Computer Note CN-14.2 August 1975

SIMPL-T:

A Structured Programming Language

by

Victor R. Basili

and

Albert J. Turner

Computer Science Center * UNIVERSITY OF MARYLAND * College Park, Md. 20742

Preface

SIMPL-T is a member of a family of languages that are designed to be relatively machine independent and whose compilers are relatively transportable onto a variety of machines. It is a procedure oriented, non-block structured programming language that was designed to conform to the standards of structured programming and modular design. There are three data types in SIMPL-T: integer, string and character.

The first member of the SIMPL family, the typeless language SIMPL-X, was bootstrapped onto the 1108 in the Fall of 1972. The implementation of SIMPL-T was completed in January, 1974.

This Computer Note is primarily intended as the reference manual for SIMPL-T. However since it is anticipated that it will be used in teaching SIMPL-T, the material has been organized so that the manual can be used in the classroom.

We would like to acknowledge the work of Hans Breitenlohner who wrote the execution time monitor and provided assistance in trying to interface with the idiosyncrasies of EXEC 8. Acknowledgments also go to Mike Kamrad and Bruce Carmichael for their work on the bootstrap and Eleanor B. Waters and Dawn Shifflett for the typing of the the main portion of this note.

This project was supported in part by the Office of Naval Research under Grant N00014-67-A-0239-0021 (NR-044-431) to the Computer Science Center of the University of Maryland, and in part by the Computer Science Center of the University of Maryland.

Victor R. Basili
Department of Computer Science

Albert J. Turner
Computer Science Center

Preface to Later Editions

Several enhancements have been made to the SIMPL-T Compiler since this document was originally written. At each new printing, however, updates have been made to maintain the accuracy of the user documentation.

The printing history and compiler version number corresponding to each printing is given below.

Computer Note	Printing Date	Compiler
14	January 1974	1.0
	August 1974	1.2
14.1	January 1975	1.5
14.2	August 1975	1.7

Contents

1. Introduction

This manual describes the implementation of the language SIMPL-T for the Univac 1106/1108 computers using the Exec 8 operating system. SIMPL-T is essentially an extension of the language SIMPL-X, which is described in: V. R. Basili, "SIMPL-X, a language for writing structured programs", Tech. Rept. TR-223, U. of Md. Comp. Sc. Center, Jan. 1973.

SIMPL-T is a procedure-oriented, non-block-structured programming language that was designed to conform to the standards of structured programming and modular design. SIMPL-T is intended to be the base language for a family of languages that will include a systems programming language and the graph language GRAAL (W. Rheinboldt, V. Basili, and C. Mesztenyi, "On a programming language for graph algorithms", BIT 12, 1972). The design and development of the SIMPL family of languages is being done at the University of Maryland Computer Science Center and Computer Science Department.

In order to avoid the inclusion of much material that would be superfluous for the anticipated readers, it is assumed that the reader has some knowledge of a general purpose programming language such as FORTRAN, BASIC, ALGOL, or PL/I.

The manual is designed so that the basic features are presented first, and more specialized features are presented later. Chapter 2 contains a description of the basic language that is sufficient to get a novice SIMPL-T programmer "on the air". Chapter 2, Chapter 3, and most, if not all, of Chapter 4 contain the material that would normally be covered in a programming course (with selected topics from Chapter 5 perhaps also being included). Most of the material in Chapter 5 is for those who are familiar with the language and have special purpose requirements. Chapter 6 contains information about using the SIMPL-T compiler, and Chapter 7 contains assorted information about the 1106/1108 Exec 8 implementation.

A language feature that has not yet been explained is sometimes used in an example in order to provide a more illustrative example.

The meaning of such a feature should be clear from its usage to those who are familiar with another general purpose programming language but if not, the feature is always explained soon after such a usage. It should also be noted that the examples are designed primarily to illustrate the SIMPL-T language and thus they may not always illustrate the best way to solve a particular programming problem.

Braces ({ }) are used to denote optional syntax and the symbols < and > are used to enclose the name of a general syntactic entity. For example, the syntax for a call statement can be specified by

CALL <identifier> {(<parameter list>)}

This means that a call statement consists of the word CALL followed by an identifier. The identifier may optionally be followed by a parameter list, enclosed in parentheses. Words such as CALL are called keywords.

2. The Basic SIMPL-T Language

2.1 Program Structure

The syntax for a SIMPL-T program is illustrated by

{<declaration list>} <segment list> START <identifier>

The <declaration list> defines the variables that may be used anywhere in the program. The <segment list> is a collection of procedures (subroutines) and functions, and <identifier> names the procedure with which execution is to begin. (The <segment list> may consist of only a single procedure.)

The following example illustrates this program structure.

```
INT X,Y                                   } declaration list

PROC PRINTSUM(INT A, INT B)
WRITE (A+B)

PROC MAINPROG                             } segment list
X := 3
Y := 4
CALL PRINTSUM(X,Y)

START MAINPROG
```

(The result of this program is that 7 is printed.)

Thus a SIMPL-T program contains a (possibly empty) set of global declarations and a set of procedures and functions. Execution begins with one of the procedures, and the procedures and functions are called as needed during execution.

2.1.1 Declarations

The initial declaration list of a program contains declarations for all variable identifier names that are global. A global identifier is an identifier that is known to all segments of a program. Only global declarations for integer variables and integer arrays are discussed in this section.

2.1.1.1 Integer Declaration

An integer variable may have any integer value between $-2^{35} + 1$ and $2^{35} - 1$, inclusive. An integer variable declaration consists of the keyword INT followed by one or more identifier names, separated by commas. Initialization may also be specified as illustrated by the following valid declaration list.

```
INT X
INT CAT, DOG1
INT M=3, N=-1, I
```

In the above example M and N are initialized to the values 3 and -1, respectively. This means that these variables will have the specified values when execution of the program begins. The value of an uninitialized variable is initially undefined. (Actually globals will have value 0 on the 1106/1108 unless the B MAP option is specified.)

2.1.1.2 Integer Array Declaration

The only data structure in SIMPL-T is the one-dimensional array. This is an ordered collection of elements, all of the same data type. The elements are numbered 0, 1,..., n-1, where n is the number of elements in the array.

Integer array declarations begin with the keywords INT ARRAY, and are completed by listing the array identifiers and the number of elements for each array. The number of elements must be a positive integer, and is enclosed in parentheses. For example,

```
INT ARRAY TOTALS(10)
```

declares an array of 10 elements: TOTALS(0), TOTALS(1),..., TOTALS(9).

An array can also be initialized by specifying a list of values for the array elements. Initialization begins with the first element (number 0) and proceeds until the list is exhausted (or all array elements are exhausted). A repetition factor can be specified by enclosing the factor in parentheses following the initialization value.

Some examples are

```
INT ARRAY A(3), BAT(95), VECTOR(20)
INT ARRAY A1(10), B(5) = (2,3,-1)
INT ARRAY C(11) = (0,1,3(9))
```

The second declaration specifies that B(0) , B(1) , and B(2) are to be initialized to 2 , 3 , and -1 , respectively. The third declaration initializes C(0) to 0 , C(1) to 1 , and C(2)-C(10) to 3 .

2.1.1.3 Declaration List

A declaration list, such as the list of global declarations at the beginning of a program, consists of one or more declarations. Declarations follow one another with no separator (except blanks). More than one declaration for the same type can appear in a declaration list. All identifiers used in a program must be declared.

An example of a declaration list is

```
INT X, Y
INT I
INT ARRAY INPUTS(100),OUTPUTS(50)
INT SUM
INT ARRAY SUMS(20) = (0(20))
```

2.1.2 Segments

A segment is a procedure or function definition. Segments contain a list of statements to be executed when the segment is invoked.

2.1.2.1 Procedures

The syntax for a procedure definition is illustrated by

```
PROC <identifier> {(<parameter list>)} {<local declaration list>}
     <statement list> {RETURN}
```

where <identifier> is the name of the procedure.

An example of a procedure definition is

```
PROC TEST (INT X, INT Y)
/* THIS PROC PRINTS THE SUM OF X AND Y */
WRITE (X+Y)
```

A procedure is a subroutine that, when invoked, executes its <statement list> and returns to the caller. A procedure may access any global identifier (unless the procedure has a local identifier by the same name) as well as its local identifiers and parameters.

The items of the <parameter list> , separated by commas, are of the form INT <identifier> or INT ARRAY <identifier> . These parameters are passed to the procedure when it is invoked (called).

Integer parameters are passed by value (unless otherwise specified as in 4.2). This means that if a procedure changes the value of an integer parameter, the new value is effective only to that procedure. For example, if procedure P is defined by

```
PROC P(INT X)
X := 7
```

and the statements

```
X := 3
CALL P(X)
WRITE(X)
```

are executed, then the number printed will be 3 (not 7).

Array parameters, however, are passed by reference. Logically, this means that the array itself is passed (rather than the value as for integer parameters). Thus any modification to an array parameter by a procedure will be a modification to the actual array passed as an argument by the caller. For example, if procedure Q is defined by

```
PROC Q(INT I, INT ARRAY A)
A(I) := 7
```

and the statements

```
A(2) := 3
CALL Q(2, A)
WRITE(A(2))
```

are executed, then 7 will be printed.

2.1.2.2 Functions

The function definition syntax is illustrated by

INT FUNC <identifier> {(<parameter list>)} {<local declaration list>} {<statement list>} RETURN(<expression>)

A function is similar to a procedure. The main differences are

1) the value of <expression> is returned (as the value of the function evaluation) to be used in the same manner as the value of a variable would be used;
2) functions may not have side effects, that is, they may not change the values of any nonlocal variables or arrays.

(Note that (2) is assumed but not enforced. Those who insist on writing functions with side effects should see 7.4 .)

2.1.2.3 Local Declarations

All local variables and arrays must be declared in the local declaration list. Local declarations are similar to global declarations, but initialization is not allowed. (The values of local variables at entry to a segment are undefined.)

2.1.3 Scope of Identifiers

Global identifiers, including segment names, are accessible from all segments unless a segment declares a local with the same name as a global. Local declarations override global declarations so that a global identifier is not available to a segment in which that identifier is declared local.

Local identifiers are only accessible to the segment in which they are declared. Both globals and locals may be passed as parameters. The value of all locals is undefined at entry to the segment, and locals do not necessarily retain their values between successive calls to the segment.

2.2 Comments and Blanks

Blanks may appear anywhere in a SIMPL-T program except within an

identifier, symbol, keyword, or constant. Blanks are significant delimiters and may be needed as separators for identifiers or constants. For example,

```
IF X
```

and

```
IFX
```

are not equivalent.

A comment is any character string enclosed by /* and */ . (See 6.1 for a modification of this convention.) A comment may appear anywhere that a blank may occur and has no effect on the execution of a program. The following illustrates a comment:

```
/* THIS IS A COMMENT. */
```

2.3 Statements

The syntactic entity <statement list> denotes any sequence of SIMPL-T statements. No separators (other than blanks) are used between statements.

2.3.1 Assignment Statement

The syntax of the assignment statement is given by

```
<variable> := <expression>
```

where <variable> is either a simple variable (i.e., an integer identifier) or a subscripted variable. The assignment statement causes the value of the <expression> to be assigned to the <variable> . Examples of valid SIMPL-T assignment statements are

```
X := Y+Z
X := Y=Z
A(I):= A(I+1)+A(J-2)*X
```

2.3.2 If Statement

The IF statement causes conditional execution of a sequence of one or more statements. The syntax is

```
IF <expression>
    THEN <statement list>1
```

{ELSE <statement list>$_2$} END

At execution, the value of the <expression> determines the action taken. If the value is nonzero, <statement list>$_1$ is executed and <statement list>$_2$ (if there is an else part) is skipped. If the value is zero, <statement list>$_2$ (if it exists) is executed and <statement list>$_1$ is not executed. Execution proceeds with the next statement (following END) after execution of either <statement list> .

Example

```
IF X<3 .AND. Y<X
    THEN
        Y:=X
    ELSE
        X:=X+1
        Y:=Y-1
        IF X>Y
            THEN
                X:=Y
            END
    END
```

Note that the ELSE part of the main IF statement also contains an IF statement that will be executed only if the ELSE part is executed.

Example.

```
IF X THEN Y:=Y/X ELSE Y:=Y/2  END
```

This statement divides Y by X if X is nonzero and divides by 2 if X is zero.

2.3.3 While Statement

The WHILE statement provides a means of iteration (looping):

WHILE <expression> DO <statement list> END

The value of the <expression> determines the action at execution time, just as for the IF statement. If the value of <expression> is nonzero, then <statement list> is executed; otherwise <statement list> is skipped and execution proceeds with the statement following END . How-

ever, if <statement list> is executed, then execution proceeds with the WHILE statement again. Thus if <expression> is nonzero, then <statement list> is executed until <expression> becomes zero.

Example. The following statement list sums the odd and even integers from 1 to 100.

```
ODD := 0
EVEN := 0
I := 0
WHILE I<100
DO
    I := I+1
    IF I/2 * 2 = I
        THEN /* EVEN INTEGER */
            EVEN := EVEN + I
        ELSE /* ODD */
            ODD := ODD + I
        END
END
```

2.3.4 Case Statement

Exactly one of a group of statement lists may be executed by using the CASE statement. The syntax is illustrated by

CASE <expression> OF
\n$_1$\ <statement list>$_1$
\n$_2$\ <statement list>$_2$
⋮
\n$_k$\ <statement list>$_k$
{ELSE <statement list>$_{k+1}$} END

where each $n_1, n_2, \ldots, n_k$ is a constant or a negated constant.

