR I=0 STEP 4 UNTIL N DO
    BEGIN
    LIST(CNTR);
    ITEM K;
    FOR K=0 STEP 1 UNTIL 3 DO
        LIST(DITM[K+1]);
    CNTR=CNTR+4;
    END
ENDL;
```

This is equivalent in FORTRAN to:

```
      CNTR=LOCF(ADDR)
      M=N-1
      DO 1 I=1,M,4
      PRINT 100,CNTR,(DITM(K+I-2),K=1,3)
    1 CNTR=CNTR+4
  100 FORMAT(1X,O6/4O30)
```

When a SYMPL program is compiled in debug mode, the compiler generates code in such a way that the program can be executed through CYBER Interactive Debug (CID). A program is compiled in debug mode when either of the following is true:

- The DB parameter is present on the SYMPL control statement.
- The control statement DEBUG,ON has been executed, putting the job in debug mode, and the SYMPL control statement does not include the option DB=0.

In order to make the CID interface possible, extra information is added to the object code by the compiler. This information includes:

- Symbol tables, which include entries for all the scalars, arrays, procedures, functions, labels, and switches in the program. No entries are made for array items.
- Line number information, to enable CID to access individual statements in the program. A line number is inserted in the object code before the executable code for each statement, and a line number loader table is provided. If more than one statement appears on a line, all the statements have the same line number.

CID treats SYMPL object code like FORTRAN object code. The features described in the CID reference manual as FORTRAN features are therefore also available to the SYMPL user. Use of SYMPL with CID is subject to the following restrictions:

- In an overlay structured program, the procedure DBUG.OM must be called from a COMPASS routine with the SYMPL calling sequence whenever a new overlay is loaded.

  A call to the procedure is also required if an already loaded overlay is being reentered and an overlay trap is to be in effect. The procedure is called with a single parameter, which is zero if the overlay in question has just been loaded, and one if an already loaded overlay is being reentered.
- CID truncates all identifiers to seven characters.
- Identifiers are not qualified. Only the outermost declaration of an identifier is recognized by CID; redeclaration at an inner level is ignored.
- All SYMPL entities referenced by CID have a type assigned to them. This type is the default type for the DISPLAY command and the PRINT command. The correspondence established is that SYMPL scalars of real type correspond to CID type F (floating point) and all other entities correspond to CID type O (octal).
- Debug does not support part word or multiword arrays. Therefore, only one identifier is used for each SYMPL array. This identifier is the array name if it has one, otherwise it is the name of the first item declared in the array. The identifier used for the array is defined to CID as a one word item occupying the first word of the array, with an offset of zero relative to the beginning of the array.
- Array bounds and subscripts are changed to FORTRAN format for referencing through the FORTRAN-only CID commands. For other CID, commands, no subscripts are used. Any dimension for which the lower bound equals the upper bound is deleted. For example, an array declared in SYMPL by the following declaration:

  ```
  ARRAY A [1:7,0:0,5:8]
  ```

  is referenced in CID as if the declaration were:

  ```
  ARRAY A [1:7,5:8]
  ```

  If the lower bound equals the upper bound for all the dimensions of the array, it is treated by CID like a scalar, since it only has one element.

  For a serial array with an entry size greater than one word, the bounds are rearranged so that the entry size is the leftmost dimension and the declared dimensions are moved one to the right. For example, if the SYMPL declaration is:

  ```
  ARRAY S [1:30] S(2);
      ITEM S0(0),
           S1(1);
  ```

  References to the array in CID are the same as if the array declaration were:

  ```
  ARRAY S [0:1,1:30];
  ```

  That is, a reference to S0[17] becomes a reference to S(0,17) (in CID, parentheses are used instead of brackets).

  For a parallel array with an entry size greater than one word, the bounds are rearranged so that the entry size is the rightmost dimension. For example, if the SYMPL declaration is:

  ```
  ARRAY P [1:100] P(3);
      ITEM P0(0),
           P1(1),
           P2(2);
  ```

  References to the array in CID are the same as if the array declaration were:

  ```
  ARRAY P[1:100,0:2];
  ```

  In CID P0[1] is referenced as P(0,1), P1[32] is referenced as P(32,1), and P2[46] is referenced as P(46,2).

- Formal parameters and based arrays must be accessed indirectly in CID. The indirect value operator ! must be used to obtain the value of these entities.

  For example, if the procedure heading is:

  ```
  PROC P(X);
  ```

  then the CID command:

  ```
  DISPLAY X
  ```

  displays the address of X, rather than its value. To display the value of X, the form:

  ```
  DISPLAY!X
  ```

  must be used.

Similary, if a based array is declared by:

