60499800

CONTROL DATA CORPORATION

SYMPL VERSION 1 USER'S GUIDE

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

REVISION RECORD

REVISION	DESCRIPTION
A	Original release.
(05-20-77)	
Publication No. 60499800	

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	A
iii/iv	A
v/vi	A
vii	A
viii	A
1-1 thru 1-2	A
2-1 thru 2-8	A
3-1 thru 3-6	A
4-1 thru 4-5	A
5-1 thru 5-10	A
6-1 thru 6-6	A
7-1 thru 7-5	A
8-1 thru 8-6	A
9-1 thru 9-4	A
10-1 thru 10-2	A
A-1 thru A-4	A
B-1	A
Index-1 thru -3	A
Comment Sheet	A
Reply Env.	–
Cover	–

Page	Revision

Page	Revision

PREFACE

SYMPL version 1.2, which is a systems programming language, operates under control of the following operating systems:

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

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

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

This manual is an introduction to the SYMPL 1.2 language and its use. It neither replaces the reference manual nor advances past the reference manual in application of the language. Rather, it presents the concepts of the language in an introductory way and emphasizes good programming practices. Not all aspects of the language are treated equally. The reader is assumed to be familiar with a FORTRAN language on a CYBER 170 compatible system.

Another publication of interest:

Publication	Publication Number
SYMPL Reference Manual	60496400

CDC manuals can be ordered from Control Data Literature and Distribution Services, 8100 East Bloomington Freeway, Minneapolis, MN 55420.

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

CONTENTS

APPENDIXES

INDEX

FIGURES

TABLES

The SYMPL language is similar to the JOVIAL language, which was derived from the ALGOL-58 language. Consequently, it has many similarities with ALGOL and ALGOL-like programming languages such as PL/I, although it has features not found in these better known languages. SYMPL also has similarities with the COMPASS assembly language and the FORTRAN Extended compiler language. SYMPL statements provide Boolean and algebraic capabilities; the declarations provide the data structures of other languages.

Since SYMPL is a systems programming language, it does not include input/output facilities. When a SYMPL subprogram is called from a FORTRAN Extended main program, however, the FORTRAN language PRINT statement capabilities can be used within the subprogram for debugging purposes.

Of more significance for input and output, however, is the fact that the calling sequence conventions for FORTRAN Extended and SYMPL are alike. Both COMPASS and FORTRAN Extended routines interface easily with SYMPL.

CHARACTERISTICS OF SYMPL

The SYMPL language is characterized by:

- Reserved words
- Orientation toward manipulation of bits and 6-bit bytes as well as words
- Free-form program format, although good programming practices and coding conventions advise use of a more rigid structure
- Nonexecutable declaratives that describe data structure and use
- Nonexecutable compiler-directing statements
- Executable statements that describe procedures to be carried out
- Block structure in which the reserved words BEGIN and END delimit compound statements and declarations

Loader capabilities for interprogram communication available through SYMPL include: the COMMON declaration that produces named or blank common blocks; the XREF declaration that produces loader external references; and the XDEF declaration that produces loader entry points.

SYMPL COMPARED WITH FORTRAN

This user guide illustrates many SYMPL concepts by comparing SYMPL with FORTRAN Extended. Figures 1-1 and 1-2 show two jobs that produce the same results through FORTRAN Extended and SYMPL. Figure 1-1 shows a job deck containing a FORTRAN Extended program that generates and prints the first 10 Fibonacci numbers. (A Fibonacci number is defined as the sum of the two immediately preceding Fibonacci numbers.)

a. Job deck

```
job statement
FTN,R=0.
LGO.
7/8/9
  PROGRAM FIBON(OUTPUT)
  INTEGER L(10)
  DATA L(1),L(2) /1,1/
  LIMIT=10
  PRINT 4,LIMIT
4 FORMAT (*1FIRST *,I2,
         * FIBONACCI NUMBERS*)
  DO 1 N=3,LIMIT
  L(N)=L (N-1) + L(N-2)
1 CONTINUE
  DO 2 N=1,LIMIT
  PRINT 3,L(N)
2 CONTINUE
3 FORMAT(1H ,I10)
  STOP
  END
6/7/8/9
```

b. Output from program FIBON

```
FIRST 10 FIBONACCI NUMBERS
         1
         1
         2
         3
         5
         8
        13
        21
        34
        55
```

Figure 1-1. FORTRAN Extended Fibonacci Numbers Example

Figure 1-2 illustrates the same task as figure 1-1. Since the SYMPL subprogram uses FORTRAN output, a FORTRAN main program is required to control the environment in which the SYMPL subprogram executes. The XREF statement is necessary to allow access to library procedures which perform output. The SYMPL subprogram statements are arranged in the order of the FORTRAN program statements. Current Control Data coding standards require the subprogram to be structured as shown in figure 1-3 and to include comments.

Syntax differences between SYMPL and FORTRAN include the following SYMPL conventions:

- All statements must be terminated by a semicolon.
- BEGIN and END delimit a compound statement that can contain other elementary or compound statements.

```
job statement
FTN,R=0.
SYMPL.
LGO.
7/8/9
  PROGRAM MAIN(OUTPUT)
  CALL FIBON
  STOP
  END
7/8/9
  PROC FIBON;
    BEGIN
    XREF BEGIN PROC PRINT; PROC LIST;
                        PROC ENDL; END
    DEF LIMIT #10#;
    ITEM N I;
    ARRAY [1:LIMIT]; ITEM L=[1,1];
    PRINT("(*1FIRST *,I2,* FIBONACCI
                        NUMBERS* ,//)");
        LIST(LIMIT);
        ENDL;
    FOR N=3 STEP 1 UNTIL LIMIT DO
        L[N]=L[N-1] + L[N-2];
    FOR N=1 STEP 1 UNTIL LIMIT DO
        BEGIN
        PRINT("(1H ,I10)");
            LIST(L[N]);
            ENDL;
        END
    END
  TERM
6/7/8/9
```

Figure 1-2. SYMPL Fibonacci Numbers Example

Reserved words exist; in figure 1-2, the following reserved words are used: PROC, BEGIN, XREF, END, DEF, ITEM, ARRAY, FOR, STEP, UNTIL, DO, and TERM.

A subprogram is called by the program name itself, without a preceding CALL.

Spaces are significant and can be replaced by, or accompanied by, one or more blanks or comments.

Comments are delimited by the mark # when the ASCII character set is used or ≡ when a CDC character set is used.

Brackets delimit array subscripts.

```
PROC FIBON;
BEGIN
XREF
      BEGIN
      PROC PRINT;
      PROC LIST;
      PROC ENDL;
      END
DEF LIMIT #10#;
ITEM N I;
ARRAY [1:LIMIT];
      ITEM L = [1,1];

PRINT("(*1FIRST *,I2,* FIBONACCI NUMBERS*)");
LIST(LIMIT);
ENDL;

FOR N=3 STEP 1 UNTIL LIMIT DO
      L[N]=L[N-1] + L[N-2];

FOR N=1 STEP 1 UNTIL LIMIT DO
      BEGIN
      PRINT("(1H ,I10)");
      LIST(L[N]);
      ENDL;
      END
END
TERM
```

Figure 1-3. PROC FIBON in Recommended Program Format

Language differences between SYMPL and FORTRAN include the following SYMPL conventions:

All variables must be declared, even those used only for loop control.

External subroutines are required for output.

FOR loops can have negative step increments.

IF statements have the form IF. . . THEN. . . ELSE.

Symbolic constants are allowed.

Array items are referenced by item name, not array name.

The SYMPL language consists of reserved words, programmer-supplied words, and expressions. These, in turn, are composed of characters from the SYMPL character set. The remainder of this section discusses each of these basic language elements.

SYMPL CHARACTER SET

The SYMPL character set is limited to 55 characters:

Letters A through Z and $ ($ is considered to be a letter)

Digits 0 through 9

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

Other characters in the computer character set (appendix A) can appear in a SYMPL program only within a character constant or a comment.

Input and output of the SYMPL marks is complicated by the different character sets available on keypunches, terminals, and printers. Not all 026 keypunches have the same characters written on the top of keys; not all characters appear on key caps of either terminals or keypunches. A character that is keypunched for a constant as ' might appear on printed output as " or ≠, depending on the type of printer.

The two marks most frequently confused among the SYMPL character set are those used to delimit comments and some types of constants. For these functions, SYMPL requires the display code values of 60 and 64, respectively. Appendix A of most CYBER 170 software manuals shows the CDC standard character set that can be used to determine which keys must be used to obtain the punch combination for the required display code.

Delimiter For	Display Code Value	ASCII Graphic	CDC Graphic
Comment	60	#	≡
Character Constant	64	"	≠ or '

In this manual, an ASCII input device and an ASCII printer are assumed.

The marks + - * and / have the same meaning in arithmetic expressions as they do in other languages, with ** representing exponentiation and = = representing interchange. The marks , and . have customary meanings. The mark = represents replacement, as in FORTRAN Extended (not as in PL/I).

The marks [ ] ( ) and < > are explained below:

[ ] Balanced brackets delimit a subscript of an array.

<> Balanced angle brackets delimit arguments for the based array P function and the bead functions B and C.

( ) Balanced parentheses delimit arguments of a function, procedure, or DEF statement. They also group expressions and denote a call-by-value argument. As in other languages, parentheses can be used to improve readability of expressions or to force a specific evaluation order within expressions.

The two marks # and " must be used in pairs.

Paired pound signs delimit:

Comment

Character string of a DEF statement

" Paired quote marks delimit a character or status constant.

The remaining marks are used as follows:

; Semicolon terminates each declaration and executable statement, and most compiler-directing statements.

: Colon is used to:

Separate bounds of array dimension

Terminate a label

Define a status constant

RESERVED WORDS

SYMPL is a reserved-word language. A complete list of the 50-plus reserved words appears in the SYMPL Reference Manual. In this user guide, reserved words are introduced as the appropriate language element is described. Reserved words identify elements of the language. Examples are PROC and PRGM which signify program headers, ITEM and ARRAY which describe data items, and IF and ELSE which form part of executable statement syntax.

Words appearing in capital letters in statement formats presented in this manual are, for the most part, reserved words. A few words or letters required in some circumstances are not reserved words. Specifically, the following are not reserved, although good programming practice restricts use of these words to situations in which meaning cannot be confused:

Data descriptions: B, I, U, S, R, C.

Control words of the CONTROL compiler-directing statement, such as EJECT, NOLIST, PRESET, PACK, and IFEQ.

PROGRAMMER-SUPPLIED IDENTIFIERS

Identifiers are programmer-supplied names that are analogous to COMPASS names and FORTRAN variable names. Identifiers cannot be constructed through micro substitution or concatenation, however, as they can in COMPASS.

Identifiers must have these characteristics:

First character must be a letter or $

Contain 1 through 12 letters, digits, or $

Must not duplicate a reserved word

Although SYMPL identifiers can have 12 characters, it is good programming practice to limit identifier length to 10 characters. This restriction allows efficiencies in tables constructed by the compiler.

Examples of valid identifiers are:

I

X1

$IGN

SEMAPHORE

FIRSTBIT

Examples of invalid identifiers are shown in table 2-1.

TABLE 2-1. INVALID IDENTIFIERS

Identifier	Why Invalid
LIM	Reserved word
1AJ	Does not begin with letter or $
LAB"I"	Contains marks
↑↑00017	↑↑ are not SYMPL characters
FIRST CASE	Contains invalid blank
TEST	Reserved word
OPEN.RM	Contains mark

EXPRESSIONS

Expressions are used within statements. SYMPL expressions are similar to those of other languages in that they are sequences of identifiers, constants, or function calls separated by operators and parentheses. Two types of expressions are:

Arithmetic expressions that yield numeric values.

Boolean expressions that yield Boolean values of TRUE or FALSE.

ARITHMETIC EXPRESSIONS

Arithmetic expressions are used in replacement statements such as the following in which identifier A receives the value of the evaluated expression:

A=arithmetic expression;

An operand in an arithmetic expression can be any of the following:

Constant.

Variable defined as data type I, U, S, R, or C. Variables can be scalars (full 60-bit word for each item) or fields in an array (number of bits determined by array declaration) or parts of a scalar or field indicated by a bead function (a bead function extracts bits or characters from an array item or scalar).

Function call.

Boolean data cannot be used in arithmetic expressions.

All manipulation of variables takes place in full words, with SYMPL aligning a partial word field in a full word before performing the expression evaluation. Alignment is as shown in table 2-2. When data of different types is used in a simple expression, the system performs conversions as necessary. The SYMPL Reference Manual contains full details of conversion.

TABLE 2-2. DATA ALIGNMENT

Data Type	Alignment
C	Left-justified and adjusted to one word length. Data less than 10 characters is blank filled; data longer than 10 characters is truncated to 10 characters.
I	Right-justified with sign extension.
U, S	Right-justified.
R	Real data always occupies a full word and need not be realigned.

When character data is used in arithmetic expressions, only a single word of characters is involved. Any character data used as an integer is assumed to be an integer; the leftmost bit is the sign bit, and other bits are the integer value. No realignment takes place when character data becomes integer data. When an integer is converted to character data, however, the rightmost 6 bits of the integer are assumed to be a single character; and they are left-justified and blank filled. Any other bits in the integer are ignored. With this exception, conversions are standard for mixed data types.

The operators in an arithmetic expression can be arithmetic or logical. For the most part, character data is used only with logical operators.

Arithmetic Operators

The arithmetic operators are:

Unary operators + - and the intrinsic function ABS(exp)

Binary operators + - * / and **

A series of operators are evaluated according to FORTRAN precedence rules in which evaluation proceeds in the following order; parentheses can force a different order:

** (exponentiation)

* or / (multiplication or division)

+ or - (addition or subtraction)

When an integer is divided by another integer, the quotient is truncated without rounding. For example, the following statements produce WORD=2 and BIT=48:

```
ITEM BIT, WORD, I;
I=18;
WORD=I/10+1;
BIT=6 * (I-I/10 * 10);
```

Exponentiation is always performed in-line for all powers of two. Other small integer powers might be performed in-line, depending on compiler optimization. Integer multiplication and division by a power of two are performed in-line and are accurate to 60 bits signed.

Masking Operators

The masking operators of arithmetic expressions perform bit-by-bit operations that yield numeric values. The operators, in order of precedence, are:

LNO Complement (set 0 to 1, or set 1 to 0)

LAN Logical product (set to 1 if both bits 1)

LOR Inclusive OR (set to 1 if either or both bits is 1)

LXR Exclusive OR (set to 1 if bits are unlike)

LIM Imply (set to 1 if first operand is 0, or if first and second operands are both 1)

LQV Equivalence (set to 1 if both bits alike)

These operands work with the full word containing a scalar. More powerful masking operations result when the items are part-word array items or bead functions as described in section 5.

An example of logical product use of a 12-bit mask of zeros that sets the twelve low order bits to zero in ABCDEFGHIJ is:

```
ITEM N C(10)="ABCDEFGHIJ";
ITEM MASK2=-O"7777";
N=N LAN MASK2;
```

The following example shows exclusive OR use that sets X=0 only if A=B:

```
ITEM A,B,X;
X=A LXR B;
```

Character data used with masking operators always involves 60 bits. Shorter strings are left-justified and blank-filled; longer strings are truncated to ten characters.

BOOLEAN EXPRESSIONS

Boolean expressions are rules for determining logical values. Such expressions always yield Boolean results; that is, the result is always TRUE or FALSE. Boolean expressions are used primarily in statements that test a condition, such as:

```
IF Boolean-expression THEN . . .

FOR I=0 STEP 1 WHILE Boolean-expression DO . . .
```

A Boolean expression also can be used as the right-hand side of a replacement statement if the type of the left-hand side identifier is Boolean, as in:

```
ITEM ERRORS B, CHARCOUNT;
ERRORS=CHARCOUNT GR 7;
```

Another use of Boolean expressions is to manipulate absolute values. In the following example, SIGNE is TRUE if NUMB is less than 0. After the absolute value is obtained, if SIGNE is TRUE, NUMB is reset to negative:

```
ITEM NUMB, SIGNE B;
SIGNE=NUMB LS 0;
NUMB=ABS (NUMB);
.
.
.
IF SIGNE THEN NUMB= -NUMB;
```

Two types of operators that can be used in Boolean expressions classify the expressions:

Logical operators AND, OR, and NOT classify the expression as a logical Boolean expression.

Relational operators EQ, GR, LS, GQ, LQ, and NQ classify the expression as a relational Boolean expression.

In the following example of a relational Boolean expression, OK is TRUE if A is greater than Q:

```
ITEM A, Q, OK B;
OK=A GR Q;
```

In the following example, relational Boolean expressions and logical Boolean expressions are combined. OK is TRUE if A is greater than Q and A is not 0:

```
ITEM A, Q, OK B;
OK=(A GR Q) AND NOT (A EQ 0);
```

Logical Operators

The logical operators for Boolean expressions are identical to the FORTRAN logical operators, although they are written without the decimal point delimiters. They are implemented in SYMPL by tests such as the ZR instruction of COMPASS.

In contrast to the masking operators of arithmetic expressions, the logical operators of Boolean expressions work with one Boolean value (TRUE or FALSE) versus another, as in:

```
IDENT1 OR IDENT2
```

The Boolean logical operators, in order of highest to lowest precedence, are:

NOT	Logical negation (TRUE if neither TRUE)
AND	Logical conjunction (TRUE if both TRUE)
OR	Logical disjunction (TRUE if either TRUE)

During execution, evaluation of a Boolean expression proceeds only as long as needed to determine the result; evaluation terminates when partial evaluation satisfies the expression. For example:

```
B = (I EQ I/6*6) OR (NAME EQ "ABC");
```

If I is a multiple of 6, B is TRUE without further evaluation.

Relational Operators

The relational operators specify a comparison between two arithmetic expressions or character operands. The relational operators for Boolean expressions are equivalent to FORTRAN relational operators, although the mnemonics of the operators differ. These operators are used only with arithmetic expressions, as in:

```
IDENT3 NQ 17
```

The relational operators are:

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

During execution, character values in the arithmetic expression of a relational Boolean expression are compared in-line if neither value crosses a word boundary. If either crosses a word boundary, a call to a SYMPL library routine is compiled with attendant increase in instruction execution time.

Character strings of unequal length can be compared; SYMPL expands the shorter with blank padding to the length of the longer before comparing. For example, at the end of the sequence shown in figure 2-1, AB, BC, and AC have the value TRUE.

```
ITEM A C(2)="XX",
      B C(4)="XX  ",
      C C(6)="XX    ";
ITEM AB B, BC B, AC B;
AB=A EQ B;
AC=A EQ C;
BC=B EQ C;
```

Figure 2-1. Unequal Length Character String Example

STATEMENTS

Statements in a SYMPL program can be classified by syntax or use.

Statement use is described by the terms declaration and executable statement.

A declaration defines data or subprograms and also directs the compiler.

An executable statement specifies the operations to be carried out.

Statement syntax is described by the terms elementary and compound.

An elementary statement consists of a single language statement terminated by a semicolon.

A compound statement begins with the reserved word BEGIN; it contains zero, one, or more elementary or compound statements, and it ends with the reserved word END. One compound statement is considered to be a single statement.

Classification of a statement as compound does not affect its use; that is, a compound statement can be part or all of either a declaration or an executable statement.

Elementary statements begin with a reserved word or a programmer-supplied identifier. Examples of elementary statements are shown below. Reserved words in these examples are: PROC, GOTO, CONTROL, ITEM, IF, LS, THEN, DO, and LAN.

```
PROC FIRSTONE (A, B, C);

GOTO LABELABC;

CONTROL NOLIST;

ITEM SIZEREC I=350;

IF A LS B THEN C=D;

DO XX [I]=9-I;

P=R LAN T;

MYPROCALL;
```

Compound statements form a single unit. They can be used in most places where an elementary statement can be used. One of the most common occurrences of a compound statement is in the declaration of a procedure. (Procedures are similar to FORTRAN subroutines.) The syntax of a procedure states that a procedure is declared by a procedure header followed by optional declarations followed by a single statement. Since the single statement can be a compound statement, a procedure has virtually unlimited length. For example:

```
PROC LONGONE;                 Procedure header
BEGIN
    ITEM I, J;
    XREF ARRAY K;             Single compound
      .                       statement
      .
      .
END
```

The compound statement structure can be part of a declaration, as in:

```
XDEF
BEGIN
    ITEM A;
    ITEM B;
    ITEM C;
END
```

The same three items could be declared as externals with three elementary declarations, as in:

```
XDEF ITEM A;
XDEF ITEM B;
XDEF ITEM C;
```

In many instances a compound statement must be written to perform several operations as a single logical unit. The syntax of a FOR statement, for example, states that a single statement must follow the reserved word DO. To perform three arithmetic replacement operations with a single FOR statement, the single statement following DO must be compound, as in:

```
FOR I=4 STEP 1 UNTIL 10 DO
    BEGIN
    A=B;
    C=D;        Single compound      Single elementary
    E=F;        statement            FOR statement
    END
```

Another instance of compound statements deals with arrays. Array declaration syntax states that the one ITEM declaration immediately following the ARRAY declaration is a named item in that array. When more than one named item occurs within the array, the ITEM declaration can be a compound statement, as in:

```
ARRAY A [0:2];
    BEGIN
    ITEM AA;
    ITEM AB;
    ITEM AC;
    END
```

The same ARRAY declaration can be written using the abbreviated format for an ITEM declaration, as in:

```
ARRAY A[0:2];
    ITEM AA, AB, AC;
```

Notice that individual declarations or executable statements within a compound statement are terminated by a semicolon, including those immediately preceding END. Statements within a compound statement are written the same way as though they were outside the compound statement context. The words BEGIN and END are reserved words and are not terminated by semicolons.

DECLARATIONS

Declarations are required in a SYMPL program to define the type and use of data and to define other entities used in the program. Each declaration begins with a reserved word. Table 2-3 shows reserved words which begin declarations.

TABLE 2-3. RESERVED WORDS THAT BEGIN DECLARATIONS

Word	Use
ITEM	Defines an item, its characteristics and, optionally, its value.
ARRAY	Defines an array, its structure, and optionally, its values for direct reference.
BASED ARRAY	For indirect reference, defines an array, its structure, and optionally the value of each item in the array.
LABEL	States that a label name is used locally as a label in the case of a duplicate name outside the subprogram when the label name has not yet been declared.
STATUS	Defines names to be associated with compiler-assigned integer values.
SWITCH	Defines a list of label names to be associated with compiler-assigned integer values.
COMMON	Defines a storage block for reference by external subprograms.
PROC	Begins a procedure subprogram to be executed when the procedure is called.
FUNC	Begins a function subprogram that results in associating a single value with the function name when the function is called during execution.
ENTRY	Defines an alternate entry point for a subprogram.
XDEF	States that a subprogram, data or switch is to be accessible external to this module.
XREF	Identifies declarations defined in an externally compiled subprogram.
DEF	Defines character strings or variables to be substituted during compilation.
CONTROL	Declares actions the compiler is to take at the time the statement is executed.

Declarations and executable statements can be intermixed in a program. However, a specific requirement concerns the placement of some declarations. For example, an item must be declared before it is referenced, and a function must be declared before it is called. A procedure, on the other hand, can be called before it is declared. These differences and requirements of each declaration are explained where each declaration is discussed in depth.