If the value of <expression> is n_j, then <statement list>$_j$ is executed and the other statement lists are not executed. If <expression> does not evaluate to any of the n_i's, then the ELSE part (<statement list>$_{k+1}$) is executed, if there is an ELSE part, and none of the state-

ment lists is executed if there is no ELSE part. The cases may be in any order, and more than one case designator \n_i\ may be used with the same statement list, as is illustrated in the following example.

Example

```
CASE X*Y+Z OF
    \1\
        X := 3
    \2\
        IF X<Y
            THEN
                X := Y
            END
        Y := Y+1
    \4\ \6\ /* CASES 4 AND 6 COINCIDE */
        X := 2
        Y := 3
    ELSE
        X := 0
    END
```

2.3.5 Call Statement

The CALL statement

CALL <identifier> {(<argument list>)}

causes the procedure named <identifier> to be executed. Each argument in the argument list may be an expression or an array, and the arguments must agree in number and type with the parameters in the procedure definition for the procedure that is called. Arguments in <argument list> are separated by commas.

Upon completion of the execution of the procedure, execution resumes with the statement following the CALL statement.

Example. To invoke the procedure DOIT with arguments X+Y and the array A , the statement

CALL DOIT (X+Y, A)

is used.

2.3.6 Example

```
PROC SORT (INT N, INT ARRAY A)

/* THIS PROCEDURE USES A BUBBLE SORT ALGORITHM TO SORT THE
    ELEMENTS OF ARRAY 'A' INTO ASCENDING ORDER.  THE VALUE
    OF THE PARAMETER 'N' IS THE NUMBER OF ITEMS TO BE SORTED.  */

INT SORTED, /* SWITCH TO INDICATE WHETHER FINISHED */
    LAST,   /* LAST ELEMENT THAT NEEDS TO BE CHECKED */
    I,      /* FOR GOING THROUGH ARRAY */
    SAVE    /* FOR HOLDING VALUES TEMPORARILY */

IF N>1
  THEN /* SORT NEEDED */
    SORTED := 0 /* INDICATE NOT FINISHED */
    LAST := N-1 /* START WITH WHOLE ARRAY */

    WHILE .NOT. SORTED
      DO /* CHECK CURRENT SEQUENCE FOR CORRECTNESS */
        SORTED := 1 /* ASSUME FINISHED */
        I := 1      /* INITIALIZE ELEMENT POINTER */

        WHILE I <= LAST
          DO /* COMPARE ADJACENT ELEMENTS UP TO 'LAST' */
            IF A(I-1) > A(I)
              THEN /* OUT OF ORDER */
                SAVE := A(I)      /* INTERCHANGE */
                A(I) := A(I-1)    /* A(I) AND    */
                A(I-1) := SAVE    /* A(I-1)      */
                SORTED := 0       /* MAY NOT BE FINISHED */
              END
            I := I+1
          END /* LOOP FOR COMPARING ADJACENT ELEMENTS */
        /* A(LAST),..., A(N-1) ARE NOW OK */
        LAST := LAST -1
      END */ LOOP FOR CHECKING CURRENT SEQUENCE */
  END /* IF N>1 */

/* END PROC 'SORT' */
```

2.4 Integer Expressions

An integer expression represents an integer value. An integer expression may be

1) a scalar integer variable (either a simple variable or a subscripted array variable);
2) an integer constant;
3) an integer function call;
4) an integer operation (such as + or -) where each operand may also be an expression;
5) an integer expression enclosed in parentheses.

2.4.1 Subscripted Array Variables

An array element is designated by following the array name with a subscript, enclosed in parentheses, whose value designates the number of the array element to be used. The subscript can be any integer expression.

For example

```
A(3)
```

designates the 4^{th} element of array A , while

```
A( X + A(Y) )
```

designates the element whose number is the value of X plus the value of the array element designated by A(Y) .

2.4.2 Function Calls

A function call has the form

<identifier> {(<argument list>)}

where <identifier> is the name of the function. The rules for <argument list> are the same as for the CALL statement.

2.4.3 Constants

An integer constant may be designated by any sequence of decimal

digits representing a valid non-negative integer value. Note that negative constants may usually be used where desired although such a constant is formally viewed as the unary minus operation on a nonnegative constant in integer expressions.

For example, the following are valid SIMPL-T integer constants.

```
3
35927
0
123456789
```

2.5 Basic Integer Operators

The operators described in this section all have integer expressions as operands and yield an integer result. Any arithmetic overflow that occurs in a calculation is ignored.

2.5.1 Arithmetic Operators

Addition (+) , subtraction (-) , and multiplication (*) are binary operators with the usual meaning. The integer divide (/) operator yields the integer quotient of its operands. Thus if the result of X/Y is Q , then X = Q*Y + R , where R is the remainder that was discarded in the integer divide.

The unary minus (-) operator yields the negative of its operand. Note that the expression -3 is formally viewed as the unary minus operation on the constant 3 although it would probably be logically (and equivalently) viewed as the constant "minus three" by the programmer. There is no unary plus operator in SIMPL-T.

2.5.2 Relational Operators

The relational operators are equal (=) , not equal (<>) , less than (<) , less than or equal (<=) , greater than (>) , and greater than or equal (>=) . The expression X=Y has value 1 if X and Y are equal, and value 0 otherwise. The remaining relational operators are similarly defined.

Note that the result of a relational operation always has value 1

or zero, depending on whether the relation is true or false, respectively. The relational operators can also be denoted by .EQ. , .NE. , .LT. , .LE. , .GT. , and .GE. , respectively.

(A note of caution regarding arithmetic overflow generated by a relational operation is in 7.5.)

2.5.3 Logical Operators

The logical operators .AND. , .OR. , and .NOT. are defined by:

X.AND.Y is 1 if both X and Y are nonzero, and is 0 otherwise

X.OR.Y is 0 if both X and Y are zero, and is 1 otherwise

.NOT.X is 1 if X is zero and is 0 otherwise

As is the case for relational operations, a logical operation always yields the result 1 or 0 .

Note that the logical operators yield the "natural" result. For example, the expression

```
X<Y .AND. Y<Z
```

will have the value 1 (i.e., will be "true") if Y is both greater than X and less than Z , and will have the value 0 (i.e., will be "false") otherwise.

2.5.4 Precedence

The precedence of the basic integer operations, from highest to lowest, is

.NOT. - (unary)	unary
* /	arithmetic
+ - (binary)	
= <> < > <= >=	relational
.AND.	logical
.OR.	

The order of evaluation between operators of equal precedence is left to right (except between unary operators, which is right to left).

As an example, the expression

```
- A + B + C * D
```

would be evaluated by

(1) negating the value of A

(2) adding the value of B to the result from (1)

(3) multiplying the value of C by the value of D

(4) adding the results from (2) and (3)

Parentheses may be used to alter the normal precedence. Thus (A+B)*C would cause the values of A and B to be added and the result to be multiplied by the value of C .

2.5.5 Examples

The following are examples of valid SIMPL-T expressions.

(1) X + Y/7 * 2

(2) X<3 .OR. X>8

(3) X>3 .AND. X+Y<10

(4) X + (X*(Y+1)<500)

For X=9 and Y=12 these expressions have the values

(1) 11

(2) 1

(3) 0

(4) 10

2.6 Identifiers

Identifiers (i.e., names) in SIMPL-T may be any string of letters or digits that begins with a letter. For usage in an identifier, the symbol $ is considered to be letter. Identifiers are used to denote variables, arrays, procedures, functions, and other entities in a program. All identifiers used in a program (except SIMPL-T intrinsic identifiers, such as READ) must be declared.

There is no formal restriction on the length of identifiers. However identifiers may not cross the boundary of a source input record (e.g., card), so that there is an actual restriction to the length of an input

record (e.g., 80 characters).

Certain reserved words (keywords) may not be used as identifiers in a SIMPL-T program. These keywords (such as IF , INT) are listed in Appendix V. Due to the special meaning given to these keywords, rather disastrous results may occur if a keyword is used as an identifier in a SIMPL-T program. This is especially true of keywords used in declarations (such as INT , ARRAY , PROC). The resulting diagnostics generated by the compiler may not be too helpful for such an error, primarily because the programmer often overlooks this type of error as a possible cause of the diagnostics.

Since many keywords are used for more specialized features of the SIMPL-T language, the list in Appendix V should be consulted before writing a SIMPL-T program.

2.7 Basic I/O

2.7.1 READ

READ may be used to read values from job stream input (card, teletype, etc.) into integer variables. Values to be read are placed on input records (cards, teletype lines, etc.) as decimal constants separated by blanks or commas (or both). Negative values are indicated by placing a minus sign before the number to be read. A value may not cross the boundary of an input record.

To illustrate the READ statement, the statements

```
READ(X,Y,A(I+J))
READ(I,J)
```

and the input

```
3, -2, 5 7
10 , 12
```

would cause the same results as

```
X := 3
Y := -2
A(I+J) := 5
I := 7
J := 10
```

Thus the input to be read in is considered to be a stream of numbers rather than a sequence of cards (or lines, etc.). The numbers are read one by one and numbers are not skipped unless explicit directions to do so are specified. Skipping to the beginning of an input record can be specified by using the arguments SKIP , SKIP0 , SKIP1 , SKIP2 ,..., SKIP9 . (SKIP is the same as SKIP1 .)

The effect of a skip argument is as follows:

SKIP0 - skip to the beginning of the current card (for reread)
SKIP , SKIP1 - skip to the beginning of the next card
etc.

The skip directive is relative to the last value read from the input stream. Thus, for example, successive

```
READ(X, SKIP)
```

statements would cause the first value to be read from each card that has a value on it, regardless of the number of values on a card.

To illustrate further, the statements

```
READ(X, SKIP, Y)
READ(SKIP, Z)
READ(SKIP)
READ(I)
```

and the input

```
3, 5, 7
2
0
1, 4
10
```

would be the same as the assignments

```
X := 3
Y := 2
Z := 0
I := 1
```

The READ statement can also be used to read in values for an entire array. For example, if X is an integer variable and A is an integer array of 10 elements, the statement

```
READ(X, A)
```

would read the next input item into X , and the following 10 items into A(0), A(1),..., A(9) .

The intrinsic function EOI may be used to determine the end of the input. The result of the function EOI is given by

$$EOI = \begin{cases} 1 & \text{if no more items are available for READ} \\ 0 & \text{if one or more items remain to be read} \end{cases}$$

Note that the value of EOI is determined on the basis of values, not input records (such as cards), **remaining** to be read. The use of EOI is illustrated in 2.7.3.

2.7.2 WRITE

Values of expressions may be printed by using WRITE . The values to be printed are considered to be a stream of values that are placed at tab positions that provide columns 8 characters in width. A line is not printed until it is filled, unless a skip or eject argument is used.

The carriage control parameters that can be used are

```
EJECT  - skip to the top of the next page
SKIP0  - start over on the current line (overprint)
SKIP , SKIP1  - print the current line
SKIP2  - print the current line and double-space
SKIP3 }
  :   } similar to SKIP2
SKIP9 }
```

Each argument of WRITE may be an expression, an array, or a carriage control specification. As an example, if X = 3 , Y = 2 , and I = 10 , the statements

```
WRITE(X, 2*X + 3*Y)
WRITE(I, I/X, SKIP, Y)
WRITE(SKIP)
```

would cause

```
3      12      10      3
2
```

to be printed.

