USA003087
32.0/17.0

Space Flight Operations Contract

HAL/S PROGRAMMER'S GUIDE

PASS 32.0/BFS 17.0

November 2005

DRD - 1.4.3.8-b

Contract NAS9-20000

United Space Alliance

USA003087
32.0/17.0

HAL/S PROGRAMMER'S GUIDE

Approved by

Original Approval Obtained
Barbara Whitfield, Manager
HAL/S Compiler and Application Tools

Original Approval Obtained
Monica Leone, Director
Application Tools Build and Data Reconfiguration

DRD - 1.4.3.8-b

Contract NAS9-20000

Revision Log

The HAL/S Programmer's Guide has been revised and issued on the following dates[1]:

Issue	Revision	Date	Change Authority	Sections Changed
29.0/14.0		03/10/1999	DR109052	4.3 pp.4-7,4-8
			DR109063	App. B p. B-4
			Cleanup	pp. 4-6, 4-8 p 8-8 pp. 11-5, 11-10, 11-12 pp. 16-1, 16-2, 16-3 pp. 19-14, 19-17, 19-21, 19-22, 19-24, 19-26
30.0/15.0		07/21/2000	CR12711	28.1 p. 28-3
			CR13212	28.2 p. 28-6
			Cleanup	Preface p. 1-6 pp. 2-1, 2-4 thru 2-7 p. 4-9 p. 5-2 p. 6-3 thru 6-7 pp. 7-2, 7-3, 7-5, 7-6, 7-8, 7-9, 7-10, 7-12, 7-14, 7-18, 7-21, 7-22, 7-23 pp. 8-2 thru 8-9 pp. 9-5, 9-6, 9-8,9-10, 9-12 pp. 10-3, 10-7, 10-10, 10-11 pp. 11-3, 11-4, 11-7, 11-10 pp. 12-6, 12-8 thru 12-11, 12-14, 12-16, 12-17, 12-19 pp. 13-2, 13-3, 13-5, 13-6, 13-9, 13-12, 13-13, 13-14, pp. 15-4, 15-5, 15-7 pp. 16-1, 16-3 pp. 17-3 thru 17-9 pp. 18-2, 18-3 pp. 19-5, 19-8, 19-9, 19-12 thru 19-24, 19-28, 19-30,19-32 pp. 20-1, 20-4, 20-6, 20-9, 20-10, 20-12 thru 20-16 pp. 21-1 thru 21-4, 21-6, 21-11, 21-13, 21-14, 21-15

[1] A table containing the Revision History of this document prior to the USA contract can be found in Appendix G.

Issue	Revision	Date	Change Authority	Sections Changed	
			Cleanup cont'd	pp. 22-2, 22-3, 22-4, 22-6, 22-7 pp. 23-3 pp. 24-4, 24-6, 24-10, 24-12, 24-13, 24-14, 24-16, 24-18 pp. 25-3, 25-4, 25-5, 25-7,25-8, 25-10, 25-11 pp. 26-2, 26-4, 26-6 thru 26-10, 26-14, 26-16, 26-17, 26-18 pp. 27-2, 27-4, 27-5, 27-7, 27-8 pp. 28-4, 28-6, 28-11, 28-15, 28-16, 28-17, 28-19, 28-20, 28-21, 28-23, 28-26 thru 28-29 pp. 29-1, 29-2, 29-3, 29-5, 29-6, 29-9 pp. 31-2, 31-3 pp. B-1, B-2 pp. C-1, C-2 INDEX-1 thru INDEX-4	
31.0/16.0		09/07/01	CR13220	24.3 24.8	p. 24-4 p.24-18
			CR13336	4.3	pp. 4-7,4-8
			Cleanup	p.6-3	
32.0/17.0		11/05	CR14215B	21.3 App. A	pp. 21-7, 21-8 p. A-1
			CR14216A		Preface
			PCR0780	28.9	p. 28-31

List of Effective Pages

The current status of all pages in this document is as shown below:

Page No.	Change No.
All	32.0/17.0

Preface

The HAL/S Programming Language was developed by the staff of Intermetrics, Inc. based on many years of experience in producing software in the aerospace field, and is currently maintained by the HAL/S project of United Space Alliance. Although HAL/S was designed to fulfill the flight software requirements of the NASA Space Shuttle program, its features are sufficiently broadly based to meet production software requirements in many other aerospace and real time applications. HAL accomplishes three significant objectives:

- problem orientation, through the use of constructs designed with specific applications in mind;
- enhanced readability, through the use of a natural mathematical format;
- increased reliability, through the incorporation of code and data protection features.

The design of HAL/S exhibits a number of influences, the greatest being the syntax of PL/1 and ALGOL, and the two-dimensional format of MAC/360, a language developed at the Charles Stark Draper Laboratory. With respect to the latter, fundamental contributions to the concept and implementation of MAC were made by Dr. J. Halcombe Laning of the Draper Laboratory.

The HAL/S Programmer's Guide presents an informal description of HAL/S aimed primarily at those unfamiliar with the language. The original version of the Guide was prepared by the staff of Intermetrics, Inc. under direction of Dr. Philip Newbold, that document's principal author.

The primary responsibility is with USA, Department, 01635A7.

Questions concerning the technical content of this document should be directed to Danny Strauss (281-282-2647) MC USH-635L.

Table of Contents

The Programmer's Guide consists of two parts. Sections 1 - 13 make up the first part, while Sections 15 - 31 constitute the second part.

Sections 1-13 of the Programmer's Guide are oriented toward those who have no knowledge of the HAL/S language. They describe the simpler versions of many of the salient features across the entire spectrum of the language. This part assumes a gradually accumulating knowledge of HAL/S from section to section. Therefore, it should initially be read through in its entirety from first section to last.

Sections 15-31 describe other more advanced language constructs important in satisfying general purpose programming needs. The topics presented in different sections are largely unrelated to one another. They may therefore be studied as required in any desired order. A detailed knowledge of the material in the first 13 sections is, however, assumed.

Paragraphs of text enclosed in horizontal bars refer to the existence of more complex HAL/S constructs described elsewhere in the Guide or in the Language Specification Document.

This page intentionally left blank.

List of Figures

This page intentionally left blank.

INTRODUCTION TO HAL/S

HAL/S is a higher order programming language developed by Intermetrics, Inc. for the flight software of the NASA Space Shuttle program. The language is expressly designed to allow programmers, analysts, and engineers to create software which is reliable, efficient, highly readable, and easily maintained.

HAL/S is intended to satisfy virtually all the flight software requirements of the Space Shuttle. To achieve this, the language incorporates a very wide range of features, including applications oriented data types and computations, real time control, and constructs for implementing systems programming algorithms.

- **DATA TYPES AND COMPUTATIONS**
  HAL/S provides facilities for manipulating a number of different data types. Its integer, scalar, vector, and matrix types, together with the appropriate operators and built-in functions provide an extremely powerful tool for the implementation of guidance and control algorithms. Bit and character string processing constructs are available. The formation and use of multi-dimensional arrays, and of tree-like organizations of heterogeneous data are also featured.
- **REAL TIME CONTROL**
  HAL/S is a real time control language. Real time processes can be scheduled and executed in a variety of different modes. Mechanisms for interfacing with external interrupts and other environmental conditions are provided.
- **ERROR RECOVERY**
  HAL/S contains an elaborate run time error recovery facility which allows the programmer freedom (within the constraints of safety) to define his own error processing procedures, or to leave control with the operating system.
- **SYSTEMS PROGRAMMING FEATURES**
  HAL/S contains a number of features especially designed to facilitate its application to systems programming, thus substantially eliminating the necessity of using an assembler language. Most important among these is a facility for creating and manipulating pointers to various kinds of data and code blocks.

Specific features of the HAL/S language have been incorporated to enhance software reliability. By various means, separate blocks of code can be isolated from one another while maintaining ease of access to commonly used data.

- **BLOCK ORIENTATION**
  HAL/S is a block oriented language: nested blocks of code may be established which define local variables that are invisible outside the block.
- **CENTRAL DATA POOLS**
  Separately compiled blocks of code can be executed together, and communicate through one or more centrally managed and highly visible data pools.
- **CONTROLLED ACCESS IN REAL TIME**
  In a real time environment, HAL/S couples the above precautions with locking mechanisms preventing uncontrolled access to sensitive data or areas of code.

HOW TO USE THE PROGRAMMER'S GUIDE

The HAL/S Programmer's Guide is primarily designed to describe the features of HAL/S and their use to programmers previously unfamiliar with the language. Once the contents of the Guide have been mastered, the HAL/S Language Specification document will serve as an additional reference source for the finer details of each language construct. For executing HAL/S programs, a user will require information contained in the HAL/S User's Manual applicable to his particular machine.

The Programmer's Guide is divided into two parts which should be read in order of their appearance,

- Section 1 through 13 describe many of the major features of the language in sufficient detail to enable a new user to begin writing useful HAL/S programs. It should initially be read through in its entirety, from first section to last, and then later referred back to as required.
- Sections 15 through 31 cover additional language forms omitted from the first segment by reason of their complexity or relative unimportance. The first portions of this guide make frequent reference to the existence of the forms described in this segment to facilitate cross referencing. Since Section 15 through 31 are a collection of largely unrelated topics, it is generally not necessary to read the sections sequentially.

It is stressed again that the HAL/S Language Specification Document is the final arbiter concerning the rules governing the form and use of all HAL/S constructs. Appropriate references to the Specification are made in the Guide where omissions have been made in order to retain clarity.

1.0 STRUCTURE OF HAL/S

This section gives an overview on an abstract level of the overall properties of HAL/S compilations, and tries to relate these properties to the need for good programming practice. Later sections of the Guide interpret these properties in terms of actual HAL/S Language constructs.

1.1 STRUCTURING AND HIGHER ORDER LANGUAGES

A common method of problem solving is the so-called "top down" approach. The algorithm for solving the problem is first outlined broadly, and then, step by step, delineated in successively deeper levels of greater detail. The success of the algorithm in arriving at the solution lies as much in its ability to break down the problem into its simplest component parts, as in its ability to resolve the problem as a whole.

If a problem is to be solved by programming it in a higher order language, then the "top down" approach is of special interest because it lends insight into how the program can be organized. Specifically, the organization takes the form of an outer program block enclosing numerous nested "subroutines"[2]. On the outermost level, the program is only concerned with the broad outlines of the solution, and relegates the first level of detail to the outer set of subroutines. These in turn relegate the next level of detail to an inner set of subroutines, and so one until each level of the problem has been relegated to the appropriate set of subroutines.

This particular programming technique is partly what is meant by "structured programming". This term also implies an ability to form nested groups of executable statements inside a program or subroutine. On each level of nesting, a statement group has the ability to behave as if it were a single executable statement.

The overall effect of structured programming techniques is to introduce an orderliness into the writing of programs that not only makes them easier to read but also far less prone to error. Most modern higher order languages possess constructs out of which structured programs can be created: the constructs of the HAL/S language have been defined deliberately with structured programming in mind.

[2] Here the term "subroutine" is loosely used in its generally recognized sense, conveying the idea of a subordinate block of code executed by calling it, and returning to the caller on completion. HAL/S uses different terminology, to be introduced later.

1.2 THE BLOCK STRUCTURE OF HAL/S

The structure of a HAL/S compilation, as indicated below, generally consists of a program block with so-called procedure and function blocks nested within it.

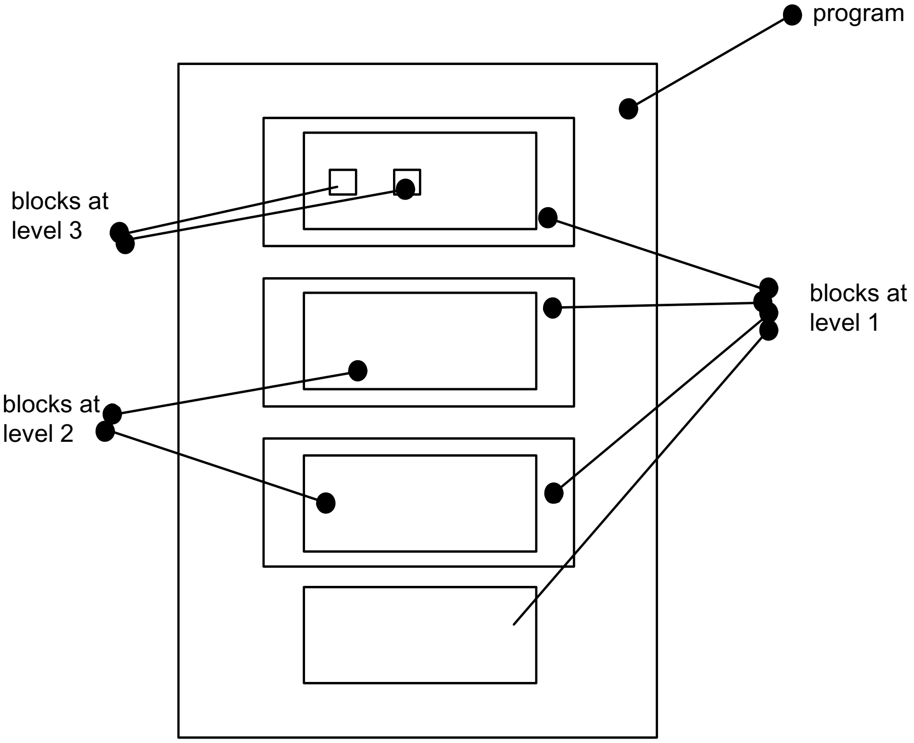


Figure 1-1

Function and procedure blocks constitute the HAL/S interpretation of the "subroutines" of Section 1.1. The more deeply such a block is nested, the greater the depth of detail of the problem solution it is supposed to handle. The blocks at each level contain executable code implementing the appropriate part of the problem solution.

Both kinds of blocks are similar in that they contain code which is executed by a call or "invocation", and which returns execution to the caller on completion. However, procedure and function blocks differ in the way they are invoked. A procedure is invoked by a CALL statement, while a function (like its mathematical counterpart) is invoked by its appearance in an expression, and returns a result[3].

Generally, the code in any block may invoke a procedure or function block defined at the same level, or in a surrounding outer level. The rules defining the region where a block may be invoked are discussed later in this Section.

[3] A procedure is therefore like a FORTRAN SUBROUTINE, and a function is like a FORTRAN FUNCTION. Note, however, that FORTRAN SUBROUTINES and FUNCTIONS are always exterior to the program calling them, while this is not true for HAL/S.

The forms of procedure and function blocks and the constructs for invoking them are described in Section 11 of the Guide. The form of the outer program block is described in Section 3.

SCOPING OF DATA

In HAL/S, all data must be defined in so-called “data declarations”. An important consequence of the structural properties of HAL/S is its ability to place data declarations so as to bound the regions in a program which may reference the declared data. This feature is called “scoping”.

Data declared at the program level may generally be used throughout the entire compilation:

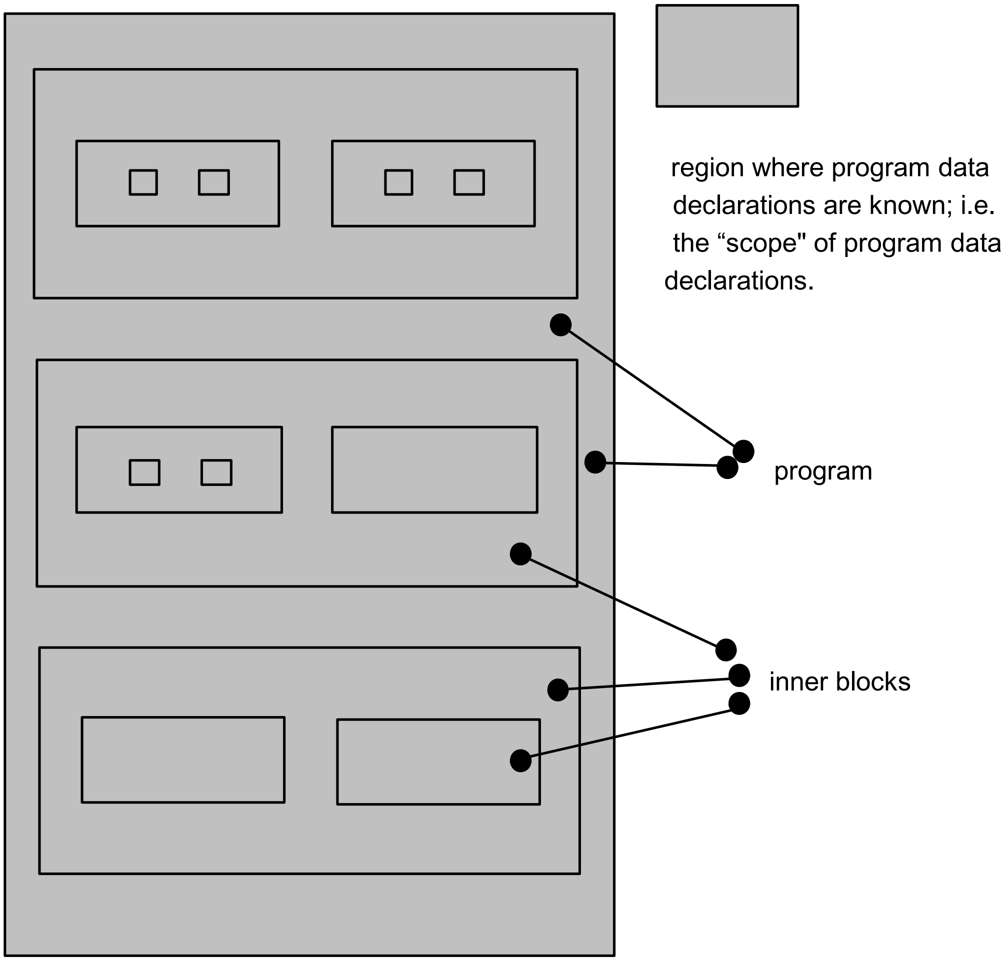

region where program data declarations are known; i.e. the “scope" of program data declarations.
program
inner blocks

Figure 1-2

In addition, any procedure or function block nested within a program block may declare local data - data known only in that particular block and in blocks nested within it - as indicated below:

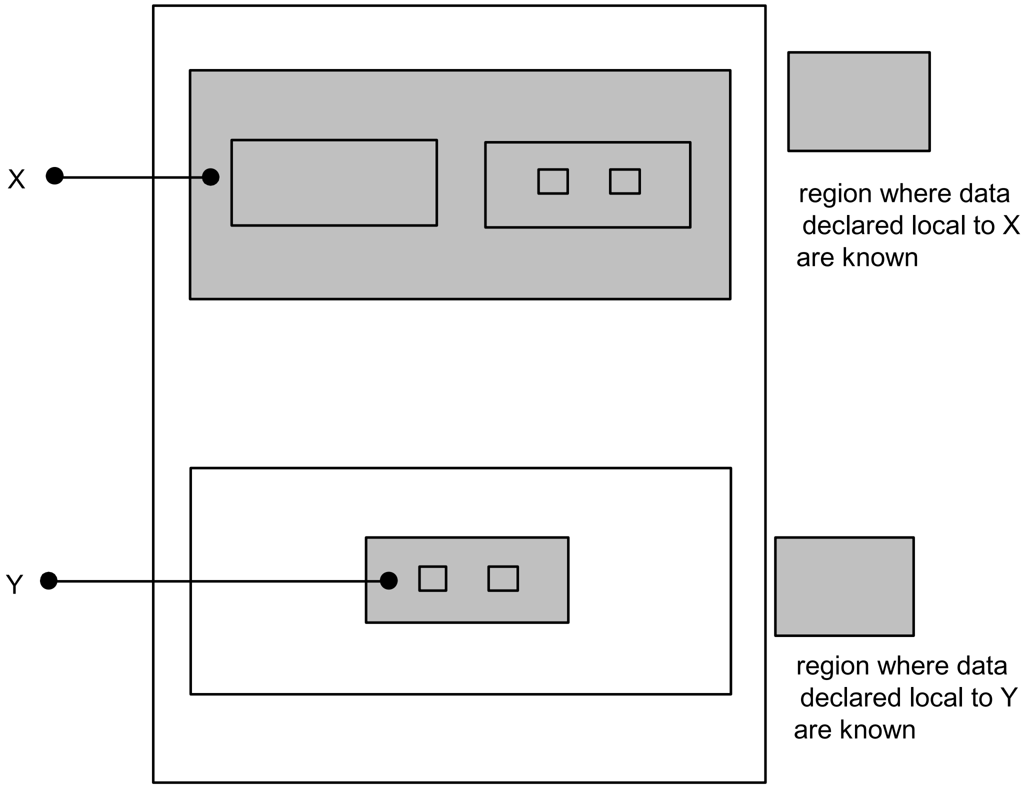

X
region where data declared local to X are known
Y
region where data declared local to Y are known

Figure 1-3

SCOPING OF BLOCK NAMES

The program block, and every procedure or function within it are named: block names have scoping rules identical with the data scoping rules already described. The name of any procedure or function block is deemed to have been "declared" in the surrounding block in which the procedure or function is nested. This bounds the region where its name is known, and therefore determines where it may be invoked. Thus, the name of any procedure or function nested at the program level is known anywhere in the program.

However, since in HAL/S recursion is not allowed, such a procedure or function may be invoked from anywhere in the program except inside itself, as indicated:

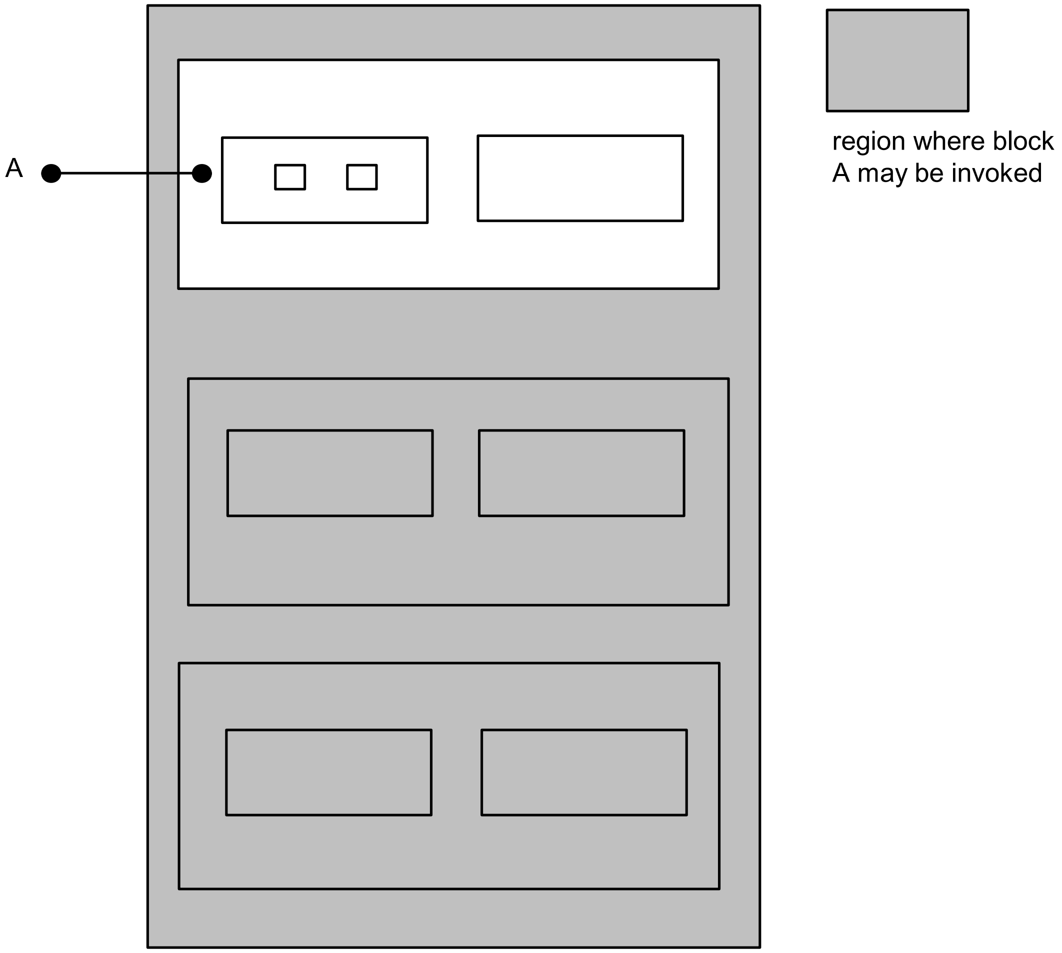


Figure 1-4

Similarly, inner procedures and functions may be invoked from anywhere in the block enclosing them except within themselves.