The following examples show the use of various declarations:

Definition of an item with a preset value:

```
ITEM PI R=3.14159;
```

Definition of an array and its structure:

```
BASED ARRAY A [0:4,3:5] P(2);
    BEGIN
    ITEM AA C(0,0,7)=["POS=",,,,"MAX="];
    ITEM BB I(0,42,18)=[5(4)];
        .
        .
        .
    END
```

Assignment of special properties:

```
STATUS MONTH JAN, FEB, MAR, APR;
```

Definition of a storage block that can be referenced externally:

```
COMMON INFO;
```

Specification of a subprogram:

```
FUNC ROUND(INNUM);
ITEM INNUM;
ROUND=(INNUM+9)/10;
```

Identification of local labels:

```
LABEL CASE3,CASE4;
```

Character string substitution during compilation:

```
DEF OFF #0#;
```

Conditional assembly:

```
CONTROL IFEQ OPSYS,"NOS";
```

EXECUTABLE STATEMENTS

Executable statements specify the operations to be carried out within the program using the elements defined in declarations. These statements execute in the order they appear in the program, allowing for transfer of control as a result of an executing statement. A complete list of executable statements is shown in table 2-4.

STATEMENT LABELS

A label is an identifier used to name a statement. Any executable statement in a SYMPL program can be labeled. Labels on declarations refer to the following executable code.

Labels are referenced by GOTO statements, which transfer control to the named label. SYMPL has neither an assigned GO TO statement nor a CASE statement such as are available in other languages. A feature similar to the computed GO TO statement of FORTRAN is provided in SYMPL by switches.

The format of a label is:

name:

name Identifier of 1 through 12 letters, digits, or $ that does not duplicate a reserved word or another identifier in the subprogram. The colon must immediately follow the last character.

TABLE 2-4. EXECUTABLE STATEMENTS

Statement	Use
Assignment statements such as A=B+C;	Replace item to left of = with value obtained by evaluating the expression to the right of =.
Exchange statements such as D==E;	Interchange the values of D and E.
Procedure call statements such as MYPROCCALL;	Cause execution of procedure named.
GOTO statement	Transfers control to the labeled statement specified.
FOR. . .STEP. . .DO. . . WHILE/UNTIL. . . statement and its associated TEST statement	Cause repetitive execution during specified conditions.
IF. . .THEN. . .ELSE. . . statement	Conditional execution depending on circumstances specified.
RETURN statement	Ends a function subprogram or procedure subprogram.
STOP statement	Terminates program.

The example in figure 2-2 illustrates the use of a label named FINAL.

```
PROC TESTADD;
BEGIN
    ITEM A, B, C, D;
    IF A GR B
    THEN
        GOTO FINAL;
    ELSE
        C=B-A;
    FINAL: D=C;
END
```

Figure 2-2. Label Example

A LABEL declaration can be used to declare a label. In some instances, it is required. When the compiler encounters a statement that references a label, it links the reference to the last declared label name whether or not that label was in the same procedure. If the label name has not yet been used within the procedure and a duplicate label name exists outside the procedure, a LABEL declaration is required to transfer control to the correct labeled statement within the procedure.

A LABEL declaration has the following format:

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

name Label that is to be declared subsequently.

In the example in figure 2-2, if FINAL were a label name outside this procedure it would be necessary to include a LABEL declaration within PROC TESTADD:

```
PROC TESTADD;
    BEGIN
    LABEL FINAL;
        .
        .
        .
```

Since any program statement can be labeled, a label statement can be labeled. This practice is recommended for program clarity when labels have different purposes. In the example in figure 2-3, the label LL is an exit from previous code, and label MM is the beginning of a loop.

```
            .
            .
            .
        GOTO LL;
            .
            .
            .
        LL:
        MM:
            IF A GR B
                THEN
                    BEGIN
                    A=A-B;
                    COUNT=COUNT+1;
                    GOTO MM;
                    END
                ELSE
                    GOTO OUT;
            .
            .
            .
        OUT:
            .
            .
            .
```

Figure 2-3. Label Statement Label Example

A labeled statement can be simply a labeled END. Although the following statement is valid for labeling an END for usage similar to the CONTINUE statement of FORTRAN, it is not particularly useful since SYMPL offers the TEST and RETURN statements which bypass the need for such labels:

```
FIN:END
```

Comments are valid after a label, as in:

```
NEXTONE: #CONTROL REACHES HERE TO GET
          THE NEXT CHARACTER#
```

STATEMENT FORMAT

SYMPL statements can be written anywhere within columns 1 through 72 on any number of cards or card images. Unlike FORTRAN and COMPASS, SYMPL attaches no significance to any particular column of a card. The compiler treats the source program simply as a stream of characters obtained from columns 1 through 72. Card boundaries are ignored.

Because SYMPL statements are format free, both of the examples in figure 2-4 are acceptable to the compiler. The first sequence not only allows the program logic to be followed more easily, it also allows easier modifications through utilities such as UPDATE or MODIFY. The conventions of the SYMPL coding form are recommended, in which labels appear in column 1 and declarations and executable statements begin in column 7, with only one statement appearing on a single card.

```
IF I LS MAX
THEN
    BEGIN
    A[I]=A[I] + 1 ;
    I=I+1;
    END
ELSE
    GOTO FIN;
NEXT:
    ENTER(SYMBOL,TABLE [I]);



    IF I LS MAX THEN BEGIN A[I]=A[I]+1;I=I+1;
    END ELSE GOTO FIN;NEXT:ENTER
        (SYMBOL,    TABLE
        [I] );
```

Figure 2-4. Statement Format Example

COMMENTS AND SPACES

A comment is written as a string of characters delimited by a pound sign, #. Comment character strings can contain any of the computer set characters except:

Pound sign

Semicolon

A null comment that consists only of two adjacent pound signs is legal also. Anyplace a space is legal, a comment is legal, with the exception of a DEF declaration or reference. In other than DEF:

A comment can substitute for a space.

A comment can be concatenated to any legal space.

The SYMPL metalanguage described in the SYMPL Reference Manual distinguishes between instances where a space or comment is required and instances where a space or comment can, but need not, appear. In general, a space or comment is required to separate reserved words and identifiers.

Instances in which spaces or comments are prohibited are:

Within reserved words: (GOTO not GO TO)

Within status constants: (S"RED" not S "RED ")

Before the colon in a label: (SUBR: not SUBR :)

Between P, B, or C and the left angle bracket of a P function or bead function: (C<2,6>ALPHA not C <2,6>ALPHA)

Spaces or comments are allowed before and after the following marks:

Semicolon that terminates statements

Comma that separates elements of a list

Arithmetic operators or unary operators

Colon in array dimensions

Brackets in array declarations

Parentheses enclosing formal parameters in a subprogram declaration

All the following statements are valid:

```
GOTO LAB;

GOTO LAB ;

GOTO##LAB;

GOTO LAB#ORDE#;

GOTO LAB # EXIT WHEN A=B # ;
```

A SYMPL program is a series of declarations and executable statements. It can be structured as a main program or as a subprogram. Since SYMPL is a systems programming language, most source code is written in the form of a subprogram rather than a main program. The two types of subprograms are procedures and functions; they differ in that:

A function returns a value through the function name. It is called when its name is used in an expression.

A procedure 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.

A program module is a separately compiled main program or subprogram. Compilation of a module is terminated whenever the compiler encounters a TERM statement.

If a subprogram is compiled in the same program module as the program or subprogram it is called by, it requires no special treatment. If a subprogram is compiled as a separate module, however, it is known as an external subprogram and any other module referencing it must acknowledge the external subprogram status.

Separately compiled programs and subprograms can communicate by any of the following ways, as described at the end of this section:

Declaring data in labeled or blank common

Declaring entities as external

Passing arguments in a procedure or function call

Passing parameters to a procedure using common instead of using formal parameters in the procedure call might improve execution speed. Differences in object program size vary depending on whether the program sets the common variables before each call, or with formal parameters, how many transfer vectors are recognized as duplicates.

General information about procedures and functions is contained in this section, along with information about alternative entries to these subprograms. Section 7 contains the details of parameters in subprogram declarations.

MAIN PROGRAMS

A main SYMPL program consists of a main program header, a single (usually compound) statement, and the ending reserved word TERM, as shown in figure 3-1. Notice that in SYMPL neither BEGIN nor END is followed by a semicolon.

Source statements between PRGM and TERM are compiled as a single relocatable binary module with a transfer address to the first executable statement.

A main program can include any number of embedded subprograms, and those subprograms also can include embedded subprograms. If TERM appears at the end of a subprogram, it stops compilation of the module in process and source statements following TERM are compiled as a separate module.

```
PRGM name;
    BEGIN
      .
      .
      .
    declarations and statements
      .
      .
      .
    END
TERM
```

Figure 3-1. Main Program Structure

The main program header establishes the program name:

PRGM name;

name Any identifier (1 through 12 letters, $, or digits not beginning with a digit) that is not a reserved word. For loader purposes, the name is truncated to seven characters.

PROCEDURES

A procedure is a subprogram that is called when its name is referenced. The procedure can, but need not, have an associated parameter list; also it can have alternative entry points. SYMPL procedures are similar to FORTRAN subroutines. They behave as PL/I or ALGOL procedures. They can be embedded within other procedures; nesting of embedded procedures is possible to any level.

SYMPL does not support recursive procedures; a procedure should neither call itself nor be called by any procedure it has called. Responsibility for avoiding recursion rests with the programmer. The SYMPL compiler, which does not have the stack mechanisms found in ALGOL and PL/I, does not check for recursion.

Procedures and functions can be nested, as shown in figure 3-2. In this figure the nested subprogram GEN has access to all data declared in the outer procedure MYSUB.

PROCEDURE DECLARATION

A declaration establishes a procedure. It can appear anywhere in a module, even after the procedure has been referenced. It is good programming practice, however, to group all procedure declarations together at the beginning of the program preceding any executable statements.

```
PROC MYSUB;
BEGIN
ARRAY T [100] ; ITEM TT;

    PROC GEN;
    BEGIN                          }
          .                        }
          .                        }   GEN declared
          .                        }   within MYSUB
       TT [I] = 0;                 }
    END # GEN #                    }

    FUNC MAX(A,B) R;               }
    BEGIN                          }
          .                        }
          .                        }   MAX declared
          .                        }   within MYSUB
    END # MAX #                    }
.
.
.
GEN; } call to procedure GEN
.
.
.
X=MAX( Y, Z ) + MAX(V, W); } use of function MAX

GEN;
.
.
.
END # MYSUB #
```

Figure 3-2. Nested Subprograms

A procedure declaration can appear in either of the following formats:

procedure header
declarations for procedure
elementary or compound executable statement

or

procedure header
compound statement including declarations and executable statements

The usual form of a procedure includes all declarations and statements within the procedure header and the END which corresponds to the first BEGIN of the procedure. The name of the procedure is not required, but it can be included as a comment on the END statement. For example:

END#FINDIT#

The format of a procedure header is:

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

name — Any valid identifier that does not duplicate a reserved word.

param — Optional formal parameter used within the procedure for which an actual parameter is to be substituted at execution time.

A more thorough discussion of procedure declarations and parameters can be found in section 7.

A procedure declaration for a procedure SETIT is:

```
PROC SETIT (optional formal parameter list);
BEGIN
    .
    .
    .
END # SETIT #
```

The two declarations shown in figure 3-3 are legal and equivalent.

```
PROC P(A, B);
ITEM A,B;
BEGIN
ITEM I,J;
    .
    .
    .
END

PROC P(A, B);
BEGIN
ITEM A, B, I, J;
END
```

Figure 3-3. Procedure Declarations

Figure 3-4 shows a nested procedure REINITIALIZE within procedure MYSUB. Procedure REINITIALIZE can use any data of MYSUB without the need to pass that data formally. Notice that only the procedure name is used in the call; the four characters CALL do not precede the procedure name. For readability, however, many programmers use a DEF declaration (DEF CALL # #) to allow CALL in source listings.

```
PROC MYSUB;
BEGIN
ARRAY [100] ; ITEM A;
.
.
.
   PROC REINITIALIZE;                  }
   BEGIN                               }
   ITEM I;                             }
   FOR I=0 STEP 1 UNTIL SIZE DO        }   procedure
      A[I]=0;                          }   declaration
   FOR I=0 STEP 1 UNTIL LENGTH DO      }
      NAME[I]=" " ;                    }
   END # REINITIALIZE #                }
.
.
.
REINITIALIZE; } call to procedure
.
.
.
REINITIALIZE; } call to procedure
.
.
.
END # MYSUB #
TERM
```

Figure 3-4. Procedure Declaration and Call Example

Notice that procedure REINITIALIZE executes when it is called, not when its declaration is encountered. When MYSUB in figure 3-4 executes, the statements of procedure REINITIALIZE are bypassed as though the program were written as shown in figure 3-5.

```
        .
        .
        .
        GOTO BYPASS;
            PROC REINITIALIZE;
                .
                .
                .
            END # REINITIALIZE #
BYPASS:
        .
        .
        .
```

Figure 3-5. Program Execution Flow

```
        .
        .
        .
        ITEM I, J;

              PROC P;
              BEGIN
              I=I + 1;
              IF I EQ 1 THEN RETURN;
              J=1;
              END # PROC P #

        I=0;
        J=0;
        P;
        P;
        .
        .
        .
```

Figure 3-6. Procedure Exit by RETURN Statement

Procedures should be called only by the procedure name or alternative entry point name.

PROCEDURE EXIT

When control passes to a procedure, execution begins at the first executable statement associated with the entry point by which the procedure was called. Execution continues within the procedure until one of the following statements occurs:

END statement of the single procedure is reached and control returns to the statement following the procedure call.

RETURN statement within procedure is executed to return control to the calling subprogram.

STOP statement within procedure is executed to return control to the operating system.

GOTO statement is executed to transfer control to a label outside the procedure.

Exit from the middle of a procedure through a RETURN statement is illustrated in figure 3-6. Execution of procedure P occurs at its first call, after I has been assigned a value 0. After the first call, J has the value 0 since the RETURN statement executes. After the second call, which is entered with I=1, J has the value 1. A jump out of a procedure is valid, although good programming practices avoid such a jump.

In the example in figure 3-7, procedure JUMP has formal parameters N, L1, and L2. (The parentheses of N indicate call-by-value.) Since declarations for L1 and L2 do not appear within JUMP, the compiler considers them to be labels. The SWITCH declaration results in the compiler associating the identifiers of list MYSW with integer values 0 through 3, respectively. The IF statement sets N to one of these values; the GOTO statement jumps to the label associated with the value of N. The call to JUMP specifies the value of N and the particular label to be associated with switch values 0 and 3.

Section 7 describes parameters for procedure calls.

```
        .
        .
        .
        PROC JUMP( (N), L1, L2 ) ;
        BEGIN
            SWITCH MYSW L1, LAB1, LAB2, L2;
            IF N LQ 0 THEN N=0;
            ELSE
                IF N GR 3 THEN N=3;
            GOTO MYSW[N] ;
        END # PROC JUMP #
        .
        .
        .
ERRMIN: . . .
LAB1: . . .
LAB2: . . .
ERRMAX: . . .
        JUMP(I, ERRMIN, ERRMAX); . . .
```

Figure 3-7. Procedure Exit by a Jump

FUNCTIONS

A function is a subprogram used within an expression. It returns a value through its function name. This value is then used in evaluation of the expression.

The two types of functions are:

Intrinsic functions that can be referenced at any time within a program without any FUNC declaration.

Programmer-supplied functions that must be declared within a program before they can be referenced.

INTRINSIC FUNCTIONS

The five intrinsic functions are:

ABS	Absolute function that obtains the absolute value of its argument.
B	Bit function that refers to bits in the specified item.

C Character function that refers to 6-bit characters in the specified item.

LOC Location function that obtains the address of its argument during execution.

P Pointer function that refers to the pointer to a based array.

The B, C, LOC, and P functions are described in section 5.

PROGRAMMER-SUPPLIED FUNCTIONS

Functions are similar to procedures in that they can be embedded within other subprograms, and they can be declared and referenced in suitably written separate modules. Parameter lists can be passed to functions. Alternative entry to a function can be declared within a function body, as described below. Recursive functions are not valid.

Functions differ from procedures in two respects:

A function must be declared before it is referenced.

A function declaration must contain an assignment statement that assigns a value to the function name.

Function Declaration

A function declaration must appear before the function is referenced in a module. It begins with a header followed by an optional series of declarations and a function body. The function body is a single elementary or a compound statement which can include the declarations as well as other elementary or compound statements. A statement assigning a value to the function name must be included in the function body.

The format of the function header is:

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

name Any valid identifier that does not duplicate a reserved word.

param Optional formal parameter used within the function body for which an actual parameter is to be substituted at execution time.

type Type of result as described in section 4.

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

If type is omitted, I is assumed.

Within the function body, the function name can be used only as the left-hand side of a replacement statement. In the example of a function declaration in figure 3-8, notice that the statement MAX=M sets the return value, thus fulfilling the requirement that a value be assigned to the function name. A real function MAX searches for the maximum value in array T[0:N]. Within the body, a statement such as IF TAB[I] GR MAX THEN. . . or MAX=MAX + 1 is invalid since SYMPL does not allow recursion.

```
FUNC MAX (T, (N)) R;
BEGIN
    ARRAY T; ITEM TAB R;
    ITEM N I, M I;
    M=TAB[0];
    FOR I=1 STEP 1 UNTIL N DO
        IF TAB[I] GR M
        THEN M=TAB[I];
    MAX=M;
END # FUNCTION MAX #
```

Figure 3-8. Function Declaration Example

A RETURN statement can appear in the function body to return control to the calling program, as long as the function name is assigned a value before RETURN executes. RETURN is not required to end a function.

Formal parameters within the function body are subject to the same scope of declaration rules as procedures.

Function Call

A function is called when its name appears in an expression. Each of the following statements is valid, assuming a prior declaration of function MAX having two formal parameters as shown in figure 3-8:

```
I=I + MAX (VECT, 17) ;

T[MAX (VECT, 17)]=K;

P(MAX (VECT, 17), X) ;
```

Actual parameters in the call must correspond to formal parameters in the function declaration. Parameters can be passed by value or address, as described in section 7 for procedure parameters.

Function calls compile as return jump instructions, with the result normally in register X6. When the result is data type C with a string of more than 10 characters, however, register X6 contains the address of the first word of a temporary storage area containing the string.

Real and integer functions are compatible with FORTRAN Extended 4; character value functions are compatible also, if the function is declared to be 10 characters or less. The function value is returned left-justified in a word, and the unused bits are not guaranteed to contain any specific value such as zeros or blanks.

ALTERNATIVE SUBPROGRAM ENTRY

Alternative entry points can be defined for both procedures and functions. The format of the declarations are, respectively:

```
ENTRY PROC name(optional formal parameter list);

ENTRY FUNC name(optional formal parameter
               list) type;
```

Entry names can be passed as parameters and declared as externals.

An example of alternative entry is shown in figure 3-9. Procedure INIT is declared with alternative entry INCREASE. When the procedure is called with INIT, array item TAB[I] is set to 0, but when called with INCREASE, array item TAB[I] is increased by 1.

As shown with procedure INIT in figure 3-9, the parameter list in an ENTRY declaration need not match the list in the subprogram declaration. If the same parameter name does appear in two entry points, it must be the same type of variable in each. A given parameter cannot be passed by value in one list and passed by address in another list. (See section 7 for parameter details.) PROC P1(A, (B)) and ENTRY PROC P2(A, B) are illegal, since B is not referenced identically in all entries to the procedure.

```
.
.
.
ARRAY TAB[0:100]; ITEM T;
INIT(TAB, 100);
INCREASE(TAB); . . .
INCREASE(TAB); . . .
INIT(TAB, 100);
.
.
.
    PROC INIT (A, (N)) ;
    BEGIN
        ARRAY A; ITEM AA;
        ITEM N, M, I, X;
        X=-1;
        ENTRY PROC INCREASE (A) ;
        X=X + 1 ;
        FOR I=0 STEP 1 UNTIL N DO
            AA[I]=X;
    END # INIT AND INCREASE #
```

Figure 3-9. Alternate Entry Example

COMMON BLOCK DECLARATIONS

Blank common and 509 labeled common blocks can be used to pass data to separately compiled programs and subprograms. The declaration for data in a given block must be the same in all program modules. The ITEM names can differ but the specifications must be the same.

To declare common storage, the format is:

COMMON name; data-declaration

name	Label for common block. Can be expressed as any legal identifier, but only the first seven characters become the block name. If omitted, storage is allocated in blank common.
data-declaration	Scalar or array declaration. Can be expressed as a compound statement. If an array declaration is BASED ARRAY, only the pointer to the array is in common.

Data is never initialized in blank common. Data is initialized in labeled common only when one of the following conditions exists:

The program or subprogram is compiled with the P parameter on the SYMPL compiler call.

A CONTROL PRESET compiler-directing statement appears at the beginning of the program module.

Labeled common blocks are listed as part of the compiler output when either the X or R parameter is selected on the compiler call. The cross-reference map also lists labeled common names.

Good programming practices require use of meaningful names to improve readability when common is used. UPDATE common decks are particularly suitable for handling common. For example, assume the description of common block PARAMS is in an UPDATE common deck, as shown in figure 3-10. Decks that include P and Q should call the deck with PARAMS. The call to Q below has no actual parameters. Without the use of common, the declaration of Q would be Q(I1, I2, R3) and the call would take a form similar to Q(A[1],10,17.4). The use of XREF PROC Q within P is required because P and Q are separately compiled.

```
PROC P;
BEGIN
XREF PROC Q;
COMMON PARAMS BEGIN
                ITEM I1, I2;
                ITEM R3 R;
                END
.
.
.
I1=A[1];
I2=10;
R3=17.4;
Q;
.
.
.
END # PROC P #
TERM

PROC Q;
BEGIN
COMMON PARAMS BEGIN
                ITEM I1, I2;
                ITEM R3 R;
                END
.
.
.
FOR I=0 STEP I1 UNTIL I2 DO . . .
END # PROC Q #
TERM
```

Figure 3-10. COMMON Declaration Example

EXTERNAL DECLARATIONS AND REFERENCES

Any of the following SYMPL entities can be declared and referenced in separately compiled subprograms:

Scalar

Array

Based array

Label

Switch

Function

Procedure

The two SYMPL reserved words used for externals are:

XDEF Used in the declaring program to define the entity. This declaration generates an entry point that can be used by the loader.

XREF Used in the referencing program. This declaration generates an external reference to the entity. Use of XREF implies that the entity has been defined in another program.

XDEF and XREF are analogous to the COMPASS pseudo-instructions ENTRY and EXT, respectively. They are not analogous to FORTRAN Extended EXTERNAL statements.

DEFINING EXTERNALS

Storage is allocated for all entities declared with XDEF, just as if they did not have the XDEF designation. The declaration for an external definition of a scalar array, based array, or switch is simply the normal declaration preceded by the reserved word XDEF, as shown by the two examples in figure 3-11.

```
XDEF ITEM NAME C(7), MSGNUM I;

XDEF BEGIN
       ITEM NAME C(7), MSGNUM I;
       ARRAY [SIZE]; ITEM AA;
       SWITCH AUTOMAT DIGIT, LETTER, POINT,
           TEN, MARKS;
       END
```

Figure 3-11. XDEF Declaration Example

When a function, procedure, or label is declared to be external, however, the XDEF indicator is separate from the normal declaration. These three entities must be identified in two declarations:

The declaration appears in its normal format.