To illustrate the use of WRITE with an array argument, if A is an array with 20 elements, the statement

```
WRITE(A)
```

is equivalent (assuming that I is not used for something else) to

```
I := 0
WHILE I<20
  DO WRITE(A(I))
     I := I+1   END
```

2.7.3 Example

```
/* THIS PROGRAM READS IN A SET OF UP TO 100 NUMBERS, SORTS
   THEM INTO ASCENDING ORDER, AND PRINTS OUT TWO COLUMNS IN
   WHICH THE LEFT COLUMN IS THE ORIGINAL SET OF NUMBERS AND
   THE RIGHT COLUMN IS THE SORTED SET.  THE PROCEDURE 'SORT'
   FROM 2.3.6 IS USED. */

INT N  /* NUMBER OF VALUES TO SORT */
INT ARRAY A(100),  /* INPUT SET */
          B(100)   /* SORTED SET */
PROC READINPUT

N := 0
WHILE .NOT. EOI .AND. N<100
  DO /* PUT NEXT VALUE INTO 'A' AND 'B' */
    READ(A(N))
    B(N) := A(N)
    N := N+1   /* COUNT VALUES */
  END
PROC PRINT
INT I

I := 0
WHILE I<N
  DO /* PRINT LINES OF OUTPUT */
    WRITE (A(I), B(I), SKIP)
```

```
        I := I+1
      END

      :    Proc SORT from 2.3.6
      :
    PROC READSORTANDPRINT

    CALL READINPUT
    CALL SORT(N,B)
    CALL PRINT

    START READSORTANDPRINT
```

An example of the result of executing this program is

```
  5      -25
 -2       -2
  3        0
  0        3
 10        5
-25       10
 17       17
```

2.8 Example

```
/* THIS PROGRAM READS A SEQUENCE OF NOT MORE THAN 100 NONZERO INTEGERS.
   THE INTEGERS MUST BE IN INCREASING ORDER AND MUST BE FOLLOWED BY A
   0 (UNLESS 100 INTEGERS ARE TO BE READ).  THE LIST OF VALUES READ
   IS PRINTED.  ADDITIONAL VALUES ARE THEN READ AND A 'BINARY SEARCH'
   IS USED TO DETERMINE IF EACH VALUE IS A MEMBER OF THE SEQUENCE
   READ INITIALLY.  EACH VALUE IS PRINTED WITH ITS POSITION IN THE
   SEQUENCE (0 IF NOT IN THE SEQUENCE).  */

  INT FUNC SIGN(INT X)
  /* FUNCTION WHOSE VALUE IS:  1 IF X>0
                               0 IF X=0
                              -1 IF X<0       */

  IF X<0
    THEN
      RETURN(-1)
```

```
    ELSE
      RETURN(X>0)
    END
  /* END FUNC 'SIGN' */

  PROC SEARCH
  /* MAIN PROCEDURE */

  INT N, /* NUMBER OF VALUES READ */
      FOUND, /* SWITCH TO INDICATE WHETHER VALUE WAS FOUND */
      INDEX, /* POSITION OF VALUE IN SEQUENCE */
      LO,    /* LOWER BOUND OF INTERVAL FOR SEARCH */
      HI,    /* UPPER BOUND OF INTERVAL FOR SEARCH */
      KEY    /* VALUE READ TO BE LOOKED FOR */

  INT ARRAY TABLE(101)  /* SEQUENCE */

  N := 0              /* INITIALIZE */
  TABLE (0) := 1      /* FIX UP FOR FIRST READ */

  WHILE N<100 .AND. TABLE(N)<>0
    DO /* READ SEQUENCE */
      N := N+1
      READ (TABLE(N))
      IF TABLE(N)<>0
        THEN
          WRITE(TABLE(N))
        END
    END /* LOOP FOR READING SEQUENCE */

  IF TABLE(N) = 0
    THEN
      N := N-1     /* FIX UP FOR COUNTING THE ZERO */
    END
  WRITE(SKIP)      /* END LINE OF SEQUENCE VALUES */

  WHILE .NOT. EOI
```

```
    DO /* READ AND LOOK UP VALUES */
      READ(KEY)
      WRITE(KEY)
      /* INITIALIZE FOR SEARCH */
      FOUND := 0
      HI := N
      LO := 1          /* INITIAL INTERVAL IS WHOLE ARRAY */
      WHILE LO <= HI .AND.  .NOT. FOUND
        DO /* BINARY SEARCH */
          INDEX := (LO+HI)/2  /* LOOK AT MIDPOINT OF INTERVAL */
          CASE SIGN(TABLE(INDEX)-KEY) OF
            \0\  /* TABLE(INDEX)=KEY -- (FOUND) */
              FOUND := 1
            \1\  /* TABLE(INDEX)>KEY */
              HI := INDEX-1  /* TRY LOWER INDICES */
            \-1\ /* TABLE(INDEX)<KEY */
              LO := INDEX+1  /* TRY HIGHER INDICES */
          END
        END /* LOOP FOR BINARY SEARCH */
      IF FOUND
        THEN
          WRITE(INDEX,SKIP)
        ELSE
          WRITE(0,SKIP)
        END
    END /* LOOP FOR READING AND LOOKING UP VALUES */
/* END PROC 'SEARCH' */
START SEARCH
```

For this program, the input

```
2   3   5   8   10   11   15   0
2   1   0   8    7        15   18
```

would produce the output

2	3	5	8	10	11	15
2	1					
1	0					
0	0					
8	4					
7	0					
15	7					
18	0					

3. String Data

3.1 Introduction

In this chapter a second data type, string, is discussed. A string is a (finite) sequence of characters. The number of characters in the sequence is called the length of the string, and the string of length 0 is called the null string. The characters may be any of the ASCII characters (see Appendix III) although most of the 1108 peripherals do not allow the use of all ASCII characters.

3.2 Constants

A string constant is denoted by enclosing the string in apostrophes. (Note that computer people usually call apostrophes "quotes".) Any apostrophe in the string is indicated by using two apostrophes. Examples are

```
'THIS IS A STRING'
'THERE IS AN APOSTROPHE ('') IN THIS STRING'
```

The length of the first string above is 16. The length of the second is 41, and printing it would yield

```
THERE IS AN APOSTROPHE (') IN THIS STRING
```

The null string constant is denoted by '' .

A string constant may not exceed 256 characters in length.

3.3 Variables

A string variable has a maximum length associated with it. The value of a string variable is a string, and the maximum length limits the length of the string value.

3.4 Declarations

3.4.1 Scalar

A string declaration includes the specification of the maximum length for the value of the string variable. This specification is made by en-

closing the maximum length (a positive integer constant) in brackets following the string identifier. The maximum length may not exceed 4095.

Examples are

```
STRING S[5],  T[50]
STRING MESSAGE [10] = 'HELLO',  RESULT[20]
```

The first declaration defines strings S with maximum length 5, and T with maximum length 50. In the second declaration, the maximum length of MESSAGE is specified to be 10, and MESSAGE is initialized with the value 'HELLO' (so that the current length is initially 5).

The value of an uninitialized string variable is undefined.

3.4.2 Array

All elements of a string array must have the same maximum length. Thus a string array declaration must include the maximum length specification as well as the number of elements (number of strings) in the array.

String array declarations are illustrated by

```
STRING ARRAY INPUT [50] (100)
STRING ARRAY MESSAGES [20] (10) = ('MESSAGE 0', 'MESSAGE 1'),
            SA[13] (25) = ('ABC', 'XYZ' (3), 'CAT')
```

In these declarations, the array INPUT contains 100 strings of maximum length 50 each. MESSAGES (0) is initialized to 'MESSAGE 0' and 'MESSAGE 1' is the initial value of MESSAGES (1) . Execution of the statements

```
I := 0
WHILE I<5
  DO
    WRITE (SA(I), SKIP)
    I := I+1
  END
```

at the beginning of the program would produce the output

```
ABC
XYZ
XYZ
XYZ
CAT
```

3.5 Operators

3.5.1 Concatenate

The concatenate operator .CON. generates a string by joining together its two operand strings end-to-end. As an illustration,

```
'ABCD ' .CON. 'EFG' = 'ABCD EFG'
```

3.5.2 Substring

The substring operator generates a string by extracting a substring from its (string) operand. In the form

```
[F1, F2]
```

this operator extracts the substring of length F2 beginning with character number F1 of the operand string. (The first character is character number 1.)

To illustrate the substring operator, consider the following.

```
'ABCDEF' [3, 2] = 'CD'
'XYZ'[3, 1] = 'Z'
'ABCD' [1,4] = 'ABCD'
```

The two fields F1 and F2 of the substring operator may be any integer expressions. The F2 field may be omitted, in which case the substring from character F1 to the end is implied. For example,

```
'DOGCAT' [4] = 'CAT'
```

(Note: The symbols << and >> may also be used for [and] , respectively.)

If the value of F2 is nonzero, then the substring defined by [F1, F2] must lie within the current bounds of its operand string; otherwise execution is terminated with an "invalid substring" error. The following are not valid.

```
'ABC'  [3 ,2]
'ABC'  [0, 2]
'ABC'  [2,-1]
```

If the value of F2 is zero, then the substring operator is always valid and returns the null string. Additionally, S[F1] returns the null string whenever the value of F1 is greater than the length of S .

(Note: Assignment to a substring is discussed in 3.12 .)

3.5.3 Relational Operators

The relational operators (=, <>, <, <=, >, >=) may be used with string operands. The result is either the integer 0 or the integer 1 , just as for integer operands, as determined by the ASCII collating sequence (see Appendix III). Strings of unequal length are never equal.

Some examples illustrating these operators are given below.

```
'ABC' < 'ABD'
'ABC' < 'ABCD'
'ABC' < 'ABC '
'ABA' > 'AB1'
'123' < '124'
 '+'  > '-'
```

3.6 String Expressions

The string operators may have string expressions as operands. The precedence for unparenthesized expressions is (highest to lowest)

1) substring
2) concatenate
3) relational operators

(Note that the result of a relational operation with string operands is of type integer, and hence a relational operation (even with string

operands) is an integer expression.)

Thus a string expression is

1) a string constant, string variable (simple or subscripted), or string function call;
2) a substring or concatenate operation, whose operands can be string expressions;
3) a string expression enclosed in parentheses.

Examples are given in later sections of this chapter.

3.7 Assignment Statement

The assignment statement for strings is similar to the assignment statement for integers. Its syntax is given by

<string variable> := <string expression>

No automatic conversion between string and integer exists in SIMPL-T. Thus strings may not be assigned to integer variables and integers may not be assigned to string variables.

If the string represented by <string expression> has a length that does not exceed the maximum length of the variable, then the value of the variable is set to the value of <string expression>. If the value of <string expression> is too long for <string variable>, then the value of <string expression> is truncated to the maximum length of <string variable> before the assignment is made. For example, if S is declared by

```
STRING S[5]
```

and the assignment

```
S := '123456'
```

is made, then S will have the value '12345', regardless of the value of S before the assignment.

3.8 Example

This example uses the built-in function LENGTH, that returns the length of a string value. The following procedure REMOVE removes

a given substring from a given string.

```
PROC REMOVE (STRING SUB, STRING STR)
/* THIS PROC PRINTS THE RESULT OF REMOVING THE (SUB)STRING
   'SUB' FROM THE STRING 'STR'. */

INT CHARPTR, FOUND

CHARPTR := 1
FOUND   := 0
WHILE CHARPTR + LENGTH(SUB) <= LENGTH(STR) + 1 .AND. .NOT. FOUND
  DO /* CHECK FOR OCCURRENCE OF 'SUB' BEGINNING AT 'CHARPTR' */
    IF STR[CHARPTR, LENGTH(SUB)] = SUB
      THEN /* FOUND */
        FOUND := 1
      ELSE
        CHARPTR := CHARPTR + 1
      END
  END /* LOOP */

IF FOUND
  THEN /* SUBSTRING 'SUB' IS AT POSITION 'CHARPTR' OF 'STR' */
  WRITE (STR[1, CHARPTR -1] .CON. STR[CHARPTR + LENGTH(SUB)])
  ELSE /* NO OCCURRENCE OF 'SUB' IN 'STR' */
    WRITE(STR)
  END

/* END PROC 'REMOVE' */
```

3.9 I/O

READ and WRITE (and EOI) also may be used for strings. The rules for strings are similar to those for integers.

Strings to be read in must be indicated on the input medium just as a string constant would be indicated in a SIMPL-T program (i.e., enclosed in apostrophes with any apostrophe in the string being indicated by two apostrophes). Commas and blanks or both may be used to separate input items, and strings and integers may be freely intermixed.

If the length of a string that is read in is greater than the maximum length of the string variable into which it is read, the input string is truncated to the maximum length of the variable. Thus if S is a string variable,

```
READ(S)
```

with input

```
'ABCD'
```

would be completely equivalent to

```
S := 'ABCD'
```

Just as for assignments, no mixed types are permitted in a READ. Thus, for example, if X is an integer variable and the statement

```
READ(X)
```

is executed with

```
'345'
```

as the next input item, an error termination will occur.

WRITE will cause string expression values to be printed at predetermined tab positions, just as for integers. However string values are left-justified in the columns, rather than right-justified as for integer values. If a string is too long for the current line, it will be continued on the following line.

3.10 Example

```
/* THIS PROGRAM READS IN A LIST OF UP TO 99 NAMES AND PRINTS THEM
   OUT IN ALPHABETICAL ORDER.  THE NAMES MAY NOT BE MORE THAN 50
   CHARACTERS LONG. */
```

```
STRING ARRAY IN[50](100) = ('') /* FOR HOLDING THE NAMES */
INT N = 0,  /* NUMBER OF NAMES */
    I,      /* FOR GOING THROUGH 'IN' */
    SAVE,   /* FOR REMEMBERING A SPOT IN 'IN' */
    FINISHED = 0  /* SWITCH */

PROC SORT

WHILE .NOT. EOI
  DO /* READ NAMES INTO 'IN' */
     N := N + 1
     IF N < 100
       THEN /* NOT TOO MANY */
         READ (IN(N))
       END
  END
IF N > 99
  THEN /* TOO MANY NAMES INPUT */
    WRITE ('TOO MANY NAMES - ONLY FIRST 99 USED', SKIP)
    N := 99
  END

WHILE .NOT. FINISHED
  DO /* PRINT OUT SORTED NAMES */
    SAVE := 0
    I := 1

    WHILE I <= N
      DO */ GO THROUGH AND GET FIRST NAME IN ALPHABETICAL ORDER */
        IF IN(I) <>''.AND. (SAVE = 0 .OR. IN(I) < IN(SAVE))
          THEN /* THIS NAME HASNT BEEN PRINTED, AND IT IS THE
            FIRST ONE FOUND OR SHOULD COME BEFORE THE CURRENT
            CANDIDATE */
          SAVE := I  /* USE THIS ONE AS THE FIRST ALPHABETICALLY
                        SO FAR */
          END
        I := I + 1  /* GO ON TO NEXT NAME */
    END  /* LOOP TO GO THROUGH AND GET FIRST NAME IN ALPHA. ORDER */
```

```
      IF SAVE <> 0
        THEN /* A NAME WAS FOUND */
          WRITE (IN(SAVE), SKIP)    /* PRINT IT */
          IN(SAVE) := ''            /* MARK IT AS 'PRINTED' */
        ELSE /* ALL NAMES HAVE BEEN PRINTED */
          FINISHED := 1
        END
    END /* LOOP TO PRINT OUT NAMES */

START SORT
```

An example of the results of executing the program are indicated below.