In the following example, inner block B and C can only be invoked from inside regions X and Y respectively:

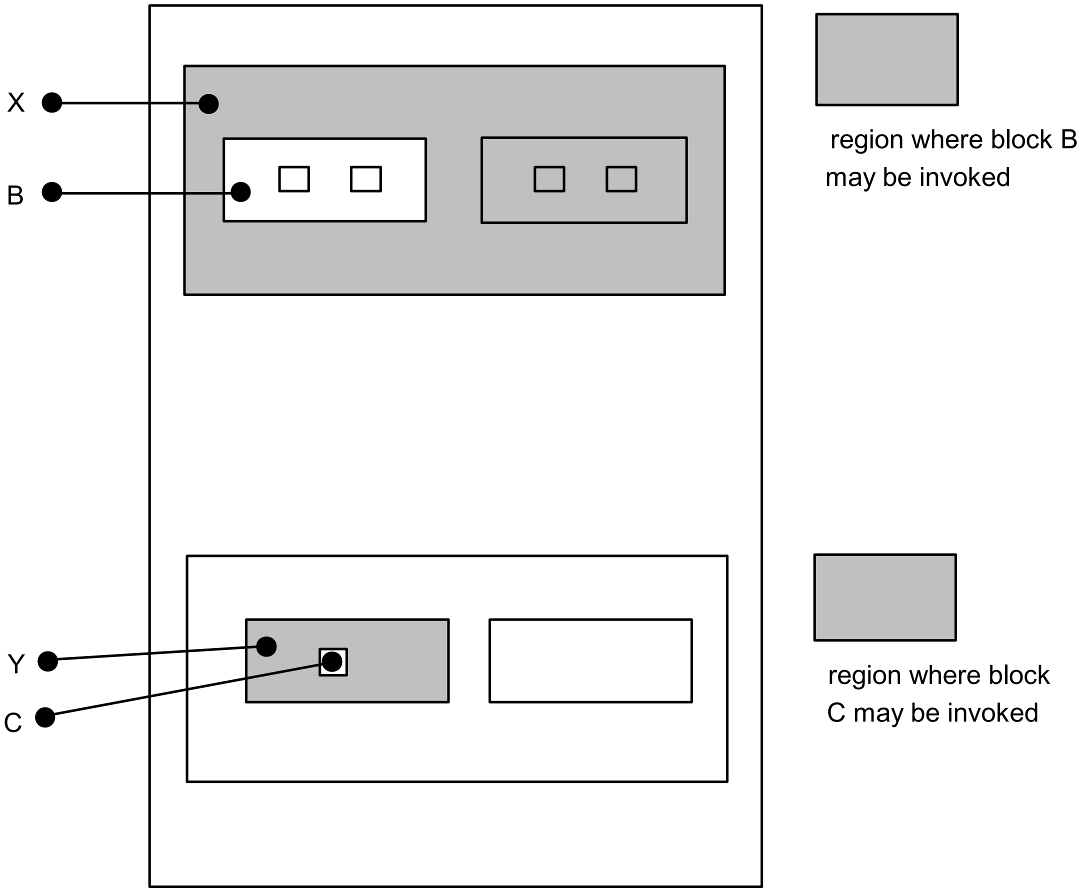

X
B
Y
C
region where block B may be invoked
region where block C may be invoked

Figure 1-5

It should be noted that all forms of recursion in HAL/S are illegal. The form of recursion not prevented by the rules given above is that in which procedures P and Q are not contained in each other, but P calls Q and Q calls P.

It is also possible for a program (or any block within it) to invoke entities outside the compilation unit; i.e., other compilation units. Procedures and functions may be compiled independently for this purpose.

See: Guide/15.

1.3 STATEMENT GROUPING IN HAL/S

In HAL/S, the actual step by step solution of a problem is performed by executable statements contained in the blocks comprising the program. Sequences of executable statements may be grouped together and treated as a single compound statement. Such statement groups are said to be "well-bracketed" - they begin with a special statement (a "DO" statement), and end with another special statement (an "END" statement). Execution of the sequence of statements in the group can be controlled in various ways depending on the form of the opening "DO" statement:

- The sequence may be executed once only;
- the sequence may be executed repetitively until specified conditions are met;
- the sequence may be executed repetitively while specified conditions are met;
- one statement in the sequence may be selected as the only one to be executed.

Sequences of compound statements may also be grouped together in the same way and, in turn, be treated as a more complex compound statement, and so on to an arbitrary degree of nesting.

Use of this grouping property in conjunction with other HAL/S constructs can substantially eliminate the need for a "GO TO" statement (in the FORTRAN sense, for example), which from the structured programming viewpoint is recognized to be "dangerous" because it destroys the readability of a program, and makes it more error-prone.

STATEMENT GROUPS AND GO TO STATEMENTS

The design of HAL/S minimizes the dangers of "GO TO" statements by limiting the regions which can be branched to by them, in a way analogous to the limits imposed on data by the scoping rules described in Section THE BLOCK STRUCTURE OF HAL/S. Consider a program containing nested groups of executable statements as show below:

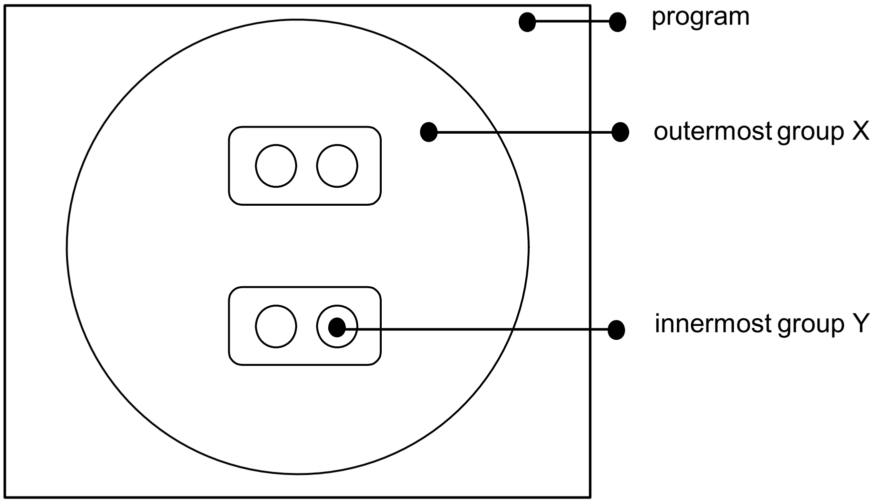


Figure 1-6

The region of legal destinations of "GO TO" statements contained in group X are as indicated below:

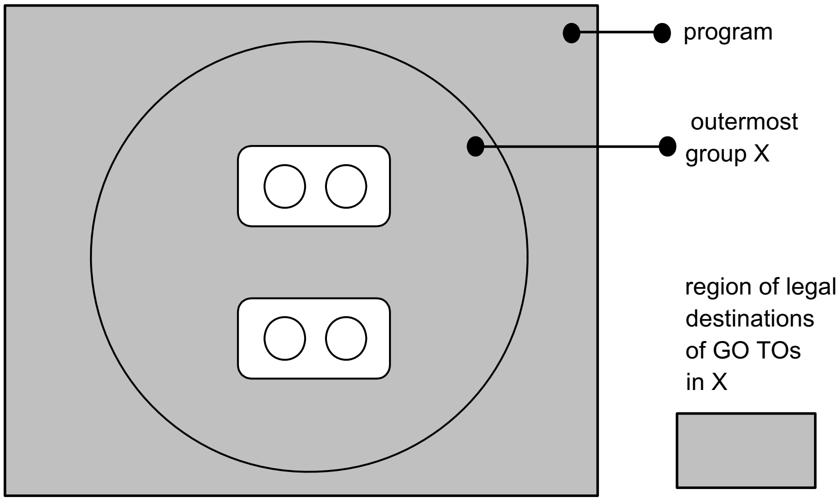


Figure 1-7

The region of legal destinations of "GO TO" statements contained in group Y are as indicated below:

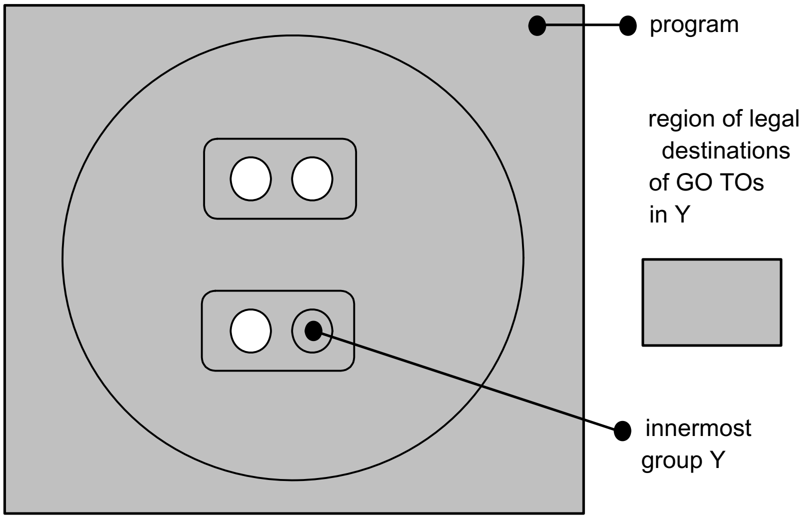


Figure 1-8

It is evident from the examples that while groups can be branched out of, or branched within, they may not be branched into.

INTERACTION WITH BLOCK STRUCTURE

Since procedure and function blocks may appear anywhere in a program, including inside statement groups, the problem arises of branches by means of "GO TO" statements in and out of such blocks.

In HAL/S, the destinations of "GO TO" statements are labels attached to executable statements. Because the scope rules for statement labels are the same as for declared data, it follows that it is impossible to branch into a procedure or function block. Additionally, a rule is made that branches may not be made out of a block (even though by scope rules the label of the destination is visible).

This leaves the reciprocal processes of call and return-to-caller as the only ways of entering and leaving procedures and functions, which is in accordance with structured programming principles.

This page intentionally left blank.

2.0 HAL/S SYMBOLOGY

HAL/S source text has its own particular characteristics; a specific character set, special combinations of characters set aside as reserved words, and certain rules dictating the form of statements. This section is an introduction to these characteristics of the HAL/S Language.

2.1 THE CHARACTER SET

The HAL/S language uses the following character set:

ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz } alphabetic characters

0123456789 } numeric characters

} alphanummeric characters

+-*./|¬&=<>#@$,;:‘ “)(_%¢ } "special" characters

(blank)

This character set is a subset of the standard character sets ASCII and EBCDIC.

Although the user really needs only the above character set when writing a HAL/S program, there are additional special characters which can be used in comments and in character string literals (described later in this section).

```
[ ]   { }   !   ?
```

The output listings produced by a HAL/S compiler may use these extra special characters for annotation.

2.2 RESERVED WORDS, IDENTIFIERS, AND LITERALS

The HAL/S language uses four kinds of primitive elements as basic constructs:

- RESERVED WORDS are a fixed part of the language and consist of combinations of upper case alphabetic characters;
- IDENTIFIERS are user-defined names used for data or labels, and consist of combinations of the alphanumeric characters;
- LITERALS express actual values, and can consist of any of the symbols in the character set;
- SPECIAL CHARACTERS serve as delimiters, separators or operators, and consist of the non-alphanumeric characters of the HAL/S set.

RESERVED WORDS

Reserved words are words having a standard meaning in the HAL/S language. As their name suggests, the user cannot use reserved words as identifier names. There are two major categories of reserved words:

- KEYWORDS are used to express parts of HAL/S statements, for example: GO TO, DECLARE, CALL, and so on. A complete list can be found in Appendix E.
- BUILT-IN FUNCTION NAMES are used to identify a library of common mathematical and other routines, for example: SINE, SQRT, TRANSPOSE, and so on. A complete list can be found in Appendix B.

IDENTIFIERS

An identifier name is a user-assigned name identifying an item of data, a statement or block label, or other entity. The following rules must be observed in the creation of any identifier name[4].

1. The total number of characters in the name must not exceed 32;
2. The first character must be alphabetic;
3. The remaining characters may be either alphabetic or numeric;
4. Any character except the first or last may be an underscore (_)

Examples:

```
ELEPHANT_AND_CASTLE
  A1
  P                          legal

  1B
  X_X_                       illegal
```

[4] Some implementations of HAL/S may place extra restrictions upon the names of identifiers. See appropriate User's Manual.

LITERALS

The three basic kinds of literals described here are arithmetic, character string, and Boolean. The utility of arithmetic literals is obvious. In simple programming problems, character string literals find most use in the generation of output. Boolean literals are used to state logical truth or falsehood.

- ARITHMETIC LITERALS express numerical values in decimal notation. The generic form of an arithmetic literal is:

$$\overbrace{\pm ddd.ddd}^{\text{mantissa}} \quad \overbrace{E\pm ddd}^{\text{exponent}}$$

1. ddd represents an arbitrary number of decimal digits.
2. The exponent is optional.
3. The + signs are optional.
4. The decimal point is optional. If absent, it is considered to be to the right of the least significant digit of the mantissa. If the decimal point is present, it may appear anywhere in the mantissa.
5. The minimum number of digits in the mantissa, and in the exponent, if present, is one. The maximum number is implementation dependent†.

† See appropriate User's Manual.

Examples:

```
0.123E16
45.9
- 4
```

It is important to note that HAL/S makes no distinction of type between a integral-valued literal and a fractional valued literal. Either integer (with possible rounding of value) or scalar (i.e., floating-point) type is assumed according to the context in which the literal is used.

The use of multiple exponents, and of binary, hexadecimal or octal exponents, is also allowed.

See: Spec./2.3.3.

- CHARACTER STRING LITERALS consist of strings of characters chosen from the entire HAL/S character set. The generic form is:

‘ccccccc’

1. The quote marks delimit the beginning and end of the literal.
2. cccc represents an arbitrary number of characters in any combination.
3. Quote marks within the literal must be represented by a pair of quote marks to avoid confusion with the delimiting quotes.
4. The minimum number of characters is zero (a ‘null’ string), the maximum is 255[†].

[†] See appropriate User's Manual.

Examples:

```
‘ ’
‘ONE two THREE’
‘DOG’ ‘S’
```

If a literal consists of a single character, or character sequence repeated many times, a condensed form of literal using a repetition factor may be used.

See: Spec./2.3.3.

- BOOLEAN LITERALS express logical truth or falsehood, and are generally used to set up the values of Boolean data items. Their forms are:

$$\left.\begin{array}{l}\texttt{TRUE}\\\texttt{ON}\end{array}\right\}\text{ expressing truth, or binary “1”}$$

$$\left.\begin{array}{l}\texttt{FALSE}\\\texttt{OFF}\end{array}\right\}\text{ expressing falsehood or binary “0”}$$

Literal strings of binary values also exist.
See: Guide/17.1

2.3 FORMAT OF SOURCE TEXT

HAL/S is a "stream-oriented" language, that is, statements may begin anywhere on a line (or card), and may overflow without special indication onto succeeding lines or cards. Several statements may be written on one line (or card) as required.

HAL/S is among the very few languages which permit subscripts and exponents to be represented as they are mathematically, using lines below and above the main line respectively as needed. This multi-line format is an optional alternative to the HAL/S single-line format.

Even when multi-line format is not used, the first character position of each line (or card) is reserved for a symbol denoting the kind of line format, subscript, main, or exponent.

SINGLE-LINE FORMAT

In single-line format, the first character position of each line is left blank, denoting a main line. An M can alternatively be used but is generally not preferred by users.

- EXPONENTS are denoted by the operator * *

  Example:

  x^{t+2} is coded as:

  ```
  X * *(T+2)
  ```

- SUBSCRIPTS are denoted by parenthesizing the subscript and preceding it with the symbol $.

  Example:

  a_{i+1} is coded as:

  ```
  A$(I+1)
  ```

MULTI-LINE FORMAT

In multi-line format, the first character of a main line is either left blank or M is inserted as before. The first character of an exponent line is E, and that of a subscript line is S.

- EXPONENTS are written on an exponent line (E-line) immediately above the main line.

  Example:

  x^{t+2} is coded as:

  ```
  |
  |  E    T+2
  |  M   X
  |
  ```

- SUBSCRIPTS are written on a subscript line (S-line) immediately below the main line.

  Example:

  a_{i+1} is coded as:

```
|
|  M A
|  S  I + 1
|
```

When using multi-line format the type indicator (- for vector, * for matrix, etc.) must overlap the identifier on the M line. Except for this, care must be taken to ensure that nothing on the E- and S-lines overlaps anything on the M-line.

Exponents of exponents and subscripts of subscripts use extra subscript and exponent lines. Special rules apply if exponents are subscripted, or if subscripts possess exponents.

See: Spec./2.4.

2.4 STATEMENT DELIMITING

As Section 2.3 indicated, HAL/S statements may be written in free form without regard for line (or card) boundaries. Because of this there is the need to explicitly indicate the end of each statement with a special symbol. HAL/S uses a semicolon for this purpose. The following statements arbitrarily selected from the language show the placement of the semicolon.

Examples:

```
|
| DECLARE I INTEGER;
| I = I + 1;
| CALL P (I,J);
|
```

2.5 COMMENTS IN HAL/S

The use of comments is a sine qua non of good programming practice. HAL/S possesses two mechanisms for the inclusion of comments in a compilation.

- IMBEDDED COMMENTS may be placed anywhere on main, exponent or subscript lines of HAL/S text.
- COMMENT LINES may appear between main, exponent and subscript lines of HAL/S text.

IMBEDDED COMMENTS

An imbedded comment takes the form:

/*...*any text*   (except */)...*/

Such comments may appear between HAL/S statements or imbedded in a statement. They may not appear in the middle of a literal, reserved word, or identifier. Nor may they overlap any source text or other comments on other lines of a group written in multi-line format. As far as the sense of the source text is concerned, an imbedded comment is treated as if it were a string of blank characters.

Examples:

```
|
|M   X = X + 1;   /* ADD ONE TO X */
|
```

```
|
|M  X = Y;
|S   1            /* BAD * /
|
|
```

illegal-comment overlap rule

COMMENT LINES

Comment lines are input lines specially reserved solely for comments by placing the character C in the first character position of the line. The rest of the line may contain any desired text.

Examples:

```
|
|M  X = X + l;
|C  ADD ONE TO X
|C  THEN CARRY ON
|
|
```

When the SRN option is specified, columns 73-80 are interpreted as a statement number. Vertical spacing and page skipping are controlled by the EJECT option.

This page intentionally left blank.

3.0 A HAL/S COMPILATION - THE PROGRAM BLOCK

The structuring of HAL/S programs was dealt with on the conceptual level in Section 1. Section 3 begins to interpret this information in terms of actual HAL/S language constructs.

For the purposes of Part I, an entire HAL/S unit of compilation is known as the "program block". The term "block" has a special connotation in this Guide. It is taken to mean a coherent body of data declarations and executable statements <u>enclosed in statements delimiting its opening and closing, and identified with a name.</u>

3.1 OPENING AND CLOSING THE PROGRAM BLOCK

The first statement of a HAL/S program is a statement defining the name of the program and opening the program block. The last statement of a HAL/S program is a statement closing the program block. Between the two are all the statements comprising the body of the program.

PROGRAM OPENING

The statement opening a program block takes the form:

```
|
| label: PROGRAM;
|
```

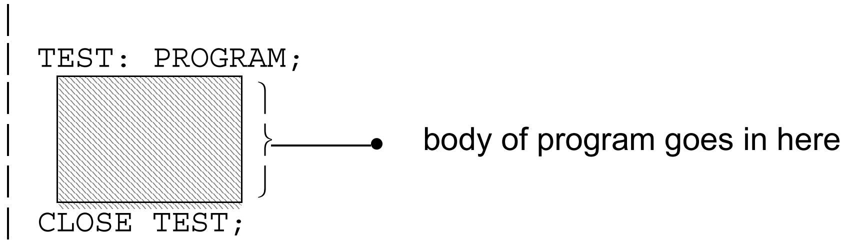
1. *label* is any legal identifier name, and constitutes the name of the program.

The program block is closed with the statement:

```
|
|   CLOSE  label;
|
```

1. The identifier *label* is optional.
2. If *label* is supplied, it must be the program name, i.e. the *label* on the opening statement of the program block.

Example:

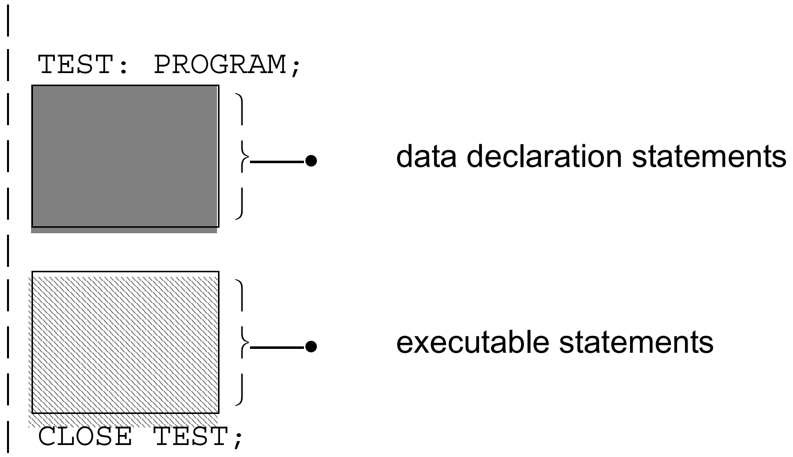


```
|
| TEST: PROGRAM;
|
|
|
| CLOSE TEST;
```

3.2 POSITION OF DATA DECLARATIONS

Normal HAL/S programs require the use of data. The names used to identify this data must be declared before use by the means of data declaration statements. Data declarations (and, additionally, certain other kinds of statements) must be placed after the program opening statement and before the first executable statement.

Example:



3.3 FLOW OF EXECUTION IN THE PROGRAM

The program begins execution at the first executable statement after the data declarations, and thereafter follows a path determined by the kinds of executable statements encountered. Unless statement groups, branches, or conditional statements intervene, execution is sequential. Finally, the path either reaches a statement terminating execution of the program, or reaches the closing statement of the program block, which has the same effect.

As described in Section 1, procedure and function definition blocks may be interspersed between the statements in a program block. The only way of executing such blocks is by explicit invocation: if they are encountered in the path of execution they are passed over as if nonexistent.

Example:

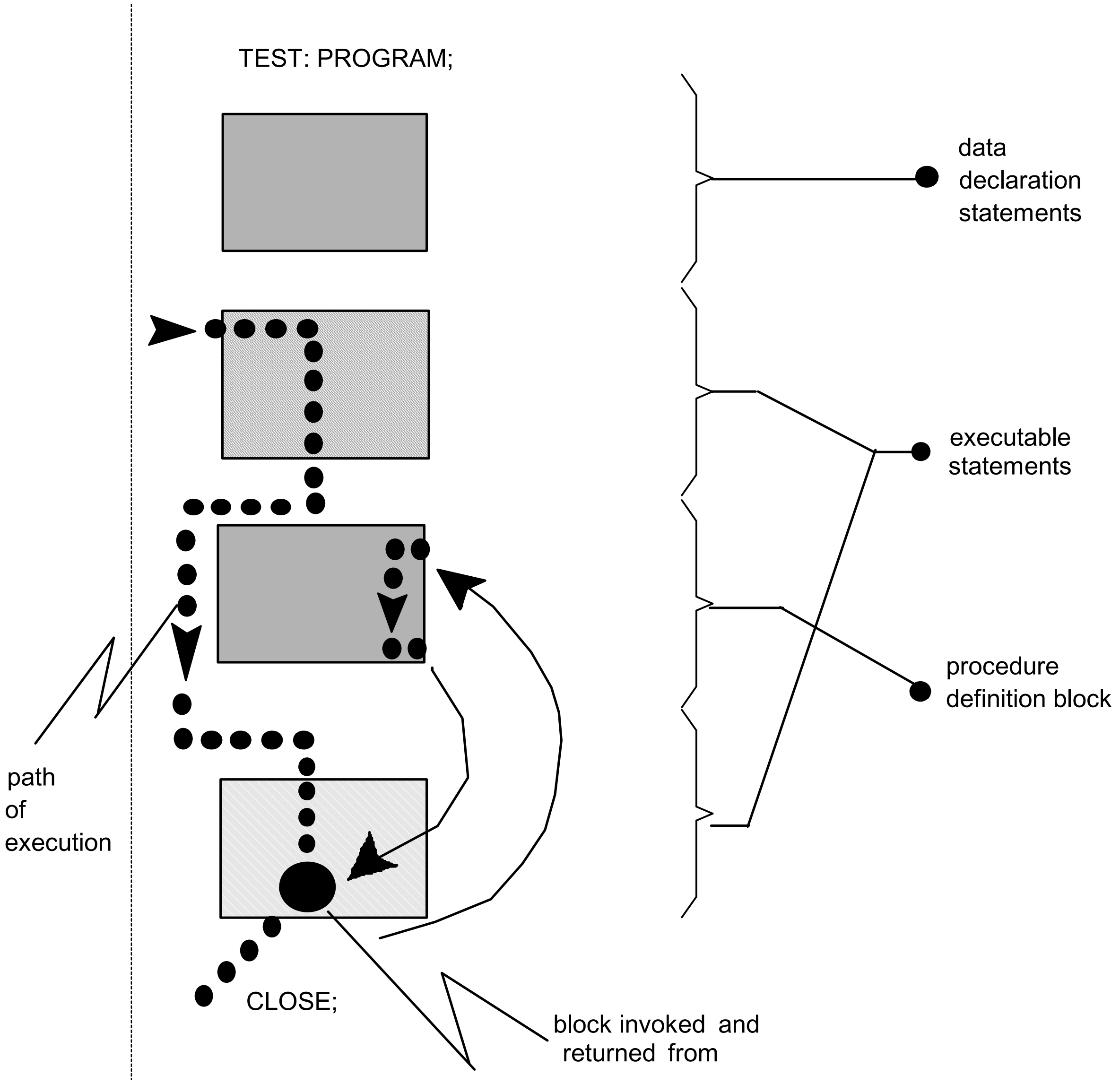


Figure 3-1

This page intentionally left blank.

4.0 DATA DECLARATION

Programming largely consists of the manipulation of numerical data. The diversity of the data types in a language determines its utility for any required task. HAL/S contains an exceptionally diverse set of data types.