The XDEF indicator formats are:

```
XDEF PROC procname;

XDEF FUNC funcname;

XDEF LABEL labelname;
```

When more than one name is declared they must be contained in a compound statement. For example:

```
XDEF
BEGIN
    PROC PRGMA;
    PROC PRGMB;
END
```

The external declaration can appear anywhere within the scope of the corresponding name. XDEF is implicit in the outermost subprogram of a module and all its alternate entry points. Outermost entities should not be specifically declared external.

REFERENCING EXTERNALS

When a program references an entity that is defined and allocated storage in a separately compiled program, the referencing program must contain a declaration that states allocation exists elsewhere. No storage is allocated for an entity declared by XREF. The form of the declaration is affected by the kind of entity, but all such declarations begin with the reserved word XREF.

The declaration for a scalar, array, or label is simply the full declaration preceded by the word XREF, as shown by the two examples in figure 3-12. The declaration for a switch is XREF followed by SWITCH and the switch name, without the list of labels. For example:

```
XREF SWITCH ACTION;
```

```
XREF ITEM NAME C(7), MSGNUM I;

XREF BASED ARRAY SIZE;
    BEGIN
        ITEM LFN C(0,0,7);
        ITEM CS(0,41,18);
    .
    .
    .
    END
```

Figure 3-12. XREF Declaration Example

The declaration for a procedure is XREF followed by PROC and the procedure name. No parameters accompany the procedure name. For example:

```
XREF PROC Q;
```

The declaration for a function is XREF followed by FUNC and the function name and type. The function declaration must appear before the function is referenced. For example:

```
XREF FUNC SEARCH B;
IF SEARCH(FET) THEN GOTO ACTION[I];
```

The XREF declaration can take the form of a compound statement, as shown in figure 3-13. XDEF declarations can appear anywhere within the corresponding program. Except for procedures and labels, however, they must appear before they are referenced. All entities declared with XREF must have a corresponding entry point generated by an appropriate XDEF declaration or by being the outermost subprogram name in a module.

```
XREF BEGIN
       ITEM DATE C(8), TIME R;
       ARRAY FET;
       FUNC SUCC B;
       END
```

Figure 3-13. XREF Declaration as a Compound Statement

Data in a SYMPL program can be classified in terms of structure, type, or use.

Data structure is described by the terms scalar and array:

A scalar is a single element that occupies at least one full word of storage. A scalar is defined by an ITEM declaration.

An array is an arrangement of elements. An array is defined by an ARRAY declaration followed by either an ITEM declaration or a compound statement containing ITEM declarations. These ITEM declarations define elements within the array.

Data type is described by the terms integer, unsigned integer, real, status, character, and Boolean:

Integer, unsigned integer, and real data represent numbers in a form suitable for arithmetic. Such data is defined by a constant in an appropriate format or by an ITEM declaration with data type I, U, and R.

Status data is a variation of integer data in which the compiler substitutes integer values with names in a list. A STATUS declaration is required when data type S is specified in a declaration for a scalar or array item.

Character data is display code representation. Such data is defined by a constant in an appropriate format or by an ITEM declaration with data type C.

Boolean data can take on only the values TRUE and FALSE. Such data is defined by the constants TRUE and FALSE or by an ITEM declaration with data type B.

Data use is described by the terms arithmetic and Boolean:

Arithmetic data used in arithmetic expressions can be any type except Boolean.

Boolean data is used only in Boolean expressions. Boolean type is considered nonarithmetic.

CONSTANTS

SYMPL has five types of constants. Real, integer, status, and character data can be used in arithmetic expressions; Boolean constants can be used only in Boolean expressions. All types of constants can be used to preset a scalar or array item.

REAL CONSTANTS

Real constants are rarely used in system programming. They represent a numeric value containing a decimal point and are written in standard scientific notation with a string of decimal digits 0 through 9, a required decimal point, and optional sign. Optionally, a real constant can be written in exponential form with the characteristic and mantissa separated by the letter E. No embedded blanks are allowed. Examples of real constants are:

```
45.      0.0
98.9     6.4E+4
.4       31.415E-01
```

INTEGER CONSTANTS

Integer constants represent either a numeric value or a bit pattern. They can take the form of a decimal constant, an octal constant, or a hexadecimal constant.

The size of any integer constant is limited by the amount of storage allocated for it. Constants to be preset in an item are limited to item size. Only character type data can cross word boundaries. Constants used in expressions can be one word in size.

Decimal Constants

Decimal constants represent numeric values without a decimal point; a preceding sign is optional. They are expressed as:

decimal-integer

Decimal-integer must be a string of decimal digits 0 through 9, with no embedded blanks.

Examples of decimal integers are:

```
6        -24          4096
```

Octal Constants

Octal constants represent bit patterns, with each digit in the constant establishing 3 bits. They are expressed as:

O"octal-integer"

Octal-integer must be a string of octal digits 0 through 7; embedded blanks are ignored.

Examples of octal constants:

Octal Constant	Resulting Bit Pattern
O"777"	111111111
O"22"	010010

Hexadecimal Constants

Hexadecimal constants represent bit patterns, with each digit in the constant establishing 4 bits. They are expressed as:

X"hex-integer"

Hex-integer must be a string of 1 through 15 hexadecimal digits 0 through 9 and A through F; embedded blanks are ignored.

Examples of hexadecimal constants are:

Hexadecimal Constant	Resulting Bit Pattern
X"F"	1111
X"4BC"	010010111100

STATUS CONSTANTS

A status constant is a mnemonic for an integer that is set at compilation time by a STATUS declaration. A status constant has meaning only in conjunction with the STATUS declaration which contains the status name and status-values. A status constant is represented internally in the same way as a U data type item. Status constants are expressed as:

S"status-value"

Status-value is the name established by a STATUS declaration. Blanks are not permitted between S and the status-value; they cannot be embedded within a status-value.

Examples of status constants, assuming a STATUS COLOR RED,ORANGE,YELLOW declaration, are:

Status Constant	Value Compiled
S"RED"	0
S"YELLOW"	2

CHARACTER CONSTANTS

Character constants represent alphanumeric data with each character in the string representing 6-bit display code. They are expressed as:

"character-string"

Character-string must be a string of any characters from the computer character set. Maximum number of characters is 240. Any character " in the string must be expressed as " ".

Examples of character constants are:

```
"THIS IS A CHARACTER CONSTANT WITH
    NON-SYMPL CHARACTERS ↑% "

"THIS ONE" "S TRICKY"
```

BOOLEAN CONSTANTS

Boolean constants represent the values TRUE and FALSE. They can be used only with Boolean expressions or items declared data type B. They are expressed as:

TRUE

FALSE

SCALAR DECLARATION

The ITEM declaration defines a scalar. SYMPL scalars are similar to FORTRAN variables, but they differ in several respects. In SYMPL:

Every scalar must be explicitly declared, including those used as DO loop variables.

The scalar declaration must appear before the first reference to the scalar.

No implicit characteristics are attached to scalar names.

Values can be preset in the scalar definition.

ITEM DECLARATION FORMAT FOR SCALARS

The format of a scalar declaration is:

ITEM name type = constant;

name — Any identifier of 1 through 12 letters, $, or digits beginning with a letter or $.

type — Data type; if omitted, I is assumed.

- I — Integer in which the leftmost bit is used for the sign and the remaining 59 bits represent the binary value. The compiler allocates a full word for an integer scalar.
- U — Unsigned integer in which all bits are used to represent the value. The compiler allocates a full word for an unsigned integer scalar.
- R — Real in which data appears in the single precision floating-point format standard for CYBER 170 systems.
- C(lgth) — Character in which data appears in standard 6-bit display code format with 10 characters to a word. The compiler allocates as many words as necessary for the character string; characters in the string are left-justified in the words. The character string length must be specified. It is a decimal integer constant of 1 through 240.

B — Boolean in which data appears as zeros or ones. The compiler allocates a full word for each Boolean scalar.

S:stlist — Status in which data appears as a small integer value assigned by the compiler from the positions of identifiers in list declared by a STATUS declaration. The compiler allocates a full word for each status scalar.

constant — Initial value of scalar to be preset at load time. Format of constant should be appropriate for the data type.

PRESET CONSTANT VALUES

A preset value can be assigned to a scalar at load time. The format of the constants are:

I or U — Integer constant in decimal, octal, or hexadecimal form.

R — Real constant with a decimal point.

C — Character constant in the form "string".

B — Boolean constant TRUE or FALSE.

S — Status constant in the form S"status-value".

Preset constant values are stored as presented by the constant form in the ITEM declaration, whether or not the constant agrees with the type specified. SYMPL neither converts nor checks for agreement between the type parameter and the preset value.

Constants preset by ITEM declarations are similar to those set by DATA statements in FORTRAN in that they are initial values only. During execution the value of a preset item can be changed, and once changed, it does not revert to its preset value even if the procedure that set it is called several times. For example, if the procedure shown in figure 4-1 is called three times, the output values of I are 1, 2, and 3, assuming OUTPUT is declared externally.

```
PROC P;
BEGIN
    ITEM I=0;
    I=I + 1;
    OUTPUT (I);
END # P #
```

Figure 4-1. Preset Constant Value Example

CONTRACTED ITEM DECLARATION FORMAT

A second ITEM declaration format allows more than one scalar to be declared.

ITEM name type=constant, name type=constant, . . . ;

Each name and type pair is independent of any other pair. Syntax is the same as described above.

EXAMPLES OF SCALAR DECLARATIONS

1. These examples show scalars without preset values:

   ITEM I;

   Integer scalar assumed for identifier I in the absence of a specified type parameter.

   ITEM OPERAND B;

   Boolean scalar.

   ITEM NAME C(7);

   Character scalar with string of 7 characters.

2. Scalars can be written in contracted form:

   ITEM I, OPERAND B, NAME C(7);

   Equivalent to example 1.

   ITEM A, B, C R;

   Equivalent to ITEM A I; ITEM B I; ITEM C R; It is not equivalent to ITEM A R; ITEM B R; ITEM C R;

3. Examples of scalars with preset values appropriate for the data type:

   ITEM NUM U=0;

   Unsigned integer scalar with 60 bits used as value.

   ITEM TOTAL=0;

   Integer scalar with rightmost 59 bits used as value and leftmost bit as + sign.

   ITEM FIRST B=TRUE;

   Boolean scalar with the value 1 in a 60-bit word.

   ITEM MESSAGE C(15)= "COMPILER ABORTS";

   Character scalar with COMPILER A in first word and BORTS with trailing blank fill in second word.

   ITEM MASK12 U=O"0101";

   Unsigned integer scalar creating bits 000001000001 at the rightmost end of the word.

4. Examples of scalars with preset values that do not correspond to the data type. Presets use the constant specified, even if it does not agree with the type declared:

   ITEM ONE R=1;

   Stored as 0. . . . .01, not normalized floating point format.

   ITEM SILLY I=FALSE;

   Stored as all 0 bits.

ARRAY DECLARATION

An array is an ordered set of entries defined by two consecutive declarations:

An array header that establishes the size and structure of the array.

A single ITEM declaration that describes the fields of the array. If more than one array item is declared, the declarations can appear between BEGIN and END.

Allocation of storage for an array depends on the array header:

An ARRAY declaration results in allocation of storage.

A BASED ARRAY declaration does not result in storage allocation for the array. Rather, it defines a structure that is to be superimposed over storage allocated elsewhere in the program and allocates one word to contain the pointer. Based arrays are described in section 5.

SYMPL arrays differ from FORTRAN arrays in several respects. In FORTRAN, an array has a name by which all elements in the array are known. Individual elements, which must be one word in length, are referenced by a subscript written in enclosing parentheses. FORTRAN numbers each dimension of the array starting with 1 and limits the number of dimensions to three.

In contrast to FORTRAN, an array in SYMPL need not have a name. Elements, which need not be all the same length and can contain more than one word, are referenced by their array item name, not the array name itself. Subscripts to an array item name are written in enclosing brackets. The number of dimensions in an array is limited to seven; each dimension can have a programmer-supplied upper bound and lower bound.

SYMPL offers many capabilities for array declaration that are not available in FORTRAN. These include:

Specifying bounds of a dimension with negative values, as in:

```
ARRAY [-10:-3];
```

Specifying array elements less than one word in size, as in:

```
ARRAY[4]; ITEM A C(5), B U(0,30,3);
```

Specifying array elements more than one word in size, as in:

```
ARRAY[4]; ITEM D C(46);
```

Presetting values in the array elements, as in:

```
ARRAY[4]; ITEM NUMS=[1,2,3,4,5];
```

Specifying storage structure for multiword elements, as in:

```
ARRAY[4] S(2);
```

Although arrays can have up to seven dimensions, system programming generally does not require multidimension structures. (The multiword element capabilities of SYMPL and its serial-versus-parallel storage structures allow results that might require more than one dimension in other languages.) Consequently, the following material deals mostly with single dimension arrays. See the SYMPL Reference Manual for a description of multidimensional arrays.

The complete format for an array declaration is shown here for reference only. Section 6 presents arrays in a tutorial manner.

ARRAY name [low:up,low:up,. . .] structure (size);

name	Identifier naming the array.
low	Lower bound of a dimension of the array.
up	Upper bound of a dimension beginning at low.
structure	Structure of the array, P (parallel) or S (serial).
size	Number of words required to hold one entry of the array.

SCOPE OF DECLARATIONS

An item declared within a subprogram is valid only within that subprogram and subprograms nested within it. Statements outside the declaring subprogram cannot reference that item by name. An item declared within a nested subprogram is valid only within that subprogram and any subprogram nested within it.

An item referenced only within the subprogram in which it is declared is called a local identifier. The compiler always allocates space for a local identifier. An item declared in one subprogram and referenced in a nested subprogram is called a global identifier.

Figure 4-2 illustrates local and global identifiers. In the procedure SUBPROG:

Identifiers for items D and E are local to procedure P. They are global identifiers for procedure R which is nested within procedure P. They are unknown outside procedure P.

Identifiers for items F, G, H, and I are local to procedure Q. They are unknown outside procedure Q.

Identifiers for items A, B, and C are local to procedure SUBPROG. They are valid anywhere in the body of SUBPROG. Therefore, A, B, and C are global identifiers for procedure P, procedure R, and procedure Q.

The statement D=A+B is valid within the body of procedure R. Within procedure Q, however, the same statement is invalid since D is unknown outside procedure P.

Procedure R cannot be called at any point marked #NO#.

If an item declared in a nested subprogram has the same name as a global identifier, the compiler allocates space for both identified scalars or arrays. The innermost declaration is valid only in the procedure in which it is declared. In case of conflict, the innermost declaration always has precedence. Consequently, two declarations can specify different types of data for the same identifier.

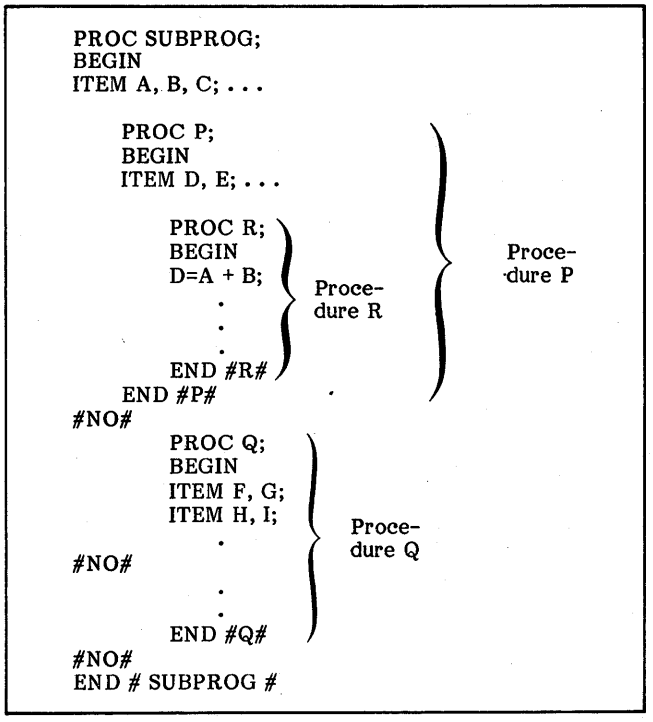


Figure 4-2. Local and Global Identifiers

In the example shown in figure 4-3, VAR is declared twice in procedure SUBPR. Since the innermost declaration has precedence, VAR in the body of procedure P is a local identifier of type Boolean and the statement VAR=TRUE is legal. The integer identifier VAR is valid anywhere in SUBPR except within procedure P. Since the two items, VAR, are in no way connected, good programming practice would be to give them unique names. In light of the systems nature of most SYMPL programs, it is also good programming practice to limit the use of global identifiers.

The subprogram in which an identifier is declared establishes the scope of declaration; the mere presence of a BEGIN . . . END sequence does not. BEGIN and END in compound statements such as FOR and IF do not affect the scope of an identifier.

```
PROC SUBPR;
BEGIN
ITEM VAR I;

    PROC P;
    BEGIN
    ITEM VAR B;
        .
        .
        .
    VAR=TRUE;
    END #P#
.
.
.
VAR=1;
.
.
.
END #SUBPR#
```

Figure 4-3. Duplicate Name Item Declarations

This section presents some of the declarations and statements that give SYMPL its power. These include:

DEF Declaration

References character strings by name for character string substitution during compilation.

SWITCH Declaration

Declares labels for a computed GOTO capability.

STATUS Declaration

Declares identifiers with implicit integer values for symbolic reference.

BASED ARRAY Declaration

Declares an array structure to be superimposed over data allocated elsewhere in program.

LOC Function

References addresses of other program entities.

Bead Functions

References part of a string of characters or bits.

DEF DECLARATION

The DEF declaration is a compiler-directing statement that allows a character string to be referenced symbolically in a program. The DEF declaration defines a name and a character-string to be substituted for subsequent occurrences of the name. The character-string, or DEF body, can be as simple as an integer constant for a DEF name used in array dimension syntax; or it can be a complete executable statement or part of such a statement.

The DEF declaration has two forms:

A DEF name without a formal parameter list resembles the COMPASS assembly language MICRO pseudo-instruction that allows symbolic reference to a character string (in SYMPL, micro delimiters are not used with micro references, however).

A DEF name accompanied by a formal parameter list resembles the COMPASS macro facility and FORTRAN statement function facility in which actual parameters are substituted for formal parameters when the DEF name is referenced.

During compilation, the character-string is substituted for occurrences of the DEF name. No computation takes place as a result of the substitution; a DEF name of ABC and a body of 3+2 results in the three characters 3+2 in place of ABC, not the single character 5.

Among the common uses of DEF are:

Improving program readability by allowing illegal words in the source listing. To allow a source statement such as IF A=0 THEN CALL ERROROUTINE:

```
DEF CALL # #;
```

Improving program execution speed by allowing in-line function code rather than the return jump execution of normal function calls:

```
DEF MODULO(X,N) #(X)-(X)/(N)*(N)#;
RANK=MODULO(LENGTH,10);
expands as RANK=(LENGTH)-(LENGTH)/(10)*(10);
```

Improving program maintainability by defining limit sizes that can be updated by future DEF changes:

```
DEF ENTRYLENGTH #3#;
ARRAY TBL[2] P(ENTRYLENGTH);
```

DEF cannot be used to redefine an identifier defined in an ITEM, ARRAY, or COMMON declaration:

```
ITEM ONE; DEF ONE #TWO#; produces ERROR nn
```

Further, DEF cannot be used to redefine the characters that serve to identify SYMPL syntax. Character substitution does not occur for the following characters used in the context of a syntax descriptor:

B, C, I, R, S, U, E, O, X, P

Substitution does occur when one of these characters is used as an identifier, nevertheless:

```
DEF C #NEW#;
ITEM ONE C(6); A=B + C;
expands as ITEM ONE C(6); A=B + NEW;
```

Similarly, DEF S #Q#; has no effect on a subsequent statement such as IF Z EQ S"SIZE". Substitution does not take place within a comment or within a character-string constant.

Any DEF declaration is effective only within the procedure or function in which it is declared, and it is effective only after the declaration appears. If a DEF name is redefined within a subprogram it does not affect the definition in an outer program. When the same DEF name is used again after leaving the subprogram, the DEF declaration in the outer program is effective.

DEF WITHOUT PARAMETERS

A DEF declaration without parameters produces straightforward expansion. The DEF format is:

```
DEF name #character-string#;
```

name — Any valid identifier by which the character-string is to be referenced.

character-string — DEF body that is to replace the DEF name during DEF expansion. From 1 to 240 characters can be used. The character # must appear as ##.

A space, but not a comment, can appear between the DEF name and the character-string.

The DEF body can contain another DEF name, as long as the definitions are not recursive or circular. The compiler checks for a single level of recursiveness only. Recursiveness obtained by nesting produces infinite loops.

DEF WITH PARAMETERS

A DEF declaration with parameters produces parameter name substitution during the expansion of the DEF body. The format for DEF with parameters is the same as without parameters, with the addition of a parameter list:

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

param — Formal parameter to be replaced by an actual parameter during DEF expansion. Must duplicate at least one identifier within the DEF body. If more than one parameter is used, they must be separated by commas.

If the number of actual parameters exceeds the number of formal parameters in the DEF declaration, a fatal error condition exists and expansion is suppressed. Expansion does occur, however, when the number of actual parameters is less than the number of formal parameters. The formal parameters without corresponding actual parameters are removed and nothing is inserted in those places. Debugging can be difficult when the number of formal parameters differs from the number of actual parameters.

During compilation, the formal parameter names within the DEF body are replaced by actual parameter names. For example, assume the following DEF declaration:

```
DEF RESET (A,N) #FOR I=0 STEP 1
    UNTIL N DO A[I]=0;#;
```

A reference RESET(T,64) produces:

```
FOR I=0 STEP 1 UNTIL 64 DO T[I]=0;
```

A DEF parameter is not recognized within a comment or a constant string. For example, assume the following DEF declaration:

```
DEF RESET (A,N) # ##SET A[2] TO "N" ## A[2]="N" #;
```

A reference RESET(T,64) produces:

```
#SET A[2] TO "N" # T[2]="N";
```

Expansion occurs in all other contexts except as a declaration name or within # or " pairs. For example, assume the following DEF declaration:

```
DEF PART (A,B,C,D) #C<A,B> C[D] = " "#;
```

A reference PART(W, X, Y, Z) produces the meaningless syntax:

```
Y<W,X> Y[Z] = " ";
```

Often, parentheses should be used within the DEF body to achieve correct results. For example, assume the following DEF declaration:

```
DEF MODULO(X,N) #X-X/N*N#;
```

A reference Y=3*MODULO(13+2, 6+2) produces:

```
Y=3*13+2-13-2/6+2*6+2; and subsequent evaluation as Y=42.
```

Yet DEF MODULO(X,N) #(X-(X)/(N)*(N))#; with the same reference produces:

```
Y=3*(13+2-(13+2)/(6+2)*(6+2)); and subsequent evaluation as Y=21.
```

Actual parameters must be separated by commas. The compiler recognizes parameters by balancing pairs of delimiters: ( ), < >, and [ ]. New parameters are not recognized within pairs of # delimiters, however. Consequently, any actual parameter that contains a comma, semicolon, right parenthesis, or an unbalanced (,), <,>, or [,], should be delimited by pairs of # marks.