Input:	Output:
'HERZOG'	BUTLER
'MCKISSICK'	COOLEY
'BUTLER'	HERZOG
'COOLEY'	LIEBERSOHN
'MANSFIELD'	MANSFIELD
'LIEBERSOHN'	MCKISSICK
'SCHMEISSNER'	ROSE
'ROSE'	SCHMEISSNER

3.11 String Functions

3.11.1 User Defined Functions

A string function is a function those value is a string (i.e., the function "returns" a string). The rules governing the use of string functions are analogous to the rules for integer functions (2.1.2.2). The following illustrates the use of string functions.

```
STRING FUNC ALPHABETIZE(STRING S)
/* THIS FUNC REARRANGES THE CHARACTERS OF STRING 'S' INTO ALPHABETICAL
   ORDER.  'S' MAY HAVE A MAXIMUM OF 256 CHARACTERS. */

STRING RESULT[256],  /* FUNCTION RESULT */
       NEXTCHAR[1]   /* NEXT CHARACTER IN ALPHABETICAL ORDER */
INT CHARNUM,  /* FOR LOOKING THROUGH CHARACTERS IN 'S' */
    NEXTNUM   /* POSITION OF 'NEXTCHAR' IN 'S' */
```

```
RESULT:=''
WHILE S<>''
  DO  /* EXTRACT NEXT CHAR (IN ALPHABETICAL ORDER) OF 'S' */
    NEXTCHAR := S[1,1]  /* START WITH FIRST CHAR */
    NEXTNUM:=1
    CHARNUM:=2
    WHILE CHARNUM<=LENGTH(S)
      DO  /* LOOK THROUGH CHARS OF 'S' FOR "SMALLEST" */
        IF S[CHARNUM,1] < NEXTCHAR
          THEN
            NEXTCHAR := S[CHARNUM,1]
            NEXTNUM:=CHARNUM
          END
        CHARNUM:=CHARNUM+1
      END  /* LOOP TO LOOK THROUGH CHARS OF 'S' */
    RESULT := RESULT .CON. NEXTCHAR  /* ADD NEXT CHAR TO RESULT */
    S := REMOVE(S,NEXTNUM)
  END  /* LOOP TO EXTRACT NEXT CHAR */

RETURN(RESULT)
/*  END FUNC 'ALPHABETIZE' */


STRING FUNC REMOVE(STRING S, INT CHARNUM)
/*  FUNC TO REMOVE CHARACTER NUMBER 'CHARNUM' FROM 'S' */
RETURN(S[1,CHARNUM-1] .CON. S[CHARNUM+1])
/* END FUNC 'REMOVE' */
```

3.11.2 Intrinsic Functions

In this section some intrinsic (built-in) functions that facilitate programming with strings are described.

3.11.2.1 LENGTH

The function LENGTH returns the length of the value of its argument. The argument may be any string expression, and the result is of type integer.

Example

```
LENGTH ('ABC') = 3
LENGTH ('ABC' .CON. 'DE') = 5
```

3.11.2.2 MATCH

The MATCH function is used to find an occurrence of a substring in a string. The syntax is of the form

```
MATCH (S1, S2)
```

where S1 and S2 may be any string expressions. MATCH returns the character number in S1 of the first character of (the first occurrence of) the string S2. If S2 is not a substring of S1 , then MATCH returns 0 .

As an illustration, suppose that S = 'ABCATDOG' and T = 'AT' . Then

```
MATCH (S, T) = 4
MATCH (T, S) = 0
MATCH (S, 'A') = 1
MATCH (S, 'CAT') = 3
MATCH (S, 'DOGS') = 0
MATCH (S, 'CATS') = 0
```

3.11.2.3 INTF

INTF is used to convert a string of decimal digits (or a minus sign followed by decimal digits) into an integer value. If the string

contains a character that is not a digit (other than a leading minus) then the program is terminated.

In the following example, let S1 = '123' and S2 = '017' . Then

```
INTF (S2) = 17
INTF (S1 .CON. S2) = 123017
INTF ('-' .CON. S2) = -17
INTF ('12345' [2,3]) = 234
```

3.11.2.4 STRINGF

STRINGF is the inverse of INTF . That is, STRINGF converts the value of an integer expression to string. As examples, let I = 22 and J = -15 . Then

```
STRINGF (I) = '22'
STRINGF (J) = '-15'
STRINGF (I+J) = '7'
STRINGF (I-I) = '0'
```

The result of STRINGF is a string with no leading zeros. Thus the length of the string returned by STRINGF is the number of significant digits in the integer value, plus 1 if the value is negative.

3.11.2.5 LETTERS

The function call LETTERS (<string expression>) returns a 1 (integer) if each character in the string is a letter (A-Z) and a 0 otherwise. The letters may be upper or lower case (or both).

3.11.2.6 DIGITS

The value of DIGITS (<string expression>) is 1 if each character in the string is a digit (0-9) and 0 otherwise.

3.11.2.7 TRIM

The result of

TRIM (<string expression>)

is the value of <string expression> truncated to remove trailing blanks.

3.12 Substring Assignment

The substring indicator may also be used on the left side of an assignment statement:

<string variable> [<first char>, <length>] := <string expression>

The rules for <first char> and <length> are the same as for the substring operator in string expressions (3.5.2). The second field (, <length>) may also be omitted, just as for the string expression operator.

When the substring indicator is used in this manner, the substring specified by [<first char>, <length>] is replaced by the first <length> characters of the (string) value of <string expression> . The remaining characters of <string variable> are not changed. If needed, the value of <string expression> is extended on the right with blanks so that its length is not less than <length> .

To illustrate, let S1 = 'ABCDE' and S2 = '123456' . Then after the assignments

```
S1 [2, 3] := 'XYZ'
S2 [2, 2] := '?'
S2 [5]    := '**XX'
```

the values of S1 and S2 will be

```
S1 = 'AXYZE'
S2 = '1? 4**'
```

Note that the length of the value of <string variable> cannot be changed by a substring assignment.

3.13 String Parameters

Strings passed as arguments to (user) procedures and functions are passed by value unless otherwise specified as in 4.2 . String arrays are passed by reference. These conventions are the same as for integers and integer arrays, as explained in 2.1.2.1 .

The maximum length of a string parameter passed by value is set to the maximum length of the argument when the call occurs. Note that

a string expression that is not a string variable (such as S.CON.T , S[I,J], or 'STRING CONSTANT') has a maximum length equal to the length of its value.

All arguments passed to intrinsic functions are by value. That is, an intrinsic function will not change the value of a parameter.

3.14 Example

```
/* THIS PROGRAM REPLACES ALL SUBSTRINGS BETWEEN '/*' AND '*/' BY
   BLANKS */

STRING INPUT [80]
INT PTR1, PTR2

PROC REMOVECOMMENTS

WHILE .NOT. EOI
  DO
    READ (INPUT)
    PTR1 := 1           /* INITIALIZE FOR SEARCH */
    WHILE PTR1 <> 0
      DO  /* REMOVE SUBSTRINGS */
        PTR1 := MATCH (INPUT, '/*')
        IF PTR1 <> 0
          THEN /* FOUND BEGINNING */
            PTR2 := MATCH (INPUT, '*/')
            IF PTR2 > PTR1 + 1
              THEN /* FOUND END (AFTER BEGINNING) */
                INPUT [PTR1, PTR2 - PTR1 + 2] := '' /* BLANK IT OUT */
              END
          END
      END
    WRITE (INPUT, SKIP)
  END
/* END PROC 'REMOVECOMMENTS' */

START REMOVECOMMENTS
```

For the input

```
'XXX /*  COMMENT 1 */ YYY /* COMMENT  2 */'
'PTR1 := 1  /* INITIALIZE FOR SEARCH */'
'WHILE PTR1 <> 0'
'  DO /* REMOVE SUBSTRINGS */'
```

the program would produce the output

```
XXX                  YYY
PTR1 := 1
WHILE PTR1 <> 0
  DO
```

4. Additional Language Features

4.1 Escape Mechanisms

4.1.1 Exit Statement

The EXIT statement provides a means of escaping from a WHILE loop. In its basic form, the statement

```
EXIT
```

causes the immediate termination of the (innermost) WHILE statement containing the EXIT statement. Execution proceeds as if the WHILE statement has terminated normally.

The use of EXIT is illustrated by the following function.

```
INT FUNC FIND (INT NUMBER, INT ARRAY VALUES, INT SIZE)
/* FUNC TO RETURN THE SUBSCRIPT OF THE ELEMENT OF 'VALUES' HAVING
  VALUE 'NUMBER'.  IF 'NUMBER' IS NOT IN 'VALUES', THEN 0 IS RETURNED.
  THE VALUES TO BE CHECKED ARE IN VALUES(1), ..., VALUES(SIZE). */

INT I

I := 1
WHILE I <= SIZE
  DO
    IF VALUES(I) = NUMBER
      THEN /* FOUND */
        EXIT
      ELSE
        I := I + 1
      END
  END
IF I > SIZE
  THEN /* NOT FOUND */
    I := 0
  END
RETURN(I)

/* END FUNC 'FIND' */
```

An exit of more than one level of nesting can also be performed by using an EXIT statement. To do so, the exit statement has the form

EXIT (<designator>)

where <designator> denotes the WHILE statement to be terminated. A <designator> is an identifier that is specified in the form

\<designator>\ WHILE ...

to designate the WHILE loop to be exited.

Consider the following program segment:

```
\LOOP1\  WHILE I <= N    /* LOOP 1 */
   DO
      .
      .
      .
      WHILE 1    /* LOOP 2 (WILL NOT TERMINATE WITHOUT AN EXIT) */
         DO
            .
            .
            .
            IF I + J = K
              THEN
                EXIT
              ELSE
                IF I + J > K
                  THEN
                    EXIT (LOOP1)
                  END
              END
              .
              .
              .
         END /* LOOP 2 */
      I := I + 2
   END /* LOOP 1 */
X := I
```

If the statement EXIT is executed, then the next statement to be executed would be I := I + 2 (which is in the WHILE statement designated by \LOOP1\ ). However if the statement EXIT(LOOP1) is executed, the next statement to be executed would be X := I (the next statement after the

WHILE loop designated by \LOOP1\ ).

A WHILE designator may be any identifier that has no other meaning in the segment (procedure or function) in which it is used.

4.1.2 Return Statement

The RETURN statement causes a return to the calling procedure or function. It may be any statement in a segment. The form

```
RETURN
```

is used for procedures, and the form

```
RETURN (<expression>)
```

is used for functions.

The function FIND of 4.1.1 may be rewritten to illustrate this statement:

```
INT FUNC FIND (INT NUMBER, INT ARRAY VALUES, INT SIZE)
INT I
I := 1
WHILE I <= SIZE
  DO
    IF VALUES (I) = NUMBER
      THEN /* FOUND */
        RETURN (I)
      ELSE
        I := I + 1
      END
  END
RETURN (0)  /* NOT FOUND */
/* END FUNC 'FIND' */
```

Note that the last statement in a function need not be a RETURN (<expression>) if the structure of the function's statement list is such that a return is always made from within the statement list.

4.1.3 Abort Statement

The statement

ABORT

causes the immediate (abnormal) termination of an entire SIMPL-T program, regardless of the location of the ABORT statement or the depth of segment call nesting.

4.2 Parameter Passing By Reference

Procedures may communicate scalar (integer or string) results through the parameters passed to it by specifying that a parameter is a reference parameter. Logically, this means that the scalar variable itself is passed to the procedure rather than the value of the variable, just as for array parameters. Thus a procedure can then change the value of a variable in a CALL argument list.

A procedure declares a scalar parameter to be a reference parameter by means of the keyword REF . The following program illustrates the difference between normal parameter passing (by value) and reference parameters.

```
INT X

PROC ADD1 (INT X, INT Y)
X := X + Y

PROC ADD2 (REF INT X, INT Y)
X := X + Y

PROC MAIN
X := 3
CALL ADD1 (X, 2)
WRITE (X)
CALL ADD2 (X, 2)
WRITE (X)

START MAIN
```

This program would print

```
3    5
```

Note that only variables (simple or subscripted) may be passed by reference. That is, neither constants nor expressions (that do not consist

of a variable only) may be passed by reference. (In particular, a substring may not be passed by reference.)

Functions may also have reference parameters.

4.3 Recursive Segments

A segment that calls itself, either directly or indirectly, must be declared recursive. This is done by including the keyword REC before the segment definition. A segment that does not call itself can also be declared recursive in order to cause the dynamic (rather than static) allocation of locals (thus using no storage for the locals of the segment until the segment is invoked).