Identifiers of the kind described in Section 2 are used to name items of data. Identifier names used to represent data items must[1] be defined in data declarations appearing in the appropriate program, procedure or function block. The effect of placing data in different blocks is described in Section 1. The position of data declarations within a program block is described in Section 2.

This Section now proceeds to describe the detailed construction of data declarations.

4.1 HAL/S DATA TYPES

In the HAL/S language, arithmetic data of the following types can be declared:

- INTEGER for the representation of integer-valued quantities;
- SCALAR for the representation of "floating-point" quantities;
- VECTOR for the representation of algebraic row or column vectors (without distinction), and each element of which is a scalar quantity;
- MATRIX for the representation of algebraic matrices, and each element of which is a scalar quantity.

These arithmetic data types may be specified in either single or double precision. In the case of integer, the precision determines the maximum absolute value the identifier might take on. In all other cases, it determines the number of significant digits in the mantissa of the value.

In addition, HAL/S also possesses the following data types:

- CHARACTER for the representation of strings of text;
- BOOLEAN for the representation of binary-valued (logical) quantities.

It is possible to declare arrays (or tables) of any of the six above types.

HAL/S possesses other data types. The Boolean data type is a degenerate form of the HAL/S "bit string" data type.

See: Guide/17.

HAL/S also possesses hierarchical organizations of data items of any type, known as "structures".

See: Guide/19.

[1] The HAL/S language prohibits the use of implicitly declared data items considering it to be an undesirable programming practice.

4.2 SIMPLE DECLARATION STATEMENTS

Data declaration statements define identifiers used to name data. The simplest forms of declaration statement for each data type listed above are examined on the following pages.

INTEGER

```
|  DECLARE name INTEGER;
|  DECLARE name INTEGER SINGLE;
|  DECLARE name INTEGER DOUBLE;
|
|
```

1. In each of the forms *name* is any legal HAL/S identifier.
2. Presence of the keyword SINGLE specifies single precision.
3. Presence of the keyword DOUBLE specifies double precision.
4. Absence of either keyword implies default of single precision.

For the integer data type, single precision usually implies halfword and double precision fullword, depending on the implementation[2].

Examples:

```
|
| DECLARE I1 INTEGER;
| DECLARE BIG_I INTEGER DOUBLE;
|
```

SCALAR

```
|
|  DECLARE name SCALAR;
|  DECLARE name SCALAR SINGLE;
|  DECLARE name SCALAR DOUBLE;
|
```

1. In each of the forms *name* is any legal HAL/S identifier.
2. Presence of the keyword SINGLE specifies single precision.
3. Presence of the keyword DOUBLE specifies double precision.
4. Absence of either keyword implies default of single precision.
5. The keyword SCALAR may be omitted.

Double precision usually implies increased range of exponent and increased number of digits in the mantissa, but it is implementation dependent[2].

[2] See appropriate User's Manual.

Examples:

```
DECLARE S1;
DECLARE S2 SCALAR;
DECLARE S3 SCALAR DOUBLE;
```

MATRIX

```
DECLARE name MATRIX(m,n);
DECLARE name MATRIX(m,n)SINGLE;
DECLARE name MATRIX(m,n)DOUBLE;
```

1. In each of the forms *name* is any legal identifier.
2. Keywords SINGLE and DOUBLE have the same significance as for scalar and vector types.
3. m and n denote respectively the number of rows and columns in the matrix. They must lie in the range $1 < m, n \leq 64$† .
4. If the size specification (m, n) is absent, a 3x3 matrix is assumed.

† This value may vary between implementations. See appropriate User's Manual

Examples:

```
DECLARE M1 MATRIX(2,4);
DECLARE M2 MATRIX(4,5) DOUBLE;
DECLARE M3 MATRIX;
                 ↑
        a 3 X 3 matrix
```

VECTOR

```
DECLARE name VECTOR(n);
DECLARE name VECTOR(n) SINGLE;
DECLARE name VECTOR(n) DOUBLE;
```

1. In each form *name* is any legal identifier.
2. Keywords SINGLE and DOUBLE have the same significance as for scalar type.
3. n specifies the length of the vector and must lie in the range $1 < n \leq 64$†.
4. If the length specification (n) is omitted a length of 3 is assumed.

† This value may vary between implementations. See appropriate User's Manual.

Examples:

```
|
| DECLARE V1 VECTOR(10);
| DECLARE V2 VECTOR(10) DOUBLE;
| DECLARE V3 VECTOR;
|              ↑
          a  3-vector
```

CHARACTER

```
|
| DECLARE name CHARACTER(n);
|
```

1. *name* is any legal identifier.
2. n specifies the maximum length of the text string that the data type may carry (i.e., the maximum number of characters). It must lie in the range of $1 \leq n \leq 255$[†].
3. The actual length of the string of text carried may vary during execution between zero (a “null” string) and the maximum n.

[†] This value may vary between implementations. See appropriate User’s Manual.

Example:

```
|
| DECLARE C1 CHARACTER(80);
|
```

BOOLEAN

```
|
| DECLARE name BOOLEAN;
|
```

1. *name* is any legal identifier.

Example:

```
|
| DECLARE B1 BOOLEAN;
|
```

ARRAYS

The properties of a data item, (its type, precision, and size), as expressed in its declaration are called the “attributes’’ of the data item. In any of the above declarations, the attributes are specified following the name of the data item.

To declare an array of any data type an ARRAY specification is inserted between the name of the data item and its attributes:

```
DECLARE name array(n) attributes;
```

1. *attributes* stands for any legal form of attributes for any data type described.
   It is possible that none appear, in which case SCALAR, SINGLE is implied.
2. n denotes the number of elements in the array (i.e., entries in the table) and must lie in the range 1 < n < 32768[†].

[†] This value may vary between implementations. See appropriate User’s Manual.

Examples:

```
DECLARE AS1 ARRAY(500) SCALAR;
DECLARE AM1 ARRAY(20) MATRIX(4, 4);
```

HAL/S also supports multidimensional arrays of any data type.

See: Guide/18.1.

COMPOUND AND FACTORED DECLARATIONS

If a program contains declarations of many data items it is tedious to repeat the keyword DECLARE in every declaration. Many separate declarations may be condensed into one compound declaration as shown below.

Example:

```
      DECLARE S;                             
      DECLARE I INTEGER DOUBLE;              
      DECLARE M3 MATRIX;                     
      DECLARE M6 MATRIX(6,6);                }  separate declarations
      DECLARE B BOOLEAN;                     
      DECLARE C ARRAY(5) CHARACTER(20);      
      DECLARE V ARRAY(3) VECTOR;             

      DECLARE S                              
         I INTEGER DOUBLE,                   
         M3 MATRIX,                          }  equivalent compound
         M6 MATRIX(6,6),                        declaration
         C ARRAY(5) CHARACTER(20),           
         V ARRAY(3) VECTOR;                  
```

Note the commas separating the declaration of each data item.

If the identifiers in a compound declaration have some attributes in common, a third, even more compact form called a factored declaration is possible. This form is as shown below.

Example:

```
DECLARE V1 VECTOR(3),        
        V2 VECTOR(3)DOUBLE,  }  compound declarations
        V3 VECTOR(3)DOUBLE;  
```

can be rewritten in the factored form:

```
DECLARE VECTOR(3),V1,              equivalent
                  V2 DOUBLE,    }  factored
                  V3 DOUBLE;       declaration
```

Note the comma separating the factored attributes and the first declared data item.

4.3 INITIALIZATION OF DATA

A HAL/S data item of any type may be initialized by incorporating the appropriate specification into its declaration. The form of such a specification differs depending on whether the data item is "uni-valued" or "multi-valued''.

- UNI-VALUED data items are those having only one element: unarrayed scalars, Booleans, and characters.
- MULTI-VALUED data items are those having more than one element: unarrayed vectors and matrices, and arrayed data items of any type.

In either case, the specification is placed after the type, precision, and size attributes of a declaration. This positioning will become apparent in the examples to follow.

UNI-VALUED DATA ITEMS

The two variations of the construct for initializing uni-valued data items are:

```
INITIAL(value)
CONSTANT(value)
```

1. The two forms have the same effect in that the data item is initialized to the literal indicated by *value*.
2. The form using the keyword CONSTANT is required only if the user wishes <u>never</u> to change the initial value during execution[†] .
3. The type of the literal *value* must be compatible with the type of the data item as determined from the following table:

data type	literal value
CHARACTER	character string[††]
BOOLEAN	Boolean
INTEGER SCALAR	arithmetic

[†] In many respects, a data item initialized in this way is akin to a literal.

[††] If the length of the literal value in the CONSTANT clause is greater than the declared length of the variable, the literal will be truncated to match the declared variable length. A warning message (DI18) will be generated.

Examples:

```
DECLARE A SCALAR INITIAL(3),
        B SCALAR CONSTANT(4.5E-3),
        C CHARACTER(80) INITIAL('YES'),
        D BOOLEAN INITIAL (TRUE);
```

Note: initial working length of C becomes 3.

MULTI-VALUED DATA ITEMS

There are two corresponding variations of the INITIAL/CONSTANT specification for multi-valued data items:

INITIAL(value^1, value^2,....)
CONSTANT(value^1, value^2,....)

1. The meaning of the keyword CONSTANT is the same as for uni-valued data items.
2. The type of each literal *value* must be compatible with the type of the data item as determined from the following table:

data type	literal value
CHARACTER	character string
BOOLEAN	Boolean
INTEGER	arithmetic
SCALAR	
VECTOR	
MATRIX	

Note that if <u>all</u> the elements of a multi-valued data item are to be initialized to the <u>same value</u> then the form used for uni-valued data items may be used.

Examples:

```
DECLARE V VECTOR INITIAL(1,2,3.5),
        S ARRAY(2) CONSTANT (1,0),
        T ARRAY(2) VECTOR(2) INITIAL(4.7,-5.3,0,0);
DECLARE V VECTOR INITIAL(0),
        S ARRAY(100) INTEGER INITIAL(256);
                                           ↑
```

all elements of these data items are identically initialized.

ORDER OF INITIALIZATION

To complete the specification of initialization, the order initialization of the elements of multi-valued data items needs to be defined.

The following ordering rules, though applied here to the initialization of multi-valued data items, holds true whenever the ordering of elements is called into question.

- VECTOR data items are initialized in order of increasing index.
- MATRIX data items are initialized row by row in order of increasing index.
- ARRAY data items are initialized array element by array element in order of increasing index. Where the array element are themselves multi-valued, each array element in turn is initialized completely according to the previous rules before going on to the next.

Example:[3]

```
DECLARE M ARRAY(2) MATRIX(2,2) INITIAL(1,2,3,4,5,6,7,8);
```

if M_1 is the first array element, and M_2 is the second, then:

$$M_1 \equiv \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}, \quad M_2 \equiv \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$$

Literal values appearing in initial lists may be expressions computable at compile time rather than literals.

See: Guide/Appendix D.

Additional more compact initialization forms are available if only partial initialization is required, or if subsets of the initial values are identical.

See: Guide/16.

[3] In this and many following examples in the Guide, the symbol ≡ means "has the value", or "having the value".

This page intentionally left blank.

5.0 REPLACE STATEMENTS

Often in writing a HAL/S program, it may be necessary to use the same language construct, identically repeated, many times. To avoid the tedium involved in rewriting it each time it is required, a so-called "replace name" can be defined to represent symbolically the text of the construct. The replace name can then be written in place of the construct each time, and the HAL/S compiler will perform the necessary substitutions.

The use of such replace names is especially useful in cases where the constructs they represent may be required to be modified from compilation to compilation.

The definition of the replace name and the text it substitutes is accomplished by a REPLACE statement.

5.1 THE REPLACE STATEMENT

The REPLACE statement is placed together with the data declarations of the program, or other block in which it is to be used. It takes the form:

```
REPLACE name BY "XXXXXXX";
```

1. *name* is the replace name chosen to symbolically represent the text, It may be any legal identifier name.
2. XXXXXXX represents the HAL/S source text which is to be substituted. The text is delimited by double quote marks, and must be written in single line format.
3. XXXXXXX may be <u>any</u> legal source text of arbitrary length. Imbedded double quote marks must be represented as a <u>pair</u> of double quote marks to avoid confusion with the delimiters,
4. The text must not begin or end in the middle of a reserved word, identifier, literal, or imbedded comments.

Examples:

```
REPLACE OUTPUT BY "WRITE (6)";
REPLACE INCREMENT BY "X=X+1;";
```

5.2 USING REPLACE STATEMENTS

The following examples show the way in which the symbol substitution defined by the REPLACE statement is used.

Examples:

```
REPLACE DV BY “VECTOR DOUBLE INITIAL(0)”;
DECLARE VEC1 DV,
        VEC2 DV,
        VEC3 DV;
```

- by expansion of DV it is evident that VEC1, VEC2, VEC3 are all double precision vectors initialized to zero.

```
REPLACE N BY “4”;
DECLARE V1 VECTOR(N),
        M1 MATRIX(N,N),
        M2 MATRIX(2,N);
```

- this shows the utility of the REPLACE statement in making it easy to change the sizes of several vectors and matrices simultaneously.

```
REPLACE X BY “VECTOR(2)”;
REPLACE Y BY “ARRAY(5) X”;
```

- this is an example of nested substitutions. The expansion of Y is ARRAY(5) VECTOR(2).

```
REPLACE X BY “REPLACE Y BY””Z”””;
X;
DECLARE Y SCALAR;
```

- although this is a legal use of REPLACE statements, it does not lend itself to clarity. The sequence of statements culminates in Z being declared as a scalar data item.

A REPLACE statement takes effect only after it appears. It does not modify the entire block, only that section that follows its appearance.

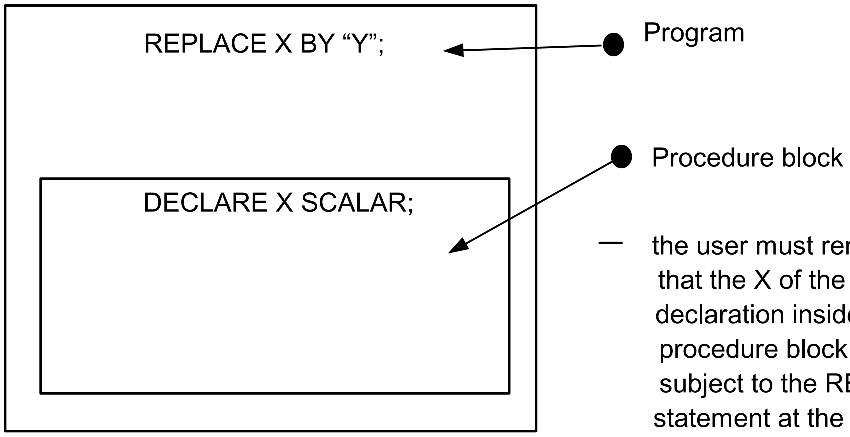
Example:

```
DECLARE V1 VECTOR(N);
REPLACE N BY “4”;
DECLARE V2 VECTOR(N);
 .
 .
 .
```

- the replace statement will be effective only starting with the second declaration statement. N is unknown in the first declaration and compilation would detect the error.

Care must be taken in using REPLACE statements because the ways in which they are affected by the block structure of the HAL/S program in which they appear are not always obvious.

Example:

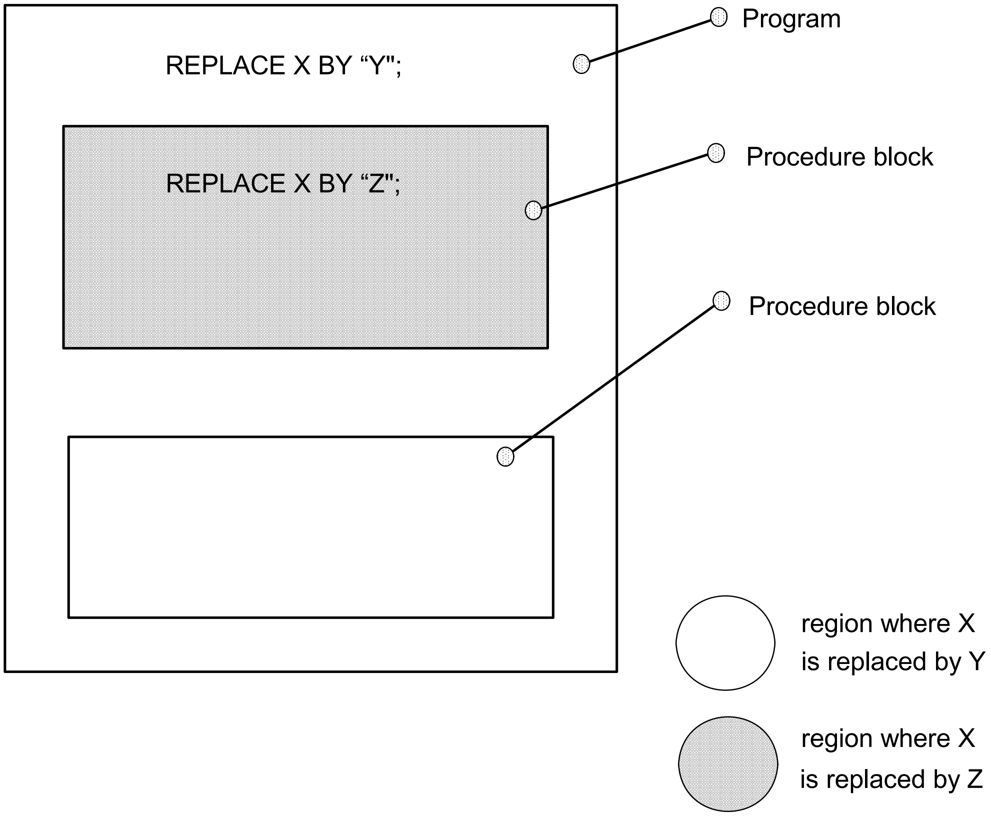


```
REPLACE X BY “Y”;
DECLARE X SCALAR;
```

— the user must remember that the X of the local declaration inside the procedure block is still subject to the REPLACE statement at the program level.

Figure 5-1

The only case in which a REPLACE statement in an outer block becomes ineffective in an inner block is when the inner block has a REPLACE statement in it with the same name.



Replace statements may also possess parameters, turningthem into a sophisticated macro expansion facility.
See Guide/29.

Figure 5-2

6.0 DATA REFERENCING AND SUBSCRIPTING

Any appearance of the name of a previously-declared data item in an executable statement constitutes a reference to its value (and possibly causes a change in its value)[10]. Sometimes it is necessary to be able to reference elements of vectors, matrices, and arrays, and also to reference parts of character strings. HAL/S has a wide range of subscript forms designed for this purpose.

Two kinds of subscripting are relevant to the data types described in Section 4.

- COMPONENT SUBSCRIPTING allows the user to select elements or subsets of elements from vectors and matrices, and to select substrings from character data items.
- ARRAY SUBSCRIPTING allows the user to select elements or subsets of elements from arrays of any data type.

Depending on the nature of a particular data item, either or both kinds of subscripting may be affixed to it.

6.1 SUBSCRIPTS OF UNARRAYED DATA TYPES

Unarrayed data types, i.e., those whose declarations contain no array specification, may at most possess only component subscripting. Unarrayed data items of integer, scalar, and Boolean types may not possess any subscripting. Allowable subscripts for the remaining types, - character, vector, and matrix - are now each described in turn.

CHARACTER

In a character data item, character positions are indexed left to right starting from 1. In the subscript forms given below, STRING represents an unarrayed data item of character type with current working length L.[11]

- To select the α^{th} character from STRING:

$$\mathtt{STRING}_{\alpha}$$

1. α is an integer expression in the range $1 \le \alpha \le L$.

- To select α characters from STRING, starting from the β^{th}:

$$\mathtt{STRING}_{\alpha\ \mathtt{AT}\ \beta}$$

1. α and β are integer expressions
2. β is in the range $1 \le\beta \le L$.
3. α is in the range $0 \le \alpha \le L - \beta + 1$.

[10] This Section, for convenience, includes appearance causing change in value under the term “reference", even though this is not the most usual meaning of the term.

[11] In the case where reference of a subscripted character data type causes a change in its value (e.g. on the left hand side of an assignment), somewhat different interpretations of the subscript form holds true. An account of these is given in Section 8.3.

- To select a substring starting with the α^{th} character of STRING, and ending with the β^{th}:

$\texttt{STRING}_{\alpha\ \texttt{TO}\ \beta}$

1. α and β are integer expressions in the range $1 \le (\alpha,\beta) \le L$.
2. $\beta \ge \alpha$.

Examples:

if C ≡ 'ABCDEF' then:

C_5 ≡ 'E' For further information refer to the Language Specification, Sec. 5.3.2 where the use of # is explained.

$C_{2\ AT\ 2}$ ≡ 'BC'

$C_{4\ TO\ 6}$ ≡ 'DEF'

VECTOR

Elements of a vector are indexed starting from 1. In the following subscript forms, VEC represents an unarrayed vector data item of length L.

- To select the α^{th} element from VEC:

$\texttt{VEC}_{\alpha}$

1. α is an integer expression in the range $1 \le \alpha \le L$.
2. The resulting data type is scalar.

- To select an α-vector partition starting from the β^{th} element of VEC:

$\texttt{VEC}_{\alpha\ \texttt{AT}\ \beta}$

1. α is an integer <u>literal value</u> in the range $2 \le \alpha \le L$.
2. β is an integer expression in the range $1 \le \beta \le L - \alpha + 1$.

- To select a partition starting from the αth element of VEC and ending with the β^{th}.

$\texttt{VEC}_{\alpha\ \texttt{TO}\ \beta}$

1. α and β are integer <u>literal values</u> in the range $1 \le (\alpha,\beta) \le L$.
2. $\beta > \alpha$.

Examples:

$$\text{if } V \equiv \begin{bmatrix} 4.5 \\ 9.3 \\ 7.1 \\ 2.7 \end{bmatrix}$$

then:

$$V_1 \equiv 4.5 \qquad \text{(scalar)}$$

$$V_{3 \text{ TO } 4} \equiv \begin{bmatrix} 7.1 \\ 2.7 \end{bmatrix} \qquad \text{(2-vector)}$$

$$V_{2 \text{ AT } 1} \equiv \begin{bmatrix} 4.5 \\ 9.3 \end{bmatrix} \qquad \text{(2-vector)}$$

MATRIX

Rows and columns of a matrix are indexed starting from 1. Any matrix subscript must consist of a row subscript followed by a column subscript. In the following subscript forms, MAT represents an unarrayed M x N matrix data item.

- To select the element of MAT common to the α^{th} row and β^{th} column:

$\text{MAT}_{\alpha,\beta}$

1. α, β are integer expressions.
2. α is in the range $1 \le \alpha \le M$, and β is in the range of $1 \le \beta \le N$.
3. The resultant data type is SCALAR.

- To select the α^{th} row of MAT:

$\text{MAT}_{\alpha,*}$

1. α is an integer expression in the range $1 \le \alpha \le M$.
2. The resultant data is N-VECTOR.
3. If the asterisk is replaced by a TO- or AT- subscript under the rules given for vector data types, a vector partition from the α^{th} row may be selected.

- To select the β^{th} column of MAT:

$\text{MAT}_{*,\ \beta}$

1. β is an integer expression in the range $1 \le \beta \le N$.
2. The resultant data is M-VECTOR.
3. If the asterisk is replaced by a TO- or AT- subscript under the rules given for vector data types, a vector partition from the β^{th} column may be selected.

- To select a α x γ matrix partition starting from the β^{th} row and σ^{th} column of MAT:

$$\texttt{MAT}_{\alpha\ \texttt{AT}\ \beta,\gamma\ \texttt{AT}\ \alpha}$$

1. α, γ are integer literal values in ranges $2 \leq \alpha \leq M$, $2 \leq \gamma \leq N$, respectively.
2. β, γ are integer expression in ranges $1 \leq \beta \leq M - \alpha + 1$, $1 \leq \sigma \leq N - \gamma + 1$ respectively.
3. Either or both the AT-subscripts may be replaced by TO-subscripts under rules already given by vector and matrix types.
4. Either of the AT- subscripts may in addition be replaced by an asterisk if all M rows or all N columns are to be included in the partition.

Examples:

$$\text{if } M \equiv \begin{bmatrix} 1.1 & 1.2 & 1.3 \\ 2.1 & 2.2 & 2.3 \\ 3.1 & 3.2 & 3.3 \end{bmatrix} \quad \text{then:}$$

$M_{2,3} \equiv 2.3$ (scalar)

$$M_{*,1} \equiv \begin{bmatrix} 1.1 \\ 2.1 \\ 3.1 \end{bmatrix} \quad \text{(3-vector)}$$

$$M_{2,\ 2\ \text{TO}\ 3} \equiv \begin{bmatrix} 2.2 \\ 2.3 \end{bmatrix} \quad \text{(2-vector)}$$

$$M_{*,2\ \text{AT}\ 1} \equiv \begin{bmatrix} 1.1 & 1.2 \\ 2.1 & 2.2 \\ 3.1 & 3.2 \end{bmatrix} \quad \text{(3x2 matrix)}$$

$$M_{1\ \text{TO}\ 2,1\ \text{TO}\ 2} \equiv \begin{bmatrix} 1.1 & 1.2 \\ 2.1 & 2.2 \end{bmatrix} \quad \text{(2x2 matrix)}$$

6.2 SUBSCRIPTS OF ARRAYED DATA TYPES

Arrayed data types, i.e., those whose declarations contain an array specification, may possess array subscripting. If the data types are vector, matrix, or character, then they may, in addition, possess component subscripting.