The delimiting # are suppressed during expansion. For example:

All the following are valid as actual parameters:

```
F(L,N)

P((A+B)/C)

C<MODULO(N,64),J>T[K,L]

"T[K,L]"
```

Each of the following, however, must be within # pairs if they are to be used as actual parameters:

2,BYTPW	must be #2,BYTPW#
A=B; C=D;	must be #A=B; C=D;#
C<	must be #C<#

A comment can be passed as an actual parameter if the comment is enclosed by double # marks, as in:

```
DEF THREE(A,B,C) #A; B C; #
THREE(X=Y, ##SET Z TO X##, Z=X);
expands as X=Y; #SET Z TO X# Z=X;
```

A consecutive set of commas in an actual parameter string is valid to indicate an empty actual parameter, as in:

```
DEF THREE(A,B,C) #A; B C; #
THREE(X=Y, , Z=X);
expands as X=Y; Z=X;
```

SWITCH STATEMENT

A switch is a SYMPL concept that is similar to the computed GO TO statement of FORTRAN. The label to which control branches depends on the value of an expression at the time the GOTO executes. SYMPL has neither the assigned GO TO statement of FORTRAN nor the CASE statement of ALGOL.

The SWITCH declaration defines a named list of labels. The compiler associates the first label in the list with unsigned integer value 0, the second label is associated with 1, and so forth, through the list.

The SWITCH declaration is:

SWITCH swname label, label, . . . ;

swname — Identifier specifying the name of the switch.

label — Identifiers of labels to be associated with the list. Labels in the list need not have been previously declared. A label identifier can duplicate identifiers in other lists. Null positions in the list can be indicated by consecutive commas. Another switch name cannot appear in the list.

The switch name can be used only in a GOTO statement. It cannot be used in P functions or as a parameter for a function or procedure.

GOTO format is:

GOTO swname [arithmetic expression];

When GOTO executes, the expression is evaluated; control then transfers to the label whose value is equal to the value of the expression:

In the following, control transfers to label LDN when 3 is the value of I:

```
SWITCH DEVELOP TTO, ARH, SVL, LDN;

GOTO DEVELOP [I];
```

If evaluation of the expression in the GOTO statement produces a result that is beyond the values associated with the switch, execution results are unpredictable. Switch limit checking can be activated by the C parameter of the compiler call. When C is selected, an out-of-bounds reference results in a diagnostic message and execution aborts.

Within the SYMPL compiler, switches are implemented as sequential jumps. Normally, one element appears in each half of the word. Less space is consumed, but execution time is increased when switch packing is selected. Both the D parameter of the compiler call and the CONTROL PACK compiler-directing statement cause switches to be packed.

In the example in figure 5-1, the jump vector was compiled when neither the D nor the C option was selected for a declaration of SWITCH EVEN ZERO,,TWO,FOUR. With this declaration, an evaluation of I with a value of 1 creates an infinite loop.

```
EVEN    JP    ZERO
-       JP    ZERO
+       JP    *
-       JP    *
+       JP    TWO
-       JP    TWO
+       JP    FOUR
-       JP    FOUR
```

Figure 5-1. SWITCH Declaration Compilation

STATUS STATEMENT

STATUS is one of the more powerful concepts of SYMPL. The functions of STATUS can be duplicated by other programming techniques using integer values, but the simplification in programming, improvement in documentation, and advantages for program maintenance cannot be duplicated. STATUS is particularly useful in decision table and syntax analysis situations. Good programming practices call for use of STATUS whenever a set of variables is to be associated with small integer values.

STATUS is a compile-time concept similar to the EQU pseudo instruction of COMPASS. No memory is assigned to the status list mnemonics during execution.

The STATUS declaration defines a named list of mnemonics. The compiler associates the first mnemonic in the list with the unsigned integer value 0, the second mnemonic is associated with 1, and so forth, through the list. All items in the list always are referenced mnemonically.

The STATUS declaration format is:

STATUS stlist status-value, status-value, . . . ;

stlist — Name by which entire list is known, called a status list name.

status-value — Identifiers to be associated with the list. An identifier cannot be duplicated within a list. Unlike other program identifiers, however, they can duplicate the name of an identifier in any other status list or in the program, or even duplicate reserved word.

The following are equivalent:

```
ITEM A=3;

STATUS NUM  ZERO, ONE, TWO, THREE;
ITEM A S:NUM=S"THREE";
```

STATUS-VALUE REFERENCES

Status-values can be referenced in several forms, depending on the needs of a program:

A status function is used in all contexts in which the list and value must be associated.

A status constant is used in contexts in which the status list name is not ambiguous.

A status item provides convenience in referencing a scalar or array item that always takes status constant values.

A status switch is a SWITCH statement in which the switch name is associated with a status list and each label is associated with a status-value.

Status Function

A status function is actually a constant. It can be used anywhere in a program in which an integer constant can be specified, including array bounds specification and item presetting. Format is:

stlist "status-value"

During compilation the function is replaced by the appropriate value:

In the following statements, code generated during compilation presets item WHICH to the unsigned integer value 2 and assigns X=0:

```
STATUS KIND DOG, CAT, BIRD;
ITEM WHICH I=KIND"BIRD";
X=KIND"DOG";
```

Status Constant

A status constant is a shortened form for a status function. The format for a status constant, as defined in section 4, is:

S"status-value"

Because status-values are not required to be unique, the compiler must have some way to relate a status-value to the appropriate status list. This can be done by presetting the list name in an ITEM declaration, as in:

```
STATUS CLR RED, GREEN, GREY;
ITEM SHADE S:CLR;
IF SHADE EQ S"GREEN". . . .
```

where CLR is the list name and GREEN is the status-value being referenced.

A status constant can be used as loop control in FOR statements if the induction variable item has a status type. In expressions, the use of a status function or status constant is not restricted. If their meanings are not obvious, however, programming comments should be used extensively.

Status Item

If a scalar or array item usually contains a value from a particular status list, it should be defined as a status item. When this scalar or array is used in an expression with a status-value, a status constant can be used instead of a status function.

A status item is declared by a data type of:

S:stlist

Any place ITEM name U can appear in a declaration, the following can appear:

ITEM name S:stlist

Once an item of data type status is declared, status-values from the named list can be specified as status constants rather than the status functions that would otherwise be required. For example:

```
STATUS BULK ROBIN, OWL, EAGLE;
ITEM INCHES S:BULK;
ITEM WEIGHT U;
INCHES=S"OWL";
```

Without the status data type declaration for INCHES, the last statement must appear as:

```
INCHES=BULK"OWL";
```

With only the above declarations, INCHES=S"HAWK" produces a compilation error since HAWK is not a status-value from the list BULK.

Further, a statement such as WEIGHT=S"ROBIN" produces an error since WEIGHT is not a status item with ROBIN as a status-value.

A status item and status constant can be combined to preset an integer value. In the following example PAGE is set to 2:

```
STATUS SP NO, SGL, DBL, TPL;
ITEM PAGE S:SP=S"DBL";
```

Status items are not limited to status-values. Good programming practice, however, prohibits usage such as assigning a status-value to a status item for which it was not originally defined.

STATUS SWITCH

A status switch is a form of the SWITCH statement. Format is:

SWITCH swname:stlist label:status-value, label:status-value, . . . ;

swname	Switch name
stlist	Name of status list previously defined in a STATUS declaration
label	Name of label
status-value	Status-value from status list stlist that is to be associated with the preceding label

Figure 5-2 is an example of status switch use. Depending on whether NAME is alphabetic (has a display code less than 33), numeric (has a display code less than 45), or neither, NEXTCHAR is set to a certain status-value. At the end of their common processing, control transfers to the switch AUTO. If NAME is alphabetic, NEXTCHAR is set to status-value LETTER which is associated with label ALPHA, and control transfers to processing at label ALPHA. Control transfers in the same manner to NUMB if NAME is numeric, or to MARK if it is neither.

Label:status-value pairs can appear in any order. Not all status-values need be referenced in the switch. The same status-value can appear with different labels, and the same label can appear with more than one status-value. Figure 5-3 shows two examples of valid switch declarations.

EXAMPLES OF STATUS USE

The example in figure 5-4 declares status lists SOP and CLASS, which are sets of operators, with related status items initialized, respectively, to the last and first status values of the related list. The unnamed array has one element for mnemonic of status list SOP. Each array element is preset to a value that indicates whether it is an arithmetic operator, relative operator, or an error.

The following IF statement determines whether or not code at level EXP should be executed by comparing the value of a current element of CLASS with the value of a status constant acceptable for arithmetic operators:

```
IF CLASS [OP]=S "ARITH" THEN GOTO EXP;
```

```
STATUS CHAR LTR, DIGIT, OTHERS;
ITEM NEXTCHAR S:CHAR;
SWITCH AUTO:CHAR ALPHA:LTR,
                 NUMB:DIGIT,
                 MARK:OTHERS;
ITEM NAME;
          .
          .
          .
IF NAME LS 33
THEN NEXTCHAR=S"LTR";
ELSE IF NAME LS 45
    THEN NEXTCHAR=S"DIGIT";
    ELSE NEXTCHAR=S"OTHERS";
          .
          .
          .
GOTO AUTO[NEXTCHAR];
          .
          .
          .
ALPHA:
          .
          .
          .
NUMB:
          .
          .
          .
MARK:
          .
          .
          .
```

Figure 5-2. Status Switch Example

```
STATUS CHAR LTR, DIGIT, OTHERS;
SWITCH LOOP:CHAR LOOPA  DIGIT,
                 LOOPB  LTR,
                 LOOPB  OTHERS;

STATUS CHAR LTR, DIGIT, OTHERS;
SWITCH LOOP:CHAR LOOPA  DIGIT,
                 LOOPB  DIGIT,
                 LOOPC  OTHERS,
                 LOOPD  LTR;
```

Figure 5-3. Valid Status Switch Declarations

```
STATUS SOP PLUS, MINUS, EQ, LS, SOP;
STATUS CLASS BAD, ARITH, REL, COMP;
ITEM OP S:SOP=S"SOP";
ITEM KLASS S:CLASS=S"BAD";
ARRAY [SOP "SOP"];
ITEM CLASS S:CLASS = [S"ARITH",  # PLUS       1#
                      S"ARITH",  # MINUS      1#
                      S"REL",    # EQUAL      2#
                      S"REL",    #LESS THAN   2#
                      S"BAD"];   # ERROR      0#
```

Figure 5-4. Preset Status Values Example

Status constants can be used also as the loop control of a FOR statement, assuming an array TAB:

```
FOR OP=0 STEP 1 UNTIL S"SOP" DO
    TAB [CLASS [OP]]=TRUE;
```

Adding a new operator to the status list SOP in the example in figure 5-4 entails changing the STATUS declaration and adding a new element to CLASS:

```
STATUS SOP PLUS, MINUS, EQ, GR, LS, SOP;

S"REL", #GREATER THAN#
```

The IF and FOR statements in the example are not affected by the addition of the new operator. The addition was accomplished even though no empty array element was left for growth.

BASED ARRAY DECLARATION AND P FUNCTION

A based array is a structure for which no storage is allocated by the compiler. All references to items within a based array are compiled relative to the contents of its array pointer. The array pointer must be set explicitly within the program through use of the P function. Usually, the P function, for which one word is allocated, is assigned a value as the result of the LOC function reference to an array for which storage has been allocated.

In concept, a based array is a structure that can be superimposed over any portion of memory. By changing the pointer, the structure can be moved to various parts of memory. Based arrays in SYMPL (which have no similarities in COMPASS, FORTRAN, COBOL, or PL/I) provide flexibility for dealing with system programming concepts.

The declaration for a based array is the same as for a fixed array, except a name is required and preset values are not relevant:

```
BASED ARRAY name structure(size); item description;
```

name	Required name of array.
structure	Indication of parallel or serial (P or S) structure of multiword entries. Default is P.
size	Number of words in each entry. Default is 1.
item description	Description of entry in array, as described in section 4.

Several based arrays can be declared in a format:

```
BASED BEGIN ARRAY name . . . ;
            ARRAY name . . . ;
      END
```

The array dimensions can, but need not, be part of the BASED ARRAY declaration. If the subscripts of the based array and the array it is to be superimposed on need to be the same, the first element of a based array should correspond to the first element of the fixed array. SYMPL adjusts each array item reference at the time it is referenced, not at the time of the pointer setting. Accessing a based array item is slower than accessing a normal array item.

The pointer to the based array must be given a value through the P function. A P function is the name of the internal variable that contains the array pointer. It can be used the same as any variable.

The format is:

P<based array name> = arithmetic expression;

based array name	Name declared in BASED ARRAY declaration.
arithmetic expression	Arithmetic expression whose evaluation results in an address. Can be a LOC function, constant, or other expression.

On a word-addressable CYBER 70 or CYBER 170 system, any location in the program field length can be accessed as a subscripted word of a based array by:

```
BASED ARRAY ANY; ITEM X;
P<ANY>=0;
X[n]= . . .
```

The combined use of the BASED ARRAY declaration and the P function is illustrated in figure 5-5. The example assumes a file information table is allocated storage in procedure R. Procedure Q is to manipulate a file information table, with the array containing the file information table being passed as a parameter to Q. A reference to LFN[0]="MYFILE" in figure 5-5 sets the characters MYFILE in the first word of array FITNAME.

```
PROC Q(FITNAME);
BEGIN
    XREF ARRAY FITNAME;
    BASED ARRAY ALLFITS S(17);
        ITEM LFN C(0,0,7),
             RL  I(1,0,24),
             MRL I(6,0,24);
    .
    .
    .
    P<ALLFITS> = LOC(FITNAME);
    .
    .
    .
END
```

Figure 5-5. P Function Example

To superimpose the FIT structure on location 1000 octal in figure 5-5:

```
P<ALLFITS> = O"1000";
```

The P function can be used to represent the based array pointer variable in an expression. For example:

Assume the value of P<FIT> has been set to location 1000 octal. To move the based array structure 1000 octal words in memory:

```
P<FIT> = P<FIT> + O"1000";
```

A reference to LFN[0] in figure 5-5 then accesses location RA+2000 octal.

A based array can be used as a formal parameter in a procedure, though it is slow to access. The actual parameter must be a P function, not a based array name. This method is useful if the procedure is going to move the array.

For example, in figure 5-6, assume a based array A is to be used with the storage to which based array B currently points. Procedure P manipulates data known as array item. At the end of the procedure, the pointer to B must be reset to the pointer of A. Normally, the formal parameter is a fixed array, and the call passes a based array as an actual parameter.

```
PROC P(B);
BEGIN
BASED ARRAY B; ITEM . . . ;
BASED ARRAY A;
    BEGIN ITEM . . .

    END
P<A> = P<B>;
.
.
.
P<B> = P<A>;
RETURN;
END
```

Figure 5-6. Based Array as a Formal Parameter

A based array also is useful when a list is built dynamically in an area it shares with many kinds of data. For example, in figure 5-7a, assume a list in which STR points to a character string and SUC points to the next location in the string. Procedure ACTION manipulates each element of the list. The first parameter is the first element of the list; the second parameter is a procedure name (and consequently must be identified by FPRC). A call to procedure ACTION that would result in the printing of all elements of the list is shown in figure 5-7b. The subscripts can be omitted on STR and SUC because L has bounds 0:0.

a.
```
PROC ACTION ((FIRST),WHAT);
ITEM FIRST;
FPRC WHAT;
BEGIN
BASED ARRAY L;
    ITEM STR C(0,0,7), SUC I(0,42,18);
ITEM DUMY;
FOR DUMY = DUMY WHILE FIRST NQ 0 DO
    BEGIN
    P<L> = FIRST;
    WHAT (STR);
    FIRST=SUC;
    END
END #ACTION#
```

b.
```
ARRAY HEAP[1:200]; . . .
    PROC OUTPUT ((S));
        ITEM S C(7);
        BEGIN
        PRINT("(1H,A7)");
        LIST(S);
        ENDL;
    END #PROC OUTPUT#
ACTION(LOC(HEAP[N]),OUTPUT);
```

Figure 5-7. Use of a Based Array for Listing

LOC FUNCTION

LOC is an intrinsic function that returns the address of the actual argument used in the function call. The most common use of LOC is to obtain an address for a based array

pointer, but LOC is not restricted to such use. The value returned from the function is an address of type I.

The function call is:

LOC(argument)

argument Can be the 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

When LOC is used with the name of a based array as an argument, the value returned is the current value of the pointer, not the address of the pointer. When LOC is used with a P function, the address of the pointer word of the based array is returned. If the argument is an array item, the value returned is the address of the word where the item resides within the element.

For example, assume array ILFIT is a file information table declared in another module. It is accessed as a based array FIT with:

```
XREF ARRAY ILFIT; ITEM LFN C(0,0,10), . . . ;

P<FIT>=LOC (ILFIT);
```

In general, LOC should not be called with the name of a function, procedure, label, or switch, except perhaps during debugging. Although an address is returned, that address is probably not useful since no inferences can be drawn about the contents of locations surrounding the address returned. Further, the results from a particular program might not be reproducible when a different version of the compiler is used or a different optimization occurs with the same compiler version. For instance, with statements L:GOTO M; GOTO N; in a program, A = LOC(L) returns the address of label L, but A+1 does not reference GOTO N because the compiler can delete the statement and reorder the physical locations. Similarly, A = LOC(L+1) has no meaning, although the compiler does not prohibit such a statement.

One use of LOC is illustrated in figure 5-8. Assume a COMPASS main program with a 1000 word buffer at tag BUFFER. The SYMPL subprogram uses the buffer for writing, accessing the array as an XREF item. The buffer pointers are on array FET. After the first LOC function, FIRST points to BUFFER; after the second LOC function, IN points to the element BUFFER [CURRENT] which would be the last word of data written.

```
PROC WRITE;
BEGIN
XREF ARRAY BUFFER;
ITEM CURRENT;
ARRAY FET; ITEM FIRST .... IN ...,...;
.
.
.
FIRST [0]=LOC (BUFFER);
IN [0]=LOC (BUFFER [CURRENT]);
```

Figure 5-8. LOC Function Example

When file environment tables or other system interfaces are involved, SYMPL code cannot be used to monitor operating system activity. Optimization considers such data to be constant and might remove the tests from loops. See the SYMPL Reference Manual appendix C for more information.

BEAD FUNCTIONS

ITEM declarations define scalars, full words of arrays, or partial words of an array. Each time an identifier of an ITEM declaration is referenced, the entire contents of the item is accessed. At times, however, only part of an item is wanted. The bead functions (a bead is one of a string) provide access to part of an item for the purpose of extracting the contents of, or storing into, partial words.

In good programming practices, bead functions are used sparingly, since a program making frequent references to these functions is hard to maintain. Declaration and use of an array with partial-word items is preferable.

The two bead functions are:

C Access specified number of 6-bit bytes as data type character.

B Access specified number of bits as data type unsigned integer.

The two functions are not interchangeable; the C function implies the result is data type C, but the B function implies the result is data type U. The source data type is assumed to match the function, even if it is a different data type item. For example, the function B<42,18>, not the function C<7,3>, should be used to access an address in the lower 18 bits of an integer item.

Numbering conventions, which for the most part are not the conventions used elsewhere in the operating system, are as follows:

Characters and bits are numbered from 0, not from 1.

The leftmost bit is numbered 0.

The leftmost character is numbered 0.

Characters are each 6 bits.

If a bead function appears within a larger expression, SYMPL moves the specified item to a full word, aligning data as appropriate for its type. Then the result is used in the expression or replacement statement.

SYMPL does not check whether the number of beads to be extracted is within the size of the item. The programmer is responsible for the use of the function.

Bead functions can be used in the following circumstances:

In place of an item name in an expression, as in:

```
IF C<0,5> NAME EQ "INPUT" THEN R=1;
```

Left side of a replacement statement. Only the beads specified are affected, with any remaining characters untouched:

```
C<9,4> STRING=C5;
```

Right side of a replacement statement. SYMPL extracts the beads, then converts to the data type of the left side of the statement:

```
LFN=C <0,7> FITQ;
```

Parameters to a function or procedure. The function has the same properties as a subscripted variable in that it is computed and stored in temporary storage, and cannot be an output parameter:

```
CALLABC (J, C<9,1> NAME);
```

Bead functions can cross word boundaries only when the bead is extracted from a data type C item. Calls to library routines are compiled when a bead function crosses, or might cross, a word boundary, thus retarding processing. If the compiler can determine that only one word is to be accessed, the function is evaluated in-line. For example, given a data item LONG, in-line code results from:

```
C<12,3> LONG;
```

On the other hand, calls to library routines are compiled from:

```
C<I,J> LONG;
```

CHARACTER (BYTE) FUNCTION

The character function, which is also known as a byte function, extracts consecutive 6-bit characters from the specified item. The function is similar to the PL/I function SUBS. The result of a character function is data type C, with values assumed to be display code.

The format of the character function is:

```
C<start,number> identifier
```

start — Arithmetic expression indicating the first character to be extracted. Character positions are numbered from 0 at the left of the item.

number — Arithmetic expression indicating the number of consecutive characters to be accessed. The value of start+number should be within the size of the item.

If a length parameter is omitted, a single character is extracted.

If the data type of the item being accessed is C, the function can cross word boundaries and the maximum value for length is 240. (240 is the maximum number of characters allowed in a string.)

If the data type of the item is not C, however, the maximum value for length is 10.

identifier — Name of scalar or array item from which characters are to be extracted. Can be any data type, except B or S, but the result is always data type C. The extraction is done without any conversion.

EXAMPLES OF CHARACTER FUNCTION USE

1. To compare the hashed value associated with an identifier, the function shown in figure 5-9 adds the display code values of all identifier characters. The modulo 100 octal (decimal 64) is established through DEF so the subprogram could be easily modified for another modulo. The C function extracts one character at a time from the identifier. As with all functions, the name is set to the return value within the function.

```
FUNC HASH (IDENT) I;
BEGIN
DEF NCH #64#;
ITEM IDENT C(12);
ITEM I, H;
H=0;
FOR I=0 STEP 1 WHILE C<I,1> NQ " " DO
    H=H + C<I,1> IDENT;
HASH=(H)-(H)/(NCH)*(NCH);
END #HASH FUNCTION#
```

Figure 5-9. Use of C Function in a Hashing Routine

2. SYMPL limits character strings to 240 characters. Longer strings can be manipulated within a program as an array of strings. Procedure ADD, as shown in figure 5-10, adds up to 10 characters to the right end of a character string. The procedure has three parameters: the name of the array to which characters are to be added, the number of characters to be added, and the characters to be added. The first call to ADD moves three characters expressed as a constant; the second call uses a bead function to specify the location of characters to be moved.

   The example in figure 5-10 moves characters to a larger array BUFFER. The two DEF statements establish a byte number of a character within a word, and the word index of a character in a string buffer given its index I. I points to the first available character.

   The IF statement handles two conditions: the THEN clause adds characters when the characters to be added reside within a single word; the ELSE clause handles the situation when all characters are not in the same word.

   Good programming practice calls for a statement similar to DEF BYPW #10# with reference to the number of bytes per word referenced as BYPW. As stated above, the example is machine dependent.