The following recursive function computes the factorial of an integer.

```
REC INT FUNC FACTORIAL (INT N)
IF N < 2
  THEN
    RETURN (1)    /* 0! = 1, 1! = 1 */
  ELSE
    RETURN (N* FACTORIAL (N-1))  /* N! = N* (N-1)! */
  END
/* END FUNC 'FACTORIAL' */
```

4.4 Access Between Separately Compiled Program Modules

It is often convenient to construct a program in two or more separately compiled modules rather than as a single compilation. The modules are compiled separately and then combined by the collector (@MAP) for execution. However, in this type of program construction it is often required that not only procedures and functions but also data in one module be accessible from within another module.

Since separate compilations are independent (that is, identifiers from one compilation are not known in any other compilation), special facilities are needed in order to provide the desired capabilities. In order for a SIMPL-T module (module 2) to access a procedure, function, or data from another module (module 1), two things are required:

a) the module (module 1) that contains the procedure, function, or data must make it available to other modules by specifying it as an entry point;

b) the module (module 2) that wishes to access the procedure, function, or data must specify that it is an external reference.

If these requirements are met, module 2 may use the identifier denoting the procedure, function, or data just as if the identifier were declared normally in module 2.

4.4.1 Entry Points

Segments and data in a SIMPL-T program module may be made accessible to separately compiled program modules by declaring such a segment or data item as an entry point. This is done by preceding the usual declaration by the keyword ENTRY .

Examples

```
ENTRY INT I, J = 2
ENTRY STRING ARRAY S[10](20)
ENTRY REC PROC P(INT X) ...
```

Only global identifiers may be entry points. Due to an EXEC 8 restriction, entry point names may not exceed 12 characters in length. Truncation to 12 characters is performed if needed.

4.4.2 External References

In order to access an entry point of a separately compiled program module, the identifier to be accessed must be declared an external reference. The keyword EXT is used for this purpose.

External declarations for data items are similar to normal declarations. The differences are that initialization of externals is not allowed, and the size specification for strings and arrays may be omitted (since they are defined in another module). Examples of external data declarations are

```
EXT INT ARRAY VALUES
EXT STRING S , T
EXT STRING ARRAY SA[17](32)
```

External segment declarations must include a specification of the types of the parameters. Illustrations are

```
EXT PROC P(INT, STRING)
EXT INT FUNC FIND (INT, INT ARRAY)
EXT PROC REMOVE (REF INT, STRING),
        INITIALIZE
```

External declarations may be global or local. Just as for entry points, an external name may not exceed 12 characters and truncation is used, if needed, to meet this restriction. The START specification at the end of a program may name an external procedure as the procedure to be initially invoked.

4.4.3 Nonexecutable Programs

Only one of a group of separately compiled program modules may specify a START procedure. The remaining modules are called nonexecutable, and are so designated by omitting the procedure name following START . (A nonexecutable module must have at least one ENTRY declaration to be of value.)

A nonexecutable module may consist of (entry point) data items only. That is, a nonexecutable module need not have any segments.

4.4.4 Example 1

The two modules given here illustrate the use of external references and entry points. The combined program reads in 50 numbers, sorts them into increasing order, and prints them.

Module 1:

```
ENTRY INT  ARRAY NUMBERS(50)
EXT PROC SORT

PROC MAIN
READ (NUMBERS)
CALL SORT
WRITE (NUMBERS)

START MAIN
```

Module 2:

```
EXT INT ARRAY NUMBERS

ENTRY PROC SORT
INT I, SAVE, LAST, INTERCHANGED

INTERCHANGED := 1
LAST := 49

WHILE INTERCHANGED
  DO /* BUBBLE SORT */
    I := 1
    INTERCHANGED := 0
    WHILE I <= LAST
      DO
        IF NUMBERS(I) < NUMBERS (I-1)
          THEN
            SAVE := NUMBERS (I-1)
            NUMBERS (I-1) := NUMBERS(I)
            NUMBERS(I) := SAVE
            INTERCHANGED := 1
          END
        I := I + 1
      END
    LAST := LAST -1
  END
/* END PROC SORT */
START
```

4.4.5 Example 2

The ability to access data in a separately compiled module is not absolutely necessary, since data can be accessed through argument lists. (The capability of data access between modules is needed for practical considerations, however.) As an illustration, consider the following modifications to the program in the previous example. The program obtained by combining modules 1A and 2A is equivalent to the program consisting of modules 1 and 2 above.

Module 1A:

```
INT ARRAY INPUT(50)
EXT PROC SORT(INT ARRAY)

PROC MAIN
READ (INPUT)
CALL SORT(INPUT)
WRITE (INPUT)

START MAIN
```

Module 2A:

```
ENTRY PROC SORT(INT ARRAY NUMBERS)
INT I, SAVE, LAST, INTERCHANGED

      .
      .  same as module 2 above
      .
```

4.4.6 Executing a Program Having Multiple Modules

The procedure for executing a SIMPL-T program consisting of separately compiled modules is similar to that for other languages. For example, the program consisting of modules 1 and 2 above could be compiled and executed by

```
      .
      .
      .
@SIMPLT,I   PROG1
      .
      .  module 1  source
      .
```

```
@SIMPLT,I   PROG2
      .
      .  module 2  source
      .
@MAP
@XQT
      .
      .  data
      .
```

Additional information can be found in various EXEC 8 user documentation.

4.5 DEFINE Facility

A restricted macro capability exists in the compiler. This facility exists regardless of options specified. Macros are declared in a manner similar to that for other SIMPL declarations, and are invoked whenever the macro name is used as an identifier in the program. Macro parameters and nested macro calls (including recursive calls) are allowed.

A brief description of this facility follows:

4.5.1 Macro Definition

A macro definition has the syntax

DEFINE <define list>

where <define list> is a list of one or more definitions, separated by commas. Each definition has the form

<identifier> = <string constant>

where <identifier> is a normal SIMPL identifier, and <string constant> is a normal SIMPL string constant (enclosed in apostrophes). Macro parameters are denoted in the defining <string constant> by &n , where n is a digit between 1 and 9, inclusive, that refers to the argument number. The &n is replaced by the argument when the macro is invoked.

4.5.2 Example

The following program (written for illustration only) prints the integers 1-10, modulo 4.

```
DEFINE
  INCR    = '&1 := &1+1' ,
  ASSIGN  = '&1 := &2' ,
  FOREVER = 'WHILE 1' ,
  MOD     = '&1-&1/&2*&2'

INT I=0, J

PROC MAIN

FOREVER
  DO
    IF I >= 10
      THEN EXIT
      ELSE
        INCR(I)
        ASSIGN(J,MOD(I,4))
        WRITE(J)
      END
  END
START MAIN
```

4.5.3 Macro Expansion

Arguments to a macro are separated by commas and the argument list is enclosed in parentheses. Each argument may be

1) a string constant (enclosed in apostrophes), in which case the value of the string constant is substituted for the formal parameter in the defining <string constant> ;
2) any string of characters not including a comma or right parenthesis, except that nested parentheses are allowed and a comma may appear between nested parentheses. In this case, the character string minus leading and trailing blanks is substituted for the formal parameters.

When a macro identifier is found during the processing of source text, the following occurs:

1) A copy is made of the macro definition;
2) The formal parameters in the copy of the defining string are replaced by the actual parameters from the argument list of the macro invocation;
3) The expanded macro then replaces the macro invocation (macro id and argument list) in the source text, and processing of the source text resumes with the expanded macro.

4.5.4 Conventions and Restrictions

1) SIMPL comments (/* ... */) are removed from a <string constant> that defines a macro.
2) The usual scope rules apply for macro declaration. This means that macros may be defined as locals if desired. Note, however, that a global macro identifier cannot be redefined as a local without first turning off the macro expansion facility (see below) since the occurrence of the identifier in the local declaration list would invoke the global macro.
3) A macro invocation occurs whenever a macro identifier is found in the text. This means that a macro cannot be invoked within a string constant or comment, for example.
4) The argument list for a macro invocation is optional. An empty list () is allowed.
5) An argument that is not a string constant may not cross an input line boundary. An argument list must begin on the same line as the macro identifier.
6) No more than 9 formal parameters are allowed. Missing arguments are considered to be null, and arguments corresponding to no formal parameter are ignored.
7) The total length of all macro definitions (concatenated) cannot exceed 4000 characters.
8) An expanded macro plus the remainder of the line where the macro is invoked cannot exceed 400 characters. The length of all arguments concatenated cannot exceed 400 characters.
9) If an expanded macro plus the remainder of the line will not fit on one line, the expanded text is split as needed at the last blank before a line boundary.
10) No more than 50 macro invocations (including nested invocations) can occur from a single line of source text.
11) Note that neither ENTRY nor EXT may be used with DEFINE.

4.5.5 Options

The directives

/+ EXPANDOFF +/ and /+ EXPANDON +/

can be used to disable the expansion facility for any portion of source text. The expansion is initially ON. The directives

/+ EXPANDPRINTON +/ and /+ EXPANDPRINTOFF +/

can be used to obtain a listing of macro expansions. The print facility is initially OFF.

5. Some Special-Purpose Language Features

5.1 Character Data Type

5.1.1 Introduction

In order to facilitate a more efficient implementation of some string handling algorithms, SIMPL-T has a third data type: character. A character is any ASCII character (see Appendix III).

The addition of character data does not facilitate the writing of SIMPL-T programs. In fact, just the opposite may be true: some algorithms are more difficult to code using character data than if strings were used. However some computers, such as the 1106 and 1108, do not have machine instructions that facilitate the efficient implementation of string operations such as substring extraction. Thus the character data type exists in SIMPL-X so that character-oriented algorithms can be implemented more efficiently (with respect to execution time) on such machines.

In general, the addition of character data to SIMPL-T involves mostly straightforward extensions of the integer and string handling concepts. Some of these extensions, and the variations that are needed for character data, are explained below.

5.1.2 Constants

Character constants are denoted by enclosing the character in quotation marks ("). A character constant may also be denoted by C '<integer constant>' where <integer constant> specifies the numerical value of the ASCII encoding of the character.

Examples of character constants are

```
"A"    "1"    "?"    """"    C'13'    C'Ø'137''
```

5.1.3 Declarations

Scalar character declarations are similar to scalar integer declarations. Examples are

```
CHAR C, C1 = "X"
ENTRY CHAR C2
EXT CHAR C3
```

Character array declarations are also similar to those for integer arrays. One difference is that a string constant can be included in the initialization list for a character array. The meaning is that the elements of the array are to be initialized with successive characters of the string. Examples are:

```
CHAR ARRAY CA1(10),
           CA2(20) = ("A", 'CAT', "X")
ENTRY CHAR ARRAY B(10)
EXT CHAR ARRAY CA
```

In this example, CA2(1) is initialized to "C", CA2(2) to "A", and CA2(3) to "T".

5.1.4 Statements

The assignment statement has the form

<character variable> := <character expression>

for character data. Both sides of the assignment statement must be of type character.

The case statement can be used in the form

CASE <character expression> OF
\c_1\ <statement list>$_1$
.
.
.
\c_n\ <statement list>$_n$
{ELSE <statement list>$_{n+1}$} END

where $c_1,\ldots,c_n$ are character constants. The form of the character case statement is illustrated by

```
CASE NEXTCHAR OF
\"A"\ CALL CASEA
\"B"\\"X"\ CALL CASEBX
\"?"\ CALL WHAT
ELSE CALL OTHERCASE
END
```

5.1.5 Operators

The only operators that may have character operands are the relational (=, <, etc.) operators. Both operands must be of type character, and the result is the same as it would be for single character strings consisting of the operand characters.

Thus a character expression can only be a character variable (simple or subscripted), a character constant, or a character function call.

5.1.6 Intrinsic Functions and Procedures

5.1.6.1 INTVAL

INTVAL (<char expr>)

returns the integer whose value is the binary ASCII encoding of the character argument (see Appendix III). Examples are

INTVAL ("A") = 65
INTVAL (" ") = 32
INTVAL ("a") = 97

5.1.6.2 CHARVAL

The result of

CHARVAL (<int expr>)

is of type character and is the inverse of INTVAL . Thus, for example,

CHARVAL (65) = "A"
CHARVAL (32) = " "

The value of the argument must have a value between 0 and 127, inclusive.

5.1.6.3 INTF

The INTF function also may be applied to character data. As examples,

INTF ("1") = 1
INTF ("9") = 9

The value of a character argument must be one of the characters "0" , "1" , ..., "9" .

5.1.6.4 STRINGF

STRINGF also may have a character argument although no function is required for this conversion (see 5.1.8). The result is a string of length 1 consisting of the character. For example

STRINGF ("A") = 'A'

STRINGF ("3") = '3'

5.1.6.5 CHARF

The function CHARF converts a string or integer argument to character. For string arguments, CHARF (<string expression>) is the first character of the string. For integer arguments,

CHARF (<int expr>) = CHARF (STRINGF(<int expr>))

Thus, for example,

CHARF ('ABC') = "A"

CHARF (7) = "7"

CHARF (-17) = "-"

5.1.6.6 LETTER

LETTER is an integer function defined by

LETTER (<char expr>) = LETTERS (STRINGF(<char expr>))

5.1.6.7 DIGIT

The integer function DIGIT is defined by

DIGIT (<char expr>) = DIGITS (STRINGF(<char expr>))

5.1.6.8 UNPACK

UNPACK is an intrinsic procedure that stores the successive characters of a string into successive elements of a character array. The format is

{CALL} UNPACK (<string expr>, <char array>)

For example, if S = 'CAT' and A is a character array, then the statement

UNPACK (S,A)

would result in A(0) = "C", A(1) = "A", etc.