ARRAY SUBSCRIPTING ONLY

Arrays are indexed starting from 1. In the array subscript forms given below, TABLE represents an array of length L of any data type.

- To select the α^{th} array element from TABLE:

$\mathtt{TABLE}_{\alpha\mathtt{:}}$

1. α is an integer expression in the range $1 \leq \alpha \leq L$.
2. The colon is <u>optional</u> only if the data type of TABLE is integer or scalar.

- To select a sub-array of length α starting from the β^{th} array element of TABLE:

$\mathtt{TABLE}_{\alpha\ \mathtt{AT}\ \beta\mathtt{:}}$

1. α is an integer <u>literal value</u> in the range $1 \leq \alpha \leq L$.
2. β is an integer expression in the range $1 \leq \beta \leq L - \alpha + 1$.
3. The colon is <u>optional</u> if the data type of TABLE is integer or scalar.

- To select a sub-array starting from the αth array element of TABLE and ending with the β^{th}.

$\mathtt{TABLE}\ _{\alpha\ \mathtt{TO}\ \beta\mathtt{:}}$

1. α, β are integer <u>literal values</u> in the range $1 \leq (\alpha,\beta) \leq L$.
2. $\beta > \alpha$
3. The colon is <u>optional</u> if the data type of TABLE is integer or scalar.

Examples:

if T is a 4-array of Booleans with
$T \equiv$ (TRUE FALSE TRUE TRUE) then:
$T_{2:} \equiv$ FALSE (unarrayed)
$T_{3\ TO\ 4:} \equiv$ (TRUE,FALSE) (still arrayed)

if T is a 4-array of integers with
$T \equiv (1\ 2\ 3\ 4)$ then:
$T_2 \equiv 2$ (unarrayed)
$T_{3\ TO\ 4} \equiv (3,4)$ (still arrayed)
} optional colon omitted

if C is a 3-array of characters, with
$C \equiv$ ('YES' 'NO' 'MAYBE') then:
$C_{1:} \equiv$ 'YES' (selects first array element)
$C_{2\ TO\ 3:} \equiv$ ('NO','MAYBE') (still arrayed)

ARRAY AND COMPONENT SUBSCRIPTING

If TABLE represents an array of vector, matrix, or character data type, then the following rule shows how array and component subscripting are juxtaposed.

$\mathtt{TABLE}_{\mathtt{array:\ component}}$

1. *array* represents array subscripting of any of the forms previously described.
2. *component* represents any form of component subscripting legal for the data type of TABLE, as described in Section 6.1.

The purpose of the colon now becomes clear: it is required to distinguish and separate array and component subscripting.

Examples:

if C is a 3-array of characters, with C ≡ ('YES','NO','MAYBE') then:

$C_{3:3}$ ≡ 'Y' (selects 3rd character from third array element)

if M is a 2-array of 2x2 matrices with

$$M \equiv \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}, \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} \quad \text{then:}$$

$M_{2:2,\ 2}$ = 8 (element in 2nd row, 2nd column of second array element)

Apparently, the colon should be optional on Boolean data types also. It is not because the Boolean data type is a degenerate case of a bit string data type which may possess component subscripting.

See: Guide/17.3.

COMPONENT SUBSCRIPTING ONLY

When an arrayed data item of vector, matrix or character type is required to be given only component subscripting, array subscripting cannot be totally omitted. Rather, it must be replaced by an asterisk. Let TABLE represent such a data item; the subscripting form is then required to be:

$\mathtt{TABLE}_{*:\mathtt{component}}$

1. *component* represents any form of component subscripting legal for the data type of TABLE, as described in Section 6.1.

Examples:

if C is a 3-array of characters, with

$C \equiv$ (‘YES’,‘NO’,‘MAYBE’) then:

$C_{*:1} \equiv$ (‘Y’,‘N’,‘M’) (makes 3-array from first character of each item)

if M is a 2-array of 2x2 matrices with

$$M \equiv \left(\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}, \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} \right) \quad \text{then:}$$

$M_{*:1,1} = (1,5)$ (2-array of scalars)

$$M_{*:*,2} \equiv \left(\begin{bmatrix} 2 \\ 4 \end{bmatrix}, \begin{bmatrix} 6 \\ 8 \end{bmatrix} \right) \qquad \text{(2-array of 2-vectors)}$$

HAL/S allows more general forms of subscript expressions than just those stated above. See Spec./5.3. In particular, a symbolic form of reference to the last array or other element of a data type is allowed.

See Spec./5.3.2.

More complex subscripting forms apply to multi-dimensional arrays, See Guide/18.3; and to the organization of data called “structures”.

See Guide/19.6.

Subscript forms stated to be literals may in fact be expressions computable at compile time.

See Guide/Appendix D.

This page intentionally left blank.

7.0 EXPRESSIONS

Section 6 dealt with the referencing of declared data items. At this point it is appropriate to describe how the values of these data items can be manipulated. In HAL/S the construct which specifies operations on data items is called an "expression"[1]. In many cases it is very close in form to the generally accepted notion of a mathematical expression.

Generally, expressions consist of sequences of operations, possibly parenthesized in places to override the precedence rules of HAL/S. Each operation is comprised of one or two operands and an operator. The very simplest form of expression is one in which there are no operations and just one operand. An operand may be a data item, possibly subscripted, or a built-in function, or an explicit conversion function. This section begins by describing the legal HAL/S operations, and then continues to show how they are combined into expressions.

Previous sections of the Guide have divided data items and literals into three broad classes: arithmetic, character, and Boolean. It is convenient to divide the operations to be described into the same three classes. The type of an expression is the type of the value resulting from its execution, and may, in general, be different from the types of some of its operands.

7.1 ARITHMETIC OPERATIONS

Arithmetic operations are the most numerous of all operations in the HAL/S language. They comprise operations on vector, matrix, integer, and scalar data types. HAL/S recognizes the following operations:

Symbol	Purpose
* *	exponentiation, inversion, transposition
(blank)	multiplication
*	vector cross product
.	vector dot product
/	division
+	addition
-	subtraction, negation

[1] The storing of the result of a HAL/S expression into a data item is performed by an ASSIGNMENT statement, of which the expression forms a part.

NEGATION

Negation is a unary operation applicable to any arithmetic data type:

Symbolic form: - *R*

1. The legal data types for *R* are given by the following table:

R-type
MATRIX
VECTOR
SCALAR
INTEGER

2. Negation of vector and matrix types implies element-by-element negation.

Examples:

if I is an integer and I ≡ 5

then -I ≡ -5

if V is a 3-vector and V $\equiv \begin{bmatrix} -1.5 \\ 4.2 \\ -5.1 \end{bmatrix}$

then - V $\equiv \begin{bmatrix} 1.5 \\ -4.2 \\ 5.1 \end{bmatrix}$

ADDITION AND SUBTRACTION

Addition and subtraction can only take place between compatible arithmetic data types:

Symbolic form: $L \pm R$

1. The legal combinations of data types are indicated by the following table:

L-type	*R*-type
MATRIX	MATRIX
VECTOR	VECTOR
SCALAR	SCALAR
INTEGER	INTEGER

2. Operations on matrix and vector operands imply element-by-element addition and subtraction.
3. The operands in a matrix addition or subtraction must have the same row and column dimensions.
4. The operands in a vector addition or subtraction must have the same lengths.
5. In a mixed integer-scalar operation, the result is scalar. The integer operand is first converted to single precision scalar.

Examples:

If I is integer with I ≡ 5

S is scalar with S ≡ -4.2

then

I + 1 ≡ 6 (integer result)

I + 0.5 ≡ 5.5 (scalar result)

S + 1 ≡ -3.2 (scalar result)

I - S ≡ 9.2 (scalar result)

if V1 is a 3-vector with V1 ≡ $\begin{bmatrix} -1.0 \\ -2.5 \\ 3.2 \end{bmatrix}$

V2 is a 4-vector with V2 ≡ $\begin{bmatrix} 0.5 \\ 0 \\ -2.2 \\ 1.5 \end{bmatrix}$

then the operation V1 + V2 is illegal because the lengths of V1, V2 do not match;

but

$V1 - V2_{1\ \mathrm{TO}\ 3} \equiv \begin{bmatrix} -1.5 \\ -2.5 \\ 5.4 \end{bmatrix}$ is legal because of the *R* operand has produced a 3-vector.

Using S, V1 above,

S + V1 is illegal because the types are incompatible;

but $S + V1_3 \equiv -1.0$ is legal and has a scalar result because subscripting has changed the *R* operand to scalar type.

If M1 is a 3 x 2 matrix with $M1 \equiv \begin{bmatrix} 1.0 & 0 \\ -0.5 & -1.0 \\ 0 & 0 \end{bmatrix}$

M2 is a 2 x 2 matrix with $M2 \equiv \begin{bmatrix} 0.5 & -0.5 \\ 1.0 & 1.0 \end{bmatrix}$

then M1 - M2 is illegal because the row dimensions of the operands do not match;

but

$M1_{2\ \mathrm{AT}\ 1,*} - M2 \equiv \begin{bmatrix} 0.5 & 0.5 \\ -1.5 & -2.0 \end{bmatrix}$ is legal because the number of rows in the *L* operand have been reduced to 2 by subscripting.

DIVISION

In division, the dividend may be any data type, but the divisor must either be integer or scalar.

Symbolic form: *L*/*R*

1. The legal combinations of data types are given by the following table:

L-type	*R*-type
MATRIX VECTOR SCALAR INTEGER	SCALAR INTEGER

2. If the dividend is of matrix or vector type, element-by-element division by the *R* operand is implied.
3. If either or both operands are of integer type, they are first converted to scalar type.

Examples:

$1/2 \equiv 0.5$ (both integer operands converted to scalar)

$$\text{if V is a 3-vector with } V \equiv \begin{bmatrix} 2.0 \\ 4.0 \\ 6.0 \end{bmatrix}$$

$$\text{then } V/2 \equiv \begin{bmatrix} 1.0 \\ 2.0 \\ 3.0 \end{bmatrix}$$

$$\text{if M is a 2 x 2 matrix with } M \equiv \begin{bmatrix} 1.0 & -0.5 \\ 0.2 & 0.6 \end{bmatrix}$$

S is a scalar with $S \equiv 0.5$

then S/M is illegal since the *R* operand may not be of matrix type,

$$\text{but } M/S \equiv \begin{bmatrix} 2.0 & -1.0 \\ 0.4 & 1.2 \end{bmatrix}$$

DOT PRODUCT

The HAL/S dot product operation corresponds to the mathematical dot or inner product of two vectors. In mathematical notation:

$$s = \langle u , v \rangle \quad \text{or} \quad s = u^T v$$

where u, v are column vectors and T denotes the transpose.

Note that HAL/S does <u>not</u> require the user to distinguish between row and column vectors because the position of the operand in the operation is sufficient in itself to allow it to be interpreted as one or the other.

Symbolic form: *L.R*

1. The operands of the dot product must be as shown:

L-type	*R*-type
`VECTOR`	`VECTOR`

2. The lengths of each operand <u>must</u> be the same.
3. The result is of scalar type.

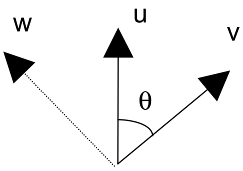
Example:

if V is a 3-vector with V $\equiv \begin{bmatrix} 0.5 \\ 1.0 \\ -0.5 \end{bmatrix}$

then V.V = 1.5

CROSS PRODUCT

The HAL/S cross product operation corresponds to the mathematical vector cross product in 3-dimensional Euclidean space:



if w is perpendicular to u, v as shown,
and | w | = | u | | v | sin θ then w = u x v

Figure 7-1

Symbolic form: *L* * *R*

1. The operands must be of type vector:

L-type	*R*-type
VECTOR	VECTOR

2. Both operands must be of length 3.

3. The result is a 3-vector.

Example:

if V1 is a 3-vector with V1 $\equiv \begin{bmatrix} 0.5 \\ 0 \\ 0 \end{bmatrix}$

if V2 is a 3-vector with V2 $\equiv \begin{bmatrix} 0 \\ 0.5 \\ 0 \end{bmatrix}$

then V1 * V2 $\equiv \begin{bmatrix} 0 \\ 0 \\ 0.25 \end{bmatrix}$

MULTIPLICATION

The HAL/S language has no explicit symbol for multiplication: the adjacency of two operands signifies this operation. Multiplication can take place with arithmetic operands of any type:

- If operand types are either integer or scalar, multiplication in the regular arithmetic sense is implied;……………………………………………………………….………CASE ①
- if one operand is integer or scalar, and the other vector or matrix, then element-by-element multiplication is implied;…………………………………………… CASE ②
- if both operands are vectors then the outer product is implied, the result being a matrix;..……………………………………………………………………........CASE ③
- if both operands are matrices, the matrix product is implied;……………...CASE ④
- if one operand is a matrix, and the other a vector, then a vector-matrix product is implied, the result being a vector..……………………………………….…...CASE ⑤

The symbolic form for multiplication is as shown:

Symbolic form: *L R*

1. At least one blank character must separate the *L* and *R* operands.

The additional rules applicable to each of the cases described above are now listed in turn.

CASE ①

2. The operand types are:

L-type	*R*-type
INTEGER SCALAR	INTEGER SCALAR

3. If both operands are integer, the result is integer, otherwise it is scalar.
4. If one operand is integer, then it is first converted to single precision scalar.

Example:

If I is integer with I ≡ 10

then 1.5E-2 I ≡ 0.15 (scalar result)

CASE ②

2. The operand types are:

L-type	*R*-type
INTEGER SCALAR	VECTOR MATRIX
VECTOR MATRIX	INTEGER SCALAR

3. Element-by-element multiplication of the vector or matrix is implied.

4. If one operand is of integer type, it is first converted to single precision scalar.

Examples:

if S is scalar with $S \equiv 1.5$

$$M \text{ is a } 2 \times 2 \text{ matrix with } M \equiv \begin{bmatrix} 0 & 0.3 \\ -0.1 & 0.4 \end{bmatrix}$$

$$\text{then } S\ \ M \equiv \begin{bmatrix} 0 & 0.45 \\ -0.15 & 0.6 \end{bmatrix}$$

$$\text{and } M\ \ S \equiv \begin{bmatrix} 0 & 0.45 \\ -0.15 & 0.6 \end{bmatrix}$$

CASE ③

2. The operand types are:

L-type	*R*-type
VECTOR	VECTOR

3. If the *L*-operand is of length m, and the *R* operand is of length n, the result is an m x n matrix.

Examples:

If V1 is a 3-vector with $V1 \equiv \begin{bmatrix} 1.0 \\ -1.0 \\ 1.0 \end{bmatrix}$

V2 is a 2-vector with $V2 \equiv \begin{bmatrix} 0.5 \\ 0.6 \end{bmatrix}$

then $V1\ \ V2 \equiv \begin{bmatrix} 0.5 & 0.6 \\ -0.5 & -0.6 \\ 0.5 & 0.6 \end{bmatrix}$ (3 x 2 matrix)

and $V2\ \ V1 \equiv \begin{bmatrix} 0.5 & -0.5 & 0.5 \\ 0.6 & -0.6 & 0.6 \end{bmatrix}$ (2 x 3 matrix)

CASE ④

2. The operand types are:

L-type	*R*-type
MATRIX	MATRIX

3. The number of columns in the *L* operand must equal the number of rows in the *R* operand.
4. If the L operand is an m x n matrix and the R operand is an n x p matrix, the result is an m x p matrix.

Examples:

If M1 is a 2 x 3 matrix with $M1 \equiv \begin{bmatrix} 1.0 & 1.0 & 2.0 \\ 0.5 & -0.5 & 1.0 \end{bmatrix}$

M2 is a 3 x 2 matrix with $M2 \equiv \begin{bmatrix} 0 & 0.5 \\ 0 & 1.0 \\ 0 & 1.0 \end{bmatrix}$

then $M1\ M2 \equiv \begin{bmatrix} 0 & 3.5 \\ 0 & 0.75 \end{bmatrix}$ (2 x 2 matrix)

$$\text{and M2 M1} \equiv \begin{bmatrix} 0.25 & -0.25 & 0.5 \\ 0.5 & -0.5 & 1.0 \\ 0.5 & -0.5 & 1.0 \end{bmatrix} \qquad \text{(3 x 3 matrix)}$$

Note that by using partitioning subscripts that $M1_{*,2\ \mathrm{TO}\ 3}$ M2 is illegal because of dimension mismatch;

$$\text{but M2 } M1_{*,2\ \mathrm{TO}\ 3} \equiv \begin{bmatrix} -0.25 & 0.5 \\ -0.5 & 1.0 \\ -0.5 & 1.0 \end{bmatrix} \quad \text{is still legal}$$

CASE ⑤

2. The operand types are:

L-type	*R*-type
VECTOR	MATRIX
MATRIX	VECTOR

3. If the *L* operand is an m x n matrix, the *R* operand must be an n-vector, and the result is an m-vector.
4. If the *L* operand is an m x n matrix, the *R* operand must be an m-vector, and the result is an n-vector.

Note that the position of the vector operand again determines its interpretation as either a row or column vector.

Examples:

$$\text{If M is a 3 x 2 matrix with M} \equiv \begin{bmatrix} 0.5 & 1.0 \\ 0 & 1.0 \\ 0.2 & 0.4 \end{bmatrix}$$

$$\text{V is a 3-vector with V} \equiv \begin{bmatrix} 1.0 \\ -1.0 \\ 1.0 \end{bmatrix}$$

$$\text{then V M} \equiv \begin{bmatrix} 0.7 \\ 0.4 \end{bmatrix} \qquad \text{(2-vector)}$$

and M V is illegal because of dimension mismatch;

$$\text{however, M } V_{1\ \mathrm{TO}\ 2} \equiv \begin{bmatrix} -0.5 \\ -1.0 \\ -0.2 \end{bmatrix} \quad \text{is legal}$$

EXPONENTIATION, INVERSION AND TRANSPOSE

In HAL/S, a single operator serves for exponentiation, matrix inversion, and matrix transpose, the operand types serving to distinguish between them.

- If both operands are integer or scalar, then exponentiation is implied;......CASE ①
- if the left operand is a square matrix, and the right is an integer-valued literal, a repeated matrix product or repeated product of inverse is implied;……......CASE ②
- if the left operand is a matrix, and the right operand is the character 'T', then the transpose is implied..…………………………………………………….......CASE ③

These operations take the general symbolic form:

Symbolic form: *L* * * *R*

1. This is the one-line format version. In multi-line format the operator symbol is omitted and *R* is placed on an exponent line. See Section 2.3.

The rules for each of the cases listed above are now described in turn.

CASE ①

2. The operand types are:

L-type	*R*-type
INTEGER	INTEGER
SCALAR	SCALAR

3. If the *L* operand is integer and the *R* operand is a non-negative integral-valued literal, then the result is integer, otherwise it is scalar.
4. Consistent with Rule 3, if the result is scalar, then any integer operands are first converted to single-precision scalar.

Examples:

If I is an integer with I ≡ 5

then I * * 2	≡ 25	(integer result)
and I * *-1	≡ 0.2	(scalar result)
also 2* *0.5	≡$\sqrt{2}$	(scalar result)

CASE ②

2. The operand types are:

L-type	*R*-type
`MATRIX`	`INTEGER`

3. The *L* operand is a square matrix.

4. The *R* operand is an integral-valued literal. The following table shows the effect of different ranges of values of the *R* operand:

value	result
$\leq$ - 2	repeated product of inverse
-1	inverse
0	unit matrix
1	no-operation
$\geq$ 2	repeated product

Examples:

If M is a 2 x 2 matrix with $M \equiv \begin{bmatrix} 0.5 & 1.0 \\ -0.5 & 0 \end{bmatrix}$

then $M^2 \equiv \begin{bmatrix} -0.25 & 0.5 \\ -0.25 & -0.5 \end{bmatrix}$

$M\text{-}^1 \equiv \begin{bmatrix} 0 & -2.0 \\ 1.0 & 1.0 \end{bmatrix}$

and $M^0 \equiv \begin{bmatrix} 1.0 & 0 \\ 0 & 1.0 \end{bmatrix}$

CASE ③

2. The operand types are:

L-type	*R*-type
`MATRIX`	`T`

3. If the *L* operand is an m x n matrix, then the result is an n x m matrix.

4. If *R* is symbolically T, then transpose is indicated even if T is a declared data item.

Examples:

If M is a 2 x 3 matrix with $M \equiv \begin{bmatrix} 1.0 & 0 & 3.0 \\ 2.0 & 0 & 4.0 \end{bmatrix}$

then $M^T \equiv \begin{bmatrix} 1.0 & 2.0 \\ 0 & 0 \\ 3.0 & 4.0 \end{bmatrix}$

If V is a 3-vector with $V \equiv \begin{bmatrix} 1.0 \\ 2.0 \\ 3.0 \end{bmatrix}$

then V^T is illegal because the L operand is not matrix type.

The transpose of a vector is not needed in the HAL/S language.

NOTE ON PRECISION CONVERSION

It is possible that the precisions of the two operands may differ in any of the operations described. In these cases, precision conversion usually takes place before the operation is executed. The rules under which it takes place are as follows:

1. No precision conversion is applicable in unary operations: transposition is considered a unary operation.
2. Where one operand only is integer, the precision of the result is the same as the precision of the other operand. Where the operation implies an integer-to-scalar conversion, the result of the conversion is generated with the precision of the other operand.
3. If Rule 2 does not apply, and the precisions of the operands differ, the single precision operand is first converted to double precision. The precision of the result of the operation is the same as the precision of the operands after the possible precision conversion.

The single to double precision conversion only takes place for a SCALAR operation if there is a double precision SCALAR variable in the expression. Double precision SCALAR CONSTANTs do not affect the precision of the operation. For Example:

```
DECLARE DOUBLE_VAR SCALAR DOUBLE;
DECLARE SINGLE_VAR SCALAR INITIAL(X);
DECLARE DOUBLE_CONSTANT SCALAR DOUBLE CONSTANT(Y);
DOUBLE_VAR = SINGLE_VAR + DOUBLE_CONSTANT;
```

The DOUBLE_CONSTANT is converted to single precision, and the operation is performed in single precision.

7.2 CHARACTER OPERATIONS

There is only one character operation in HAL/S: the catenation of character strings.

Symbol	Purpose
|| `CAT`	catenation

CATENATION

The utility of catenating character strings is obvious in the generation of output listings. The rules related to the catenation operation are as follows:

Symbolic form: *L* || *R*
CAT

1. The *L* and *R* operands are not just restricted to character type: some degree of implicit type conversion is allowed. The following types are legal.

L-type	*R*-type

`INTGER`
`SCALAR`
`CHARACTER`
	`INTGER`
	`SCALAR`
	`CHARACTER`

2. The rules for converting integer and scalar types to character type are to be found in Appendix A.

Examples:

If C is a character item with C ≡ ' UNITS'

I is integer with I ≡ 10

then 'TEN' || C ≡ 'TEN UNITS'

I || C ≡ '10 UNITS'

and

I || I ≡ '1010'

7.3 BOOLEAN OPERATIONS

Boolean operations are logical (binary) transformations on Boolean operands. HAL/S recognizes the following operations:

Symbol	Purpose
& AND	logical intersection
| OR	logical conjunction
¬ NOT	logical complement

COMPLEMENT

The complement operation complements the logical value of a Boolean operand. It takes the following form:

Symbolic form: ¬ *R*
NOT

1. The *R* operand is of Boolean type.

Example:

If B is Boolean with B ≡ TRUE

then ¬ B ≡ FALSE

CONJUNCTION

The conjunction operation causes the logical values of two Boolean operands to be OR’ed together.

$$\text{Symbolic form:}L \begin{Bmatrix} | \\ \text{OR} \\ \ \end{Bmatrix} R$$

1. The *L* and *R* operands are of Boolean type.
2. The truth table for the resulting Boolean is as follows:

T=TRUE
F=FALSE
		L	

		T	F
R	T	T	T
	F	T	F

Examples:

If B is Boolean with B ≡ FALSE

then

B | B ≡ FALSE

B | TRUE ≡ TRUE

INTERSECTION

The intersection operation causes the logical values of two Boolean operands to be AND'ed together.

$$\text{Symbolic form:}L \begin{Bmatrix} \& \\ \text{AND} \\ \ \end{Bmatrix} R$$

1. The L and R operands are of Boolean type.
2. The truth table for the resulting Boolean is as follows:

T=TRUE
F=FALSE
		L	

		T	F
R	T	T	F
	F	F	F

Examples:

If B is Boolean with B ≡ FALSE

then

B & TRUE ≡ FALSE

B & B ≡ FALSE

7.4 COMBINING OPERATIONS & PRECEDENCE

It is obviously desirable to be able to combine operations so as to create expressions of any required complexity. In combining operations, the following information is necessary:

- The order in which operations are executed (the order of "precedence");
- the way in which the precedence order can be overridden.

ARITHMETIC AND CHARACTER PRECEDENCE

The precedence of HAL/S operations on arithmetic and character data types are shown in the following table:

SYMBOL	PRECEDENCE	PURPOSE
	FIRST	
* *	1	exponentiation, etc.
(blank)	2	multiplication
*	3	cross product
.	4	dot product
/	5	division
+	6	addition
-	6	subtraction, negation
||, CAT	7	catenation
	LAST	

Two rules clarify and modify this information:

- Sequences of operations of the same precedence are evaluated left to right, except for * * and /, which are evaluated right to left.
- Sequences of multiplications are sometimes reordered to minimize the number of elemental products required. Dot and cross products are involved in this process.