```
BASED ARRAY B S(3);
    ITEM B0(0),
         B1(1),
         B2(2);
```

then the CID command DISPLAY B displays the value of the P function of B, DISPLAY!B displays the value of B0, DISPLAY!B + 1 displays the value of B1, and DISPLAY!B + 2 displays the value of B2.

The advantage of using an implementation language for system software is lost if the code is hard to read. The coding standards listed in this section represent a set of conventions that experience has shown results in readable code. The standards are divided into the following groups:

- Rules for layout of individual statements.
- Rules for format declarations.
- Rules that pertain to the format and flow of control within a whole procedure.
- Rules for efficient coding.
- Rules for comments in code.
- Rules designed to improve the transportability of code to other systems.

The rules in this appendix are recommendations, not requirements.

RULES FOR STATEMENT LAYOUT

Labels should being in column 1. Labels should appear on lines by themselves except for optional comments.

CONTROL statements which govern global conditions should be at the beginning of a compilation unit. CONTROL statements which govern local conditions should be indented with the code. Blank lines should surround all conditionally compiled code blocks.

Declarations and executable code should appear only in columns 7 through 72.

The standard indentation level is 2 columns. Indentation is recommended for IF and FOR sublevels, as well as ARRAY and COMMON declarations.

No more than one statement should appear on one line. Statements can be broken across lines to clarify the meaning.

The words BEGIN, END, THEN, ELSE, and DO should all appear on lines by themselves. They can have comments on the same line.

An IF/THEN/ELSE sequence should be written:

```
IF
...
THEN
  BEGIN
  ...
  END

ELSE
  BEGIN
  ...
  END
```

The Boolean operators of compound conditional expressions in IF statements should be indented one level more than the IF. Each condition should appear on a line by itself. Explicit parentheses should be used when AND and OR are used in the same expression. Do not assume the precedence used by the SYMPL compiler. Nested conditional expressions should be indented the standard indentation level to reflect the logical levels. For example:

```
IF RC EQ 1
  AND (READAREA GR 0
    OR READAREA LS LASTAREA)
  AND NOT RECORDING
THEN
  BEGIN
  ...
  END
```

A FOR statement should have the UNTIL or WHILE part on a separate line, indented to the next indentation level, and the DO on a line by itself aligned under the FOR. Compound conditions in the WHILE part should be indented according to the rules for compound conditions in an IF statement. For example:

```
FOR I=1 STEP 1
  WHILE NOT FINISHED
  AND (LINK [I] NQ 0
    OR MAMA [I] EQ BABY [I+1])
DO
  BEGIN
  ...
  END
```

A properly indented BEGIN/END pair should surround the body of the controlled statement of IF and FOR statements even if there is only one controlled statement.

Second and subsequent lines of statements which overflow one line should be indented at least one additional level from the level of the original statement. Assignment statements which overflow one line should be split into two statements.

Operands and operators should be separated by spaces. Left parentheses should be preceded by blanks except for intrinsic functions.

A blank line or an END should follow each unconditional transfer - GOTO, RETURN, or TEST.

The use of blank lines to improve readability is encouraged, especially surrounding block comments and after an END.

DECLARATION CONVENTIONS

Each symbol being declared should be on a line by itself, and should have a comment describing the declaration and its use. This includes all items, array items, DEF, XREF, and XDEF declarations, and status lists.

Array item declarations should be enclosed by a properly indented BEGIN/END pair and each item definition should be preceded by the keyword ITEM and end with a semicolon. An attempt should be made to align all parts of the declaration neatly. Thirteen spaces are recommended for the item name field of declarations which occur in common decks. Two digits should be used for entry, fbit, and nbit. For example:

```
ITEM FIRST              I    (00,00,60);
ITEM SECOND             B    (01,00,01);
ITEM A$LONG$NAME$       U    (01,01,29);
```

The declaration of all items should contain all specifications needed in explicit form. Each item declaration should begin with the keyword ITEM and end with a semicolon. Defaults should be avoided. For example:

```
ITEM NAME               C(10);
ITEM I                  I;
ARRAY ANY [BOUNDS]      P(ENTSIZE);
    BEGIN
    ITEM FIRST          I(00,00,60);
    END
```

All data structures and definitions, as well as procedure and function names, should be as descriptive as possible of their use. External names should be seven or fewer characters, including main program names. Item names within structures may be formed by concatenating an abbreviation of the array name and the field name; for example, the items AT$FORWARD and AT$DBPROC in the array AREA$TABLE.

All declarations should appear at the beginning of the procedure. All similar items should be grouped together; that is, all XREF declarations together, all DEF declarations together, and so forth. The following order of definitions is suggested: common decks, parameters, XDEF, XREF, DEF, status lists, COMMON, items, arrays, switches. Alphabetic ordering of items within each group is preferred. Array items should be ordered either alphabetically or numerically by entry and position within the word.

The keyword XREF and XDEF should be repeated for each declaration. The form