3. An integer value between 0 and 9 can be converted to a decimal digit in display code by adding the character constant 0 to the integer. This is machine dependent, in that it depends on contiguous numbers in the character set.

   As shown in figure 5-11, function DECIMAL is a character function that converts N to a string of digits. Boolean item NEG is used to determine whether the leftmost character in working string STR is to be a minus sign. The absolute value function, ABS, is used with N prior to conversion.

```
ARRAY BUFFER [1000]; ITEM BUFWD C(10);
ITEM I;
DEF JB(I) #I-(I)/10*10#;
DEF JW(I) #I/10+1#;
ITEM LETTERS C(26)=
    "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
I=0;
ADD(BUFFER,3,"ABC");
ADD(BUFFER,10,C<3,10>LETTERS);
        .
        .
        .
    PROC ADD(SBUF,(NCHARS),(CHARS));
    BEGIN
    ARRAY SBUF; ITEM SBUFW C(10);
    ITEM NCHARS, CHARS C(10);
    ITEM I, L;
    XREF PROC ERROR;
    IF NCHARS LQ 0 OR NCHARS GR 10
    THEN
        ERROR ("ILLEGAL NCHARS");
    IF JB(I)+NCHARS LQ 10
    THEN
        C<JB(I),NCHARS>SBFW[JW(I)] =
            C<0,NCHARS>CHARS;
    ELSE
        BEGIN
        L=BYPW-JB(I);
        C<JB(I),L> SBFW[JW(I)] = C<0,L>CHARS;
        C<0,NCHARS-L> SBUFW[JW(I)+1] =
            C<L,NCHARS-L>CHARS;
        END
    I=I+NCHARS;
    END #ADD#
```

Figure 5-10. Use of C Function to Increase Character String Size

```
    FUNC DECIMAL((N)) C(20);
    BEGIN
    ITEM N;
    ITEM K, DIGIT;
    ITEM NEG B;
    ITEM STR C(20);
    STR=" ";
    IF N EQ 0 THEN BEGIN
                  C<19,1> STR="0";
                  RETURN;
                  END
    NEG=N LS 0;
    N=ABS(N);
    K=21;
    FOR DIGIT = N-N/10*10 WHILE N NQ 0 DO
        BEGIN
        K=K-1;
        C<K,1> STR = DIGIT + "0";
        N=N/10;
        END
    IF NEG THEN C<K-L,1> STR="-";
    DECIMAL=STR;
    END
```

Figure 5-11. Use of C Function for Number Conversion

BIT FUNCTION

The bit function extracts the specified number of consecutive bits from any specified item of type I, U, R, or C. The result of a bit function always is data type U, even if the bits are extracted from a different type of item.

A bit function cannot be used to obtain the absolute value of an integer. In the following example where INT contains a negative value, the result is probably a very large, positive number:

```
B<1,59>INT
```

The format of the bit function is:

```
B<start,number> identifier
```

start — Arithmetic expression indicating the first bit to be extracted. Bit positions are numbered from 0 at the left of the item.

number — Arithmetic expression indicating the number of consecutive bits to be accessed. The value of (start + number) should be within the length of the item. If a number parameter is omitted, a single bit is extracted. Number can be a maximum of 60.

If the data type of the item accessed is C, the function can cross word boundaries and the maximum value for (start + number) is 1440 (240 is the maximum number of characters allowed in a string). If the data type of the item is not C, however, the maximum value for (start + number) is 60.

identifier — Name of scalar or array item from which bits are to be extracted. Can be item of any data type except B or S, but the result is always data type U.

Bit functions extract beads from the item specified, not from the word in which the item is located. For instance, assuming a one-word array as shown in figure 5-12, B<29,2> BB[I] extracts bits 29 and 40 from item BB, which are bits 39 and 40 of the full word WW[I]. B<30,10> WW[I] is equivalent to XX[I].

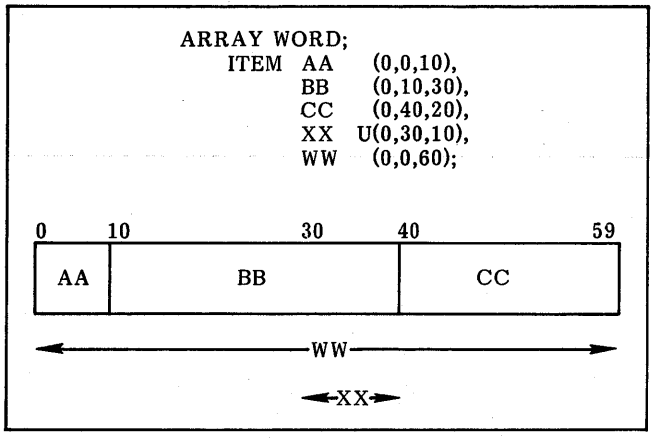


Figure 5-12. Bit Function Example A

EXAMPLES OF BIT FUNCTION USE

The first two examples show machine-dependent code.

1. An infinite operand if hardware compatible with CYBER 170 hardware (coefficient of 4000 or 3777 depending on a positive or negative operand) can be checked by:

```
DEF INF(X) # B<0,12>X EQ O"4000"
    OR B<0,12>X EQ O"3777" #;
```

2. An octal number in item N is converted to a string to be displayed as shown in figure 5-13.

```
FUNC DISPLAY ((N)) C(20);
BEGIN
ITEM N;
ITEM I, DIS C(20);
FOR I=0 STEP 1 UNTIL 19 DO
    C<I> DIS=B<I*3,3> N+"0";
DISPLAY=DIS;
END
```

Figure 5-13. Bit Function Example B

3. An integer variable from status list OPTION is used as a Boolean array as shown in figure 5-14. The item SET might be flag bits.

```
STATUS OPTION LIST, MAP, CROSS, TRACE;
ITEM SET;
DEF OFF # EQ 0 #;
DEF ON # EQ 1 #;
.
.
.
B<OPTION"TRACE"> SET=1;
.
.
.
IF B<OPTION"MAP"> SET ON AND
    B<OPTION"CROSS"> SET OFF THEN . . .
```

Figure 5-14. Bit Function Example C

4. Fields within an array item T are accessed by a subscripted array reference, even though the array is declared to have full-word items. The bead function defines three 20-bit fields in each array item. The example in figure 5-15 sets the pseudo-array item TT(10) to 0.

```
ARRAY [26]; ITEM T;
DEF TT(I) # B<( (I) - (I) / 3*3) * 20, 20> T[I/3] #;
ITEM A, B;
    A=6;
    B=4;
    TT (A + B)=0;
```

Figure 5-15. Bit Function Example D

MULTIWORD AND PART-WORD ARRAYS 6

An array is declared by an array neader followed by an item declaration describing the named entity in that array. In section 4 the declaration was shown with the format:

```
ARRAY name[dimension bounds];

    ITEM name type;
```

For example, ARRAY[9]; ITEM ENTRY; defines an array of 10 entries, each one word in size. A reference to ENTRY [2] obtains the third word in the array. The total number of entries must be less than 65535. SYMPL arrays are not limited to one-word entries, however.

Assume an entry of 30 characters occupying 3 words. The array would be declared:

```
ARRAY [9] P (3);
    ITEM THREEWDS C(0,0,30);
```

A reference to THREEWDS[2] then obtains the third entry in the array, which is 30 characters long.

Suppose the entry has three words of related, but not identical, items. Rather than a 30-character string, an entry might consist of three separate items: a header word, an identifier name, and a pointer. The first and last words are data type integer, while the middle word is data type C. One means of describing this situation is through three arrays, but such declarations neither show the relationships between arrays nor offer the convenience of passing a single parameter to a subprogram. By using multiword array entries, the relationship can be maintained. To describe the three words suggested above:

```
ARRAY ALLTHREE[9] S(3);
    ITEM HEAD I(0),
         IDENT C(1,0,10),
         PTR I(2);
```

Subscripts (0), (1), and (2) determine the word in the entry in which the named item is to appear.

A reference to IDENT[9] picks out the second word of the last occurrence of the entry. Because the array is described with an S instead of a P, IDENT[9] occupies the next to the last word of storage allocated for that array. Use of ALLTHREE as a parameter to a procedure makes all occurrences of HEAD, IDENT, and PTR available to that procedure. The called procedure must have a formal parameter array with fields defined the same.

As an alternative to having more than one word, an entry can have less than one word. Consider a list of characters A through Z right-justified and zero-filled such as the FORTRAN compiler establishes for an array described by DIMENSION ALPHA(26) and DATA/ALPHA/. This array could be declared:

```
ARRAY[25] P(1);
    ITEM ALPHA C(0,54,1) =
        ["A", "B", "C", . . . "Z"];
```

A reference to ALPHA[10] obtains the 6-bit letter K, not 60 bits.

Further, one physical word of the array can have more than one item defined within it. Consider the 26 letters left-justified in the left-hand side of the word, with an address in the lower 18 bits of the word. The array with such an entry could be described:

```
ARRAY WHERE[25] P(1);
    ITEM ALPHA C(0,0,1),
         ADDR U(0,42,18);
```

Notice that the two item descriptions define only the leftmost 6 bits and the rightmost 18 bits of the word. Array item descriptions need not account for all bits in a word.

One further capability, overlapping, is possible in SYMPL arrays. A given bit in a word can be defined as part of more than one item. Suppose, in array WHERE above, the program needed to access the entire word, not just a part of a word. A third item, integer ALLOFIT, can be declared as shown in figure 6-1a.

Efficient overlapping of Boolean items can provide for testing many conditions with one elementary statement. In the example shown in figure 6-1b, testing B3 combines the tests on B1 and B2. B3 is true if either B1 or B2 is true. Testing is accomplished with one IF statement. The overlapping feature gives SYMPL capabilities achieved in FORTRAN by the EQUIVALENCE statement and in COMPASS by EQU.

a.
```
ARRAY WHERE[25] P(1);
    ITEM ALPHA C(0,0,1),
         ADDR U(0,47,18),
         ALLOFIT;
```

b.
```
ARRAY [10];
    ITEM B1 B(0,0,1),
         B2 B(0,1,1),
         B3 B(0,0,2);
    IF B3 THEN GOTO FINAL;
```

Figure 6-1. Item Overlapping

The only limits in combining multiword and part-word item descriptions are governed by physical word size:

All character data must be aligned at 6-bit boundaries.

Only items of data type C can cross word boundaries.

Items of data types other than C must be restricted to word boundaries.

COMPLETE ARRAY DECLARATION SYNTAX

The array discussion in section 4 and the examples above made use of abbreviated forms of array declarations. The

full array declaration, which allows multiword, fullword, or part-word entries to be combined in one of two storage formats is:

ARRAY name [low:up,low:up, . . .] st (size);

name Identifier specifying the name of the array. It can be omitted unless the array is referenced in an XDEF, XREF, or BASED ARRAY declaration or a LOC function. No type is associated with the name.

low Lower bound of a dimension of the array. Must be expressed as an integer constant. Any value, including a negative value, can be specified, although a value of 0 offers execution efficiencies. If omitted, a value of 0 is assumed and the following colon must also be omitted.

up Upper bound of a dimension of the array. Must be expressed as an integer constant. Can be positive or negative. Must be equal to or greater than the preceding low with which it is paired.

st Structure of the array in storage:

- S Serial in which all the words of one element are allocated contiguously.
- P Parallel in which the first words of each entry are allocated contiguously, followed by the second word of each entry, and so forth.

size Number of words required to hold one entry, expressed as an unsigned integer. If size is omitted, 1 is assumed. Size must be less than 2048 words.

The ITEM declaration must immediately follow the ARRAY declaration. The format of the ITEM declaration for an array is:

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

name Name of element in the array.

type Type of element: I, U, R, C(lgth), B, or S:stlist. If omitted, I is assumed.

ep Entry position. Word number within the element where the high-order bit of the item occurs, starting from 0. Must be expressed as an unsigned integer constant. If omitted, 0 is assumed.

fbit First bit. Beginning position of item within the word ep, counting from 0 on the left. Must be expressed as an unsigned integer constant. For a character item, character bit position 0,6,...,54. For other type items, bit number 0 through 59. If omitted, 0 is assumed.

size Item length expressed as an unsigned integer constant. For a character item, length is the number of characters not to exceed 240. For other type items, length is in bits not to exceed 60. R type data must have a size of 60. If size is omitted, 1 is assumed for Boolean and character, and 60 for all other data types. Only C type data can cross word boundaries.

preset Initial value for the item expressed as a series of values. The values must be arranged in the same order as the allocation of storage order and separated by commas. If omitted, no values are preset at load time.

If the entire field descriptor (ep,fbit,size) is omitted, defaults are as described above. If one parameter appears within the parentheses, it is assumed to be ep; two parameters are assumed to be ep and fbit.

The type indicator for an array item must specify the position the item occupies in the entry. For 60-bit items of type I, U, B, R, or S, position information can be abbreviated, but the word position is required. The three examples shown in figure 6-2 are all valid for an array declared as ARRAY LOOK[100] (3).

```
ITEM A    (0),
     B    B(1,0,60),
     C    C(2);


BEGIN
ITEM A    I(0,0);
ITEM B    B(1);
ITEM C    R(2,0,10);
END


ITEM B    B(1),
     C    C(2,0),
     A     (0,0,60);
```

Figure 6-2. Array Item Declarations

Notice that for data type C the format is not the same as for a scalar. For a 12-character item:

Scalar C(12)

Array item C(ep, fbit, 12)

An example of part-word items in an array with single-word entries is shown in figure 6-3.

```
ARRAY DESCRIPTOR S (1) ;
ITEM    N       B(0,0,1),
        V       B(0,1,1),
        A       B(0,2,1),
        KIND    U(0,27,5),
        TYPE    U(0,32,5),
        SCOPE   U(0,37,5),
        ADDR    I(0,42,18);
```

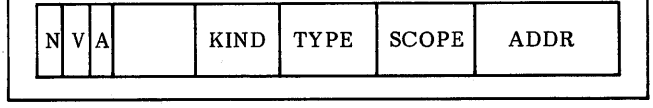


Figure 6-3. Array With Part-Word Items

When more than one item references the same field in the same word, the programmer is responsible for the results when data types are not alike. In the example in figure 6-4, INT[1] and CHAR[1] refer to the same word, but the types are different. Because of the differences in data type, different results are obtained from an assignment statement:

INT[1]=1 Sets right-justified integer 1.

CHAR[1]=1 Sets left-justified character A.

```
ARRAY CONST[24] S(3);
BEGIN  ITEM IDENT C(0,0,8);
       ITEM KIND  I(1);
       ITEM INT   I(2);
       ITEM CHAR  C(2,0,10);
END
```

Figure 6-4. Duplicate Field Item References

PARALLEL AND SERIAL ARRAYS

The capability to control array storage allocation is one of the outstanding features of SYMPL. For arrays with one-word entries, the distinction between P and S is meaningless. When multiword arrays exist, however, a parallel structure can decrease execution time.

For serial arrays, all words of the entry appear together. This is the normal structure for arrays such as the file name table in central memory resident or a FORTRAN double precision or complex array where entry size would be 2.

For parallel arrays, only entry words with the same entry position appear together. The structure can be visualized as an array of word [0] followed by an array of word [1], and so forth.

During execution, subscript calculations are faster for parallel arrays. Consequently, parallel arrays should be specified whenever possible. To access ONE[I], for instance, requires calculation of:

Parallel Address of ONE[0] + I

Serial Address of ONE[0] + 3 * I

Assuming an array SHOWIT with a 3-word entry containing three integer items, the different storage structures for serial and parallel allocation are shown in figure 6-5.

When an array item contains character data of more than 10 characters, the serial and parallel allocations still pertain to each 10-character word of the item within each word of the entry. Only items of data type C can cross word boundaries. Figure 6-6 compares serial and parallel allocation when multiword items are declared. For serial entry S, the two words of S[1] are contiguous. All entries with subscript [1] appear before any entries with subscript [2]. For parallel entry S, the two words of S[1] are not contiguous. All of entry S appears before any of entry T. The parallel structure is maintained within each entry.

PRESETTING ARRAYS

The first 6000 words of an array can have preset values. An array item is initialized by a string that contains one value for each occurrence of the item within the array. Presetting occurs for each item. The array declaration specifies what an item should be, not what a word should be. For example:

For a three-word array with one item in which occurrences are to be initialized to increasing integer values:

```
ARRAY LOOK [2];
   ITEM ONE=[0,1,2];
```

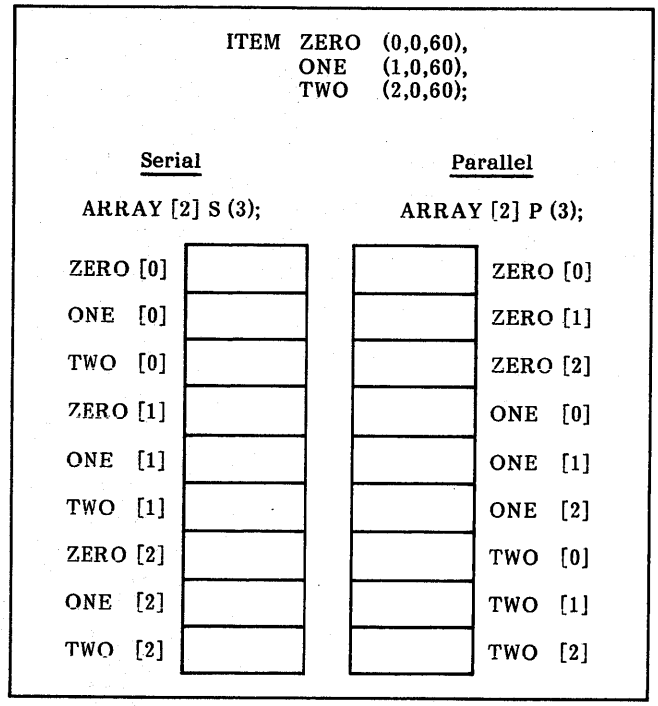


Figure 6-5. Serial and Parallel Allocation

A three-word array, SHOW, with two items in a single word, in which the occurrences of the first item are to be 1, 2, 3 and the second item A, B, C, is shown in figure 6-7. Bits 30 through 53 are undefined in the resulting array.

The string of values can be specified in abbreviated form, depending on the program needs:

If not all occurrences are to be initialized, a null value must be established by consecutive commas.

Trailing commas can be omitted.

If all occurrences are to be preset to the same value, an abbreviated format can be specified. To set ONE to all 0, for example:

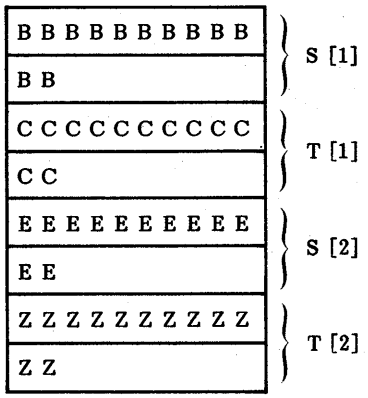
ONE=[3(0)]

An example of array presetting when not all occurrences are to be initialized is shown in figure 6-8. The elements without preset values are undefined, not zero. An example of array presetting with character data when not all occurrences are to be initialized is shown in figure 6-9.

If a number of occurrences are not to be initialized, an abbreviated format can be specified. For example, [10( ),1] is equivalent to [,,,,,,,,,,1].

Serial

```
ARRAY  SER[1:2] S (4);
ITEM   S C(0,0,12)=["BBBBBBBBBBBB",
                    "CCCCCCCCCCCC"],
       T C(2,0,12)=["EEEEEEEEEEEE",
                    "ZZZZZZZZZZZZ"];
```



Parallel

```
ARRAY  PAR [1:2] P (4);
ITEM   S C(0,0,12)=["BBBBBBBBBBBB",
                    "CCCCCCCCCCCC"],
       T C (2,0,12)=["EEEEEEEEEEEE",
                    "ZZZZZZZZZZZZ"];
```

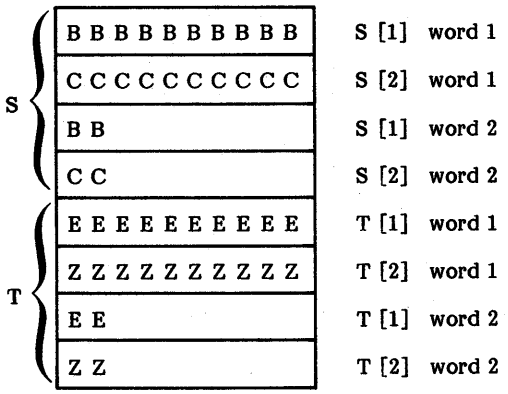


Figure 6-6. Serial and Parallel Allocation of Multiword Items

Presetting occurs without any check of array size. As a result, overlapping of values can occur. No error occurs if array bounds are exceeded. In the example in figure 6-10a, item X is initialized here only because of its position in relation to T. The compiler does not guarantee that array X immediately follows array T unless these declarations are within a COMMON block.

In the example in figure 6-10b, the preset values for BB, and then CC, overlap the preset values for AA. DD is initialized to the preset values of CC. This order is not guaranteed unless these declarations are in COMMON.

```
ARRAY SHOW[2];
ITEM FIRST (0,0,30)=[1,2,3],
     SECOND C(0,54)=["A", "B", "C"];
```

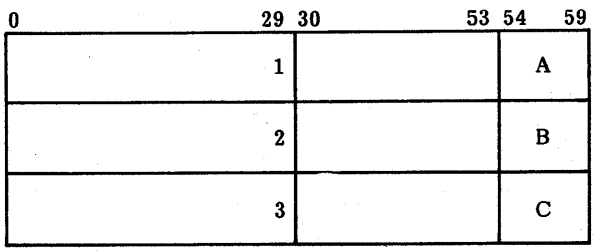


Figure 6-7. Array Presetting Example A

```
ARRAY [10];
ITEM T=[,2(1,,2),,,3];
```

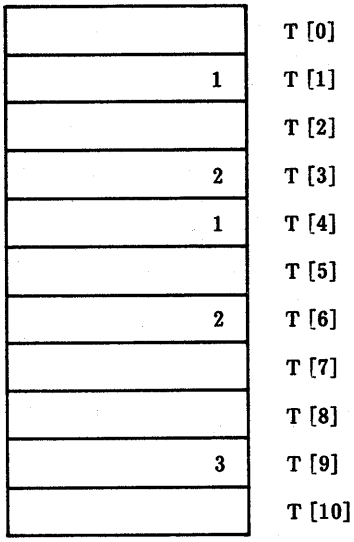


Figure 6-8. Array Presetting Example B

```
ARRAY [3] S (2);
ITEM H C(0,0,12) = ["T", ,"EEEEEEEEEEEE"];
```

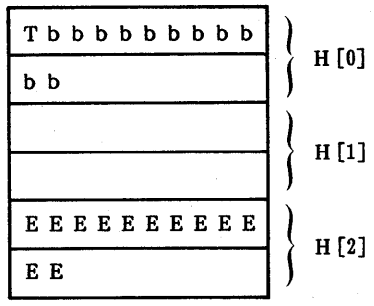


Figure 6-9. Array Presetting Example C

a. ARRAY [1]; ITEM T=[1,2,3,4,5];
ARRAY [4]; ITEM X;

1	T [0]
2	T [1]
3	X [0]
4	X [1]
5	X [2]
	X [3]
	X [4]

b. ARRAY A;
ITEM AA=[1,2,3,4,5];
ARRAY B;
ITEM BB=[6,7,8,9,10];
ITEM CC=11;
ARRAY D [1:3];
ITEM DD;

1	AA [0]
6	BB [0]
11	CC
8	DD [1]
9	DD [2]
10	DD [3]

Figure 6-10. Array Presetting Example D

Arrays can be initialized through status constants as well as the more common integer, Boolean, or character constants. Assume an array with two one-word entries as shown in figure 6-11. The second entry is associated with a status value from status list STAN.