Examples:

In the following expression, the numbered pointers show the order of execution of operations:

```
                                      2
'RESULT OF STEP ' ||N|| 'IS ' || S1+S2  - V1.V2/2/2
                  ↑  ↑        ↑    ↑  ↑ ↑   ↑  ↑ ↑
                  ①  ②        ⑨    ④  ③ ⑧   ⑤  ⑦ ⑥
```

BOOLEAN PRECEDENCE

The precedence rules for Boolean operations are stated separately because there are no implicit conversions causing interaction with arithmetic and character operations.

SYMBOL	PRECEDENCE	PURPOSE
	FIRST	
¬, NOT	1	complement
&, AND	2	intersection
|, OR	3	conjunction
	LAST	

Sequences of operations of the same precedence are evaluated left to right.

Examples:

In the following expression, the numbered pointers show the order of execution of operations:

```
 ¬  B1 | B2   & ¬ B3
 ↑     ↑      ↑ ↑
 ①     ④      ③ ②
```

OVERRIDING PRECEDENCE ORDER

In HAL/S, the order of precedence can be overridden at will by the use of parentheses, nested to any arbitrary depth.

Examples:

In the following Boolean expression,

```
  B1 | B2   &  B3 | B4   & B5
     ↑      ↑     ↑      ↑
     ②      ①     ④      ③
```

parentheses may change the precedence order as shown:

```
  (B1 | B2) & ((B3 | B4) & B5)
      ↑     ↑      ↑     ↑
      ①     ④      ②     ③
```

In the following arithmetic expression,

$$S1 \underset{\substack{\uparrow \\ \textcircled{2}}}{+} S2^{\underset{\substack{\uparrow \\ \textcircled{1}}}{2}} \underset{\substack{\uparrow \\ \textcircled{4}}}{+} S3\underset{\substack{\uparrow \\ \textcircled{3}}}{/}S2$$

parentheses may change the precedence order as shown:

$$((S1 \underset{\substack{\uparrow \\ \textcircled{1}}}{+} S2)^{\underset{\substack{\uparrow \\ \textcircled{2}}}{2}} \underset{\substack{\uparrow \\ \textcircled{3}}}{+} S3)\underset{\substack{\uparrow \\ \textcircled{4}}}{/}S2$$

> HAL/S allows the operands in an expression to be arrayed, causing parallel evaluation on an element-by-element basis See: Guide/20.1.

7.5 SOME EXPLICIT CONVERSIONS

As evidenced in Section 7, there are few implicit type conversions in the HAL/S language. However, there is a comprehensive range of explicit conversions, some of which are now described.

PRECISION CONVERSION

Any arithmetic expression may have its precision explicitly changed as follows:

$$(\mathit{expression})_{@\ \mathrm{DOUBLE}}$$
$$(\mathit{expression})_{@\ \mathrm{SINGLE}}$$

1. In the first form, if *expression* is a single precision arithmetic precision, it is converted to double precision. If it is already double precision, the conversion has no effect.
2. In the second form, if *expression* is a double precision arithmetic expression it is converted to single precision. If it is already single precision, the conversion has no effect.
3. If *expression* is of scalar type, conversion to single precision implies rounding. If it is of integer type, it entails loss of most-significant digits. See Appendix A.

Example:

If A and B are single precision, then the result of

$$(A + B)_{@\ \mathrm{DOUBLE}}$$

is double precision, the type remaining unchanged.

VECTOR CONVERSION

A vector can be synthesized from a list of scalar or integer expressions using the construct shown in the following table:

$\mathtt{VECTOR}_{\mathtt{n}}\mathtt{(}exp^1\mathtt{,}exp^2\mathtt{, ...)}$

1. The subscript number n specifies the length of the vector to be created, and lies in the range $1 < n \leq 64$[†].
2. If n is omitted the resulting vector is assumed to be of length 3.
3. Each *exp* is a scalar or integer expression .
4. The number of expressions in the list must match the implicit or explicit result length.
5. The result of the above conversion is in single precision.

[†] This value may be implementation dependent. See appropriate User's Manual.

Examples:

```
VECTOR(1,2,3)
```

creates a 3-vector with value ≡ $\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$

if S is a scalar with S ≡ 0.5 then

$\mathtt{VECTOR}_4\ \mathtt{(S,S^2,S+1,0)}$

creates a 4-vector with value $\begin{bmatrix} 0.5 \\ 0.25 \\ 1.5 \\ 0 \end{bmatrix}$

Note that even if the arguments are double precision the result is in single precision. To specify double precision in a vector conversion, the following modified form is used:

$\mathtt{VECTOR}_{\mathtt{@\ DOUBLE,n}}\mathtt{(}exp^1\mathtt{,}\ exp^2\mathtt{, ...)}$

1. The meanings of *exp* and n are as before.
2. If n is not specified, the preceding comma is also omitted.

Examples:

$\texttt{VECTOR}_{@\ \texttt{DOUBLE}}\texttt{(1,2,3)}$

creates a double precision 3-vector with value ≡ $\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$

$\texttt{VECTOR}_{@\ \texttt{DOUBLE,4}}\texttt{(1,2,3,4)}$

creates a double precision 4-vector with value ≡ $\begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}$

MATRIX CONVERSION

There exists a method of synthesizing a matrix from a list of integer or scalar expressions analogous to the vector conversion described:

$\texttt{MATRIX}_{\texttt{m,n}}\texttt{(exp1, exp2,......)}$

1. The subscript numbers m, n specify the row and column dimensions of the matrix to be created, and must lie in the range 1 < (m, n) ≤ 64[†].
2. The subscript may be omitted, in which case the resulting matrix is assumed to be 3 by 3.
3. Each *exp* is a scalar or integer expression.
4. The number of expressions must match the total number of elements in the resulting matrix.
5. The result of the above conversion is in single precision.
6. The matrix is assembled row by row from the list.

[†] This value may be implementation dependent. See appropriate User's Manual.

Examples:

```
MATRIX(1,2,3,4,5,6,7,8,9)
```

creates a 3 x 3 matrix with value $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$

$\texttt{MATRIX}_{\texttt{2,3}}\texttt{(1.5,0,0,0,0.5,0)}$

creates a 2 x 3 matrix with value $\begin{bmatrix} 1.5 & 0 & 0 \\ 0 & 0.5 & 0 \end{bmatrix}$

Note the order of assembly in each case.

As in the case of vector conversion, a modified form is required if the result is to be in double precision:

$\mathtt{MATRIX}_{@\ \mathtt{DOUBLE},m,n}(exp^1,\ exp^2,.....)$

1. The meanings of m, n and *exp* are as before.
2. If the dimension subscript is omitted, the preceding comma is also omitted.

Examples:

$\mathtt{MATRIX}_{@\ \mathtt{DOUBLE}}(1,2,3,4,5,6,7,8,9)$

creates a double precision 3 x 3 matrix with value $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$

$\mathtt{MATRIX}_{@\ \mathtt{DOUBLE},2,3}(1.5,0,0,0,0.5,0)$

creates a double precision 2 x 3 matrix with value $\begin{bmatrix} 1.5 & 0 & 0 \\ 0 & 0.5 & 0 \end{bmatrix}$

The explicit conversions described are those most commonly required for numerical analysis. However, HAL/S contains many other explicit conversion function forms corresponding to conversions between most data types.

See: Guide 21.

7.6 BUILT-IN FUNCTIONS

HAL/S possesses a comprehensive range of library or "built-in" functions that can be used as operands in expressions. Built-in functions have zero, one, or two arguments, and are written in a form akin to standard mathematical notation.

Built-in functions are divided into five different classes, roughly according to purpose:

- arithmetic
- algebraic
- vector-matrix
- character
- miscellaneous

A full description of all built-in functions is given in Appendix B. A brief explanation of some of the more important functions in each class is given below.

ARITHMETIC FUNCTIONS

Arithmetic functions perform simple arithmetic operations on scalar or integer arguments. Some arithmetic functions are:

Function	Comments
`ABS(`α`)`	returns | α | (the absolute value of α). α may be integer or scalar.
`DIV(`α,β`)`	returns the result of integer division of α by β. α and β may be scalar or integer: scalar values are rounded to integer before use.
`ROUND(`α`)`	rounds a scalar α to an integer.
`ODD(`α`)`	returns a Boolean result, which is TRUE if α is odd, and FALSE if α even.
`SIGN(`α`)`	returns +1 if $\alpha \geq 0$ and -1 if $\alpha < 0$.

ALGEBRAIC FUNCTIONS

Algebraic functions perform trigonometric and other transformations on scalar arguments. Some common algebraic functions are:

Function	Comments
`COS(`α`)`	returns cos α
`EXP(`α`)`	returns e^{α}
`LOG(`α`)`	returns $\log_e \alpha$
`SIN(`α`)`	returns sin α
`SQRT(`α`)`	returns $\sqrt{\alpha}$
`TAN(`α`)`	returns tan α

VECTOR-MATRIX FUNCTIONS

Vector-matrix functions perform operations on vectors or matrices. Common vector-matrix functions are:

Function	Comments
`ABVAL(`α`)`	returns length of vector α
`INVERSE(`α`)`	returns inverse of square matrix α
`UNIT(`α`)`	returns unit vector in same direction as vector α

CHARACTER FUNCTIONS

Character functions perform operations on character data. Some common character functions are:

Function	Comments
LENGTH(α)	returns current length of character string α
TRIM(α)	strips leading and trailing blanks from string α

MISCELLANEOUS FUNCTIONS

Some of the more important miscellaneous functions are:

Function	Comments
DATE	returns date at time of execution
MAX(α)	returns the maximum value in the integer or scalar array α
MIN(α)	returns the minimum value in the integer or scalar array α
RANDOMG	returns random number from Gaussian distribution with mean zero, variance 1.

Examples of use:

```
|
| SINE = SIN(X/2) + SIN(Y/2);
| X = ABVAL(V1*V2);
| IF ODD(X) THEN RETURN;
|
```

8.0 ASSIGNMENTS

Section 7 described, in detail, the creation of HAL/S expressions used in numerous places in the language. The assignment statement is one such instance in which the value of an expression is assigned to a data item.

For convenience, an assignment is classified according to the type of the receiving data item; that is, the data item being assigned into. Because HAL/S allows implicit type conversion, this type is not necessarily the same as the expression whose value is used in the operation.

- Arithmetic assignments are assignments to matrix, vector, integer or scalar data items.
- Character assignments are assignments to character data items.
- Boolean assignments are assignments to Boolean data items.

8.1 GENERAL FORM OF ASSIGNMENT

The assignment statement is an instance of a HAL/S executable statement. It has a general form applicable to all types of assignment:

Symbolic form *L* = *R;*

1. *L* is the receiving data item. It may be subscripted or unsubscripted.
2. Usually, *R* is an expression whose resultant value is to be used in the assignment. It may, of course, consist merely of a single operand.

Additional assignment rules are applicable which differ according to assignment type.

8.2 ARITHMETIC ASSIGNMENTS

Arithmetic assignments are those in which the receiving data type is matrix, vector, integer or scalar.

MATRIX

The receiving data item is a matrix.

1. The operand types are:

L-type	*R*-type
MATRIX	MATRIX INTEGER(rule 3)

2. The number of rows and columns of the R-expression must be the same as those of the receiving data item.

3. The only condition under which the *R*-type is integer is if it is the literal value zero. The assignment then creates a null matrix.

Examples:

If M1 is a 2x3 matrix with M1 ≡ $\begin{bmatrix} 1.0 & 1.0 & 2.0 \\ 0.5 & -0.5 & 1.0 \end{bmatrix}$

M2 is a 2x2 matrix,
M3 is a 2x3 matrix;

then

```
|
|M3 = -M1;
|
```

results in M3 ≡ $\begin{bmatrix} -1.0 & -1.0 & -2.0 \\ -0.5 & 0.5 & -1.0 \end{bmatrix}$

|
| `M2= M1;` is illegal (column mismatch)
|

but

|
| $M2 = M1_{*,2 \text{ AT } 2};$
|

results in M2 ≡ $\begin{bmatrix} 1.0 & 2.0 \\ -0.5 & 1.0 \end{bmatrix}$

```
|
|M3 = 0;
|
```

results in M3 ≡ $\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$

but

```
|
|M3 = 1;
|
```

is illegal

VECTOR

The receiving data item is a vector.

1. The operand types are:

L-type	*R*-type

`VECTOR`	`VECTOR`
	`INTEGER(rule 3)`

2. The length of the *R*-expression must be the same as that of the receiving data item.
3. The <u>only</u> condition under which the *R*-type is integer is if it is the <u>literal value</u> zero. The assignment then creates a null vector.

Examples:

If V1 is a 3-vector with V1 $\equiv \begin{bmatrix} 1.0 \\ 2.0 \\ 0 \end{bmatrix}$

M2 is a 3x3 matrix,
V2 is a 3-vector;

then

```
|
| V2 = -V1;
|
```

results in V2 $\equiv \begin{bmatrix} -1.0 \\ -2.0 \\ 0 \end{bmatrix}$

```
|
| M2 ½ V1;
|
```

is illegal (type mismatch),

but

$|M2_{1,*}$ `= V1;` is legal since subscripting reduces the L-type to 3-vector.

and results in M2 $\equiv \begin{bmatrix} 1 & 2 & 0 \\ ? & ? & ? \\ ? & ? & ? \end{bmatrix}$

(? indicates values unchanged by assignment).

Note

`V2 = 0;` creates a null vector.

INTEGER/SCALAR

Integer and scalar assignments can be treated together because their rules are nearly identical.

1. The operand types are:

L-type	*R*-type
INTEGER	INTEGER
SCALAR	SCALAR

2. The length of the *R*-expression must be the same as that of the receiving data item.
3. The only condition under which the *R*-type is integer is if it is the literal value zero. The assignment then creates a null vector.

Examples:

If I is an integer,

S is a scalar, and

M a 2x2 matrix, then

`I = 5;` results in I ≡ 5

`I = 7.7;` results in I ≡ 8

`S = 7.7;` results in S ≡ 7.7

Given the last values above for S, I

$M_{2,\ 2}$ `= I - S;`

results in $M \equiv \begin{bmatrix} ? & ? \\ ? & 0.3 \end{bmatrix}$

(? indicates values unchanged by assignment)

$M_{2,*}$ `= I;` is illegal (type mismatch)

NOTE ON PRECISION CONVERSION

In an arithmetic assignment, the precisions of the receiving data item and of the *R*-expression may differ. In these cases, precision conversion of the latter takes place before assignment, under the following rules:

1. The R-expression is converted to the precision of the receiving data item as necessary before assignment.
2. If type conversion from integer to single precision scalar is implied, its result is generated with the same precision as the receiving data item.

8.3 CHARACTER ASSIGNMENTS

The receiving data item is character type.

1. The operand types are:

L-type	*R*-type
CHARACTER	CHARACTER INTEGER SCALAR

2. *R*-expressions of integer or scalar type are converted before assignment to character type. Conversion rules are to be found in Appendix A.

Examples:

If C is a character string with C ≡ ‘ABCDE’ and C2 is a character,

then

|

| `C2 = `C_3`;` results in C2 ≡ ‘C’

|

| `C2 = 1573;` results in C2 ≡ ‘1573’

|

These apparently straightforward rules can become more complex in some situations.

Generally, when the receiving data is unsubscripted, its working length becomes the same as the length of the *R*-expression. However, if this would cause the declared maximum length of the receiving data item to be exceeded, then truncation of the excess from the right takes place.

Examples:

If C1 is character of maximum length 10
C2 is character of maximum length 1,

then

```
|
|C1 = ‘ABCDE’;
|
```

results in C1 ≡ ‘ABCDE’ of working length 5

but

```
|
|C2 = ‘ABCDE’;
|
```

results in C2 ≡ ‘A’ of working length 1

If the receiving data item is subscripted, then this causes an additional complication. The rules applicable in such a case are as follows:

Let,

$$\texttt{STRING}_{\alpha}$$

denote a receiving data item of character type:

N is declared maximum length

n is working length <u>before</u> assignment

1. The range of the subscript expression α is presumed to be in the range 1 - N; otherwise an error results.
2. The length of the R-expression is adjusted to the length implied by α, either by truncation of the excess from the right, or by padding on the right with blanks.
3. If the range of α lies inside the range 1 - n, then simple substitution of the character positions implied takes place.
4. If the range of α lies partly beyond the range 1 - n, then the working length of STRING is increased appropriately.
5. If the range of α lies totally beyond the range 1 - n, the working length of STRING is increased appropriately, and the gap between the n^{th} character and the first position implied by α (if any) is filled with blanks.

Examples:

Let C1 be character of declared maximum length 10 with value C1 ≡ ‘ABCD’

Then by Rules 2 and 3:

|
| $\texttt{C1}_{\texttt{2 TO 3}}$ = ‘QQ’;
|

results in C1 ≡ ‘AQQD’

|
| $\texttt{C1}_{\texttt{2 TO 3}}$ = ‘1234’;
|

results in C1 ≡ ‘A12D’

|
| $\texttt{C1}_{\texttt{2 TO 3}}$ = ‘X’;
|

results in C1 ≡ ‘AX D’

By Rules 2 and 4:

|
| $C1_{4\ \mathtt{TO}\ 5}$ = ‘QQ’;
|

results in C1 ≡ ‘ABCQQ’ (working length increased by 1)

|
| $C1_{4\ \mathtt{TO}\ 5}$ = ‘X’;
|

results in C1 ≡ ‘ABCX ’ (working length increased by 1)

By Rules 2 and 5:

|
| $C1_{5\ \mathtt{TO}\ 6}$ = ‘QQ’;
|

results in C1≡ ‘ABCDQQ’ (working length increased by 2)

|
| $C1_{7\ \mathtt{TO}\ 9}$ = ‘FGH’;
|

results in C1 ≡ ‘ABCD FGH’ (working length increased by 5)

|
| $C1_{6}$ = ’FGH’;
|

results in C1 ≡ ‘ABCD F’ (working length increased by 2)

8.4 BOOLEAN ASSIGNMENTS

The receiving data item is of a Boolean type.

1. The operand types are:

L-type	*R*-type
BOOLEAN	BOOLEAN

2. The logical value of the *R*-expression is transferred to the receiving data item.

Example:

If B is Boolean, then

|
| B = FALSE;
|

results in B ≡ FALSE

8.5 MULTIPLE ASSIGNMENTS

Several data items may be assigned to the same *R*-expression in the same statement. The general form of such a multiple assignment is as follows:

Symbolic form:

$L^1, L^2, ...L^n = R$;

1. The value of the *R*-expression is assigned to all L^1 ... L^n in turn.
2. Any *L*-type must be compatible with the *R*-type according to the rules stated in Sections 8.2 through 8.4.
3. No particular <u>order</u> of assignment is guaranteed.

Examples:

If M1 is a 2x2 matrix,
V1 is a 3-vector

```
|
| M1, V1 = 0;
|
```

results in M1 ≡ $\begin{bmatrix}0 & 0\\0 & 0\end{bmatrix}$, V1 ≡ $\begin{bmatrix}0\\0\\0\end{bmatrix}$

If C is a character,
I is an integer,

```
|
| C, I = 127.2;
|
```

results in C ≡ '1.2720000E+02', I ≡ 127

With the above data items,

```
|
| M1, C = 5;
|
```

is illegal because of data type mismatch between M1 and the *R*-expression.

The following example illustrates the importance of Rule 3:

If further I ≡ 2, then

|
| $V1_I$,I = I + l;
|

has an ambiguous result, depending on the order of assignment.

If I is assigned before $V1_I$,

$$\text{then } V1 \equiv \begin{bmatrix} ? \\ ? \\ 3 \end{bmatrix}, \text{ otherwise } V1 \equiv \begin{bmatrix} ? \\ 3 \\ ? \end{bmatrix}$$

(? indicates values unchanged by assignments)

In HAL/S, the receiving data item or items may be arrayed. This can produce varying effects depending on whether or not the *R*-expression also is arrayed (i.e. has arrayed operands).

See: Guide/20.3.

9.0 CONDITIONAL STATEMENTS AND BRANCHES

Section 9 is primarily concerned with the HAL/S conditional statement, by which other executable statements may be conditionally executed (or by which their execution may be conditionally avoided). Together with statement groups, which will be described in Section 10, they form a crucially important part of the HAL/S language.

The HAL/S language encourages programmers to avoid using GO TO statements to cause branches in execution. Their total elimination, however, is not desirable. This Section therefore also describes the HAL/S GO TO statement, and statement labels, which are their destinations. Statement labels are, in addition, needed for other constructs to be described in Section 10.

9.1 THE CONDITIONAL STATEMENT

In HAL/S, the simple version of the conditional statement is an "IF clause" containing an expression evaluable as either TRUE or FALSE, followed by a "true part" which is executed only if the IF clause is TRUE. The simple version may be augmented by a "false part" which is executed only if the IF clause is FALSE.

SIMPLE IF STATEMENT

The form of the simple version is:

```
|
| IF exp THEN statement;
|
```

1. *exp* is an expression which is evaluable as either TRUE or FALSE. It may be either a BOOLEAN expression or a relational expression (these are described in Section 9.2).
2. *statement* constitutes the true part of the conditional statement. It may be any executable statement, either simple or compound.
3. *statement* may possess a label but cannot be branched to from outside the IF statement.

   If *exp* is FALSE, execution proceeds to the next statement. If TRUE, *statement* is executed first.

Examples:

```
|
| IF B|C THEN X = 0;
| Y = 1;
```

X is set to 0 if either B or C or both is true: the flow diagram for these events is:

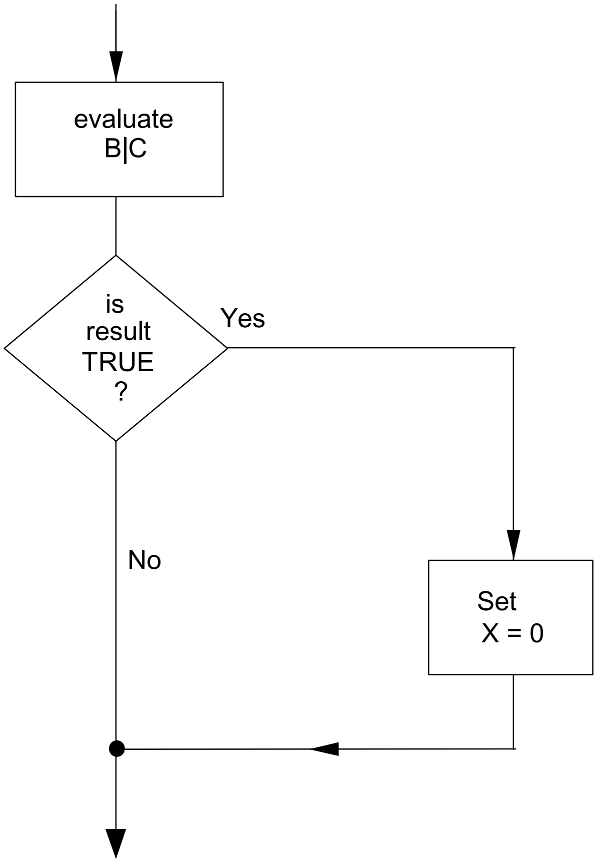


Figure 9-1

```
IF B|C THEN DO;
       X = X - l;
       Y = Y + l;
END;
```

The true part is a compound statement containing two assignments.

```
IF B THEN
   IF C THEN
      D = 0;
```

This shows that one can nest IF statements.

AUGMENTED IF STATEMENT

When augmented with a false part, the IF statement takes the form:

```
IF exp THEN statement;
ELSE else statement:
```

1. The form of the IF clause and true part are the same as in the simple conditional statement.
2. *else statement* constitutes the false part of the conditional statement. It may be any executable statement either simple or compound.
3. *else statement* may possess a label but cannot be branched to from outside the IF statement.
4. If *exp* is FALSE, execution proceeds to the next statement via *else statement*. If TRUE, it proceeds to the next statement via *statement*.
5. An ELSE clause may only be used if it is immediately preceded by an IF THEN statement (This eliminates the “DANGLING ELSE” problem found in some other higher level languages).