```
XREF BEGIN . . . (declarations) . . . END
```

should not be used.

Items whose use is strictly local to a procedure, such as induction variables or values needed across procedure calls, should be declared local to that procedure. All variable names should be unique within a deck.

CONTROL LIST and CONTROL NOLIST declarations should always be matched. A comment describing what is not being listed should always appear on the same line as the CONTROL NOLIST statement. One standard is to place the CONTROL NOLIST as the first line of the common deck and the CONTROL LIST as the last card of the common deck. CONTROL NOLIST should appear after each *CALL statement occurring in the middle of an unlisted common deck.

All declarations that are needed in more than one compilation unit should be declared in an Update common deck. All code that is needed in more than one place should be made into a procedure.

Constants should never be hard coded. Except for the use of -1, 0 or 1, where the meaning is intuitively obvious, DEF names or status declarations should be used instead. If an item can take on only a few definite values, these values and the meaning of each value should be identified in a comment. If the values are not externally defined, a status list should be used. If a status list defines external data it should be clearly documented. If a status list cannot be used then the use of DEF names for the discrete values should be considered. Special uses of status lists, such as assuming contiguous subranges of values, should be clearly documented.

Only symbolic and numeric constants should be used in DEF declarations. The use of DEF declarations to abbreviate keywords or code is discouraged.

All left-justified zero-filled strings should be declared in DEF declarations, and the name should be enclosed in dollar signs ($). The suggested format for full word octal constants is in 5-digit parcels. For example:

```
DEF $LGO$ # O"14071 70000 00000 00000" #;
```

GLOBAL CONVENTIONS

The executable code of a main program should be the last code in that program. Nested procedures should be ordered alphabetically. There should be no more than one level of nesting for procedures or functions. All procedure and function definitions should begin in column 7 and be preceded by a CONTROL EJECT. The BEGIN should also be in column 7 and should appear on the first line after the declaration. The Update deck name should be the same as the main program.

Self-modifying code should never be used.

Label names should be unique within a deck.

A branch should not be made into the middle of a FOR loop.

TEST should never be used without explicitly stating the induction variable it is testing.

An attempt should be made to keep procedures less than two pages long.

An attempt should be made to avoid splitting code blocks across a page boundary.

Sophisticated use of FOR loops, such as modifying the induction variable if it is used in a STEP clause, or modifying the STEP or UNTIL clause variables, is discouraged. For example:

```
FOR I=1 STEP J
UNTIL K
DO
    BEGIN
    . . .
    END
```

In this example, I, J, and K should not be changed within the scope of the loop. If a FOR loop with an UNTIL part can be terminated before the UNTIL condition is met, the loop should be rewritten with a WHILE and the exit from the loop should be caused by a condition in the WHILE part. Exit from any FOR loop should be caused by the UNTIL part or by a condition for the WHILE part.

A switch should be used only to imitate a CASE statement. All code referenced by the switch should be enclosed with a properly indented BEGIN/END pair, and the GOTO should precede it. The last label in the block should be an exit label. Exit from the switch should be through the exit label. Each section of code in the switch body should be self-contained; one section should not fall through to another section. A jump to another label later in the code is permitted.

GOTO should be used only when no other alternatives exist. If GOTO is used, it should be a forward jump, unless a backward jump is necessary. Jumps to the exit label of a switch are allowed when imitating a computed GOTO statement.

The use of ENTRY procedures is strongly discouraged.

EFFICIENT CODING

The use of a variable character bead function that might cross word boundaries should be avoided. For example:

```
C<I,J>CHAR[K]
```

where CHAR is longer than 10 characters.

The use of an associative bit vector should be considered in preference to a compound conditional wherever feasible. For example, instead of writing:

```
IF      CLAS [MAMA] EQ S"DATA"
   OR CLAS [MAMA] EQ S"CONS"
   OR CLAS [MAMA] EQ S"TEMP"
   OR CLAS [MAMA] EQ S"LOOP"
THEN
   ...
```

A preferable form is:

```
IF CKLOAD [CLAS[MAMA]]
   #DATA,CONS,TEMP OR LOOP#
THEN
   ...
```

DOCUMENTATION

The closing comment delimiter should always appear in column 72 of every line containing any comments.

Block comments should have the comment delimiters in columns 1 and 72 for every line in the comment. Blank lines should precede and follow the comment to separate it from the surrounding code. The comment body should be indented to the current indentation level.

Comments which are on the same line as code should start in column 36 and end in column 72. If the code extends past column 34, two blanks should separate the end of the code and the beginning of the comment. If the comment is continued to another line, the beginning of the continuation comment should start in the same column as the original comment and nothing else should appear on that line. A space should separate the comment delimiter from the body of the comment. A block comment is preferred to in-line comments that take more than two lines.

All nested procedures should begin with a block comment following the BEGIN. The block comment should contain a line with the procedure name clearly visible, and should be preceded and followed with a comment line of minus signs, as shown in Figure G-1.

```
#-------------------------#
#                         #
#  proc   name            #
#                         #
#  block comments         #
#                         #
#-------------------------#
```

Figure G-1. Block Comment Format

The block comment should contain information describing the parameters, input information, output information, and a description of important details of the processing.

A CONTROL EJECT and the block comment for the main procedure should precede the first line of executable code in the main procedure, which is the last code in the compilation unit.

A block of code should be preceded by a block comment that describes the overall action of that block. A block is any group of code that performs an identifiable macroscopic function.

At the programmer's discretion, a comment can appear on the same line as any BEGIN or END. The comment can begin two or more spaces after the keyword BEGIN or END and should describe the action of that block in one or two words. In this one case, the end-of-comment delimiter # should directly follow the body of the comment, and not appear in column 72. For example:

```
IF REGMEM[I] EQ REGI
THEN
   BEGIN   # MATCH #
   ...
   END     # MATCH #
```

A comment is recommended for the BEGIN and END of any FOR or IF body that is over one page in length. A comment with the name of the procedure is recommended for the END of the procedure.

TRANSPORTABLE CODE

All system-dependent characteristics should be isolated for easy identification and eventual change. The following characteristics are system-dependent:

- word size
- address space size
- character size
- system functions
- input/output interfaces

All system-dependent DEF declarations should be grouped in one Update common deck.

DEF declarations should be used for system-dependent declarations and code.

Good comments should accompany all system-dependent code. For example, the use of RA0 to terminate parameter lists is system-dependent.

External interfaces should be designed to be as general as possible. Users of system functions and external input/output routines should be isolated into separate routines, logically separated from the main code.

Bit functions should be used for numeric items, character functions for character strings. Type should not be mixed in bit and byte functions. Implicit conversions should not be depended on.

Character strings should be used for character information, and octal type for octal information. For example, do not use colons or semicolons for masks, and do not use octal representation for character items.

System-independent arrays should be used whenever possible.

PROGRAMMING SUGGESTIONS H

COMPILER

Space required for compilation is proportional to the number of symbols in the source program. Approximately five words of memory 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 increase compilation time and DEF expansions increase field length.

OBJECT CODE

Object code can be improved by attention to the following areas:

- Subscripts
- Arrays
- Costs of accessing data types
- FOR loops
- Data conversions
- Subprograms
- FUNC subprograms

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 three or four 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 because 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 exception 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, make the code system-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.

SUBPROGRAMS

Formal parameters should be called by value whenever possible. If a procedure must reference its formal reference 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. Reference parameters are addressed 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

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] = INT), two SAi k instructions are saved per call. The saving is realized because 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 OPTIMIZATION

The SYMPL language permits the compiler to move code to achieve optimization. SYMPL 1.2 and later versions do not perform global flow analysis. They do, however, perform many local optimizations including: compile-time computation of constant expressions, conversion of many multiplication operations to shift-and-add operations, 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. See section 5 for a more complete discussion of variable attributes.

INDEX

CONTROL DATA CORPORATION

COMMENT SHEET

MANUAL TITLE: SYMPL Version 1 Reference Manual

PUBLICATION NO.: 60496400 **REVISION:** F

NAME: _____

COMPANY: _____

STREET ADDRESS: _____

CITY: _____ **STATE:** _____ **ZIP CODE:** _____

This form is not intended to be used as an order blank. Control Data Corporation welcomes your evaluation of this manual. Please indicate any errors, suggested additions or deletions, or general comments below (please include page number references).

CUT ALONG LINE

AA3419 REV. 4/79 PRINTED IN U.S.A.

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

FOLD ON DOTTED LINES

TAPE

TAPE

FOLD

FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 8241 MINNEAPOLIS, MINN.

POSTAGE WILL BE PAID BY

CONTROL DATA CORPORATION

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

CUT ALONG LINE

FOLD

FOLD

TAPE

TAPE