Presetting of items that occupy only part of a word is the same as for whole word or multiword items. The setting of arrays TENSER and TENPAR in serial and parallel structures, respectively, is shown in figure 6-12.

PART-WORD ITEM EFFICIENCY

When an array with part-word items is being designed, efficiencies in access can be planned. Although SYMPL allows fields to occupy almost any position in an entry, good programming practice favors certain constructions for data of certain types.

When two items in different arrays are frequently exchanged, execution proceeds more quickly when the items occupy the same position within a word.

BOOLEAN DATA

The most efficient length for Boolean data is one bit. When a Boolean item is one bit in length, a shift to bit 0 and a sign test are the only instructions needed to determine whether it is TRUE or FALSE. If the item is more than one bit, however, the field must be masked and tested for a value other than 0.

```
STATUS KIND NUMB, INT, REAL, BOOL, CHAR;
ARRAY STAN [4] S(2);
BEGIN
    ITEM IDENT C(0,0,10)=
        ["SIN",,,"SUBS", "ABS"];
    ITEM SKIND S:KIND(1)=
        [S"REAL",,,S"CHAR", S"NUMB"];
END
```

S I N	IDENT [0]
2	SKIND [0]
	IDENT [1]
	SKIND [1]
	IDENT [2]
	SKIND [2]
S U B S	IDENT [3]
4	SKIND [3]
A B S	IDENT [4]
0	SKIND [4]

Figure 6-11. Array Presetting Example E

An exception exists when one item overlays two others. In the following, item B1ANDB2 can be used to test for B1 or B2:

```
ITEM B1 B(0,18,1),
     B2 B(0,19,1),
     B1ANDB2 B(0,18,2);
```

Boolean data is most efficient in bit 0. No shifts or masks are required to access it.

Efficiencies in decision tables can be achieved when Boolean values are packed within a single word, as shown in figure 6-13. The STATUS function assigns integer values 0 through 3 to FO. The dimensions of array CHARACTERS are assigned through status functions and are [0,3]. To check whether a delete operation is valid, the following IF statement can be used:

```
IF VALIDDELETE [FO] THEN ZAPIT;
    ELSE ERROR;
```

INTEGER DATA

Signed integer data (declared by data type I) can be accessed more quickly on CYBER 170 compatible systems when the field is 60 bits or 18 bits and the field begins in bit 0 or the field occupies bits 42-59. Signed integers are faster than unsigned integers. Unsigned integers (declared by data type U) are accessed more quickly when the field ends with bit 59.

Serial allocation:

```
ARRAY TENSER[4] S (2);
BEGIN
ITEM A I(0,0,30)=[4,,3,,6];
ITEM B I(0,30,15)=[,3,,7];
ITEM C C(1,0,5)=["LLLLL", "BBBBB", "CCCCC",
    "TTTTT", "EEEEE"];
END
```

0 29	30 44	45 59	
4			A [0] , B [0]
L L L L L			C [0]
	3		A [1] , B [1]
B B B B B			C [1]
3			A [2] , B [2]
C C C C C			C [2]
	7		A [3] , B [3]
T T T T T			C [3]
6			A [4] , B [4]
E E E E E			C [4]

Parallel allocation:

```
ARRAY TENPAR[4] P(2);
BEGIN
ITEM A I(0,0,30)=[4,,3,,6];
ITEM B I(0,30,15)=[,3,,7];
ITEM C C(1,0,5)=["LLLLL", "BBBBB", "CCCCC",
    "TTTTT", "EEEEE"];
END
```

0 29	30 44	45 59	
4			A [0] , B [0]
	3		A [1] , B [1]
3			A [2] , B [2]
	7		A [3] , B [3]
6			A [4] , B [4]
L L L L L			C [0]
B B B B B			C [1]
C C C C C			C [2]
T T T T T			C [3]
E E E E E			C [4]

Figure 6-12. Array Presetting Example F

```
STATUS FO SQ, WA, IS, AK;
ARRAY CHARACTERS [FO"SQ":FO"AK"];
    BEGIN ITEM  D1 B(0,0,1) = FALSE,
                D2 B(0,1,1) = FALSE,
                D3 B(0,2,1) = TRUE,
                D4 B(0,3,1) = TRUE,
    VALIDDELETE  B(0,0,4);
    END
```

Figure 6-13. Packed Boolean Array

ACCESSING ARRAY ITEMS

A particular occurrence of an array item is referenced by:

item-name [subscript]

subscript — Arithmetic expression indicating occurrence of item. Can be signed integer constant, an unsigned integer constant, a scalar, array item, or an expression that provides such a value.

Both of the following are valid:

TAB [I + TAB[3] * 2]

X [-3] where ARRAY [-5:5]; ITEM X;

SYMPL does not check array bounds during execution. If the subscript is omitted from a reference to an array item, 0 is assumed. A diagnostic is generated unless the array bounds are 0:0.

When multiword items are referenced, the format is the same. The compiler generates code needed to extract the item from its serial or parallel array. Code generated for parallel arrays is more efficient.

When a part-word item is referenced, the compiler generates code that:

Masks the item to extract it from its word.

Shifts it to the position appropriate for its data type. Character data is left-justified and signed integer data is right-justified with sign extension, as described in section 4.

Only the referenced item is affected by access. No other part of the word in which the item is positioned is disturbed.

The usual form of a procedure declaration is shown in figure 7-1. The formal list of parameters identifies parameters to be passed to the procedure when the procedure is called by a reference to its name.

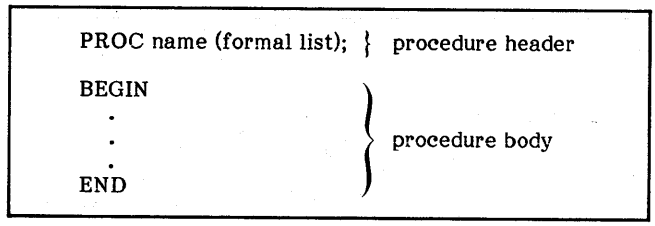


Figure 7-1. Procedure Declaration Structure

The procedure declaration establishes formal parameters that are used within the procedure body declaration. At the time the procedure is to execute, the actual parameters accompanying the procedure reference take the place of the formal parameters. The programmer is responsible for correspondence between the formal parameters and actual parameters. SYMPL checks neither the number nor type of parameters on a call during compilation or execution.

Formal and actual parameters are illustrated in figure 7-2a. Assume procedure SUB. The header for nested procedure P defines three formal parameters: A, BOUND, and S. Parameters A and BOUND are used within P to check whether the array bound is positive and to initialize the array items to a zero value. Parameter S returns a character constant to the calling procedure SUB. The header specifies that parameters A and S are to be passed by their address, but BOUND is to be passed by value.

When procedure P is called, actual parameters TAB, 64, and ETAT are substituted, respectively, for formal parameters A, BOUND, and S. Procedure P executes as if the lines containing the comment * were written as shown in figure 7-2b.

PROCEDURE DECLARATION AND CALL

The format for a procedure header with formal parameters is:

PROC name (param1, param2, . . .);

name Any SYMPL identifier (1 through 12 letters, $, or digits beginning with a letter) that is not a reserved word. Names of intrinsic functions are not reserved words. If the procedure is to be called by a program written in a language other than SYMPL, only the first seven characters are used and the first character cannot be the dollar sign.

a.
```
PROC SUB;
BEGIN
    PROC P(A,(BOUND),S);
    BEGIN
    ARRAY A; ITEM AA;
    ITEM BOUND, S C(10);
    ITEM I;
    XREF PROC ERROR;
#*# IF BOUND LS 0
#*# THEN ERROR("BOUND NEGATIVE");
#*# FOR I=0 STEP 1 UNTIL BOUND DO
        AA[I]=0;
#*# S="INIT";
    END # PROC P #
ITEM ETAT C(10);
ARRAY TAB[64]; ITEM T;
P(TAB, 64, ETAT);
    .
    .
    .
END # PROC SUB #
```

b.
```
BOUND=64;
IF BOUND LS 0
    THEN ERROR ("BOUND NEGATIVE");
FOR I=0 STEP 1 UNTIL BOUND DO
    T[I]=0;
ETAT="INIT";
```

Figure 7-2. Formal and Actual Parameters Example

param Name of any SYMPL entities listed below. Later in this section, each type of parameter is discussed separately.

Array	Based array
Function	Procedure
Label	Item

If the name specifies an item, it can be enclosed in parentheses to indicate a call-by-value rather than a call-by-address.

Within the procedure body, all formal parameters of any type except label must be declared. Any formal parameter not declared in the body is assumed to be a label parameter.

SYMPL makes certain assumptions about the formal parameters, depending on their type, as shown in table 7-1. Again, depending on the type, table 7-2 lists reasonable entities to pass as actual parameters. Notice that an array or based array formal parameter can be passed in several forms. Switch elements cannot be passed as parameters. (A switch can be used as an external entity, however.)

TABLE 7-1. FORMAL PARAMETER ASSUMPTIONS

Formal Parameter Declaration	Assumed Content of Parameter Word
ITEM I	Address of item I
ARRAY A	First word address of array A
BASED ARRAY BA	Address of pointer to BA
LABEL L	Address of label L
FPRC FP	Address of entry to procedure FP
FUNC F	Address of entry to function F

TABLE 7-2. POSSIBLE ACTUAL PARAMETERS

Formal Parameter	Reasonable Actual Parameter
Item for call-by-address	Item name, (item name)
Item for call-by-value	Item name, arithmetic expression, subscripted array item
Array	Array name, based array name
Based array	P function, item name, expression whose result is a pointer
Label	Label name
Procedure	Procedure name
Function	Function name

SCALAR AND ARRAY ITEM NAMES AS PARAMETERS

Any scalar or array item of any type (I, U, R, C, S, or B) can be specified as a formal parameter. Within the procedure, the formal declaration syntax is the same as a declaration outside the procedure.

The formal parameter list indicates whether a given parameter is to be passed by value or address. This is illustrated in figure 7-3. The corresponding actual parameter for W or Z is assumed to be a local variable with the address of W or Z; for X or Y, the assumption is that they are the actual values to be used for X or Y. Call-by-address is required for any parameter whose value is to be returned to the calling subprogram since call-by-value parameters work with a temporary copy of a variable.

An actual parameter can be a scalar name, constant, array item name, or expression. (When an actual parameter is the name of an item enclosed in parentheses, SYMPL considers it to be an expression.) Consequently, the procedure receives the address of a temporary location containing the

```
PROC P(W, (X), (Y), Z);
BEGIN ITEM  W R,
            X I,
            Y C(14),
            Z B ;
.
.
.
END #P#
```

Figure 7-3. Passing Parameters by Value or Address

scalar value instead of the address of the scalar itself. Such a parameter does not create the instruction savings of a call-by-value parameter. It does, however, provide the protection for the scalar value accorded all call-by-value parameters.

SYMPL performs no conversions when the type of a formal parameter is not the same as the type of the actual parameter:

In the following, X=FALSE at the end of the procedure since 1.0 is type real and has a format that is not the same as integer type 1:

```
R(1.0, 1, X);
    PROC R( (A), (B), C );
    ITEM A B,B B,C B;
    C = A EQ B;
```

No BEGIN and END pair is associated with procedure R. The declarations for a procedure can appear before the executable statement. Since C=A EQ B constitutes the entire executable portion of R, a compound statement is not required. Many of the remaining examples in this section use such a single elementary statement in the procedure called.

EXPRESSIONS AS ACTUAL PARAMETERS

Expressions are evaluated and the result is passed to the procedure as a temporary storage word. The procedure receives the address of a temporary location containing the result of the evaluation.

In figure 7-4, when procedure P is called, the statement changing W in the procedure has no effect. The temporary storage word for A+B is changed.

```
ITEM A=1, B=2;
P(A+B, B, 3);
      .
      .
      .
    PROC P(W, (X), (Y));
    BEGIN ITEM W, X, Y;
    ITEM I;
    X=X+1;
    FOR I=0 STEP 1 UNTIL X DO
    W=W+Y;
    END # PROC P #
```

Figure 7-4. Expression as Parameter

SUBSCRIPTED VARIABLES AS ACTUAL PARAMETERS

Subscripted variables are considered to be expressions. The procedure receives the address of a temporary location containing the result. As with parameters called by value, subscripted variable parameters modified within the procedure cannot be passed out of the procedure. For example, assume procedure P is defined by:

```
PROC P(A);
ITEM A;
A = 0;
```

When procedure P is called from a program containing the following statements, T[12]=2 when the calling program resumes execution:

```
ARRAY TT; ITEM T;
T[12]=2;
P(T[12]);
```

If a procedure must modify a subscripted variable, the array name and the subscript must be passed as separate parameters. In the formal array, the variable to be modified must be described as the same field as the actual item in the actual array. For example, assume a procedure Q defined, as shown in figure 7-5. When procedure Q is called from a program containing the following statements, T[12]=0 at the end of the procedure. Items X and T must have identical field descriptions:

```
ARRAY TT[100]; ITEM T;
T[12]=2;
Q(TT,12);
```

```
PROC Q(XX,Y);
BEGIN
ARRAY XX; ITEM X;
ITEM Y;
X[Y]=0;
END
```

Figure 7-5. Subscripted Variable as Parameter

CHARACTER STRINGS AS PARAMETERS

Character strings are passed to a procedure without any accompanying information about length. The programmer writing the procedure is responsible for knowing the length.

The declared length of the string cannot be passed to the procedure through a variable in the parameter list. ITEM STRING C(N) is illegal in a procedure since the syntax of an ITEM declaration requires a character string length to be expressed as an integer constant. The compiler generates code based on the declared length of the formal parameter. If the actual parameter is not the same length, unexpected results can occur.

The actual parameter string should have a length longer than or equal to the formal parameter string length. If it is longer, only the number of characters specified by the ITEM declaration are used or altered. An actual parameter string shorter than the formal parameter string can produce unpredictable results, since characters following the actual parameter are accessed. The compiler does not guarantee the contents of those characters.

No padding occurs when an actual parameter string is shorter than a formal parameter string. In the example in figure 7-6, the call to procedure Q sets the first 10 characters of LEFT to the value RIGHT; the last 10 characters are undefined.

```
ITEM LEFT C(20), RIGHT C(10), JUNK C(10);
Q(LEFT,RIGHT);
        .
        .
        .
    PROC Q(S,T);
    ITEM S C(20), T C(20);
    S=T;
```

Figure 7-6. Character Strings as Parameters

LABEL NAMES AS PARAMETERS

A label name can be used as an actual parameter. A formal parameter declaration for the label can, but need not, appear in the procedure declaration. It makes debugging easier and is generally good programming practice to declare it, however. The parameter in the transfer vector is assumed to be the address of a label.

PROCEDURE NAMES AS PARAMETERS

A procedure name can be specified as a formal parameter. Within the procedure, the formal parameter declaration is not the same as a procedure declaration elsewhere. The parameter in the transfer vector is assumed to be the address of the entry to the procedure. SYMPL calls the procedure by simulating a return jump.

A formal parameter that is a procedure name must be declared with:

```
FPRC name, name, . . . ;
```

name Identifier of a procedure.

The formal declaration of a procedure name does not include any parameters to that procedure. Such parameters must be established for use in the procedure. Assuming procedures P and S as shown in figure 7-7, a call P(17,S) results in a call to procedure S with 17 as a parameter. The programmer writing procedure P is responsible for knowing that procedure S requires an integer parameter X.

```
PROC P (N, Q);
BEGIN ITEM N;
    FPRC Q;
Q(N);
END # P #

PROC S (X);
BEGIN ITEM X;
    .
    .
    .
END # S #
```

Figure 7-7. Procedure Name as Parameter

A procedure name in a parameter list should be programmed carefully. Since the called procedure must supply parameters and SYMPL checks neither the number or type of

parameters, any execution-time errors are difficult to debug. In the example in figure 7-7, calls to procedure P must supply only the names of the procedures, all of which require exactly the same type of parameters.

ARRAY NAMES AS PARAMETERS

Any array or based array can be specified as a formal parameter. Within the procedure, the formal parameter declaration syntax is the same as array declaration outside of the procedure, including the descriptions of items in the array.

When the formal parameter is specified with a BASED ARRAY declaration, the actual parameter must be a pointer or LOC function, or an expression whose value is a pointer. Access of a formal based array is inefficient and should be avoided. Such access is justified only when the intent of the procedure is to move the based array.

When the formal parameter is specified with an ARRAY declaration, the actual parameter must be an array or based array. An array name can be subscripted; this has the effect of imposing the first element of the formal array onto the designated element of the actual array.

The first word address of an array is passed to a procedure without any accompanying information about array bounds, and SYMPL performs no subscript checking. Consequently, the array bounds are not required in the formal array declaration. The programmer writing the procedure is responsible for bounds and subscript checking.

If the size and structure are not the same for the formal and actual arrays, the wrong elements are accessed. The programmer is responsible for defining the correct field positions in the formal array, and for extracting or storing the desired fields in the actual array.

For single-dimension single-word arrays, bounds can be omitted in the formal declaration since parameters passed to a procedure can control array size. In the example in figure 7-8a, calls to procedure Q set both array A and array B to zero. For multidimension arrays or multiword array items, the formal declaration must be correct to ensure proper results. In figure 7-8b, the first call to procedure P sets array A to zero. The second call, however, erroneously sets more than the 37 items of array B.

EFFICIENCY IN PARAMETER LISTS

The calling sequence for a procedure with parameters is lengthy. Several techniques can be used on source programs to reduce the size of the generated code or to reduce the time required for execution. Three such techniques are: use of call-by-values for scalars or array items, reuse of a single parameter list, and the DEF capability.

a.
```
ARRAY A[0:64]; ITEM AA;
ARRAY B[27:63]; ITEM BB;
Q(A,64);
Q(B,63-27);
      .
      .
      .
PROC Q (X, (N));
BEGIN
ITEM N, I;
ARRAY X; ITEM XX;
FOR I=0 STEP 1 UNTIL N DO
    XX[I]=0;
END # Q #
```

b.
```
ARRAY A[64]; ITEM AA;
ARRAY B[27:63]; ITEM BB;
P(A, 64);
P(B, 63-27);
      .
      .
      .
PROC P (T);
BEGIN
ARRAY T; ITEM TT;
ITEM I;
FOR I=0 STEP 1 UNTIL 64 DO
    TT[I]=0;
END
```

Figure 7-8. Array Names as Parameters

CALL-BY-VALUE PARAMETERS

SYMPL calls subprograms through a return jump instruction. Actual parameters are passed to the subprogram through a transfer vector list.

The address of a parameter list is passed in register A1. If the F parameter appears on the SYMPL compiler call, the last word in each list contains all zeros as required by the FORTRAN Extended calling sequence.

The transfer vector list contains local copies of all parameters used. The two types of parameters are:

A scalar or array item parameter enclosed in parentheses in the formal parameter list indicates that the parameter is to be called by value rather than by address. The transfer vector points to a temporary storage word containing the value. The corresponding actual parameter is protected by SYMPL.

All other parameters appear in the transfer vector lists as addresses of memory words containing their values.

Call-by-address parameters require two memory references to access the parameter. This indirect addressing is less efficient than the direct addressing possible for call-by-value parameters.

For program efficiency, call-by-value should be specified for scalars or array items in a formal parameter list as often as possible. Call-by-address should be used only when the parameter is modified within the procedure and the new value of the parameter is to be returned to the calling subprogram.

REUSING A PARAMETER LIST

The SYMPL compiler uses the same transfer vector as many times as possible. Consequently, the size of generated code can be reduced by rewriting some calls to reference global identifiers. Consider the following:

A declaration for procedure P is identical to that for procedure Q:

```
PROC Q (R, S, T, U, (V));
```

If the calls are P(A, B, C, D, F+1) and Q(A, B, C, D, E), the same transfer vector cannot be used. These two calls do allow the same transfer vector:

```
H=1; P(A, B, C, D, H);

H=F+1; Q(A, B, C, D, H);
```

Use of global identifiers, external identifiers, and common variables must be considered in relation to other modular programming needs.

The IF statement allows alternative statements to execute, depending on whether a Boolean expression is TRUE or FALSE. The FOR statement simplifies coding of repetitive operations.

IF STATEMENT

The IF statement has three clauses:

The IF clause specifies the Boolean condition to be tested.

The THEN clause specifies the statement to execute when the result of the IF clause evaluation is TRUE.

The ELSE clause specifies the statement to execute when the result of the IF clause evaluation is FALSE. This clause is optional; if omitted, the statement following the THEN clause executes when the result is FALSE.

The IF statement syntax is:

IF Boolean expression THEN statement ELSE statement

Boolean expression	Boolean expression specifying the condition to be tested.
statement	Any elementary statement or compound statement. All statements must be terminated with semicolons just as if they were not associated with IF.

Since the ELSE portion of the IF statement is optional, the simplest form of the IF statement is:

IF Boolean expression THEN statement;

Both of these are valid IF statements:

```
IF A EQ 0 THEN T[I]=0; ELSE T[I]=2;

IF A EQ 0 THEN T[I]=1;
```

ELSE distinguishes between statements that are always executed and those that execute only when a condition is false.

The logic of the IF statement is shown in figure 8-1. The THEN statement executes only when the Boolean expression is TRUE; the ELSE statement executes only when the Boolean expression is FALSE.

The differences in execution between the following two statements

```
IF A EQ B THEN C=D; E=F; G=H;
```

and

```
IF A EQ B THEN C=D; ELSE E=F; G=H;
```

appears in the logic diagrams shown in figure 8-2. The second diagram illustrates the execution of E=F; only when A EQ B; is false.