Examples:

```
IF B|C THEN X = 0;
ELSE X = 1;
```

X is set to 0 if B or C or both is true, otherwise X is set to 1. The flow diagram for these events is:

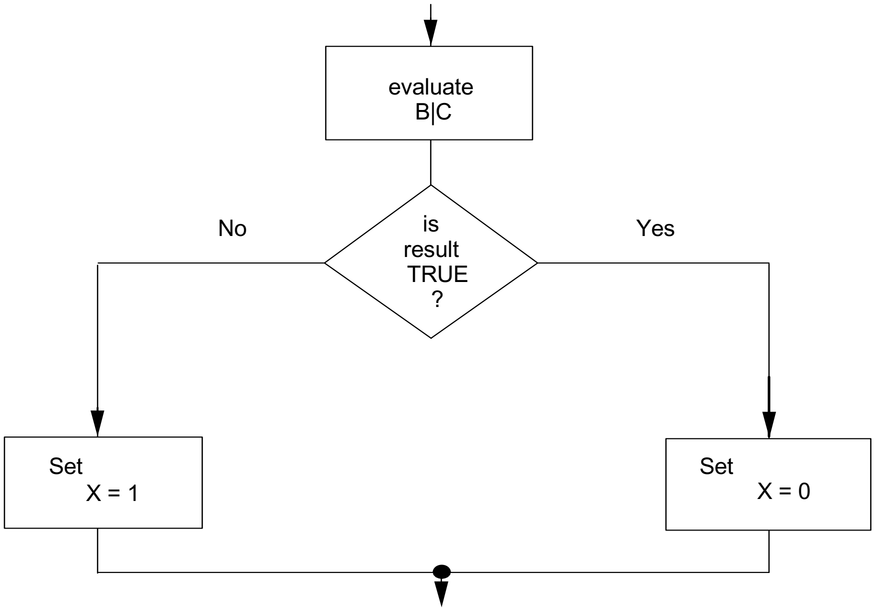


Figure 9-2

```
IF B|C THEN DO;
        X = 1;
        Y = 2;
 END;
 ELSE DO;
        X = 2;
        Y = 1;
 END;
```

Here, both true and false parts are compound statements each containing two assignments each.

```
IF B THEN X = 0;
ELSE IF C THEN X = 1;
Y = 2;
```

This is legal: the false part of a conditional statement may itself be another conditional statement: the flow diagram for these events is:

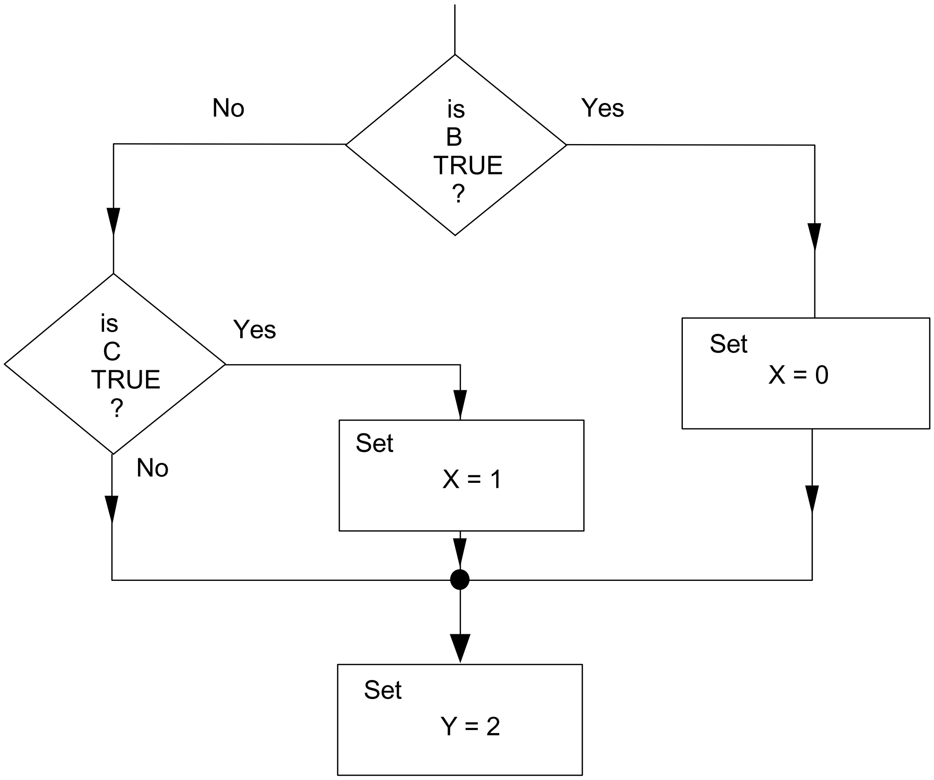


Figure 9-3

```
IF B THEN
  IF C THEN
     X = 0;
  ELSE
     X = 1;
ELSE X = 2;
```

Illegal because the last ELSE clause is not immediately preceded by an "IF exp THEN statement" statement. If the intent is to make the last ELSE effective on the first "IF exp THEN" clause, one can use the DO-END grouping in the following manner:

```
IF B THEN
  DO;
    IF C THEN
       X = 0;
    ELSE
       X = 1;
  END
ELSE X = 2;
```

This is legal because the DO-END group collects any number of statements within its scope and makes them look like a single statement. The flow diagram for these events is:

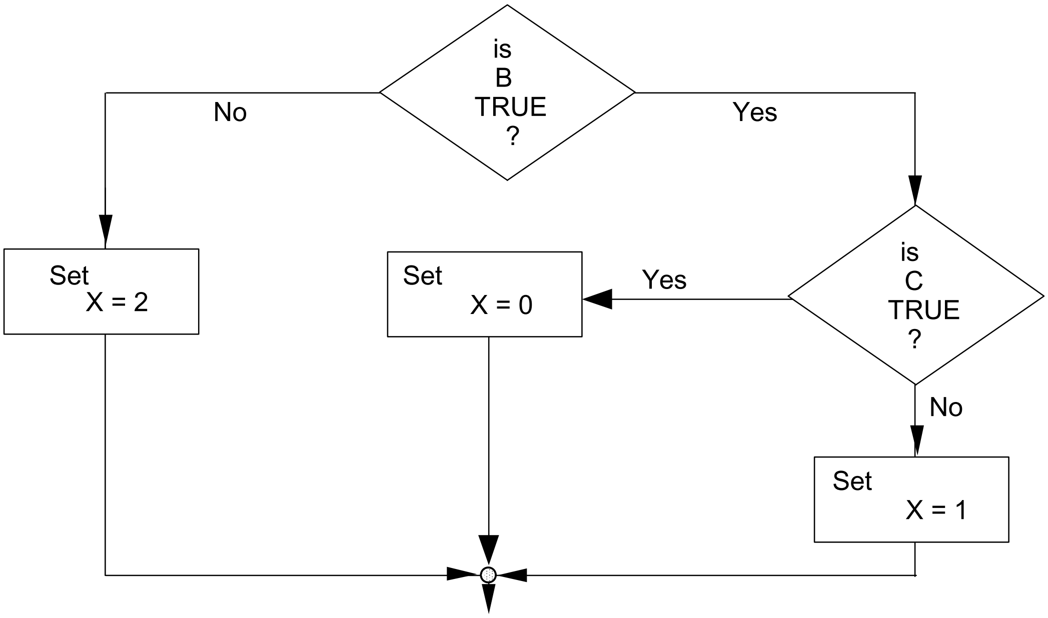


Figure 9-4

9.2 RELATIONAL EXPRESSIONS

As was stated in Section 9.1, there are two valid forms of expression in an IF clause, BOOLEAN, and relational. BOOLEAN expressions were described in Section 7, relational expressions only appear in a limited number of HAL/S constructs, among them conditional statements, and are now described.

The simplest form of a relational expression is merely a comparison between two like quantities. The result is either TRUE or FALSE. More complex forms of relational expressions result from combining comparisons with the BOOLEAN operators &, |, and ¬.

COMPARATIVE OPERATIONS

HAL/S recognizes the following comparative operators:

SYMBOL	PURPOSE	CLASS
>	greater than	I
<	less than	
<=	less than or equals	
NOT > ¬ >	not greater than	
>=	greater than or equals	
NOT < ¬ <	not less than	
=	equals	II
NOT = ¬ =	not equal	

The operands of comparative operations may, in general, be expressions of any of the types described in Section 7 Depending on the type of operand, the operators may be restricted to Class II only, or may be either Class I or Class II.

- CLASS II ONLY

Symbolic form: *L* $\begin{matrix} = \\ \text{NOT} = \\ \neg = \end{matrix}$ *R*

1. Legal combinations of data types are indicated by the following table:

L-type	*R*-type
VECTOR	VECTOR
MATRIX	MATRIX
BOOLEAN	BOOLEAN

2. Comparison of vector and matrix operands implies element-by-element comparison.
3. The operands in a vector comparison must be the same length.
4. The operands in a matrix comparison must have the same row and column dimensions.

Examples:

If V, V1 are 3-vectors with

$$V \equiv \begin{bmatrix} -1 \\ -1 \\ 2 \end{bmatrix}, \quad V1 \equiv \begin{bmatrix} 1 \\ 1 \\ -2 \end{bmatrix}$$

then `V = V1` is FALSE,

`V - V1 = 2V` is TRUE.

If further V2 is a 2-vector with $V2 \equiv \begin{bmatrix} 1 \\ 1 \end{bmatrix}$

then V1 = V2 is illegal because of length mismatch,

but $\texttt{V1}_{\texttt{1 TO 2}}$ `= V2` is TRUE.

- CLASS I AND CLASS II

$$\text{Symbolic form: } L \left\{ \begin{matrix} > \\ < \\ >= \\ <= \\ \text{NOT}> \\ \neg > \\ \text{NOT}< \\ \neg < \\ = \\ \text{NOT}= \\ \neg = \end{matrix} \right\} R$$

1. Legal combinations of data types are indicated by the following table:

$$\begin{array}{rl} \text{L-type} & \text{R-type} \\ \left. \begin{matrix} \texttt{INTEGER} \\ \texttt{SCALAR} \\ \texttt{CHARACTER} \end{matrix} \right\} & \left\{ \begin{matrix} \texttt{INTEGER} \\ \texttt{SCALAR} \\ \texttt{CHARACTER} \end{matrix} \right. \end{array}$$

2. In a mixed scalar-integer operation, the integer operand is converted to scalar before the comparison takes place.
3. For character string comparisons, the standard dictionary collating sequence[†] is used. For strings of equal length, a string is greater than another string if at the first miscompare (going left to right) the character string in the first string is greater than the character in the second string. If the lengths are unequal, the shorter one is padded with blanks on the right, then the comparison used for strings of equal length is used.

[†] The collating sequence is implementation dependent. See appropriate User's manual.

Examples:

If I is an integer with I ≡ 5
then `I = 5` is TRUE
`I < 4` is FALSE
`I >= 5` is TRUE

If C is a character data item with C ≡ ‘ABC’
then `C = ‘ABC’` is TRUE
`C = ’BCD’` is FALSE
`C > ‘AB’` is TRUE
`C < ‘ABCD’` is TRUE
`C > ‘ABB’` is TRUE

NOTE ON PRECISION CONVERSION

Precision conversion may be required where both operands of the comparison are arithmetic.

Where the types of the operands are the same, but the precisions differ, the single precision operand is converted to double precision before the comparison is made.

In a mixed integer/scalar comparison the result of the integer-to-scalar conversion is generated with the same precision as the scalar operand.

COMBINING COMPARATIVE OPERATIONS

Comparative operations may be combined as if they were BOOLEAN operands, using the rules for Boolean operations described in Section 7. It is important to note however, that comparative operations are not BOOLEAN operands in the sense that they can be mixed with actual BOOLEAN data items.

- Boolean expressions may contain no comparative operations.
- Relational expressions may contain no Boolean operands.

Examples:

If V1, V2 are 3-vectors with

$$V1 \equiv \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \quad V2 \equiv \begin{bmatrix} 3 \\ 2 \\ 1 \end{bmatrix}$$

and C is character with C ≡ ‘ABC’
then
$\mathtt{V1 = V2|C_1 = ‘A’}$ is TRUE
$\mathtt{V1 = V2\ \&\ C_1 = ‘A’}$ is FALSE

If B is Boolean then
`B|V1 = V2` is illegal
but
`B = ON|V1 = V2` is legal

PRECEDENCE

The following table shows the precedence of operations involved in a relational expression:

Symbol	Precedence	Purpose
	FIRST	
	1	operations involving operands of comparisons
> < <= NOT>,¬> >= > NOT<,¬< = NOT=,¬	2	comparative operations
¬, NOT	3*	logical operations on comparisons
&, AND	4	
|, OR	5	

* Any operand of this operator must always be parenthesized.

Example:

In the following expression, the numbered pointers show the order of execution of operations:

```
IF S1 + S2 = 0 | ¬ (S3 > 0) & ¬ (S4< 0 | S5 > 0) THEN
      ↑    ↑   ↑ ↑     ↑    ↑ ↑    ↑   ↑    ↑
      ①    ②   ⑩ ④     ③    ⑨ ⑧    ⑤   ⑦    ⑥
```

Section 9.2 ends with some more examples designed to clarify the foregoing.

Examples:

Let V be a 3-vector with V ≡ $\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$

```
|IF V = 1 & V = 2 THEN V = 0;
|S   1       2          3
|IF V > 0 | V < 0 THEN V = 0;
|S   3       2
```

The first statement will cause V_3 to be set to zero since both comparisons are TRUE. Then

$$V \equiv \begin{bmatrix} 1 \\ 2 \\ 0 \end{bmatrix}$$

In the second statement, neither comparison in the relational expression is true. Hence, the "true part" is not executed and finally

$$V \equiv \begin{bmatrix} 1 \\ 2 \\ 0 \end{bmatrix}$$

as before.

Relational expressions may be arrayed, additional rules being required to determine if the result is TRUE or FALSE.

See: Guide/20.5.

9.3 LABELS AND BRANCHES

In HAL/S, there are two entities involved in the branching operation: a GO TO statement, which, when executed causes the branch; and a "statement label" which is the destination of such a branch. HAL/S also uses statement labels for other purposes, which will become clear in Section 10.

LABELS

Labels are names chosen by the programmer and attached to statements. More than one label may be attached to a statement. The way of attaching a single label to a statement is as follows:

```
|
| label: statement;
|
```

1. *statement* is any executable statement or statement group (see Section 10).
2. *label* is a user-defined identifier name (see Section 2.2).

Examples:

```
|
| ONE: X = X +1;
| TWO: Y = 0;
|
| IF X = 0 THEN ONE: Y = 0;
| IF X = 0 THEN X = 1;
| ELSE TWO: X = 3:
|
| THREE: IF X = 0 THEN Y = 1;
|
```

If more than one label is required, then they follow each other in sequence.

```
|
| ONE: TWO: THREE: X = X +1;
|
```

GO TO STATEMENT

The GO TO statement specifies the label to which execution branches: it takes the form:

```
|
| GO TO label;
|
```

1. *label* is a label attached to some statement to which execution is to branch.

Examples:

```
|
| GO TO ONE;
|
```

The GO TO statement itself may be labeled:

```
|
| TWO: GO TO THREE;
|
```

It is important to note that HAL/S places relatively severe restrictions on the placement of GO TO statements and where they may cause execution to branch to. Section 1.3 described this on the abstract level, and Section 10 further discusses it in connection with statement groups.

ELIMINATING GO TO STATEMENTS

The Guide has stressed throughout that, according to structured programming principles, GO TO statements are inherently undesirable because they tend to disguise the program's flow of execution.

It will be found that HAL/S contains a sufficient number of other constructs to allow GO TO statements to be substantially eliminated from a program. Following is an example showing the elimination of GO TO statements.

Example:

```
          IF X > 1.5 THEN GO TO ALPHA;
          IF X < 1.5 THEN GO TO BETA;
          Y = Y + l;
          GO TO GAMMA;
ALPHA:    X = X - 0.05;
          GO TO GAMMA;
BETA:     X = X + 0.05;
GAMMA:
.
.
.
.
```

This example is programmed in HAL/S in the simplest way (possibly having been translated from FORTRAN or an assembly language). The profusion of GO TO statements disguises the simple flow of execution, which is interpreted by the following flow diagram:

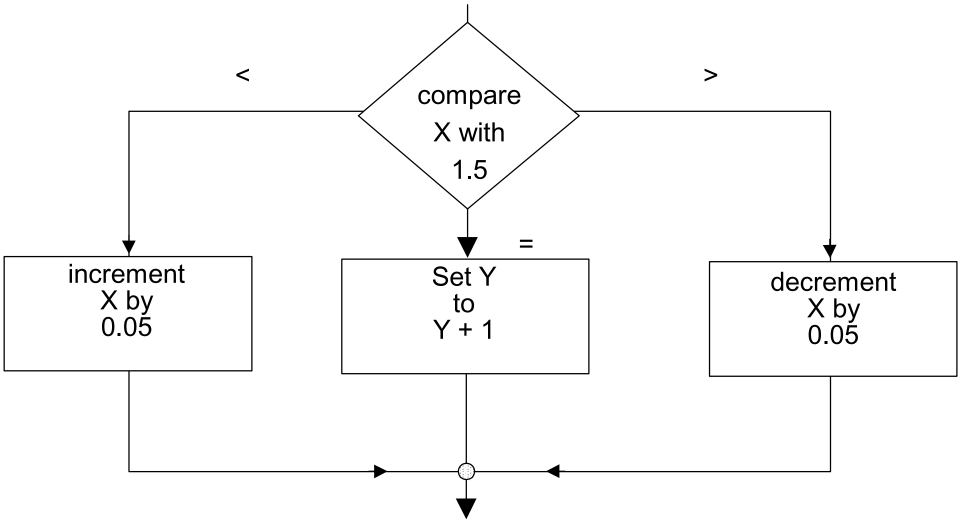


Figure 9-5

The same algorithm is more clearly programmed as follows:

```
IF X   >  1.5 THEN
  X   =  X - 0.05;
ELSE
IF X   <  1.5 THEN
  X   =  X + 0.05;
ELSE
  Y   =  Y + 1;
 .
 .
 .
```

10.0 STATEMENT GROUPS

Section 1.3 of the Guide introduced, on an abstract level, the idea of “statement groups’’, which could be treated as if they were simple executable statements, and could be nested one inside the other. The power of such a facility can be seen, for example, when it is used in conjunction with the conditional statement: (this is demonstrated later in Section 10.1).

There is, in fact, a second, equally important reason for grouping statements in HAL/S: the execution of such groups can be controlled in a variety of ways. If no explicit specification is made, the sequence of statements is executed once only. By explicit specification:

- the sequence may be repetitively executed until some condition is satisfied;
- a single executable statement (or nest statement group) of the group, selectable at execution time, may be executed.

Section 10 explains in detail how statements are grouped, and how execution control of the groups is specified.

10.1 DELIMITING STATEMENT GROUPS

In HAL/S, groups of statements are said to be “well-bracketed”: they are delimited explicitly by opening and closing statements which are themselves considered executable.

THE DO STATEMENT

Every statement group is opened with a “DO” statement which is also used to specify control of execution within the group. It takes the generic form:

```
|
| DO control;
|
```

1. *control* is a construct to be described. It specifies the manner in which the sequence of statements is to be executed.
2. *control* is optional. If it is absent, the sequence of statements within the group is executed in order once only.
3. The DO statement is executable in that it may be labeled according to the Rules of Section 9.

The particular instances of DO statements will be explained in Section 10.2.

THE END STATEMENT

Every statement group is closed with an END statement:

```
|
| END label;
|
```

1. The END statement is executable in that it may be labeled according to the Rules of Section 9.
2. *label* is optional: if present, the opening DO statement of the group must be labeled with *label*.

The label specification in an END statement is never functionally necessary in HAL/S. However, it should be regarded as good programming practice because it facilitates cross-checking by the compiler.

Examples:

Two instances of statement groups are shown below. Even though details of execution control have not yet been explained, the form of the construct should be clear.

```
|     DO WHILE I> 0;  }  opening Do statement
|           I = I -1; ⎫
|           A = 0;    ⎬ group of statements
|S           I        ⎭
|     END             }  closing END statement

| FIX: DO FOR I = 1,25,16,2;
|         A =  -A  ;  }  one statement in group
|S         I     I
|     END FIX;        }  label specification in the END
                           matches label of DO
```

The following examples show the importance of being able to group statements together for use in conjunction with a conditional statement.

```
|
| IF S = 0 THEN I = 2;
| C = 'RESET VALUE OF I TO ' || I;
| .
| .
| .
|
```

It is required to conditionally execute <u>both</u> assignments: one solution is -

```
IF S ¬ = 0 THEN GO TO NOSET;
I = 2;
C = ‘RESET VALUE OF I TO ’ || I;
NOSET:
.
.
.
```

This solution is error prone and not in accordance with structured programming concepts: a better solution is -

```
IF S = 0 THEN DO;
   I = 2;
   C = ‘RESET VALUE OF I TO ’ || I;
END;
```

The whole of the group enclosed by DO...END is subject to conditional execution.

10.2 REPETITIVE EXECUTION OF STATEMENT GROUPS

The sequence of statements in a group can be executed repetitively until some condition is satisfied. In this section, two basic forms of DO statement causing repetitive execution are described:

- The DO WHILE statement, in which execution is repeated while a relational or Boolean expression remains true in value;
- The DO FOR statement, in which the sequence is executed once for each of a set of assigned values of a “control variable”.

THE DO WHILE STATEMENT

The form of the DO WHILE statement is:

```
|
| DO WHILE condition;
|
```

1. *condition* is any relational or BOOLEAN expression. It is evaluated prior to each cycle of execution of the statement sequence in the group.
2. The next cycle of execution of the group proceeds if the value of *condition* is TRUE.
3. If the value of *condition* is FALSE, the stopping condition is satisfied. Execution proceeds to the statement following the END statement of the group.

Examples:

```
|
| I = 9;
| DO WHILE I > 0;
|    I = I - 2;
| END;
|
```

Here the group is executed 5 times, after which the value of I is -1. In flow diagram form, the sequence of events is:

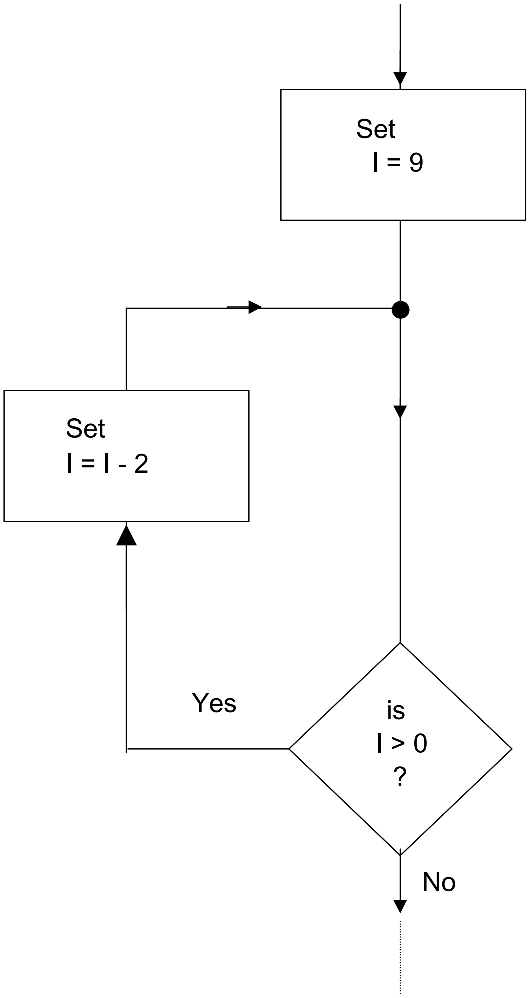


Figure 10-1

It is possible for a group never to be executed:

```
DO WHILE FALSE;
   I = I - 2;
END;
```

It is also possible for a group to be executed forever:

```
|
| I = I - 0;
| DO WHILE TRUE
|   I = I - 2;
| END;
| .
| .
| .
|
```

Normally in this case, the programmer would insert statements in the group removing this possibility:

```
|
| I = 9;
| DO WHILE TRUE
|   I = I - 2;
|   IF I < 0 THEN GO TO ALL_DONE;
| END;
| .
| .
| .
|
```

If the keyword UNTIL is substituted for the keyword WHILE, then the group is always executed at least once. After the first cycle, the relational or Boolean expression is evaluated at the beginning of each cycle as in the DO WHILE, except that the logic of the test is inverted: cycles of execution continue until the result of the expression becomes TRUE.

Example:

```
|
| I = 0;
| DO UNTIL I <= 0;
|   I = I - 1;
| END;
| .
| .
| .
|
```

The group is executed once, and the final value of I is -1.

THE DO FOR STATEMENT

The most widely used form of the DO FOR statement is:

```
|
| DO FOR var = initial TO final BY increment;
|
```

1. *var* is an unarrayed INTEGER or SCALAR data item (it may be subscripted if required). It is called the "control variable" of the DO FOR statement.
2. *initial, final* and *increment* are integer or scalar expressions:

   initial is the initial value assigned to *var*.

   increment is the amount by which *var* is incremented on each cycle of execution of the sequence of statements in the group.

   final is the value against which *var* is tested at the start of every cycle to determine if the stopping condition is satisfied.

   All three expressions are evaluated <u>once</u> prior to the first cycle of execution.
3. The stopping condition is met when the value of *var* lies outside the range bounded by *initial* and *final*.
4. *increment* may be either positive or <u>negative</u>. The phrase

   BY *increment*

   is optional. If omitted, the implied increment is +1.