The string argument is extended on the right with blanks or truncated

so that its length is the same as the number of elements in the array. Thus every element of the array is given a value. Note that the string argument is first and the character array second.

5.1.6.9 PACK

The intrinsic procedure PACK is the reverse of UNPACK. The format is

{CALL} PACK (<char array>, <string variable>)

All characters of the character array are used unless the maximum length of <string variable> is too small, in which case only the first k characters of the array are used, where k is the maximum length of <string variable> .

5.1.7 I/O

READ and WRITE are extended in a straightforward manner for character data, except as noted below. Characters to be read are enclosed in quotation marks just as for character constants in a SIMPL-T program.

One difference in I/O for character data is that a string can be read into a character array. This works just as if the string were read into a string variable and then UNPACKed into the array. Similarly, a WRITE of a character array is the same as doing a PACK and a WRITE of the resulting string.

5.1.8 Characters as Strings

The set of characters is considered to be a subset of the set of strings. Thus a character is also considered to be a string (of length 1) and may be used as a string without explicit conversion (STRINGF). Note that the converse is not true: no string may be used as a character without explicit conversion (CHARF).

5.1.9 Summary

Due to the special nature of character data, its usage has not been explained fully. It is expected that those who wish to use it will be advanced enough to extend the integer and string features logically to character. Other features not mentioned above, such as character functions,

character parameters, etc., extend in a straightforward manner (e.g., characters are passed by value unless declared by reference, but character arrays are passed by reference).

5.2 Bit Representation for Integer Constants

Integer constants may be specified in binary, octal, or hexadecimal, as well as decimal. However, these additional representations specify the bit pattern for the word in which the integer is stored, rather than the value of the integer. Thus a maximum of 36 bits may be specified for the 1106/1108.

A bit representation consists of the letter B , O , or H , followed by the binary, octal, or hexadecimal, respectively, constant enclosed in quotes. (Embedded blanks are permitted.) For example, integer value 23 can also be specified by any of the following:

```
B'10111'  B'010 111'
O'27'
H'17'
```

Similarly, -23 can be specified as O'777777777750' or H'FFFFFFE8' .

Trailing zeros may conveniently be specified by ending the constant in quotes by the letter Z followed by the decimal number of zeros to be included. For example,

```
B'11Z3' = B'11000'
O'75Z6' = O'75000000'
```

A bit representation may occur anywhere in a SIMPL-T program that an integer constant may occur. A bit representation may not be used for READ , however. No blanks may be imbedded in a bit representation for an integer constant.

STRINGF may be used to convert an integer value to a string whose characters are the digits of a bit representation by using

STRINGF (<int expr>, <base indicator>)

where <base indicator> has value 2, 8, or 16. (10 is also permissible and is the default value.) Leading zeros are not included in a result from STRINGF . Similarly,

INTF (<string expr>, <base indicator>)

may be used to convert a string of binary, octal, or hex digits to integer.

5.3 Bit Operators

5.3.1 Shift Operators

There are four shift operators in SIMPL-T: left logical shift (.LL.), left circular shift (.LC.), right logical shift (.RL.), and right algebraic shift (.RA.). These are binary operators that are used in the form.

<integer expression> <shift operator> <shift count>

where <shift count> is an integer expression whose value is the number of bits to shift.

These operations are similar to the corresponding 1106/1108 hardware instructions as illustrated below.

```
O'3275Z4' .RA. 6 = O'3275Z2'
O'73Z10' .RL. 5 = O'166Z8'
O'73Z10' .RA. 5 = O'7766Z8'
O'77' .LL. 6 = O'7700'
O'3275Z4' .LC. 15 = O'275000000003'
```

5.3.2 Bit Logical Operators

The bit logical operators complement (.C.), and (.A.), or (.V.), and exclusive or (.X.) also function the same as the corresponding 1106/1108 hardware instructions. Examples are

```
.C. O'1234567' = O'777776543210'
B'110101' .A. B'11001' = B'010001'
B'110101' .V. B'011001' = B'111101'
B'110101' .X. B'011001' = B'101100'
```

5.3.3 Precedence

Bit complement (.C.) has the same precedence as the other unary operators but the binary bit operators have precedence over all other binary integer operators. Among the binary bit operators, the precedence (highest first) is

.LL. .LC. .RL. .RA.	shift
.A.	bit logical
.V. .X.	

5.4 Partwords

The partword operator is similar to the substring operator. In the form

<int expr> [F1,F2]

the partword operation has an integer value whose binary specification consists of the F2 bits beginning at bit F1 of the value of the expression. Thus the partword operator extracts F2 bits beginning at bit F1 and generates an integer value by adding leading zero bits to fill a word.

F1 and F2 may be any integer expressions. The first bit of a word is bit number 0 (not 1 as for the first character of a string). Thus the values of F1 and F2 must satisfy all of the following:

$0 \le F1 \le 35$

$1 \le F2 \le 36$

$1 \le F1 + F2 \le 36$

The F2 field may be omitted, in which case F2 = 36 - F1 (the rest of the word).

As examples, consider the following:

```
17 [31,3] = 4
O'573201577123' [6,6] = O'32'
O'573201577123' [18] = O'577123'
```

Partwords may also be specified on the left of an assignment:

<integer variable> [F1,F2] := <integer expression>

In this case, F2 bits of the value of the variable, beginning with bit F1 , are replaced by the rightmost F2 bits of the value of the expression.

The remaining bits in the variable value remain unchanged.

For example, if integer variable X has value O'6327' , then after

```
X[27,6] := O'7415'
```

X would have the value O'6157' .

5.5 Record I/O

5.5.1 Introduction

Since I/O using READ and WRITE is not very flexible, there is also record-oriented I/O in SIMPL-T. This allows a program to read input images into string variables and write one line of output text from a string expression. Thus while record I/O is more primitive than stream I/O, it gives the user complete flexibility as to the format of input and output (although he must scan the input images and build the output images himself).

5.5.2 READC

The intrinsic procedure READC may be used to read an entire input record (card, line, etc.) into a string variable. The syntax of a READC statement is

{CALL} READC({<skip>,} <readc item>{,<int variable>})

A <skip> for READC is the same as for READ , except that SKIP0 has no meaning for READC . The effect of a skip specification is different for READC , however, since a READC operation includes an implicit SKIP . Thus, for example, successive

```
READC (SKIP, S)
```

statements would read every other card.

A <readc item> may be a string variable, character array, or string array. These function as follows:

1) string variable - The next input record is read into the string variable. The input image is placed into the string just as it appears in the input (e.g., the character in the first card column

becomes the first character in the string, etc.). All of the trailing blanks are removed. The input string is truncated, if needed, to the maximum length of the string variable.

2) character array - The characters of the input image are placed into successive elements of the array, beginning with element 0. If the input image is too long, it is truncated. If it is too short, it is padded with trailing blanks, so that the entire array is filled, unless the <int variable> is included.

3) string array - Successive input records are read into the (string) elements of the array. There must be enough input records to fill the array.

The optional <int variable> is designed for use with a character array. If included in a character array read, the array will not be padded out with blank characters. Instead, the integer variable will be set to the number of characters read.

If used with a string variable, the integer variable is set to the length of the image read. Note that this can also be obtained by using the LENGTH function after the READC . For a string array, the integer variable is set to the length of the last image read.

End of file for record input is determined by the intrinsic function EOIC , which is similar to EOI . The difference is that EOIC asks "Is there another input record?", while EOI asks "Is there another input item?"

READ and READC (and hence EOI and EOIC) are not designed to be intermixed, and a user does so only at his own risk.

5.5.3 WRITEL

The analogue to WRITE for record output is WRITEL . This intrinsic may be used to write out a string onto a line of output, with each string written on a new line. Thus the statement

```
WRITEL(S)
```

is roughly the same as

WRITE(SKIP, S)

The syntax for WRITEL is

{CALL} WRITEL(<writel list>)

where <writel list> is one or more items, separated by commas. The items that may be used in <writel list> are

1) a skip or eject specification - Note that SKIP0 permits overprinting with WRITEL , and successive WRITEL (S, SKIP) statements would print on every other line.

2) a string expression - The string is printed on the next line, truncated to 132 characters if needed.

3) a character array - The array is PACKed and the resulting string is printed on the next output line.

4) a string array - The array elements (strings) are printed on successive output lines.

WRITEL and WRITE are not intended to be intermixed, although the problems are somewhat less severe than for intermixing READ and READC .

5.6 File I/O

5.6.1 Introduction

External data files can be used by a SIMPL-T program. The files must be EXEC 8 SDF files if not generated by a SIMPL-T program. Files generated by a SIMPL-T program will be EXEC 8 SDF ASCII files provided only strings are written out.

Logically, a SIMPL-T file is considered to be a sequence (stream) of scalar data items. A previously created file can be used provided it is assigned to the run before execution of the SIMPL-T program. If no file with the proper name is assigned, then the SIMPL-T file routines will assign a temporary file and free it at the end of the program.

5.6.2 File Declaration

Files must be declared. A file declaration contains the keyword FILE followed by a list of identifiers. The identifiers are the (internal) file

names to be used in the standard manner for EXEC 8 files. (Note that the qualifier is not included and that file name identifiers may not exceed 12 characters in length.)

Files may also be declared as entry points or external references. A file may not be local unless it is an external reference. A file may be used as a parameter to a segment.

Examples of file declarations

```
FILE DATA1, DATA2
ENTRY FILE DATA3
EXT FILE DATA4
```

5.6.3 READF

Files are read using READF . The syntax for this statement is

{CALL} READF (<file name>, <readf list>)

Each item in <readf list> may be

1) an integer, string, or character variable;

2) an integer, string, or character array, in which case successive values are read into successive elements of the array.

The types of the items in the file must be compatible with the types of the items in the <readf list> . No error checking is performed.

End of file may be determined by EOIF , which has value 1 if all items have been read and value 0 otherwise. The syntax for this function is

EOIF (<filename>)

5.6.4 WRITEF

Items are written into a file using WRITEF . The syntax

{CALL} WRITEF (<filename>, <writef list>)

is similar to that for READF . Each item in the <writef list> may be an expression of any data type or an array of any data type. WRITEF functions as a counterpart for READF .

5.6.5 Control Operations

A file is viewed logically as a sequence of items ending with a special end-of-file indicator. Two control operations are used in creating and positioning files.

ENDFILE is used to generate an end-of-file indicator. The statement

{CALL } ENDFILE (<filename>)

must be used after the last WRITEF statement that is used in generating the items in a file.

In order to return to the beginning of a file, REWIND is used. The statment

{CALL } REWIND (<filename>)

repositions the file at its first item

5.6.6 Example

To illustrate the use of files, the following program reads in a set of integers, writes them into a file, and then reads them from the file and prints them.

```
INT NUMBER
FILE DATA

PROC MAIN

WHILE .NOT. EOI
  DO /* READ IN NUMBERS AND CREATE FILE */
    READ (NUMBER)
    WRITEF (DATA,NUMBER)
  END
ENDFILE (DATA)
REWIND (DATA)

WHILE .NOT. EOIF(DATA)
  DO /* READ FROM FILE AND PRINT */
    READF (DATA,NUMBER)
    WRITE (NUMBER)
  END

START MAIN
```

5.6.7 Conventions and Restrictions

As mentioned above, SIMPL-T files are compatible with SDF ASCII files, provided only strings are used. SIMPL-T can read any SDF file, and any program that can read SDF ASCII files can read SIMPL-T files if only string data is read and written by the SIMPL-T programs.

The sequence of file operations is important. The normal sequence for creating and then reading a file is illustrated in the example of 5.6.6 . The restrictions are

1) a READF may only follow another READF or a REWIND , unless it is the initial operation on an already existing file;

2) a WRITEF may be the first operation on a file or may follow a REWIND (or another WRITEF);

3) an ENDFILE may be the first operation or may follow a WRITEF or a REWIND ;

4) a REWIND may only follow an ENDFILE , a READF , or another REWIND .

For these rules, EOIF is equivalent to READF .

The READF statements that read a file are completely independent from the WRITEF statements that create the file. A file is a sequence of scalar data items so that, for example, the elements of an array may be written out using WRITEF of an array and then read back in using READF into scalar variables.

5.7 Multiple Input-stream Files

Input stream files may be partitioned by use of the @EOF control statement (or other equivalent statement). When this statement is encountered, the SIMPL-T input routines will cause a 1 to be returned by EOI or EOIC . However this is considered to be a "soft" end-of-file, and reading may then continue with the next set of data.

There is no way to distinguish between a "soft" and "hard" end-of-file in a SIMPL-T program. (Any attempt to read after a "hard" end-of-file causes the program to be terminated.) However since repeated calls to EOI or

EOIC are allowed, a variable number of partitions can usually be handled by making a second end-of-file test. A "hard" end-of-file would cause successive values of 1 to be returned, while @EOF followed by more data would cause the second end-of-file test to return the value 0 .