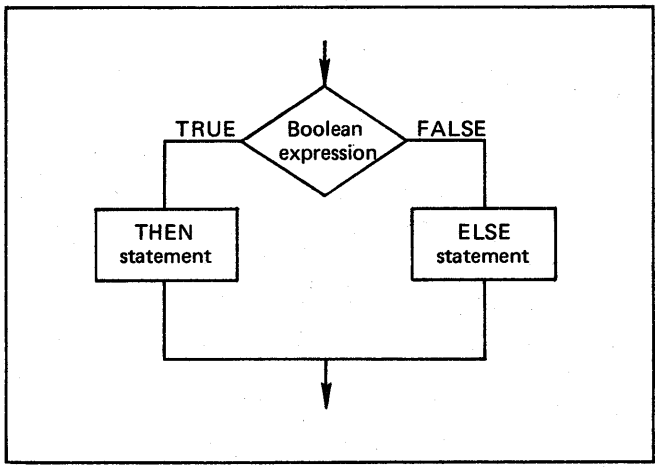


Figure 8-1. IF Statement Logic

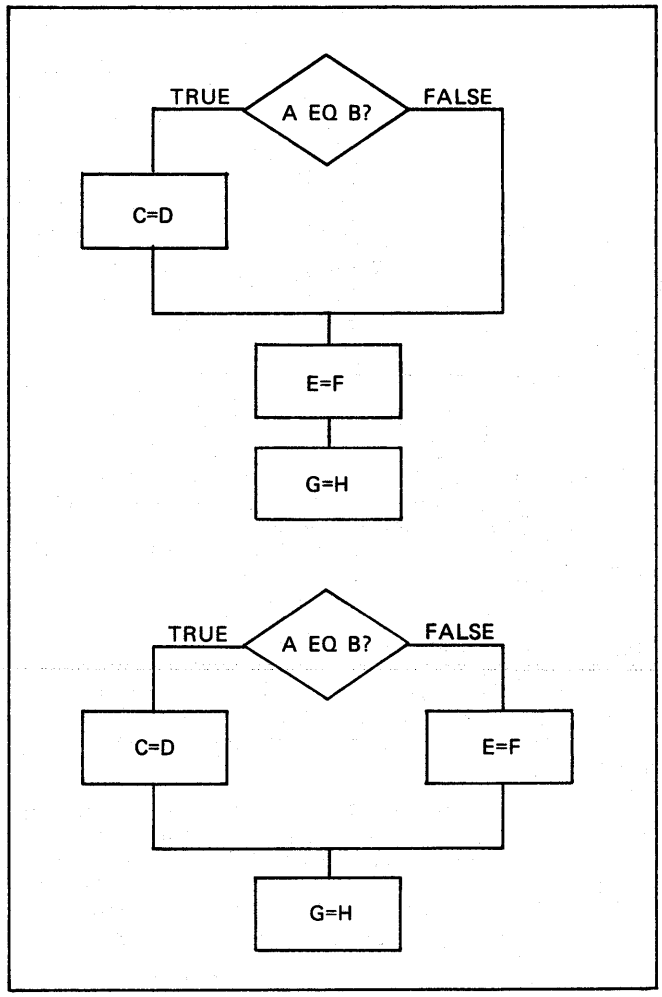


Figure 8-2. ELSE Statement Logic

All the statements in an IF construct are subject to the same rules, including punctuation, as other statements in SYMPL.

The statement can be an elementary statement such as:

```
BIRD="TROCAN";
```

The statement can be a compound statement, as shown in figure 8-3a. The statement can be another IF statement, FOR statement, STOP statement, and so forth, as shown in figure 8-3b.

```
a. BEGIN
   BIRD="TROCAN";
   TREE=24;
   END

b. IF A EQ 0 THEN
       IF B EQ 0 THEN
           C=1;
       ELSE
           C=2;
```

Figure 8-3. IF Statement Example A

A common programming practice is to write every statement following THEN and ELSE as a compound statement. In this instance the BEGIN and END visually delimit nested statements, as shown in figure 8-4.

```
IF A EQ 0 THEN
    BEGIN
    IF B EQ 0 THEN
        BEGIN
        C=1;
        END
    ELSE
        BEGIN
        C=2;
        END
    END
```

Figure 8-4. IF Statement Example B

Punctuation within an IF statement follows the rule that each elementary statement must be terminated by a semicolon. Each statement in the IF construct has a following semicolon. No semicolons are associated with BEGIN and END.

NESTED IF STATEMENTS

When IF statements are nested, the ELSE portion of an IF statement is always associated with the innermost nested IF. A nested IF statement and its corresponding logic flow are shown in figure 8-5.

It is a better practice, however, to write nested IF statements as compound statements to avoid confusion on this point. It makes the code more obvious, and, in terms of execution time and space, is no more costly. The statement in figure 8-5 should be written as shown in figure 8-6.

Another example of a nested IF statement, in which C=4 only if neither A nor B is 0, is shown in figure 8-7.

```
IF A EQ 0
    THEN
    IF B EQ 0
        THEN
        C=1;
        ELSE
        D=1;
```

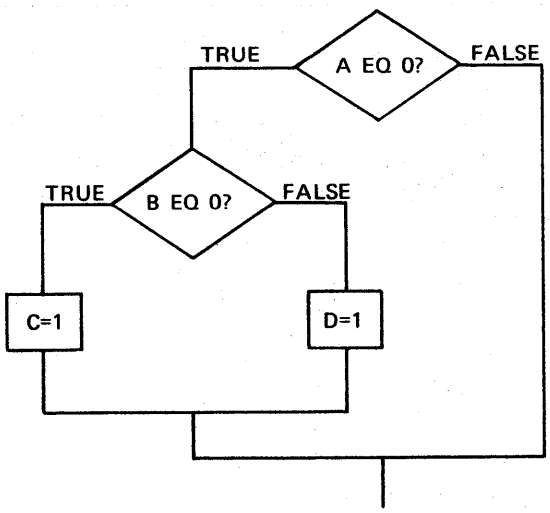


Figure 8-5. Nested IF Statement Example A

```
IF A EQ 0 THEN
    BEGIN
    IF B EQ 0
        THEN C=1;
        ELSE D=1;
    END
```

Figure 8-6. Nested IF Statement Example B

```
IF A EQ 0 THEN IF B EQ 1 THEN C=1;
                         ELSE  C=2;
               ELSE IF B EQ 0  THEN C=3;
                         ELSE  C=4;
```

Figure 8-7. Nested IF Statement Example C

BOOLEAN EXPRESSIONS IN IF STATEMENTS

Any Boolean expression can be used in an IF statement. Evaluation of the expression terminates as soon as any part of the expression determines the results. The example in figure 8-8a is evaluated as if it were written as shown in figure 8-8b.

This feature avoids wasteful tests and can result in valuable protection in a program. For example, in the following the procedure SQROOT is not called when X is negative:

```
IF X GQ 0 AND SQROOT(X) EQ Y
    THEN . . .
```

```
a. IF I GR 0 AND T[I] EQ 0
       THEN X=0;
       ELSE X=1;

b. IF I GR 0
       THEN IF T[I] EQ 0
            THEN X=0;
            ELSE X=1;
       ELSE X=1;
```

Figure 8-8. Boolean Expression in an IF Statement

Evaluation of two Boolean expressions can be forced by an expression of the proper form. For example:

IF A EQ B AND A EQ C THEN . . .

can be written in a faster executing form:

IF (A-B LOR A-C) EQ 0 THEN . . .

However, clarity and maintainability should be considered when code is written for faster execution time.

FOR STATEMENT

The FOR statement should be used any time a statement is to execute at least 3 or 4 times, or any time the conditions for execution might not exist.

The FOR statement has three clauses:

The FOR clause specifies the conditions under which the DO clause is to be executed. Those conditions might result in zero executions.

The WHILE clause or the UNTIL clause specifies the conditions that terminate the DO clause executions. The WHILE clause offers execution advantages in certain cases.

The DO clause specifies the operations to be repeated. In most instances, the DO clause includes a compound statement.

An example of a FOR statement that sets each element of array T to 0 is shown in figure 8-9.

```
DEF SIZE #1024#;
ARRAY [SIZE]; ITEM T;
ITEM I;
FOR I=0 STEP 1 UNTIL SIZE DO T[I]=0;
```

Figure 8-9. FOR Statement Example

The FOR statement is an extension of the DO statement of FORTRAN. It differs from DO in several respects, however. In SYMPL:

The induction variable (loop counter) must be declared as a scalar before it can be used.

The step value can be negative.

A loop is not necessarily executed once.

The TEST statement can be included in the loop to cause remaining computations inside the loop to be bypassed.

A CONTROL statement can affect the optimization the compiler performs with the statement.

SYMPL version 1.2 introduces program control over the code generated for FOR loops. Through a CONTROL FASTLOOP or CONTROL SLOWLOOP compiler-directing statement, a SYMPL 1.2 program can specify the implementation of the loop within each individual FOR statement. Execution advantages can be gained by specifying FASTLOOP; on the other hand, this specification puts restrictions on the format and use of the FOR statement.

SYMPL versions prior to 1.2 always produced slow loops that could not be optimized since the compiler could not ascertain the permanence of all statement characteristics. The default condition for version 1.2 is SLOWLOOP.

When the programmer knows that a loop has certain characteristics, however, CONTROL FASTLOOP should be specified to obtain optimization. The following characteristics are required for optimization:

The induction variable is type integer or type unsigned integer with an absolute value that can be expressed in 17 bits.

The variables in the arithmetic expression of the STEP clause must not be modified.

The variables in the arithmetic expression of the UNTIL clause, if present, must not be modified inside the controlled statement.

The controlled statement of the loop is executed at least once.

The variables in a WHILE clause can be changed in the loop; however, the current value is always used.

For both fast loops and slow loops, the programmer can affect code efficiency by properly planning the loop.

Faster execution for slow loops can be achieved by moving arithmetic expressions outside the loop. In the example in figure 8-10a, TAB[J]-1 is evaluated once, but N+2 and TAB[J] are evaluated every repetition. To avoid evaluation of arithmetic expressions within the loop, the FOR statement in figure 8-10a could be written as shown in figure 8-10b. Further, any call to a procedure or function within a loop inhibits optimization.

```
a. FOR I=TAB[J]-1
   STEP N+2 UNTIL TAB[J]-1 DO
   TAB[I]=0;

b. S=N+2;
   L=TAB[J]-1;
   FOR I=TAB[J]-1
   STEP S UNTIL L DO
   T[I]=0;
```

Figure 8-10. Evaluation of Arithmetic Expression in a FOR Statement

FOR SYNTAX

The general format of the FOR statement is:

FOR induction variable = loop control DO statement;

induction variable — Identifier of data type I, U, S, or R to be used as the loop counter. Data type R is not often used. The type of this induction variable establishes the mode for evaluation of arithmetic expressions in the FOR statement. When the loop terminates, the current value of the induction variable is available only if a jump exits from the loop. If the loop terminates normally, the induction variable is not defined.

loop control — Condition under which the loop is to be executed. It can take several forms as noted below.

statement — Statement to be executed as long as the loop control condition exists. This statement, which is called the controlled statement, can be any elementary or compound statement, including an IF statement or a FOR statement. Good programming practice is to write the controlled statement as a compound statement at all times.

LOOP CONTROL

For slow loops, evaluation of the test condition occurs at the beginning of each loop before the controlled statement is executed. Consequently, the controlled statement can be bypassed. In the following example T[I]=0 is never executed:

```
L=3;
FOR I=4 STEP 1 UNTIL L DO
    T[I]=0;
```

Both the test for loop terminating conditions (WHILE Boolean expression or UNTIL arithmetic expression) and the increment to the induction variable take place within the loop.

The loop control has these five forms (the fourth and fifth forms produce an infinite loop; the programmer is responsible for coding an exit jump):

1. Initial WHILE Boolean expression
2. Initial STEP arithmetic expression WHILE Boolean expression
3. Initial STEP arithmetic expression UNTIL arithmetic expression
4. Initial STEP arithmetic expression
5. Initial

initial — Arithmetic expression giving the initial value of the induction variable. The expression is evaluated once at the start of the FOR statement.

Boolean expression — Boolean expression specifying the condition under which looping is to continue. As long as the expression is TRUE, looping continues; when the expression is FALSE, looping does not take place.

arithmetic expression — Arithmetic expression indicating:

STEP Clause — Increment to the induction variable to be added each loop. This constant or variable can have a positive or negative value.

UNTIL Clause — Value after which looping terminates.

The expression can have a negative, as well as a positive, value.

The logic of a statement with a WHILE clause and STEP clause with a slow loop is shown in figure 8-11. For an UNTIL clause with a positive step with a slow loop, the logic is as shown in figure 8-12. Figure 8-13 shows the logic of an UNTIL clause with a fast loop, when a STEP expression can be positive or negative.

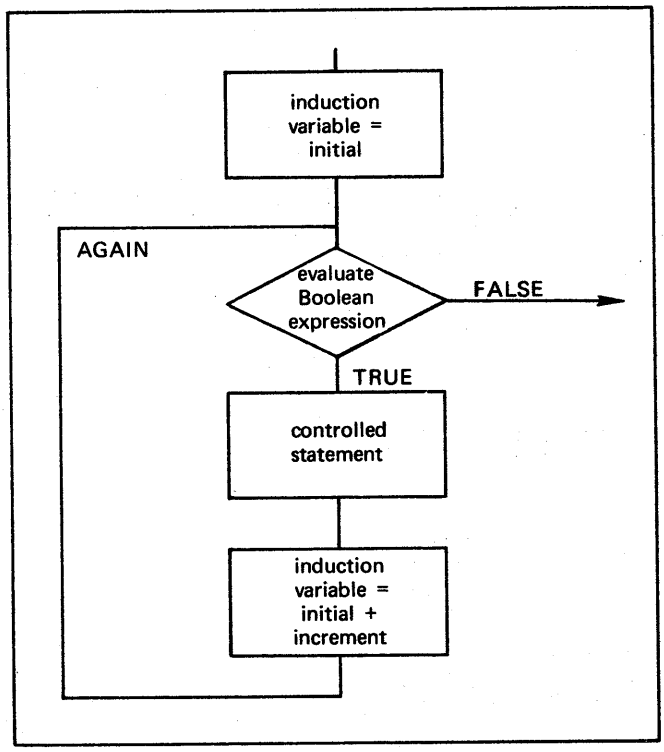


Figure 8-11. Slow Loop Logic Example A

WHILE Clause

The WHILE clause of the FOR statement combines the capabilities of an IF statement with the looping capabilities of FOR. For example, the sequence shown in figure 8-14 assigns SOL the minimum value of I, if any, when T[I]=0.

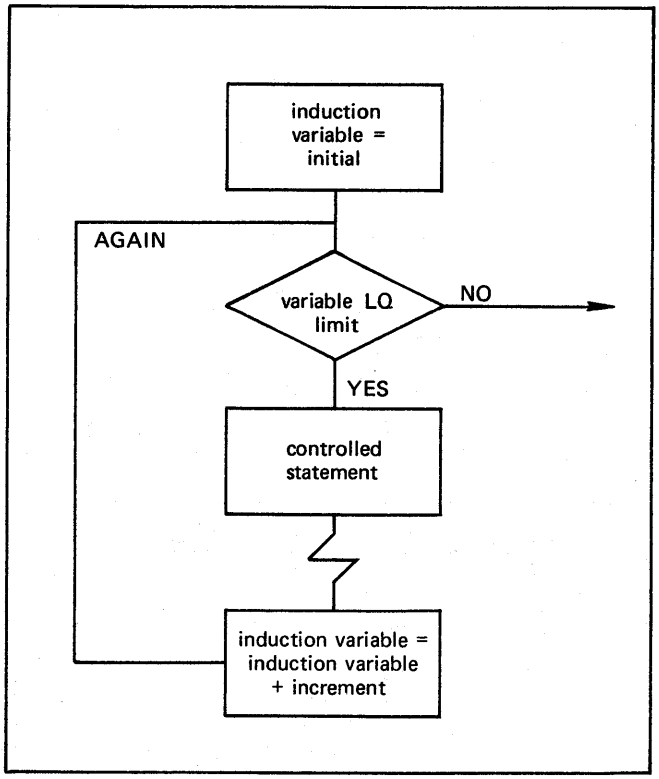


Figure 8-12. Slow Loop Logic Example B

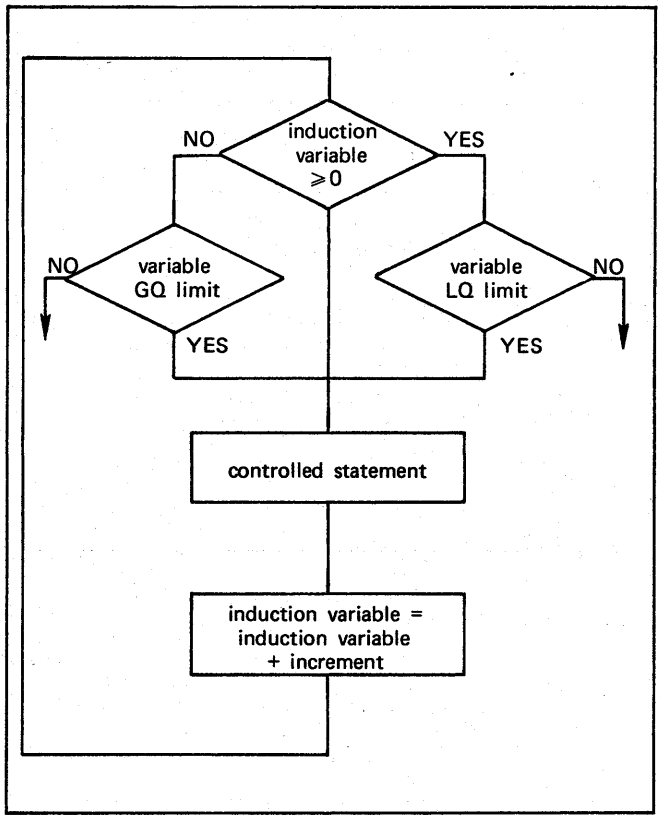


Figure 8-13. Fast Loop Logic Example

```
SOL=0;
FOR I=0 STEP 1 UNTIL 100 DO
    IF T[I] EQ 0
    THEN
        GOTO FOUND;
    .
    .
    .
FOUND: SOL=I;
```

Figure 8-14. WHILE Clause Example A

Using the WHILE clause, a FOR statement can be written to accomplish the same function if it is certain that T[I]=0 exists to stop the loop:

```
SOL=0;
FOR I=0 STEP 1 WHILE T[I] NQ 0 DO
    SOL=I;
```

Empty compound statements are often useful in FOR statements with WHILE clauses. For example, the statement shown in figure 8-15 exits from the loop with MIN having the minimum value such that T[MIN] exceeds 0. This technique is valid only with a slow loop. With a fast loop the value of the induction variable is undefined on a normal exit.

```
FOR MIN=0 STEP 1
WHILE T[MIN] LQ 0 DO
    BEGIN
    END
```

Figure 8-15. WHILE Clause Example B

No form of the FOR statement exists in which the reserved words FOR and the initial value of the induction variable can be omitted. That is, a WHILE B DO statement is not valid. The same results can be achieved, nevertheless, through use of the DEF statement to generate a valid FOR statement, as shown in figure 8-16.

```
ITEM DUMMY;
DEF ASLONGAS #FOR DUMMY=DUMMY WHILE#;
#SET NEXT TO FIRST ELEMENT OF LIST#
ASLONGAS NEXT NQ 0 DO
    BEGIN
    .
    .
    .
    END
```

Figure 8-16. WHILE Clause Example C

Controlled Statement

The controlled statement can be any valid statement. Examples of common types of controlled statements are given below; they assume all variables have been defined previously.

1. The statements necessary to initialize three arrays are shown in figure 8-17.

2. The IF statement as a controlled statement is illustrated in figure 8-18.

3. The FOR statement as a controlled statement which sets the lower triangle of MATRIX to 0 is shown in figure 8-19.

4. A compound statement nesting within a controlled statement is shown in figure 8-20.

```
FOR I=1 STEP 1 UNTIL N DO
    BEGIN
    T[I]=0;
    U[I]=0;
    V[I]=I;
    END
```

Figure 8-17. Controlled Statement Example A

```
FOR I=M STEP 1 UNTIL N DO
    IF T[I] EQ 0
    THEN
    GOTO L;
    ELSE
        BEGIN
        K=K + 1;
        U[K]=U[K] + 1;
        END
```

Figure 8-18. Controlled Statement Example B

```
ARRAY [1:10, 1:10]; ITEM MATRIX;
FOR I=1 STEP 1 UNTIL 10 DO
    FOR J=1 STEP 1 UNTIL I DO
        MATRIX[I,J]=0;
```

Figure 8-19. Controlled Statement Example C

```
FOR I=1 STEP 1 UNTIL N DO
    BEGIN
    TAM[I]=0;
    FOR J=1 STEP 1 UNTIL N DO
        BEGIN
        MAT[I,J]=0;
        TAB[I,J]=TAB[I,J] + 10 * I + J;
        END
    END
```

Figure 8-20. Controlled Statement Example D

A jump out of the controlled statement is valid. Under such circumstances, the current value of the induction variable is preserved and can be used outside the statement.

A jump into a controlled statement from outside the controlling FOR statement is possible, although such an action generally has no meaning and produces errors. Although the induction variable can be modified within the controlled statement on slow loops, good programming practice avoids such code.

TEST STATEMENT OF FOR

The TEST statement has meaning only within the FOR controlled statement. TEST, which allows the remaining part of a loop to be bypassed, is equivalent to a FORTRAN statement that jumps to a CONTINUE statement in a DO loop.

The use of TEST, in which the statement V[I]=0 is bypassed for values of I such that U[I]=VAL, is illustrated in figure 8-21. Without TEST, the sequence shown in figure 8-21 appears as shown in figure 8-22.

When loops are nested, the induction variable name can be added to the TEST statement to specify which loop is to be bypassed, as illustrated in figure 8-23. The logic of the code in figure 8-23 is as if it were written as shown in figure 8-24.

```
FOR I=0 STEP 1 UNTIL N DO
    BEGIN
    T[I]=0;
    IF U[I] EQ VAL
        THEN
        TEST;
    V[I]=0;
    END
```

Figure 8-21. TEST Statement Example A

```
        I=0;
AGAIN:
    IF I LQ N
        THEN
        BEGIN
        T[I]=0;
        IF U[I] EQ VAL
        THEN
        GOTO NEXT;
        VAL[I]=0;
NEXT:
        I=I + 1;
        GOTO AGAIN;
        END
```

Figure 8-22. TEST Statement Example B

```
FOR I=0 STEP 1 UNTIL N DO
    FOR J=0 STEP 1 UNTIL M DO
        BEGIN
        A[I,J]=A[I,J] + 1;
        IF A[I,J] EQ VAL THEN TEST I;
        IF A[I,J] EQ LAV THEN TEST J;
        B[I,J]=0;
        END
```

Figure 8-23. TEST Statement Example C

```
AGAINI:  IF I LQ N THEN
             BEGIN
             J=0;
AGAINJ:  IF J LQ M THEN
             BEGIN
             A I,J=A I,J + 1;
             IF A I,J EQ VAL THEN GOTO NEXTI;
             IF A I,J EQ LAV THEN GOTO NEXTJ;
             B I,J=0;
NEXTJ:       J=J+1; GOTO AGAINJ;
             END
NEXTI:       I=I+1; GOTO AGAINI;
             END
```

Figure 8-24. Logic of TEST Statement

COMPILATION CONTROL 9

SYMPL compilation is controlled by:

DEF statements in the program that are similar to COMPASS macros, as described in section 5.

CONTROL compiler-directing statements in the program.

$BEGIN and $END debugging code delimiters.

SYMPL compiler call itself.

The CONTROL statement is a compiler-directing statement rather than an executable statement in a program. The words used in the CONTROL statement are not reserved words: ITEM NOLIST, for example, is legal. Also, these words can be expanded by DEF.

Several types of actions are influenced by CONTROL, including:

Source listing control.

Compilation options affecting packed switches, preset of common, and FORTRAN compatibility.

Characterization of variables and arrays for optimization purposes.

Conditional assembly.

Table 9-1 shows all control-words of the CONTROL statement and the range of compiler action in regard to each statement.