Examples:

```
|
| DO FOR I = 1 TO 10;
|    X = I;
|S   I
| END;
|
```

Here the group is executed 10 times. I is initially 1, and increments each time until 10 is reached. At the end of execution of the group, the value of I is 11. In flow diagram form, the sequence of events is:

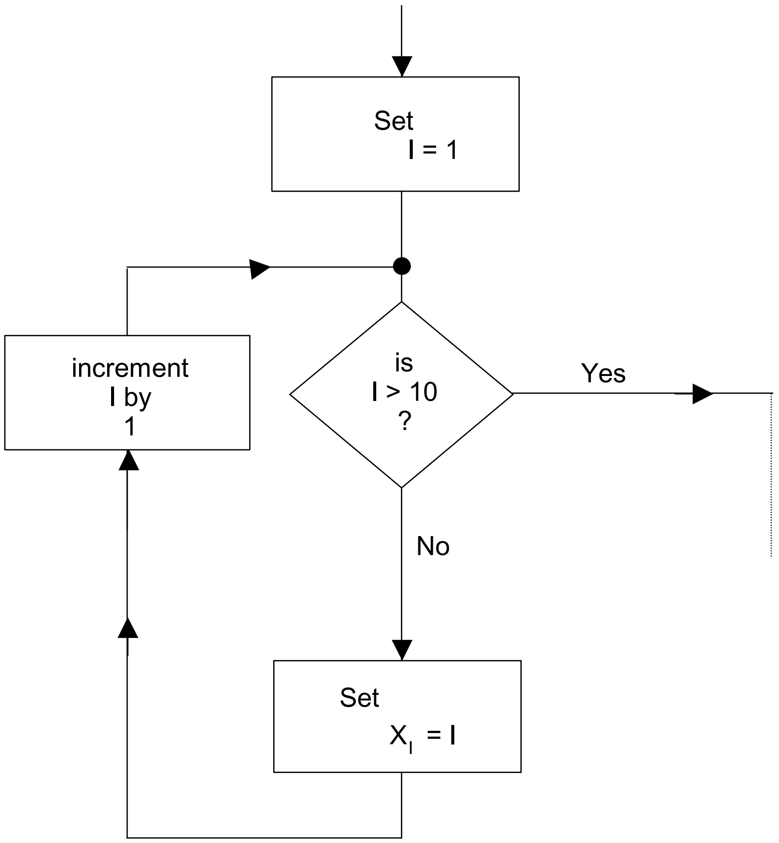


Figure 10-2

Example:

```
|
| I = 7;
| DO FOR I = I + 5 TO I - 3 BY -2;
|   X = X + I;
| END;
|
```

This example demonstrates some of the subtleties of the DO FOR statement. The initial and final values are precomputed as 12 and 4 respectively. Then I is reused as the control variable: the group is executed 5 times, and after the last cycle of execution, I retains the value 2.

Care must be taken if the control variable is integer and the range expressions are scalar: rounding occurs during assignment of values in such cases.

See: Spec. /7.6.5.

This DO FOR statement may possess a WHILE or UNTIL clause which furnishes a supplementary stopping condition.

See: Spec./7.6.5.

The DO FOR statement has a second form which is used if the values of the control variable do not form a regular progression:

|
| DO FOR *var* = *exp*^{1},*exp*^{2},...*exp*^{n};
|

1. *var* is the control variable as before.
2. Each *exp* is an integer or scalar expression. Values of the *exp*'s are assigned to *var* in turn prior to the execution of each cycle, on a left-to-right basis.
3. Each *exp* is evaluated immediately prior to the cycle of execution in which it will be used.

Examples:

```
|
|  DO FOR I = 17,5,12,4;
|    X = I;
|S      I
|  END;
|
```

Here, I takes the successive values 17, 5, 12, and 4. After the end of the last cycle, the value of I remains at 4.

```
|
| I = 7;
| DO FOR I = I + 5, I + 3, I + 1, I - 1, I - 3;
|    X = X + I;
| END;
|
```

Superficially, this example looks like a different way of expressing the second example for the first form of DO FOR statement:

```
I = 7;
DO FOR I = I + 5, TO I - 3 BY -2;
  X = X + I;
END;
```

However, the successive values of I in the new form (by Rule 3) are:

12, 15, 16, 15, 12

as opposed to

12, 10, 8, 6, 4

in the old form.

Rounding also occurs if the control variable is integer and any of the control expressions are scalar.

See: Spec./7.6.4.

As before, the DO FOR statement may possess a WHILE or UNTIL clause which furnishes a supplementary stopping condition.

See: Spec./7.6.4**.**

10.3 SELECTIVE EXECUTION OF STATEMENT GROUPS

One statement of a group may be selected for execution by means of the DO CASE statement. The form of the DO CASE statement is:

```
DO CASE exp;
```

1. *exp* is an integer or scalar expression.
2. If its value is k (after rounding if necessary), then the k^{th} statement of the group is selected for execution.
3. A run time error results if k ≤ 0 or k is greater than the number of statements in the group.

The flexibility of a DO CASE statement lies in that the selected statement may be a compound statement (i.e. it may be itself a statement group).

```
Example:
   I = 3;
   DO CASE I;
       X = 4;      case 1
       X = 3;      case 2
       DO
         X = 7;⎫
         Y = 3;⎬ case 3
               ⎭
       END;
       X = 1;      case 4
       X = 0;      case 5
   END;
```

Execution results in the third statement being scheduled for execution, and the following values being set:

X ≡ 7, Y ≡ 3

> An ELSE clause may be added to the DO CASE statement which is executed if the value of the case variable is outside the legal range for the statement group.
>
> See: Spec./7.6.2.

10.4 BRANCHING IN STATEMENT GROUPS

Execution may branch out of any statement group via a GO TO statement. In those cases where the group is being repetitively executed, execution obviously ceases before the stopping criterion is satisfied. Because GO TO statements are viewed unfavorably from the standpoint of structured programming, HAL/S possesses two statements expressly for executing controlled branches in statement groups.

- The EXIT statement is, in effect, a controlled branch out of a statement group.
- The REPEAT statement applies only to statement groups executed repetitively, and is a controlled branch back to the beginning of the group.

THE EXIT STATEMENT

The simplest form of the EXIT statement is:

```
|
| EXIT;
|
```

1. Its execution causes an immediate branch out of the innermost statement group in which it is enclosed.
2. Execution is directed to the first statement following the END of the group branched out of.

Examples:

```
|
| DO
|    X =1;
|    Y =2;
|    IF Z = 3 THEN EXIT;
|    Z = 4;        |
| END;             |
| X = X + 1;  ←────┘
|
```

Arrow shows branch in execution if Z ≡ 3

```
|
|  DO WHILE X > 0;
|     X = X -1;
|     IF X > 2 THEN DO;
|         IF Y = 3 THEN EXIT;
|         Y = Y + 1;       |
|     END;                 |
|  END;         ←──────────┘
|
```

Arrow shows branch in execution if Y ≡ 3: execution branches to the end, but not out of DO WHILE group.

There exists a second form of the EXIT statement to allow branches out of other than the innermost statement group:

```
|
| EXIT  label;
|
```

1. Its execution causes a branch out of the enclosing statement group whose DO statement possesses the label *label*.
2. Execution is directed to the first statement after the END of the group branched out of.

Example:

```
|
| ONE: DO WHILE X > 0;
|         X = X -1;
|         DO FOR I  = 1 TO 10;
|            A = A + X ;
|S            I   I
|            IF X = 1 THEN EXIT ONE;
|            IF X = 0 THEN EXIT;   |
|         END;                |       |
|       END;   <──────────────┘       |
|       X =0;  <──────────────────────┘
```

The first EXIT statement causes a branch out of the outer group rather than the inner, by virtue of its label.

THE REPEAT STATEMENT

The simplest form of the REPEAT statement is:

```
|
| REPEAT;
|
```

1. It must be enclosed in a DO FOR or DO WHILE group.
2. Its execution causes an immediate branch to the beginning of the innermost enclosing DO FOR or DO WHILE group.
3. The next cycle of execution of the group then starts (unless of course the stopping condition is satisfied).

Examples:

```
|
|  DO WHILE X > 0; ←──────
|      X = X -1;          \
|      IF X = 4 THEN DO;   \
|           Y = Y + X;      \
|          IF Y = 1 THEN REPEAT;
|      END;
| END;
|
```

If Y ≡ 1 then a branch back to the beginning of the DO WHILE is made. Note that although the DO WHILE is not the innermost group, it is the innermost repetitive group.

```
|
|  X = 4;
|  DO WHILE X > 1; ←───
|      X = X - 1;      \
|      IF X = 1 THEN REPEAT;
|      Y  = X;
|S       X
|  END;
```

When X is decremented to 1 the REPEAT branch is executed: a new cycle of execution does not begin, however, because the initial test shows that the stopping condition is satisfied.

As with the EXIT statement, there exists a second form of the REPEAT statement allowing branches back to the beginning of other than the innermost DO WHILE or DO FOR group:

```
|
| REPEAT  label;
|
```

1. Its execution causes an immediate branch to the beginning of the enclosing DO FOR or DO WHILE group whose DO statement possesses the label *label*.
2. The next cycle of execution of the group then starts (unless the stopping condition is satisfied).

Example:

```
|
|   ONE: DO FOR I = 1 TO 10; ←──────┐
|          J = I;                   │
|          DO WHILE J > 0; ←─────┐  │
|             J = J - 1;         │  │
|             X    = X    + J;   │  │
|S             J      J          │  │
|           IF X = 25 THEN REPEAT;  │
|S              J                   │
|           IF X = 0 THEN REPEAT ONE;
|S             J
|          END;
|         END;
|         Z = 0;
```

The second REPEAT statement restarts the outer DO FOR group rather than the inner DO WHILE by virtue of its label.

This page intentionally left blank

11.0 PROCEDURES AND FUNCTIONS

Section 1.2 of the Guide introduced the block structure of HAL/S programs on the abstract level. To summarize, any program can contain nested procedure and function blocks, which are two levels of "subroutines" characterized by the sequence:

invocation → execution → return to caller

The invocation of procedures and functions is governed by well-defined name scoping rules.

This section explains how, in practice, procedure and function blocks are defined in HAL/S, and describes how they are invoked and returned from.

11.1 INTRODUCTION

A procedure is a subroutine block invoked by a CALL statement. It may have two kinds of parameters:

- INPUT PARAMETERS - by which values may be passed into a procedure only.
- ASSIGN PARAMETERS - by which values may be passed into and out of a procedure.

A function is a subroutine block invoked by the appearance of its name in an expression. It returns a value and therefore has a defined HAL/S data type. It may possess input parameters only.

RELATIVE POSITION OF BLOCK DEFINITIONS

Section 1.2 described the scoping rules which determine the regions of a program where any given procedure or function block may be invoked.

An important consequence of these rules is that a procedure invocation may either follow or precede its block definition. However, for other reasons, the invocation of a function block should normally always follow its block definition.

A number of rules restrict the kind of function which may be invoked preceding its block definition.

See: Spec./4.6 & 6.4.

11.2 BLOCK DEFINITIONS

Procedure and function block definitions have forms very similar to the form of a program block, which was described in Section 3. The first statement is one defining the name and type of block, and listing its parameters. The last statement is a statement closing the block.

PROCEDURE OPENING

The statement opening a procedure block takes the form:

label: PROCEDURE (*i^1*,*i^2*,...) ASSIGN(α^1,α^2,...);

1. *label* is any legal identifier name, and constitutes the name of the procedure.
2. i^1ʼi^2,... are legal identifier names defining input parameters. If the entire parenthesized list is omitted, then the procedure has no input parameters.
3. α^1,α^2, ... are legal identifier names defining assign parameters. If the entire parenthesized list and the keyword ASSIGN are omitted, then the procedure has no assign parameters.

FUNCTION OPENING

The statement opening a function block takes the form:

label: FUNCTION (i^1 i^2,...) *attributes*;

1. *label* is any legal identifier name, and constitutes the name of the function.
2. i^1 i^2,... are legal identifier names defining input parameters. If the entire parenthesized list is omitted, then the procedure has no input parameters.
3. *attributes* defines the type of attributes and, where applicable, precision and size. The form of specification is the same as used in data declarations (see Section 4.2). If no attributes are supplied, the function is assumed to be single precision scalar.
   Note that function specification may include ARRAYs.

BLOCK CLOSING

Both procedure and function blocks are closed with the statement:

CLOSE *label;*

1. The identifier *label* is optional.
2. If supplied, it must be the name of the procedure or function block.

Examples:

```
ONE: PROCEDURE;
▨  }──────── procedure body
CLOSE ONE;

TWO: PROCEDURE ASSIGN (ARG1):
                           ↑
▨               single assign parameter -
                may be used to return
CLOSE TWO;       values from procedure

THREE: FUNCTION MATRIX(4,4) DOUBLE;
▨
CLOSE THREE;

FOUR: FUNCTION (ARG1,ARG2) BOOLEAN:
                       ↑
▨                two input parameters -
                 for passing values into
CLOSE:           function only
```

11.3 DECLARATION OF PARAMETERS AND LOCAL DATA

Procedures and functions commonly require the use of locally-defined data. As with program-level data, all data names must be declared before use by means of declaration statements. In addition, all input and assign parameters must appear in local declaration statements.

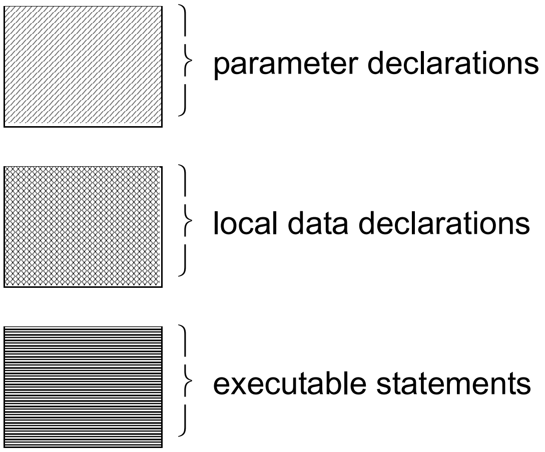
Data and parameter declarations must be placed after the procedure or function opening statement, and before the first executable statement. It is good practice, and mandatory in some implementations[12], to place parameter declarations before local data declarations. The forms of local data and parameter declarations are identical, and are as described in Section 4.

[12] See the User's Manual for any given implementation.

Examples:

General positioning -

```
ONE: PROCEDURE(ARG1) ASSIGN(ARG2);
```



```
CLOSE ONE;
```

Particular instance -

```
ONE: PROCEDURE(ARG1) ASSIGN(ARG2);

   DECLARE ARG1 MATRIX(4,4);                } parameters
   DECLARE ARG2 ARRAY(100) SCALAR DOUBLE;
   DECLARE TEMP MATRIX(4,4);                } local data
     .
     .
     .
      .
CLOSE ONE;
```

CHARACTER PARAMETER DECLARATIONS

Parameters of character type may be declared to possess an indefinite maximum length. By this means problems of truncation of character data during argument passage can be avoided.

The basic form of declaration is:

```
   DECLARE name CHARACTER(*);
```

1. The asterisk denotes an indefinite maximum length.

Example:

```
ONE: PROCEDURE(A);
     DECLARE A CHARACTER(*);
       .
       .
       .
```

11.4 FUNCTION INVOCATIONS

A function is invoked by the appearance of its name as an operand in an expression. If the function is defined with input parameters, a list of arguments to be passed must follow the appearance of the name. The precise form of invocation is:

label (i^1, i^2, ...)

1. *label* is the defined name of the function.
2. i^1, i^2 ,... is a list of arguments, which must correspond in number with the parameters of the function invoked. Each argument is a HAL/S expression.
3. If the function has no parameters, then the entire parenthesized argument list must be absent.

The transmission of the argument list during function invocation may be viewed as the assignment of the value of each expression in turn to its corresponding input parameter (although in any given implementation this may not actually be the mechanism of transmittal). A set of rules governing type and precision conversion, and dimension matching similar to the assignment rules of Section 8 are applicable. These are classified below according to parameter type.

MATRIX PARAMETER

1. The corresponding argument must be of matrix type.
2. The number of rows and columns of the argument must be the same as those of the parameter.
3. Precision conversion is allowed.

VECTOR PARAMETER

1. The corresponding argument must be of vector type.
2. The length of the vector argument must be the same as that of the parameter.
3. Precision conversion is allowed.

INTEGER/SCALAR PARAMETER

1. The following table gives the legal argument types:

parameter	argument
INTEGER SCALAR	INTEGER SCALAR

2. Conversion of the argument takes place where necessary. Scalar-to-integer conversion implies rounding of the value of the expression.
3. Precision conversion takes place when necessary and is applied at the same times as type conversion.

CHARACTER PARAMETER

1. The allowable argument types are given by the following table:

parameter	argument
CHARACTER	CHARACTER INTEGER SCALAR

2. Rules for the conversion of integer or scalar values to character type are given in Appendix A.

Generally, the working length of the parameter becomes equal to the length of the expression (after conversion, where applicable). However, if this would cause the declared maximum length of the parameter to be exceeded, truncation of the excess from the right takes place.

BOOLEAN PARAMETER

1. The corresponding argument must be of Boolean type.

The following examples show a selection of both legal and illegal function invocations. Examples:

Suppose the following functions are defined:

```
ONE: FUNCTION INTEGER;
▨
CLOSE;

TWO: FUNCTION(A,B) MATRIX(4,4) DOUBLE;
    DECLARE A MATIRX(4,4);
    DECLARE B SCALAR;
▨
CLOSE;
```

Let also the following data be declared:

```
DECLARE M1 MATRIX(4,4),
        M2 MATRIX(4,4) DOUBLE,
        M3 MATRIX(3,3),
        S SCALAR,
        I INTEGER;
```

Invocations of the above procedure are illustrated in the following constructs:

```
   S=S + ONE;

   S=S + M       ;
S          1,ONE
              ↑—   Note: subscripts may be integer
                   expressions of any kind.
   M2 = TWO(M2,S) + M2;
             ↑— M2 is converted to single precision
                during transmission.
   M2 = TWO(M2, I) ;
                ↑— I is converted to scalar type
                   during transmission.
```

The following are illegal invocations:

```
M2=TWO(M3,1.5)
          ↑__ row and column dimensions of M3 do
               not match those of parameter A.

M2=TWO(M1,'ARGUMENT'|| I);
```

transmission of character type argument
to scalar parameter B incurs an illegal
type conversion.

Arguments may possess arrayness. The effects of this depend on whether or not the corresponding parameter is declared to be an array.

See: Guide/ 20.5.

11.5 PROCEDURE INVOCATIONS

A procedure is invoked by the use of a CALL statement, which may, in the case of a procedure with parameters, also specify the arguments to be passed. The precise form of invocation is:

CALL *label* (i^1,i^2,...) ASSIGN(a^1,a^2,...);

1. *label* is the defined name of the procedure.
2. i^1,i^2, ... is a list of input arguments which must correspond in number with the input parameters of the procedure invoked. Each input argument is a HAL/S expression.
3. If the procedure has no input parameters, then the entire parenthesized argument list must be absent.
4. a^1,a^2, ... is a list of assign arguments which must correspond in number with the assign parameters of the procedure invoked. Each argument must be a HAL/S data item.†
5. If the procedure has no assign parameters, then the entire parenthesized list of assign arguments, and the ASSIGN keyword, must be absent.

† Or an assign parameter, if the invocation is nested within a procedure block.

The transmission of the input argument list during procedure invocation is identical in nature to function argument list transmission. The related rules are given in Section 11.4.

The transmission of the assign argument list follows stricter rules since values are passed both into and out of a procedure by this mechanism

ASSIGN ARGUMENTS

1. An assign argument must be a declared HAL/S data item. †
2. An assign argument must match the corresponding assign parameter in type and precision.
3. A matrix or vector argument must match the corresponding parameter in dimension.
4. Only matrix and vector arguments may be subscripted. Such subscripting must reduce the argument to scalar type by specifying one element only.

† Or an assign parameter, if the invocation is nested inside a procedure block.

The following examples show a selection of both legal and illegal procedure invocations.

Examples:

Suppose the following procedures are defined:

```
ONE: PROCEDURE;

CLOSE;

TWO: PROCEDURE(A, B) ASSIGN(C);
DECLARE A MATRIX(3,3);
DECLARE B INTEGER;
DECLARE C INTEGER;

CLOSE;
```

Let also the following data be declared:

```
DECLARE M1 MATRIX(3,3),
        M2 MATRIX(3,3) DOUBLE,
        M3 MATRIX(4,4),
        S SCALAR,
        I INTEGER,
        ID INTEGER DOUBLE;
```

Invocations of the above procedure are illustrated in the following constructs:

```
    CALL ONE;
    CALL ONE(I);
             ↑__ illegal: ONE possesses no parameters.
E              T
    CALL TWO(M2 ,S+1)ASSIGN(I);
             ↑    ↑         ↑_ values may be passed in
             |    |           ‾and out of TWO through I.
             |    ⊥
             |  type conversion required here.
             ⊥
       precision conversion  required here.

    CALL TWO(M3,ID) ASSIGN(S);
             ↑   ↑          ↑_ type conversion illegal for
             |   |              assign arguments.
             |   ⊥
             |    precision conversion required
             ⊥
        dimension mismatch: parameter  is a 3 x 3 matrix.

    CALL TWO(M1,I) ASSIGN(I);
                ↑         ↑
       appearance in both places is legal.
```

The last example introduces an interesting side effect which occurs when the same data item appears both as an input argument and as an assign argument. In the example, changing the value of assign parameter C during execution of the procedure may, depending on the implementation and the data type of I, result in a simultaneous change of input parameter B. The effect does not occur if type or precision conversion is required for transmission of the input argument. The side effect arises as a result of the actual mechanism used in argument transmission in particular implementations.

Both input and assign arguments may possess arrayness, in which case the corresponding parameters must have an array declaration.

See: Guide/ 20.5.

11.6 RETURNS FROM PROCEDURES AND FUNCTIONS

When execution reaches the CLOSE statement of a procedure block, an automatic return to caller takes place. However, if execution reaches the CLOSE statement of a function block, a run time error results since the function has no value to return to the caller. Hence a function block needs an explicit RETURN statement to cause the return to take place. In addition, if returns are required from parts of the code in a procedure block other than at the CLOSE, an explicit RETURN statement is required.

PROCEDURE RETURN

The RETURN statement of a procedure takes the form:

```
RETURN;
```

Example:

```
CHOICE: PROCEDURE(FLAG) ASSIGN(DIR);
  DECLARE FLAG BOOLEAN;
  DECLARE DIR VECTOR(3);
  IF FLAG THEN RETURN;
  DIR = UNIT(DIR);
CLOSE;
```

If FLAG ≡ TRUE then procedure merely returns execution at RETURN.

If FLAG ≡ FALSE then 3-vector DIR is normalized, and procedure returns execution at CLOSE.

FUNCTION RETURN

The RETURN statement of a function takes the form:

```
RETURN exp;
```

1. The resultant value of the expression *exp* is returned when the function returns to its caller.

The return of an expression by a function is similar in nature to the transmission of an input argument of a function to the corresponding parameter, the function itself playing the role of parameter. During return, type and precision conversions are possible, but dimension matching must be ensured. The relevant rules are the same as those described for argument transmission in Section 11.4.

Note that since a function block may not be defined with an array specification, no function may return an array result.

Examples:

```
  FUNC1: FUNCTION(A) SCALAR;
   DECLARE A MATRIX(3,3) DOUBLE;
   DECLARE I INTEGER;
    .
    .
    .
    .
    .
    .
    RETURN I+5;         ←conversion to scalar required.
    .
    .
    .
    .
    .
    RETURN A    ;         ←conversion to single precision required.
S           1,1
    .
    .
    .
    .
    .
    RETURN ‘I = ‘ || I;   ←illegal type conversion required.
    .
    .
    .
    .
    .
    CLOSE;
```

12.0 INPUT/OUTPUT STATEMENTS

Higher order languages possess I/O statements to provide programs with a means of communicating with their environment. In HAL/S, simple forms of I/O statement provide for the sequential input or output of data, including the generation of printed listings.

This section first introduces the HAL/S concept of sequential I/O and then goes on to describe the construction of I/O statements.

12.1 HAL/S INPUT/OUTPUT CONCEPTS

The form of sequential I/O statements in HAL/S is based on a specific conceptualization of the input-output process. In this conceptualization, I/O takes place through a number of "channels", each identified by an integer code. Each channel is connected to an "I/O device", of which there are two kinds, "unpaged", and "paged".

UNPAGED DEVICES

An "unpaged I/O device" can be used for both input and output. It can be visualized as consisting of a "device mechanism" which performs I/O on a continuous strip, across which data is written. The data is organized in "columns" across the strip, and in "lines" down it:

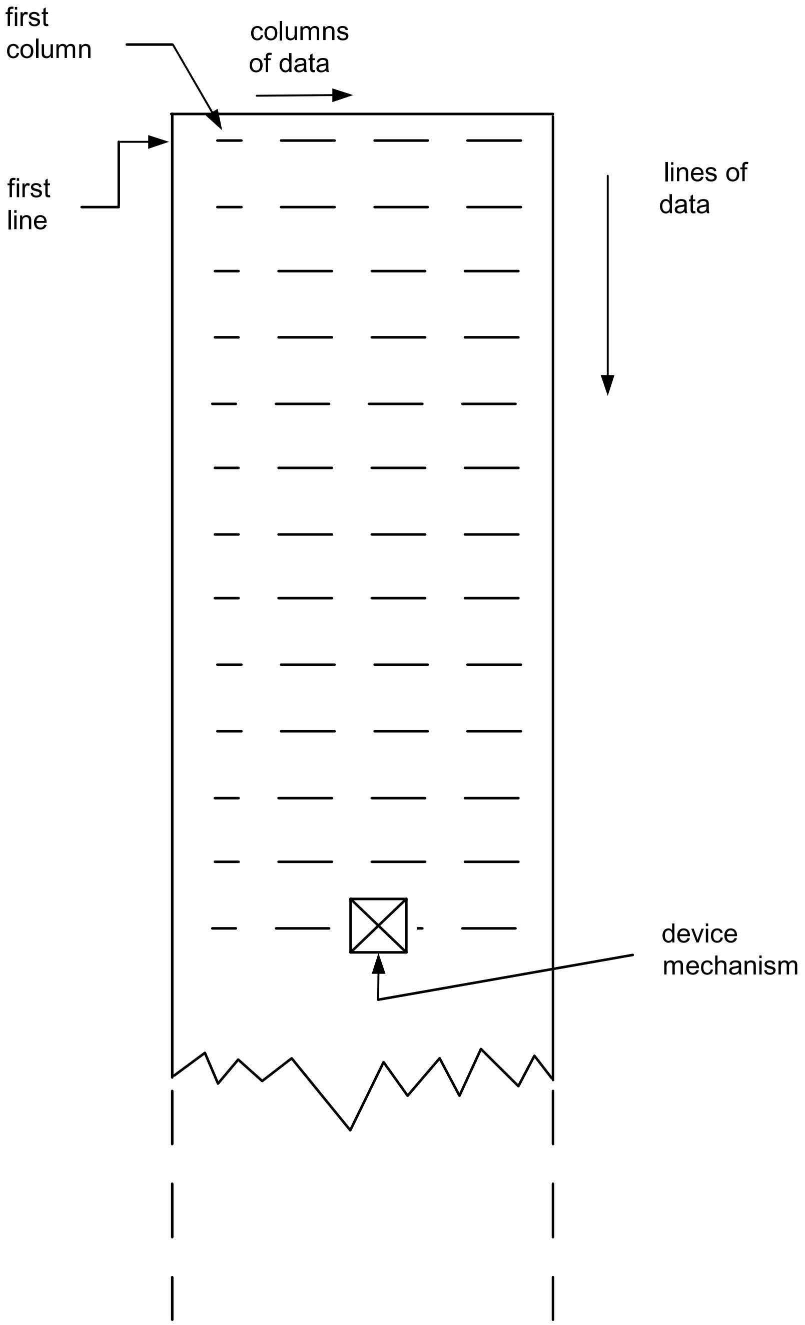

first
column
columns
of data
first
line
lines of
data
device
mechanism

Figure 12-1

The device mechanism moves from column to column along each line, and from line to line as it performs I/O. Normally, the performance of I/O is accompanied by movement from left to right across each line, and downwards from one line to the next. However, special positioning commands can modify this behavior.