This partitioning capability applies only when the SIMPL-T program is executed via an @XQT control statement. Programs invoked by a processor-call control statement (such as the SIMPL-T compiler) cannot use this facility.

5.8 Obtaining the Execution Time Options

The options specified on the @XQT or processor-call statement that causes a SIMPL-T program to be executed can be obtained by the program as it begins execution as described below.

The procedure initially invoked (via the START <identifier> specification) may have one parameter. The type of this parameter must be string. If the procedure does have this parameter, then it will initially be passed a string whose characters are the option letters specified on the control statement that invoked the program execution.

For example if a program has the procedure

```
PROC MAIN (STRING S)
        .
        .
        .
```

and the START specification

```
START MAIN
```

and is executed via the control statement

```
@XQT,ABX
```

then the value of the parameter S when the procedure MAIN is initially called will be 'ABX' .

5.9 Generating Relocatable Output

5.9.1 Introduction

To support its use as a compiler-writing language, there are statements

in SIMPL-T that can be used to generate relocatable output. These very special-purpose statements are described in this section. General information regarding the execution of a SIMPL-T program that uses these statements and the relocatable element to be generated are included in 5.9.8 .

To create a relocatable element the OPENOBJ , DEFEP , DEFXREF , GENOBJ , DEFLC , AND CLOSEOBJ intrinsic procedures are used. OPENOBJ and CLOSEOBJ initiate and terminate, respectively, the relocatable output. DEFEP , DEFXREF , and DEFLC are used to define entry points, external references, and location counters. GENOBJ is used to generate relocatable text.

The reader is assumed to have some knowledge of the terms "entry point", "external reference", "location counter", and "relocatable text" as they apply to EXEC 8 relocatable elements.

5.9.2 OPENOBJ

OPENOBJ must be the first relocatable intrinsic procedure used and may be called only once. The syntax is indicated by

{CALL} OPENOBJ (<locctrs>, <xrefs>)

where <locctrs> is the number of location counters to be used, and <xrefs> is the maximum number of external references to be used.

5.9.3 DEFEP and DEFXREF

These procedures are used to define entry points and external references, respectively, of the relocatable element. An entry point or external reference may be defined at any time between the OPENOBJ and the CLOSEOBJ .

DEFEP is used as indicated by

DEFEP (<name>, <locctr>, <offset>)

where <name> is a string whose value is the entry point name, and <locctr>, <offset> gives the location counter and offset of the entry point.

The syntax for DEFXREF is

DEFXREF (<name>, <number>)

where <name> is the (string) name of the external reference, and <number> is the number by which the external reference will be referenced in the relocatable text.

DEFEP and DEFXREF may be used at any time (between OPENOBJ and CLOSEOBJ) during the generation of the relocatable element. The sequence in which the definitions occur is not important, except that an external reference must be defined before it can be used in a GENOBJ statement.

5.9.4 GENOBJ

The GENOBJ procedure is the primary generator of the object program. The syntax is given by

GENOBJ (<text>, <tlc>, <offset> {,<rlc> {,<xref>}})

This generates the data <text> into the relocatable element at location counter <tlc> , offset <offset> . The argument <text> may be of type integer, character, or string. Integer and character data generate one word of relocatable text. (The ASCII 7-bit code, right-justified with leading zeros, is generated for CHAR data.) A string is generated into zero or more successive words, using the 9-bit ASCII encoding (4 characters per word).

If the <rlc> argument is included (without the <xref>), the address portion (rightmost 16 bits) of the integer argument <text> is relocated with respect to location counter <rlc> . If the <xref> argument is included (i.e., if there are 5 arguments), relocation is with respect to external reference number <xref> .

For example

GENOBJ (NEXTWORD, CTR, ADDR)

would generate the contents of NEXTWORD at the offset given by ADDR relative to the location counter whose value is in CTR . Similarly,

GENOBJ (INSTRUCTION, 1, IC, 2)

would generate the contents of INSTRUCTION at offset IC relative to location counter 1, and the rightmost 16-bits of the word will be relocated with respect to location counter 2.

5.9.5 DEFLC

DEFLC is used to specify the number of words to be reserved for a location counter. The syntax is indicated by

DEFLC (<loc ctr>, <size>)

DEFLC may be used any time between OPENOBJ and CLOSEOBJ . The number of words to be reserved for a location counter need not be specified before text is generated for that location counter.

5.9.6 CLOSEOBJ

The end of the relocatable output is specified by CLOSEOBJ . If arguments are included, they specify the start address of the generated program. If the argument list is omitted, a nonexecutable relocatable element is produced. The syntax for CLOSEOBJ is

CLOSEOBJ{(<loc ctr>, <offset>)}

5.9.7 Example

In this example, location counter 1 is used for instructions and 2 for data. Loops are not indicated but would clearly be used.

```
INT IC = 0,          /* INSTRUCTION COUNTER */
    XREF = 0,        /* NEXT AVAILABLE EXT REF NUMBER */
    INSTRUCTION,     /* INSTRUCTION TO BE GENERATED */
    XREFNO,          /* XREF NUMBER FOR RELOCATION */
    TYPE,            /* RELOCATION TYPE */
    DC = 0,          /* DATA COUNTER */
    MAXXREFS,        /* MAXIMUM NUMBER OF XREFS */
    ENTRYPOINT = 0 /* SWITCH */
STRING NAME [12] /* ENTRY/EXTERNAL NAME */
        :  Determine max number of external refs
        :
OPENOBJ (3, MAXXREFS)
        :  Find external ref and put into 'NAME'
        :
DEFXREF (NAME, XREF)
XREF := XREF + 1
        :  Save XREF number
        :
        :  Set up instruction
        :
```

```
CASE TYPE OF
    \0\  /* NO RELOCATION */
         GENOBJ(INSTRUCTION, 1, IC)
    \1\  /* LC 1 RELOCATION */
         GENOBJ(INSTRUCTION, 1, IC, 1)
    \2\  /* LC 2 RELOCATION */
         GENOBJ(INSTRUCTION, 1, IC, 2)
    \3\  /* XREF RELOCATION */
         GENOBJ(INSTRUCTION, 1, IC, 0, XREFNO)
    END
IF ENTRYPOINT
  THEN
    DEFEP(NAME, 1, IC)
    ENTRYPOINT := 0
  END
IC := IC + 1
              :  Generate data
              :
DEFLC(1, IC)
DEFLC(2, DC)
CLOSEOBJ(STARTLC, STARTADDR)
              :
              :
```

5.9.8 Conventions and Restrictions

Location counters 0-63 may be used. Even-numbered location counter text is placed in the D-bank, and odd-numbered location counter text is placed in the I-bank. (Location counters 1 (for instructions) and 2 (for data) are more or less standard.) If n location counters are specified as being "used" (by OPENOBJ), then location counter numbers 0,1,...,n-1 are allowed. Note that not all of these must actually have text generated for them. No DEFLC is needed for an unused counter.

The external reference numbers begin with zero. If n is specified (via OPENOBJ) as the maximum number of external references, then external reference numbers 0,1,...,n-1 may be used.

External reference and entry point names may not exceed 12 characters in length. Truncation is performed if needed.

A SIMPL-T program that generates a relocatable element must be executed by a processor-call EXEC 8 control statement:

@<processor>{,<options>} {<spec1>}{,<spec2>}

The usual default rules apply. The <spec1> field identifies the source input element and <spec2> identifies the relocatable output element. (Note that the source input records are read by READC in this case.)

The intrinsics PROPEN and PRCLOSE must be used to set up the processor call conventions. These intrinsics are described in section 5.10.

5.10 Programs That Execute as Processors

A SIMPL-T program can be made to execute as an EXEC 8 Processor. Such a program is invoked by a processor call card

@<name>{,<options>} {<spec 1>}{,<spec 2>}

rather than by @XQT . SIMPL-T programs that execute as processors have the standard EXEC 8 source input options performed for them. For example, the input (READ or READC) comes from the <spec 1> element unless the "I" option is specified, in which case it comes from the run stream and is copied into the <spec 1> element.

A SIMPL-T program can be made to execute as a processor by calling the intrinsic PROPEN before any I/O operation is done, and calling PRCLOSE after the completion of all I/O operations. This will establish the required interfaces for source input as well as for relocatable (5.9) or symbolic (5.11) output to the <spec 2> file.

5.11 Symbolic Output

The intrinsics WRDATA and WREND may be used to write symbolic output into the <spec 2> element of a processor call card. The statement

WRDATA (<string>)

writes out a string, and

WREND

closes the output.

A program that uses WRDATA and WREND must be executed as a processor (see 5.10).

6. Using SIMPL-T on the 1106/1108

6.1 Source Input Format

The normal scan of SIMPL-T program text is the first 80 characters of each input record (e.g., card columns 1-80). This text is free format and is essentially viewed as one continuous string of program text.

Keywords are reserved identifiers and may not be used as identifiers in a SIMPL-T program. Keywords are listed in Appendix V.

Input record (e.g., card, teletype line) boundaries are meaningful only in that no keyword, identifier, integer or character constant, or symbol may be split across a record boundry. This restriction does not apply, however, to string constants and comments.

Comments can be nested; that is, a comment can contain other comments. Thus a comment consists of all text between the characters /* and the first occurrence of the characters */ for which all occurrences of /* and */ in the text of the comment are themselves comment delimiters. (Thus, for example, it is always possible to temporarily remove a portion of a program by enclosing it in comment delimiters.)

Several compiler directives are available for program listing control, debugging aids, etc. Some of these may be specified by using EXEC 8 control card options (these are listed in Appendix I) and some may be specified by using a compiler directive in the source program text. A compiler directive is delimited by the characters /+ and +/ , and may occur anywhere that a comment may occur (except within a comment).

The scan width for input text can be changed at any time by using the directive

```
/+ SCANLIMIT {<value>} +/
```

where <value> is the positive decimal integer number of the last character to be included as program text on each input record. The new scan limit becomes effective on the next input record after the one on which the directive occurs. If <value> is omitted, then 80 is assumed. (The initial value is also 80 .) This feature is included primarily to allow the in-

clusion of information other than program text (e.g., sequence numbers) on the input records.

No validity checks are performed on <value> . Thus, for example,

```
/+ SCANLIMIT 1 +/
```

would render the remaining input text records useless.

6.2 Debugging Aids

6.2.1 Traces

6.2.1.1 Program Flow Traces

Two traces for program flow are available: a trace of proc/func calls and a line number trace. The call trace prints a message whenever a call to a procedure or function is executed and also prints a message when a return occurs. The messages include the names of the calling and called segments as well as the line numbers involved. (Only the first 8 characters of a segment name are printed.)

A line trace causes the number of a line to be printed when the statement on it is executed. The segment names (first 8 characters) are also printed as the segments are invoked.

The call and line traces can be activated by compiling with the T and Y options, respectively. They can also be activated and deactivated by using the compiler directives

```
/+ CALLTRACEON +/
/+ CALLTRACEOFF +/
/+ LINETRACEON +/
/+ LINETRACEOFF +/
```

These directives bracket the program statements for which a trace is to be activated. A call or line trace will be in effect for all statements between ON and OFF directives.

6.2.1.2 Variable Trace

An execution trace for the value of variables is also available. This trace is activated by the directive

/+ TRACE <id list> +/

and is turned off by

/+ TRACEOFF <id list> +/

The <id list> is a list of identifiers, separated by blanks or commas; these identifiers must be known by the usual scope rules at the place where the TRACE or TRACEOFF occurs.

The TRACE and TRACEOFF directives bracket the part of the program for which the trace is to be performed. At execution, the name and value of a traced variable is printed after execution of an assignment statement in which the variable was the left side, and after execution of a call that passes the variable as an argument by reference. The line number of the statement is also printed.

An array trace will print values of elements used as scalars whose value may be changed and will include the value of the subscript. Arrays passed as arguments will be signalled by a message, but no values will be printed.

Note that only data identifiers may appear in <id list>. Thus an array element may not be traced. If <id list> is empty in a TRACEOFF directive, all active traces are terminated.

6.2.2 Subscript Checking

Subscript checking can be requested for an entire program by using the C compilation option, and for portions of a program by using the directives

/+ SUBSCRIPTON +/
/+ SUBSCRIPTOFF +/

The directives function similarly to those for the trace options. When subscript checking is activated, the value of any subscript that is outside the bounds of the array is printed, along with line number where the error occurred.

6.2.3 Omitted Case Check

Compiling with the D option causes checking for the occurrence of an unspecified case value in a CASE statement. If the expression value for a CASE statement does not correspond to any of the case numbers (or characters) and no ELSE part was specified, then a message is printed. This check can also be activated for portions of a program by using the directives

```
/+ CASECHECKON +/
/+ CASECHECKOFF +/
```

6.2.4 Conditional Text

In many instances the best way to debug a program is to include extra statements in the program that provide information about the execution of a program as it executes. (An example of such a statement is a WRITE statement that prints the values of certain key variables.) Such statements are often somewhat cumbersome to put into a program at the right places, only to be removed after the bug has been found. The conditional text feature of the SIMPL-T compiler provides a convenient means for handling such a situation.

The conditional text facility allows any string of source text to be either included or ignored as program text by the compiler. Such text is denoted by

```
/+ <indicators> <text> +/
```

where <indicators> is a string of digits and <text> is any SIMPL-T program text that would be valid if the delimiters /+ and +/ , and the <indicators> were removed. For example,

```
/+ 23 WRITE (X, Y, SKIP) +/
```

is an example of the conditional text

```
WRITE(X, Y, SKIP)
```

with indicators 2 and 3 .