TABLE 9-1. CONTROL-WORDS OF CONTROL

Control-Word	Function	Extent of Effect
DISJOINT variable	Characterize variable as having single name.	Entire module
EJECT	Skip to new page of source listing output.	Single compiler action
ENDIF	End conditional assembly begun by IFxx.	Immediate compiler action
FASTLOOP	Generate FOR loop similar to FORTRAN DO loop.	Until a subsequent FASTLOOP, SLOWLOOP, or TERM
FI	Same as ENDIF.	Same as ENDIF
FTNCALL	Turn on compiler call F parameter.	Entire module
IFxx condition	Compile code if condition true.	UNTIL balanced ENDIF or FI
INERT array	Characterize array as not having overlapping subscript references.	Entire module
LEVEL n block	Specify memory residence of common block or based array.	Entire module
LIST	Resume source listing.	Until subsequent NOLIST or TERM
NOLIST	Suspend source listing unless H parameter on compiler call or OBJLIST appears.	Until subsequent LIST or TERM
OBJLIST	List object code, overlapping compiler call list parameter and LIST or NOLIST.	Entire module
OVERLAP variable	Characterize variable as having more than one name.	Entire module
PACK	Pack switch code.	Entire module
PRESET	Preset items in common.	Entire module
REACTIVE array	Characterize array as possibly having overlapping subscript references.	Entire module
SLOWLOOP	Generate FOR loop without FORTRAN similarities.	Until a subsequent FASTLOOP, SLOWLOOP, or TERM
TRACEBACK	Generate traceback information.	Entire module

CONDITIONAL COMPILATION

The IFxx form of the CONTROL statement allows conditional compilation that resembles COMPASS conditional assembly. SYMPL offers fewer capabilities than COMPASS, with no statements equivalent to COMPASS pseudo-instructions ELSE and IF DEF.

For instance, to compile source statements only when DBUG=0 in COMPASS and SYMPL, the statements shown in figure 9-1 can be used. In each case, the code that produces an error message and aborts the program is not assembled when DBUG=0. The COMPASS code that conditionally assembles the range identified by name B is in figure 9-1a. The same function in SYMPL is performed as shown in figure 9-1b.

```
a.      DBUG EQU 1
            .
            .
            .
      B  IFNE DBUG,0
         SA1  MSGVECT
         RJ   ERROR
         JP   ABORT
      B  ENDIF

b.      DEF   DBUG #1#;
            .
            .
            .
        CONTROL IFNQ DBUG,0;
        ERROR(MESSAGE);
        GOTO ABORT;
        CONTROL ENDIF;
```

Figure 9-1. CONTROL Statement Example A

In both languages, the conditional source statements are bracketed between a statement defining the conditions and a statement ending conditional assembly. In SYMPL, the ending statement can be either:

CONTROL ENDIF; or CONTROL FI;

However, this statement must not be generated by a DEF. When the IF condition is false, DEF statements are not expanded.

The format of a conditional assembly statement is:

CONTROL IFxx constant1, constant2;

xx — Condition that compiler is to test constants for in a constant1 xx constant2 situation:

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

constant1, constant2 — Constants or status functions to be tested. Generally, at least one constant is defined through DEF. Both constants must be the same type since SYMPL does not convert types in this context. Data type B and C should be compared only with IFEQ and IFNQ. Blanks are significant in character strings, whether the blanks are within the string or at the end of the string.

If only one constant appears, it is assumed to be constant1, and constant2 is assumed to have a value of 0.

When the condition is false, assembly continues with the next statement after the balancing CONTROL ENDIF of CONTROL FI. The source listing produced shows a minus sign in the left margin.

An example in which code is generated to call procedure S when FAST=0 is shown in figure 9-2.

```
DEF FAST #0#;
.
.
.
CONTROL IFEQ FAST;
S;
CONTROL FI;
```

Figure 9-2. CONTROL Statement Example B

A capability similar to ELSE of COMPASS can be simulated by the negation of the direct IF control statement. In the example in figure 9-3, MODEL is defined through DEF (as in DEF MODEL #76# or DEF MODEL #74#). Depending on the model, a one-bit is tested for 0 or 1.

```
CONTROL IFEQ MODEL, 76;
IF B<MFLAG> WORD [OPTS] EQ 0
    THEN RETURN;
CONTROL ENDIF;
CONTROL IFNQ MODEL, 76;
IF B<MFLAG> WORD [OPTS] EQ 1
    THEN RETURN;
CONTROL FI;
```

Figure 9-3. CONTROL Statement Example C

Similarly, a logical product (AND) of conditions can be satisfied by nested CONTROL statements. In the example in figure 9-4, a call to LOAD (TBL, XDEFNAME, FALSE) is generated when the model is not 76 and the system is neither ATS nor KRONOS. Notice that DEF is used within the conditional code to redefine SKIP.

OPTIMIZATION CONTROL

The SYMPL version 1.2 compiler introduces four CONTROL statement control-words that can be used to influence optimization performed by the compiler. None of these statements (OVERLAP, DISJOINT, INERT, REACTIVE) is required. In their absence, the compiler proceeds with its normal optimization. Because the consequences of some optimizations are unpredictable, default optimization is limited.

When the programmer informs the compiler that variables and array subscripts have been limited to uses with known consequences, the additional optimization can occur. Programs with such limits are called behaved, as opposed to unbehaved programs.

```
STATUS SYS ATS, INTCOM, KRONOS, S34, S2;
DEF SYSTEM . . . ;
DEF MODEL . . . ;
.
.
.
CONTROL IFNE MODEL, 76;
    DEF SKIP #1#;
    CONTROL IFEQ SYSTEM, SYS"ATS";
        LOAD (TAB, XDEFNAME);
        DEF SKIP #0#;
    CONTROL ENDIF;
    CONTROL IFEQ SYSTEM, SYS"KRONOS";
        LOAD (TAB, XDEFNAME, TRUE);
        DEF SKIP #0#;
    CONTROL ENDIF;
    CONTROL IFNE SKIP;
        LOAD (TAB, XDEFNAME, FALSE)
    CONTROL ENDIF;
CONTROL ENDIF;
```

Figure 9-4. CONTROL Statement Example D

The SYMPL Reference Manual contains details of the compiler optimization and the use of the optimization control-words. Future versions of the compiler might require these statements.

With or without the optimization CONTROL control-words, the SYMPL compiler performs optimization that moves code as it sees fit. A SYMPL programmer should not assume locations of any executable code.

To allow more, rather than less, optimization, a programmer should consider:

Initialization of a program in one procedure and the body of a program in another. (SYMPL does not move code from one procedure to another.)

Limiting of array subscripts to the bounds of the array, so that A[n] and B[m] are not the same word.

$BEGIN/$END DEBUGGING COMPILATION

Statements in a source program that are delimited by $BEGIN and $END are compiled only when the E parameter is specified on the SYMPL compiler call. Without the E parameter, such statements are shown in the source listing with a minus sign in the left margin, but they are not compiled. The $END statement must not be generated by a DEF. DEF is not expanded within $BEGIN and $END without the E parameter.

An example of this feature used to affect error output is shown in figure 9-5. CURSTAT is not allocated any memory space unless the E parameter is selected. The check of BYTETYP NQ S"INT" always compiles; in debug mode it produces a message, and in normal mode it does nothing.

SYMPL COMPILER CALL

The SYMPL compiler calls follow the conventions of other language processors, with I=INPUT, L=OUTPUT, and B=LGO parameter defaults. The compiler call using all defaults is:

SYMPL.

```
PROC PASSN;
BEGIN
    $BEGIN
    ITEM CURSTAT;
    $END

    PROC MISTAKE(CODE, AUX1, AUX2);
    BEGIN
    ITEM CODE, AUX1, AUX2;
        $BEGIN
        ERPRINT(CODE, CURSTAT, AUX1, AUX2);
        RETURN;
        $END
    END #PROC MISTAKE#
.
.
.
IF BYTETYP NQ S"INT"
THEN MISTAKE(ERR"NOTINT",IN[0], INX);
.
.
.
INPARMX=INPARMX+1;
    $BEGIN
    CURSTAT=CURSTAT+1;
    IF INPARMX GR INTYPE"MAX"
    THEN ERROR(ERR"INMAX");
    IF DEBUG0 THEN TRNACINT(INPARMX);
    $END
.
.
.
END #PROC PASSN#
```

Figure 9-5. Use of $BEGIN and $END

Other compiler call parameters are summarized in table 9-2. The SYMPL Reference Manual describes all parameters in detail.

Listings are controlled by any combination of LXOR=lfn:

X	Storage map and common block listing
O	Object code, lfn/line/line lists only code for source lines indicated by number
R	Cross reference map and common block listing

The time required to compile a program depends more on the length of the source code than on the number of declarations. On a CYBER 70 Model 73 system, about 2000 lines can be compiled per minute when full compilation is selected.

The total field length required for a given compilation depends on the length of the symbol table which, in turn, is dependent on the number of declarations rather than length of source code or statements. For each entry in the table, five words are required.

Field length requirements are, at minimum:

51K octal under NOS 1 and NOS/BE 1

41K octal under SCOPE 2

The SYMPL compiler is written, for the most part, in SYMPL.

Generated code might reference the FORTRAN library and SYSIO (NOS 1 and NOS/BE 1) or SYMIO (SCOPE 2) library. The FORTRAN library is expected to contain routines XTOI and ITOJ for exponentiation and routines for print input/-output. The SYSIO or SYMIO library is expected to contain the SYMPL execution-time routines SYMSM$, SYMSC$, and SYMSG$ for the more complex bit and character processing routine, SYMBSW$ for switch packing, and the SYMPL interface routines to the print facilities.

TABLE 9-2. COMPILER CALL PARAMETERS

Parameter	Significance
A	Abort after error
C	Check switch references for range
D	Pack switches two per word
E	Compile debugging statements within $BEGIN and $END
F	Generate procedure call parameter lists compatible with FORTRAN Extended
H	List all source statements despite any CONTROL NOLIST statement
I	Designate input file to be other than INPUT
N	List unreferenced items on cross reference map
P	Initialize (preset) items in labeled common
S=0	Suppress LDSET table generation
S=lib/lib	Generate LDSET table with entries for named libraries. Default is S=SYSIO/-FORTRAN for NOS 1 and NOS/BE 1; S=SYMIO/FORTRAN for SCOPE 2.
T	Suppress code generation
W	Single statement scheduling for closer correspondence between source statement order and object code order
Y	Suppress diagnostic 136, SEMI ENDS COMMENT

SYMPL has no input/output facilities. The SYMPL library does, however, contain procedures that are links to the PRINT routines of FORTRAN Extended.

To use the output features:

A FORTRAN Extended main program must call the SYMPL subprogram. The PROGRAM statement of the main program must specify the file OUTPUT.

The SYMPL program must specify the library procedures in an XREF declaration. Procedures PRINT and ENDL always are required; LIST is optional.

The SYMPL program must call both procedure PRINT and procedure ENDL for each output list to be printed. If variables are to be output, a LIST procedure call is required for each variable. PRINT, LIST, and ENDL form a single output sequence and must appear in that order, although intervening statements can appear.

The library procedures have alternative names PRINT$, LIST$, and ENDL$ for use when PRINT, LIST, or ENDL conflicts with a name used elsewhere in a program. The required externals are specified with an XREF declarative as shown in figure 10-1.

```
XREF BEGIN
     PROC PRINT;
     PROC LIST;
     PROC ENDL;
     END
```

Figure 10-1. Output XREF Declarations

The parameters for the SYMPL procedure calls are based on the FORTRAN statements. A FORTRAN Extended PRINT statement and its associated FORMAT statement have this format:

```
     PRINT label, parameter1, parameter2, . . .
label FORMAT (format specification)
```

The label of the FORMAT statement is not required for SYMPL output. The format specification specifies the format in which the parameters are to be output, including carriage control or Hollerith constant specifications. In SYMPL, this entire format, including its enclosing parentheses, must appear as a character string in a PRINT procedure call. Each FORTRAN parameter specifies a variable or array to be printed. In SYMPL, each item or array to be printed must appear in an individual LIST procedure call.

Any errors in the format specification and LIST arguments are detected during execution by the FORTRAN routines. The FORTRAN Extended Reference Manual explains any error messages that might result.

PRINT PROCEDURES

PRINT specifies the format in which information is to be output. Information appears on the file OUTPUT. Another library procedure, PRINTFL, is available for writing to files other than OUTPUT, as described in the SYMPL Reference Manual for PRINTFL discussion. The procedure call is:

```
PRINT("(specification)");
```

specification — String of characters duplicating the specification of a FORTRAN Extended PRINT statement. The specification can be any legal FORTRAN specification. Parentheses are required to be part of the string.

Examples of PRINT procedure calls are:

Assume a literal is to be printed. Either of the following can be specified:

```
PRINT ("(10H DISASTER)");

PRINT ("(* DISASTER*)");
```

Assume a character string item defined by:

```
ITEM SYNTABFORM C(40)=
#(6H HASH=O6, 11X6, 6HIDENT=2A10, . . .)#;
```

The string can be specified by simply:

```
PRINT(SYNTABFORM);
```

Assume an array item defined by ARRAY [1:9]; ITEM NDIGITS C(2)=[#I 1#,#I2#, . . . ,#I9#];

The entire array is specified by:

```
PRINT (NDIGITS[I]);
```

LIST AND ENDL PROCEDURE CALLS

LIST identifies one expression to be output. The procedure call is:

```
LIST(expression);
```

expression — Any item, subscripted array item, or expression to be output.

LIST must follow a PRINT procedure call or another LIST call. One LIST call must appear for each variable element of the PRINT specification.

The order of execution of the multiple LIST calls must correspond to the format of the preceding PRINT statement, just as the output specifications of a FORTRAN FORMAT statement must correspond to the order of parameters in the FORTRAN PRINT statement.

ENDL is required to end each output list. If no LIST calls appear, ENDL is still required. The procedure call is:

```
ENDL;
```

EXAMPLES

1. FORTRAN Extended statements and SYMPL statements that produce the same result are shown in figure 10-2a and figure 10-2b, respectively.

```
a.      PRINT 10
     10 FORMAT (*1 LIST OF IDENTIFIERS *)
        PRINT 20, LNAME, RNAME, HASH
     20 FORMAT (1H0,2A10,3X,I2)

b.      PRINT ("(*1 LIST OF IDENTIFIERS*)");
        ENDL;
        PRINT ("(1H0,2A10,3X,I2)");
        LIST(LNAME); LIST(RNAME); LIST(HASH);
        ENDL;
```

Figure 10-2. Output in FORTRAN and SYMPL

Output written is: LIST OF IDENTIFIERS lname rname hash where LNAME and RNAME are each 10 alphanumeric characters and HASH is a two-digit integer. In the SYMPL code, each variable is a parameter to a LIST procedure call.

2. The SYMPL code to output FATAL or NON-FATAL, depending on the current value of B, is shown in figure 10-3.

```
PRINT("(LX,AL)");
IF B THEN STR="FATAL";
     ELSE STR="NON-FATAL";
LIST(STR);
ENDL;
```

Figure 10-3. SYMPL Output Example A

3. To repeat the format for each iteration of a loop, the FORTRAN Extended routines perform the implicit DO loop, as shown in figure 10-4.

```
PRINT ("(11X,I10)");
FOR I=STKTOP STEP -1 UNTIL 0 DO
    LIST (STACK[I]);
ENDL;
```

Figure 10-4. SYMPL Output Example B

4. The SYMPL code to list array FLAG is shown in figure 10-5.

```
PRINT("(LX,10L3)");
FOR I=1 STEP 1 UNTIL 10 DO
    LIST(FLAG[I]);
ENDL;
```

Figure 10-5. SYMPL Output Example C

STANDARD CHARACTER SETS

A

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

CDC 64-character set

CDC 63-character set

ASCII 64-character set

ASCII 63-character set

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

Depending on another installation option, the system assumes an input deck has been punched either in 026 or in 029 mode (regardless of the character set in use). Under NOS/BE 1, the alternate mode can be specified by a 26 or 29 in columns 79 and 80 of the job statement or any 7/8/9 card. The specified mode remains in effect through the end of the job unless it is reset by specification of the alternate mode on a subsequent 7/8/9 card.

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

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

STANDARD CHARACTER SETS

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

†Twelve or more zero bits at the end of a 60-bit word are interpreted as end-of-line mark rather than two colons. End-of-line mark is converted to external BCD 1632.

††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 from ASCII/EBCDIC % yield a blank (55_8).

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

CDC CHARACTER SET COLLATING SEQUENCE									
Collating Sequence Decimal/Octal		CDC Graphic	Display Code	External BCD	Collating Sequence Decimal/Octal		CDC Graphic	Display Code	External BCD
00	00	blank	55	20	32	40	H	10	70
01	01	≤	74	15	33	41	I	11	71
02	02	%	63 †	16 †	34	42	∨	66	52
03	03	[	61	17	35	43	J	12	41
04	04	→	65	35	36	44	K	13	42
05	05	≡	60	36	37	45	L	14	43
06	06	∧	67	37	38	46	M	15	44
07	07	↑	70	55	39	47	N	16	45
08	10	↓	71	56	40	50	O	17	46
09	11	>	73	57	41	51	P	20	47
10	12	≥	75	75	42	52	Q	21	50
11	13	¬	76	76	43	53	R	22	51
12	14	.	57	73	44	54	]	62	32
13	15	)	52	74	45	55	S	23	22
14	16	;	77	77	46	56	T	24	23
15	17	+	45	60	47	57	U	25	24
16	20	$	53	53	48	60	V	26	25
17	21	*	47	54	49	61	W	27	26
18	22	-	46	40	50	62	X	30	27
19	23	/	50	21	51	63	Y	31	30
20	24	,	56	33	52	64	Z	32	31
21	25	(	51	34	53	65	:	00 †	none†
22	26	=	54	13	54	66	0	33	12
23	27	≠	64	14	55	67	1	34	01
24	30	<	72	72	56	70	2	35	02
25	31	A	01	61	57	71	3	36	03
26	32	B	02	62	58	72	4	37	04
27	33	C	03	63	59	73	5	40	05
28	34	D	04	64	60	74	6	41	06
29	35	E	05	65	61	75	7	42	07
30	36	F	06	66	62	76	8	43	10
31	37	G	07	67	63	77	9	44	11

†In installations using the 63-graphic set, the % graphic does not exist. The : graphic is display code 63, External BCD code 16.

ASCII CHARACTER SET COLLATING SEQUENCE

Collating Sequence Decimal	Collating Sequence Octal	ASCII Graphic Subset	Display Code	ASCII Code	Collating Sequence Decimal	Collating Sequence Octal	ASCII Graphic Subset	Display Code	ASCII Code
00	00	blank	55	20	32	40	@	74	40
01	01	!	66	21	33	41	A	01	41
02	02	"	64	22	34	42	B	02	42
03	03	#	60	23	35	43	C	03	43
04	04	$	53	24	36	44	D	04	44
05	05	%	63†	25	37	45	E	05	45
06	06	&	67	26	38	46	F	06	46
07	07	'	70	27	39	47	G	07	47
08	10	(	51	28	40	50	H	10	48
09	11	)	52	29	41	51	I	11	49
10	12	*	47	2A	42	52	J	12	4A
11	13	+	45	2B	43	53	K	13	4B
12	14	,	56	2C	44	54	L	14	4C
13	15	-	46	2D	45	55	M	15	4D
14	16	.	57	2E	46	56	N	16	4E
15	17	/	50	2F	47	57	O	17	4F
16	20	0	33	30	48	60	P	20	50
17	21	1	34	31	49	61	Q	21	51
18	22	2	35	32	50	62	R	22	52
19	23	3	36	33	51	63	S	23	53
20	24	4	37	34	52	64	T	24	54
21	25	5	40	35	53	65	U	25	55
22	26	6	41	36	54	66	V	26	56
23	27	7	42	37	55	67	W	27	57
24	30	8	43	38	56	70	X	30	58
25	31	9	44	39	57	71	Y	31	59
26	32	:	00†	3A	58	72	Z	32	5A
27	33	;	77	3B	59	73	[	61	5B
28	34	<	72	3C	60	74	\	75	5C
29	35	=	54	3D	61	75	]	62	5D
30	36	>	73	3E	62	76	^	76	5E
31	37	?	71	3F	63	77	_	65	5F

†In installations using a 63-graphic set, the % graphic does not exist. The : graphic is display code 63.

GLOSSARY B

Actual parameter – The name of an entity, or a value, that is passed to a procedure or function when the procedure or function is referenced.

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. The compiler creates a pointer variable which is set with a specific value during program execution.

Bead – One of a string of bits or characters.

Bounds – The upper and lower limits of an array dimension.

Call-by-address – A formal parameter that requires an actual parameter to be an address rather than a value. Call-by-address is required for any parameter whose value is to be returned to the referenced procedure or function.

Call-by-value – A scalar name parameter which is enclosed in parentheses in the formal parameter list to indicate to the compiler that a value rather than an address is to be passed as an actual parameter.

Comment – A string of characters, except the semicolon, enclosed within pound signs.

Common – Storage that is referenced by more than one subprogram.

Compound statement – A statement which begins with the reserved word BEGIN, and ends with the reserved word END.

Declaration – Defines the type and use of data and entities used in a program.

Dimensionality – The number of bounds specifications associated with an array.

Entities – A generic term which refers to any combination of items, arrays, based arrays, labels, procedures, and so forth.

Expression – A sequence of identifiers, constants, or function calls, separated by operators and parentheses, the evaluation of which yields a resultant value.

Fast loop – A type of loop within a FOR statement where test and branch are contained within the loop, so that the loop must execute at least once.

Formal parameter – The name of an entity, which appears in the header of a procedure or function declaration, for which a value or the name of an entity is passed when the procedure or function is referenced.

Function – A subprogram, headed by a function declaration, used within an expression. The value returned through the function name is used in evaluation of that expression.

Identifier – A string of 1 through 12 letters, digits, or $ beginning with a letter, that is used to name an entity within a program.

Intrinsic function – A function that can be referenced without any declaration.

Parallel array – An array in which the first words of each entry are allocated contiguously, followed by the second words of each entry, and so forth.

Procedure – A subprogram, headed by a procedure declaration, that executes when its name, or one of its alternative entry points, is called.

Programmer-supplied function – A function that must be declared before it can be referenced.

Scalar – A single element of data that occupies at least one word of storage.

Serial array – An array in which all the words of one element are allocated contiguously.

Slowloop – A type of loop within a FOR statement where test and branch occur at the beginning of the loop, so that the loop need not execute at all.

Statement – Specifies the operations to be performed during execution of the program.

Switch – A concept similar to the GO TO statement in FORTRAN. The compiler assigns a value, starting at 0, to each label named in the SWITCH declaration.

INDEX

COMMENT SHEET

CONTROL DATA CORPORATION

TITLE: SYMPL Version 1 User's Guide

PUBLICATION NO. 60499800 REVISION A

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

Applications for which you use this manual.

Do you find it adequate for your purpose?

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

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

General comments:

FROM NAME: _____ POSITION: _____

COMPANY NAME: _____

ADDRESS: _____

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

FOLD ON DOTTED LINES AND STAPLE

STAPLE

STAPLE

FOLD

FOLD

FIRST CLASS
PERMIT NO. 8241

MINNEAPOLIS, MINN.

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

POSTAGE WILL BE PAID BY

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

FOLD

FOLD