The "indicators" 0 - 9 are all initially off. To turn one or more indicators on, the directive

```
/+ SET <indicators> +/
```

is used. To turn indicators off,

```
/+ CLEAR <indicators> +/
```

is used.

Whenever conditional text is encountered by the compiler, the <text> is included in the program if, and only if, any of its <indicators> is on. Note that <text> need not be a complete statement. For example,

```
WRITE ('X=', X /+ 6 ,'Y=', Y +/)
```

could be used to easily compile a program to print either the value of X only, or the values of both X and Y .

Note that a conditional compiler directive would be specified, for example, by

```
/+ 7 /+ LINETRACEON +/ +/
```

6.2.5 User Contingency Interrupt

A contingency interrupt from teletype (@@X C) during execution of a SIMPL-T program will generate a message that gives the line number where the interrupt occurred. The user will then be given the option of resuming or terminating execution. Note that input that has already been entered will be read as a response to the resume query. Also note that the @@X C interrupt cannot be serviced if the program is waiting for input until after the input has been read.

If execution is terminated following a contingency interrupt, all usual end-of-execution functions are performed. Thus, for example, execution statistics are printed if they were specified at compilation.

6.3 Messages Generated by SIMPL-T

Both compile-time and execution-time diagnostic messages attempt to give the line number where the error occurred. The execution time error messages also include the first 8 characters of the segment name of the segment that was executing when the error occurred.

The messages generated by the compiler actually give the line number at which the error was discovered. (Thus it is possible that the spacing of the text of a program can cause a line number to be given in a diagnostic message that is one or more lines after that on which the error occurred.) Similarly, splitting statements across card boundries can sometimes make it difficult for the exact line number to be given in an execution-time diagnostic message.

6.4 Source Listing

A source listing may be requested by using the S option. If the S option is not specified, no listing of the source program will be printed (unless a print directive is used) but diagnostic messages will be given. The N option may be used to suppress the printing of diagnostics.

Program listings include up to 3 numbers to the left of each line of source text. The first (leftmost) number is the line number. The second number (if any) is the statement number for the first statement that begins on that line. The third number (if any) is the nesting level number for the first statement that begins on that line. The statement number and level number are omitted if no statement begins on that line.

Statements are numbered consecutively throughout a compilation, beginning with statement 1. The first statement in a procedure or function has nesting level 1, and the level increases by 1 inside a WHILE, IF or CASE statement.

Several directives are available to control the printing of a source listing. The directives

```
/+ PRINTON +/
/+ PRINTOFF +/
```

may be used to print selected portions of a program. The directive

```
/+ EJECT +/
```

will cause the next line to be printed at the top of the next page. Similarly,

```
/+ SKIP {<count>} +/
```

or

```
/+ SPACE {<count>} +/
```

will cause <count> blank lines to be skipped before printing the next line, where <count> is a positive decimal integer with 1 as the default.

If a listing control directive begins at the first character of a line of program text and if the line contains no text other than the directive, then that line will not be printed. Otherwise, the line will be printed as usual. For example the line

```
/+ SPACE 2 +/
```

will not be printed, but the lines

```
/+ SPACE +/  X := 3
        /+ SPACE 3 +/
```

will be printed.

6.5 Attribute and Cross-reference Listing

An attribute and cross-reference listing may be requested by using the F option or by including the directive

```
/+ ATTRIBUTES +/
```

in the program text. The attribute listing includes the characteristics, (relative) core address or internal number, and line number where defined for each identifier in the program. The cross-reference listing gives the line numbers where each identifier was used. If the value of a variable may be changed, an asterisk follows its line number.

6.6 Keywords and Intrinsic Identifiers

Keywords (see Appendix V) may not be used as identifiers in a SIMPL-T program. However, intrinsic identifiers (Appendix VI) may be redefined by the user. An intrinsic identifier is considered to be global. Thus if a program redefines an intrinsic in a global declaration (including segment names) then that intrinsic cannot be used anywhere in the program. A local redefinition, however, only prohibits the use of the intrinsic in the segment containing the local redefinition.

6.7 Other Options

The B option can be used to turn off the debug aids, such as keeping track of line numbers, that are otherwise performed. This would normally be used only on "debugged" "production" programs.

The directives

```
/+ RECURSIVEON +/
/+ RECURSIVEOFF +/
```

may be used to specify that all segments in a portion of a program are to be recursive, whether or not the keyword REC is included in the declarations.

The R option causes no relocatable output to be generated. This is useful for doing syntax checks and generating listings when no relocatable output is needed since it is faster than a full compile. Note that normal control card rules apply even when the R option is used. This means that a valid relocatable element (<spec 2> or default) must be specified even though it is not generated.

6.8 Program Analysis Facilities

6.8.1 Program Statistics

If the directive

```
/+ STATISTICS +/
```

is included anywhere in the program text, the SIMPL-T compiler will print

statistical information about a program. The statistics include

a) counts of the number of each type of statement (assignment, IF, etc.) used in the program;
b) counts of the number of procedures, functions, and function calls;
c) the average nesting level for statements in the program;
d) the number of tokens generated for the program;
e) the average number of tokens per statement.

(A token is a syntactic entity, such as a keyword, operator, or identifier, that occurs in a program statement.)

6.8.2 Execution Statistics

A statistical summary of program execution will be printed following the execution of a SIMPL-T program if the directive

```
/+ EXECUTIONSTATISTICS +/
```

is included anywhere in the source text. The following are included in the statistical summary:

a) counts of the total number of times each type of statement (assignment, IF, etc.) was executed;
b) counts of the number of times certain compound statement components (THEN parts, WHILE statement lists, etc.) were executed;
c) counts of the number of times executed for the first statement in each statement list;
d) maximum recursion level for each procedure or function that was actually called recursively. (The initial entry to a procedure is at level 0 . The first recursive call is at level 1 .)

Execution statistics for a program execution are printed, if requested, even if the execution is terminated by a program contingency. (At the present time this does not include exceeding the estimated run time, but this contingency will also be included as soon as EXEC 8 facilities permit.)

It should be noted that the use of the execution statistics feature will significantly increase the size of most programs. The use of this facility with an overlay structure is discussed in 6.8.4.

6.8.3 Execution Timing

Execution timings for each procedure and function are provided if the directive

```
/+ TIMING +/
```

is included in the program text. Two timings are given:

a) CPU time excluding non-intrinsic calls. This represents the time actually spent in the code for a procedure or function, plus the time spent in library (intrinsic) calls.

b) CPU time including all calls. This represents the time from entry at recursion level 0 to exit at the same level.

The times given are in seconds, rounded to 3 places (msec.).

Since large fluctuations in timing can occur, depending mostly on system loading factors, several runs on the same data should be done, preferably when the system is not heavily loaded, in order to obtain more reliable results. These timings are intended for use in determining program bottlenecks and for most programs are accurate enough for that purpose. Procedures that execute for very short times (less than 1 msec.) are more likely to incur inaccuracies than are procedures that require more execution time.

The overhead required for execution timing is quite significant if the number of procedure and function calls is high. Since it is not unusual for execution times to be several times higher with timing, this feature should be used only when the timings are worth the extra cost in execution time.

6.8.4 Execution Statistics or Timing with Multiple Modules

Execution statistics may be specified for any of the separately compiled modules of a program that uses more than one separate compilation. Timing may also be specified for any module desired, except that

if used in any of the modules, it must be specified for the module containing the START procedure designation in order to properly initialize the timing routines. Only those modules specified are monitored for statistics or timing.

To use the execution statistics or timing facility with an overlay structure, location counters 4 and 6 of the SIMPL modules must be placed in the root segment (by using appropriate collector (@MAP) control statements).

6.9 Macro Pre-compile Pass

The macro pre-processor described in the University of Maryland Technical Report TR-297 has been incorporated into the compiler as on optional pre-compilation pass. The initial pass creates a source text file which is then fed to the compiler.

Some relevant information for the macro pre-processor are:

1) The macro pass is invoked by using the "M" option. If this option is not specified, no pre-compile pass will be performed.
2) If the macro pass is done, the <spec 1> field on the processor call card denotes the macro source file, and the <spec 2> field denotes the relocatable output. There is no way to create a SIMPL source file or element by using the macro pre-compilation pass.
3) If the macro pass is done, all line numbers (source listing and diagnostics) will refer to the macro source, rather than the generated SIMPL source.
4) The "S" option generates a listing of the SIMPL source only. To list the macro source, the macro directive !OPTION(LIST) must be used.

6.10 Program Execution Time

The execution time (memory time) is printed at program termination. This can be eliminated, if desired, by setting

```
EXT INT S$TIMEMSG
```

to zero.

7. Additional Notes on the 1106/1108 Implementation of SIMPL-T

7.1 SIMPL-T Object Code

A relocatable element produced by a SIMPL-T compilation has instructions in the I-Bank under location counter 1 and static data in the D-Bank under location counter 2 . A small dynamic area is initially included for the allocation of locals for recursive segments and for string workspace. MCORE is used as needed to obtain additional core.

The SIMPL-T compiler generates re-entrant code, and the SIMPL-T library routines are re-entrant.

7.2 Interface with Other Languages

Programs that use FORTRAN calling conventions can be called from a SIMPL-T program and may call a SIMPL-T segment. The following rules apply:

1) To call a FORTRAN subroutine or function, the subroutine or function name must be declared as

```
EXT OTHER PROC
```

2) A SIMPL-T segment to be called from a FORTRAN program must be specified as

```
OTHER ENTRY PROC ...
```

An OTHER segment can also be called from another SIMPL-T segment.

3) A recursive SIMPL-T segment can be called only if the initial execution began with a SIMPL-T program.

4) A nonrecursive SIMPL-T segment may be called from an execution that was not initially in a SIMPL-T program, but only if compiled with the B option.

5) Only integer or integer array arguments can be passed to an OTHER procedure or function.

Arrays passed between SIMPL-T and FORTRAN programs will retain the subscript numbering of the program in which they were declared. Thus if

a SIMPL-T array is passed to a FORTRAN program, then element number 0 of the array would be logically inaccessible in the FORTRAN program. Similarly, if a FORTRAN program passes an array to a SIMPL-T program, then element 0 of the array may not normally be used in the SIMPL-T program.

The standard SIMPL-T linkage conventions are available upon request for those who wish to use these conventions in assembly language subroutines.

7.3 Some Comments on Efficiency

This section contains some random comments regarding the efficiency of certain SIMPL-T features.

1) Recursive procedures and functions incur relatively little additional overhead. It may well be reasonable, in fact, to declare nonrecursive segments that use a large amount of local storage as recursive in order to avoid the static allocation of the local storage. A possible exception here is that local string arrays in a recursive segment require the initialization of the dope vectors for the elements at entry, and this could prove costly if a recursive segment is invoked often.

2) The passing of a string argument by value (the default) means that the string must be copied into the called segment, whereas an argument passed by reference is not copied. This is unlikely to be significant unless segments with value string parameters are very heavily used.

3) If the value of a logical operation can be determined from the first operand only, then the second will not be evaluated. For example, for the operation

   $\langle expr \rangle_1$ .OR. $\langle expr \rangle_2$

   if $\langle expr \rangle_1$ is nonzero, then $\langle expr \rangle_2$ will not be evaluated. Thus the operands in a sequence of logical operations should be in the order that would usually determine the result most quickly, if possible.

4) Hardware partword operations are used when a partword operator specifies a (constant) half-, quarter-, or sixth-word. (All SIMPL-T programs are quarter-word sensitive.)

7.4 Functions with Side Effects

Functions are assumed to have no side effects. Thus some function calls may be eliminated in order to optimize the code generated. For example, function calls involved in an unevaluated operand of a logical operation (see 7.3) and successive function calls whose arguments are unchanged need not be made under the assumption of no side effects.

Those who write functions that have side effects should insure that the elimination of function calls by an optimization process will not adversely affect their program.

7.5 Arithmetic Overflow

Arithmetic overflow that occurs in calculating the results of an integer operation is ignored. This applies to intermediate, as well as final, results. For example, in calculating

```
A + B - C
```

not only the final value, but also the intermediate value of `A + B`, must be in the proper range of integer values or the result will be incorrect.

Appendix I - Executing a SIMPL-T Program on the 1106/1108

The following illustrates a run stream sequence for compiling and executing a SIMPL-T program.

```
@SIMPLT,IS
      .
      .     SIMPL-T program
      .
@XQT
      .
      .     Data for program
      .
```

Normal processor source input options and conventions are used. The primary input options are

I - source input is from the run stream
U - update source element (<specl> field of @SIMPLT card)

and the usual EXEC 8 conventions regarding correction cards apply.

The compiler options are

A - go even if severe errors are found
B - turn all debug aids off
C - check for array subscript out of bounds
D - check for omitted case
F - generate attribute and cross-reference listing
L - print object code
N - suppress printing of diagnostics
R - do not generate a relocatable element
S - print source listing
T - activate call trace initially
X - abort if any diagnostic occurs
Y - activate line number trace initially

Note that the sequence of control cards given above applies only to the situation in which only one execution is to be done in a run. If ad-