On output, the strip continually lengthens as new lines are written on the device. On input, the strip is of fixed length, and a run time error occurs if the device mechanism is requested to read off the lower end.

Data output to an unpaged device is physically written so that it may, on some future occasion, be read in again via an unpaged device.

PAGED DEVICES

A "paged I/O device" can only be used for output. It can be visualized in much the same way as an unpaged device, except that the lines of data are organized into "pages'':

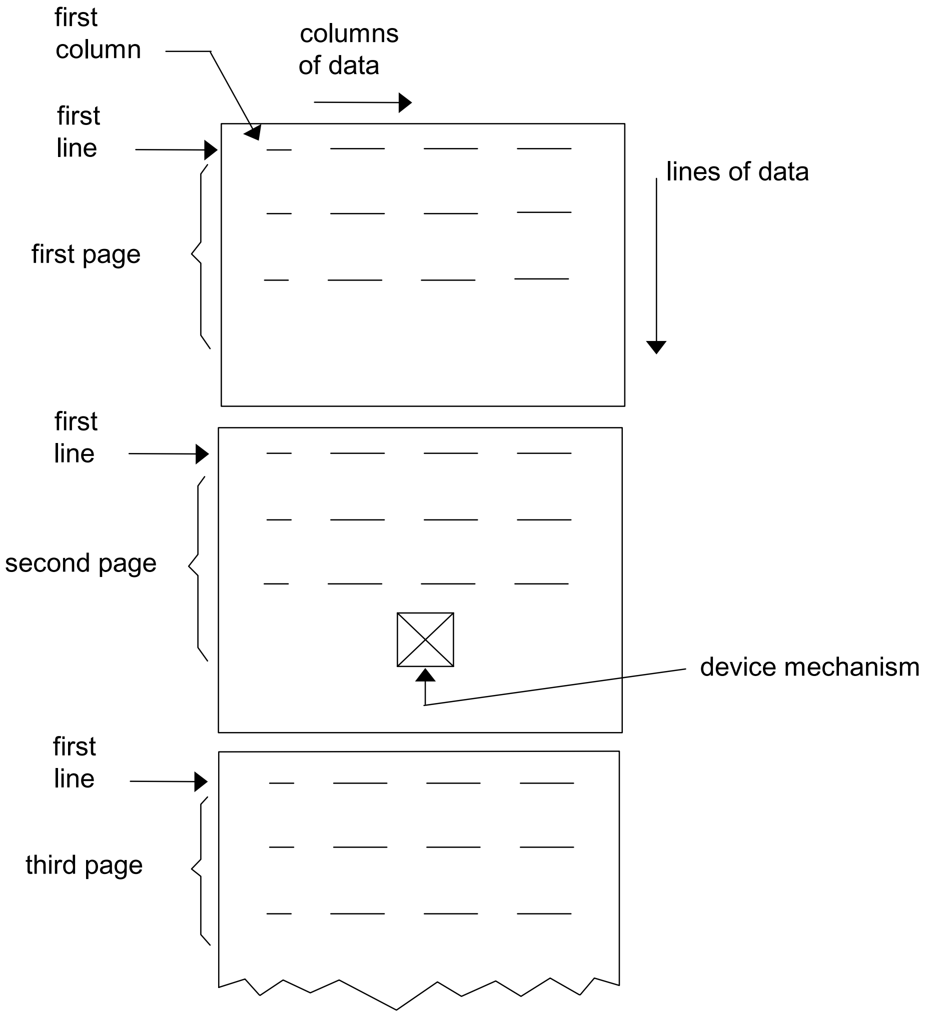


Figure 12-2

The paged device is designed to generate printed listings. The form in which data is physically written on the device is different from that on an unpaged device. Such data cannot normally be read back again via an unpaged device.

DATA STORAGE

Data is conceived as being "stored" on a device, even though in physical reality the device may be a line printer, the data becoming inaccessible to the computer.

In HAL/S, data is written on the I/O device in "fields" which can be separated by blank columns, or by a separator character. The I/O process is stream-oriented: within the confines of a single I/O statement, the column and line alignment of data fields need be of no consequence. Data fields may even be broken over line or page boundaries.

12.2 THE WRITE STATEMENT

The WRITE statement is an executable statement for the output of data to a paged or unpaged I/O device. The form of the WRITE statement is as follows:

WRITE(n) *exp*^{1},*exp*^{2},...*exp*^{n};

1. n is the channel code number, and lies in the range $0 \leq n \leq 9^\dagger$.
2. Each *exp* is a HAL/S expression whose value or values are to be written on the device. The list of expressions may be arbitrarily long. Alternatively, none need be supplied.
3. Each expression in turn from left to right is evaluated, and its value (or values) written on the specified device.

† This value may be implementation dependent. See appropriate User's Manual.

In execution, the sequence of events is as follows:

- If the WRITE statement is the first to be executed for the specified device, the device mechanism positions itself at column 1 of line 1 (on page 1 if the device is paged). Otherwise, the device mechanism moves down one line from its current position, and repositions itself at column 1.
- Data fields are written from left to right along the line, each field being separated from the next by 5 blanks[14].
- When the end of a line is reached, the device mechanism moves to column 1 of the next line and continues writing data fields. Unless the data field is of character type, the device does not attempt to break it over a line boundary if there is not room for it at the end of a line. Instead, it begins writing it on the next line.

[14] This value may be implementation dependent. Some implementations may allow the user to vary the value by a run-time option. See appropriate User's Manual.

- After finishing execution, the device mechanism is left positioned one column to the right of the end of the last data field written. Alternatively, if the data field abuts the end on a line, it is positioned at column 1 of the next line.
- If no expressions are supplied in the WRITE statement, the device merely performs its initial positioning.

DATA FORMATS

The format of a data field depends on the type of expression whose resultant value is being written on the device, and on whether or not the device is paged. The formats are, in general, implementation dependent. Typical formats are shown in Appendix F.

Uni-valued expressions each give rise to a single data field. Multi-valued expressions each give rise to a series of data fields, which are written on the device sequentially in the following way:

- an ℓ-vector expression yields ℓ scalar data fields, one for each element. The data fields are laid out along a line, separated from each other by the standard number of blanks, and overflowing onto succeeding lines as required.
- an m x n matrix expression yields mn scalar data fields, one for each element. The matrix is laid out row by row. Each row is written as if it were an n-vector. The first element of the second and subsequent rows begin a new line, vertically aligned under the first element of the first row.
- arrays are written array element by array element, completing the requirements for one element before going on to the next. The last data field of one array element is separated from the first data field of the next element by the standard number of blanks, or starting a new line if required.

Examples:

Let: M be a 3x3 matrix with $M \equiv \begin{bmatrix} 0.5 & 1.5 & 0.0 \\ 2.5 & 1.0 & 1.0 \\ 0.5 & 0.1 & 10.0 \end{bmatrix}$

I be a 3-array of integers
with I ≡ (4 6-2)

C be a character with C ≡ 'VALUE'

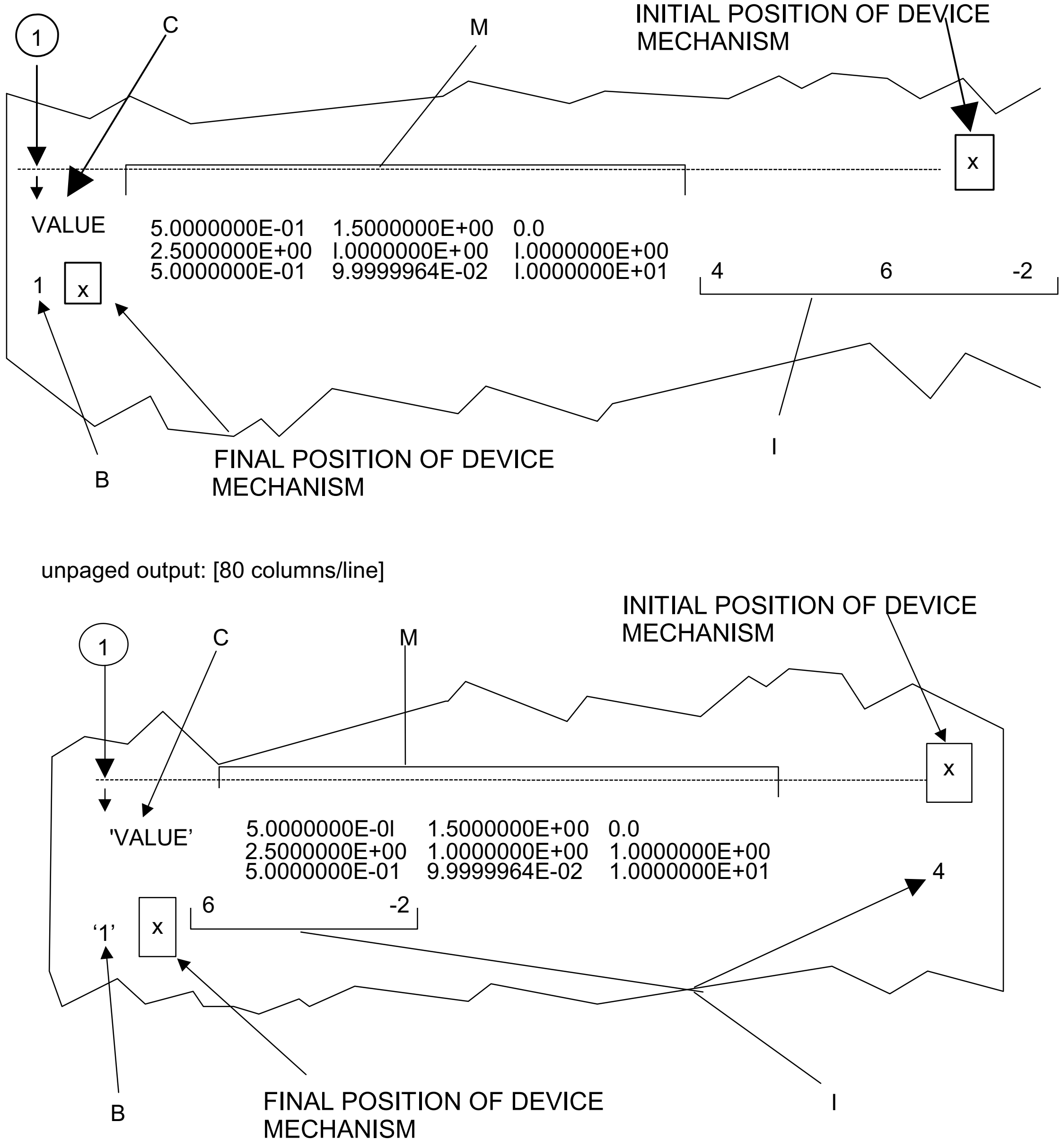
B be a Boolean with B ≡ TRUE

then

```
WRITE(6) C,M,I;
WRITE(6) B;
```

would result in output of the following form:

paged output:[132 columns/line]


1
C
M
INITIAL POSITION OF DEVICE MECHANISM
x
1
x
B
FINAL POSITION OF DEVICE MECHANISM
I

```
VALUE    5.0000000E-01   1.5000000E+00   0.0
         2.5000000E+00   I.0000000E+00   I.0000000E+00
1  x     5.0000000E-01   9.9999964E-02   I.0000000E+01   4          6          -2
```

unpaged output: [80 columns/line]

1
C
M
INITIAL POSITION OF DEVICE MECHANISM
x
'1'
x
B
FINAL POSITION OF DEVICE MECHANISM
I

```
'VALUE'   5.0000000E-0l   1.5000000E+00   0.0
          2.5000000E+00   1.0000000E+00   1.0000000E+00
          5.0000000E-01   9.9999964E-02   1.0000000E+01   4
'1'  x    6          -2
```

NOTES:

single precision scalar data fields are a fixed 14 columns wide.

single precision integer data fields are a fixed 11 columns wide.

Figure 12-3

12.3 THE READ STATEMENT

The READ statement will compile successfully, but will generate incorrect results for BFS and produce an error in the linkage editor for PASS. The error is not generated for BFS because it uses a different linkage editor. The user will see the following message in the map file for PASS:

IEW0264 - TABLE OVERFLOW - INPUT LOAD MODULE CONTAINS TOO MANY EXTERNAL SYMBOLS IN ESD

This is accepted as a known error due to the fact that neither the BFS or PASS flight software use either the READ or READALL statements (DR102959, 10/22/90).

The READ statement is an executable statement for the input of data from an unpaged I/O device. The form of the READ statement is as follows:

READ(n) *var*^{1},*var*^{2},...*var*^{n};

1. n is the channel code number, and lies in the range of 0 ≤ n ≤ 9[†] .
2. Each *var* is any type of data item, either subscripted or unsubscripted. The list of items may be arbitrarily long. Alternatively, none need be supplied.
3. The specified device reads values into each data item in turn from left to right.

[†] This value may be implementation dependent. See appropriate User's Manual.

In execution, the sequence of events is as follows:

- If the READ statement is the first to be executed for the specified device, the device mechanism positions itself at column 1 of line 1. Otherwise, the device mechanism moves down one line from its current position and repositions itself at column 1.
- Data fields are read from left to right along the line. The device expects each data field to be separated from the next by a comma and/or at least one blank.
- When the end of a line is reached, the device mechanism moves to column 1 of the next line and continues reading. Data fields may be broken over the line boundary.
- After finishing execution, the device mechanism is left positioned one column to the right of the end of the last data field read in. Alternatively, if the data field abuts the end of a line, it is positioned at column 1 of the next line.
- If no list of data items is supplied in the READ statement, the device merely performs its initial positioning.
- If the device reads two consecutive separating commas, then the value of the data item which would have been changed by reading a data field between the commas, is instead left untouched. For further clarification, refer to Sec. 10.1.1 of Language Specification.

DATA FORMATS

The formats of data fields expected by a device on input depend on the type of data item being read into. The formats are, in general, implementation dependent. Typical formats are shown in Appendix F.

Uni-valued data items cause single data fields to be read. Multi-valued data items cause a series of data fields to be read sequentially.

- A vector data item causes one data field per vector element to be read.
- A matrix data items causes one data field per matrix element to be read. Values are read into the matrix row by row.
- Arrayed data items are read into array element by array element, completing the read requirements for each element before going on to the next.

Examples:

Let M be a 3x3 matrix with initial values given

$$\text{by } M \equiv \begin{bmatrix} 0.5 & 1.5 & 0.0 \\ 2.5 & 1.0 & 1.0 \\ 0.5 & 0.1 & 10.0 \end{bmatrix}$$

Let I be a 3-array of integers,
C be a character data item of maximum length 10,
B be a Boolean.

Then

```
READ(5) M,I,C;
READ(5) B;
```

using the following data:

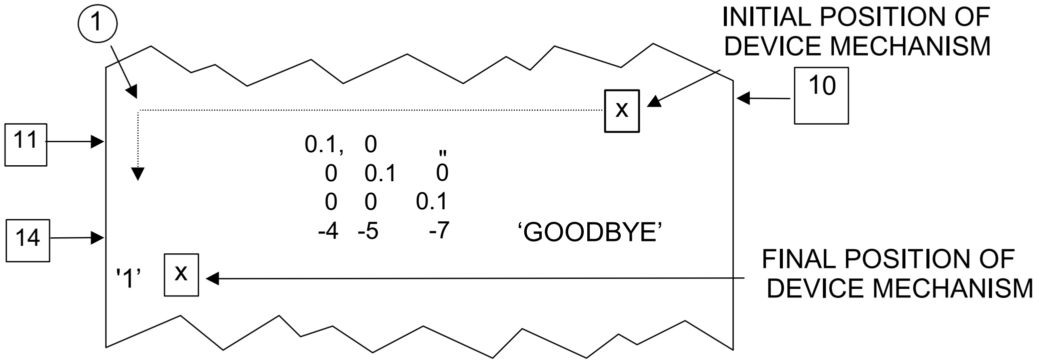

1
INITIAL POSITION OF DEVICE MECHANISM
10
x
11
0.1, 0
"
0 0.1 0
0 0 0.1
-4 -5 -7
'GOODBYE'
14
'1'
x
FINAL POSITION OF DEVICE MECHANISM

Figure 12-4

would result in:

$$M \equiv \begin{array}{l} \lceil 0.1 \quad 0.0 \quad 0.0 \rceil \neg \text{this value not changed by READ statement.} \\ |\, 0.0 \quad 0.1 \quad 0.0 \,| \\ \lfloor 0.0 \quad 0.0 \quad 0.1 \rfloor \end{array}$$

$I \equiv (-4\ -5\ -7)$

$C \equiv$ 'GOODBYE'

$B \equiv$ TRUE

12.4 INPUT/OUTPUT FORMATTING

The formatting of I/O embraces two separate concepts:

- the position of data fields;
- the shape of data fields.

Data field positioning is effected by direct movement of the device mechanism. Commands in the form of pseudo-functions can be inserted into READ and WRITE statements to cause repositioning of the mechanism.

In terms of input, formatting implies that a device can be made to recognize different shapes of data fields in a variety of positions. In terms of output, formatting implies that a device can generate different shapes of data fields in a variety of positions.

The shape of data fields can be controlled using formatted I/O statements. The equivalent of arbitrary data field shaping can be achieved using HAL/S character string handling facilities.

> There exists a second type of input statement called the READALL statement, which can be used to input arbitrary strings of characters. This can form the basis for arbitrary data field shape recognition on input.
>
> See: Guide/ 22.1.

DEVICE MECHANISM POSITIONING

HAL/S possesses five pseudo-functions which can reposition a device mechanism during execution of a READ or WRITE statement. The pseudo-functions are placed in the READ or WRITE statement as if they were normal data items or expressions.

Three basic rules underlie the operation of the pseudo-functions in positioning device mechanisms:

- Horizontal and vertical positioning are separately and independently controlled.
- The operations of the pseudo-functions are independent of whether a device is being used for input or output.

- An explicit repositioning command taking effect at a particular point in execution overrides the default movement in the same direction (horizontal or vertical) which would otherwise be made by the device mechanism.

Particular instances of these rules are noted as the device positioning pseudo-functions are described below.

HORIZONTAL POSITIONING

The two pseudo-functions TAB and COLUMN serve to position a device mechanism horizontally on a line. Their form is as follows:

```
TAB(α)
COLUMN(β)
```

1. α and β are integer expressions.
2. TAB(α) moves the device mechanism left or right by the number of columns specified by α. Negative values of α denote movement to the left; positive values, movement to the right.
3. COLUMN(β) moves the device mechanism left or right to the column indicated by β.
4. Values of α or β must not be such as to try to move the device mechanism left past column 1, or right past the rightmost column[†].

[†] The number of columns on any device (i.e. the logical record length) is assumed constant but implementation dependent. See appropriate User's Manual.

If a TAB or COLUMN pseudo-function appears at the beginning of a READ or WRITE statement, it overrides the default positioning at column 1.

It does not of itself inhibit movement onto the next line.

If a TAB or COLUMN appears between two expressions in a WRITE statement, it overrides the standard data field separation.

Successive TABs are cumulative in action.

Example:

If C1, C2, C3 are character data items
with C1 ≡ 'FIRST'
C2 ≡ 'SECOND'
C3 ≡ 'THIRD'
and if channel 6 is a paged device
then

```
WRITE(6)TAB(-50),C1,COLUMN(5),C2,C3,TAB(2);
```

produces output of the following form:

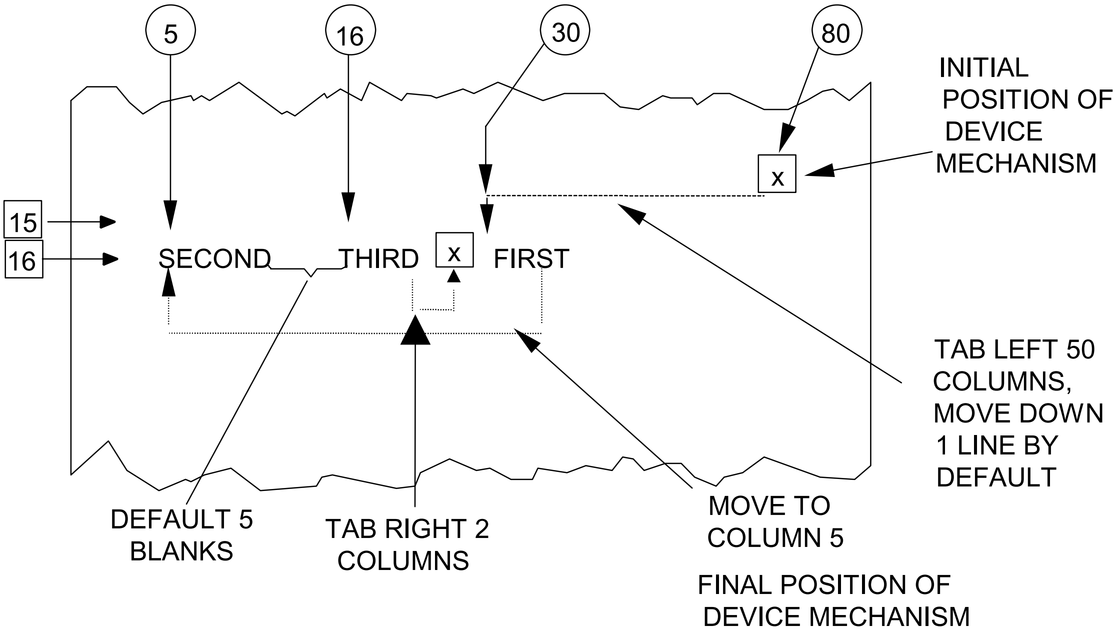

5
16
30
80
INITIAL POSITION OF DEVICE MECHANISM
x
15
16
SECOND
THIRD
x
FIRST
TAB LEFT 50 COLUMNS, MOVE DOWN 1 LINE BY DEFAULT
DEFAULT 5 BLANKS
TAB RIGHT 2 COLUMNS
MOVE TO COLUMN 5
FINAL POSITION OF DEVICE MECHANISM

Figure 12-5

VERTICAL POSITIONING

The three pseudo-functions SKIP, PAGE, and LINE serve to position a device mechanism vertically. PAGE can be used only in I/O via a paged device; the behavior of LINE is different depending on whether a device is paged or unpaged.

The form of the three pseudo-functions is as follows:

```
SKIP(α)
PAGE(β)
LINE(γ)
```

1. α, β *and* γ are integer expressions.
2. SKIP(α) moves the device mechanism downward by the number of lines specified by α. The value of α may be zero, in which case SKIP can suppress a default line advancement. However, α may not be negative (indicating upward movement). SKIPs over page boundaries are allowed.
3. PAGE(β) moves the device mechanism downward by the number of pages specified by β. As in SKIP, β may not be negative in value. The relative line number remains unchanged.
4. For unpaged devices, LINE(γ) positions the device mechanism at line γ. The value of γ must not be such as to cause upward movement of the device mechanism.
5. For paged devices, LINE(γ) has a different behavior. Let the device mechanism be on line ℓ prior to execution of LINE (γ). If $\gamma < \ell$ then the device mechanism moves to line ℓ on the next page. If $\gamma \geq \ell$ then the device mechanism moves to line γ on the current page. The value of γ must lie in the range $1 \leq \gamma \leq L$, where L is the number or lines per page[†].

[†] The number of lines per page is implementation dependent. See appropriate User's Manual.

If a SKIP, LINE, or PAGE pseudo-function appears at the beginning of a READ or WRITE statement, it overrides the default downward movement of one line.

SKIP, LINE and PAGE pseudo-functions do not of themselves inhibit the default horizontal movement to column 1. Neither does their appearance between two expressions in a WRITE statement affect the standard data field separation.

Successive SKIPs and PAGEs are cumulative in effect.

Examples:

If C1, C2, C3 are character data items
with C1 ≡ 'FIRST'
C2 ≡ 'SECOND'
C3 ≡ 'THIRD'

and if channel 6 is a paged device

then

```
WRITE(6) SKIP(0),C1, LINE(1),C2,C3;
```

produces output of the following form:

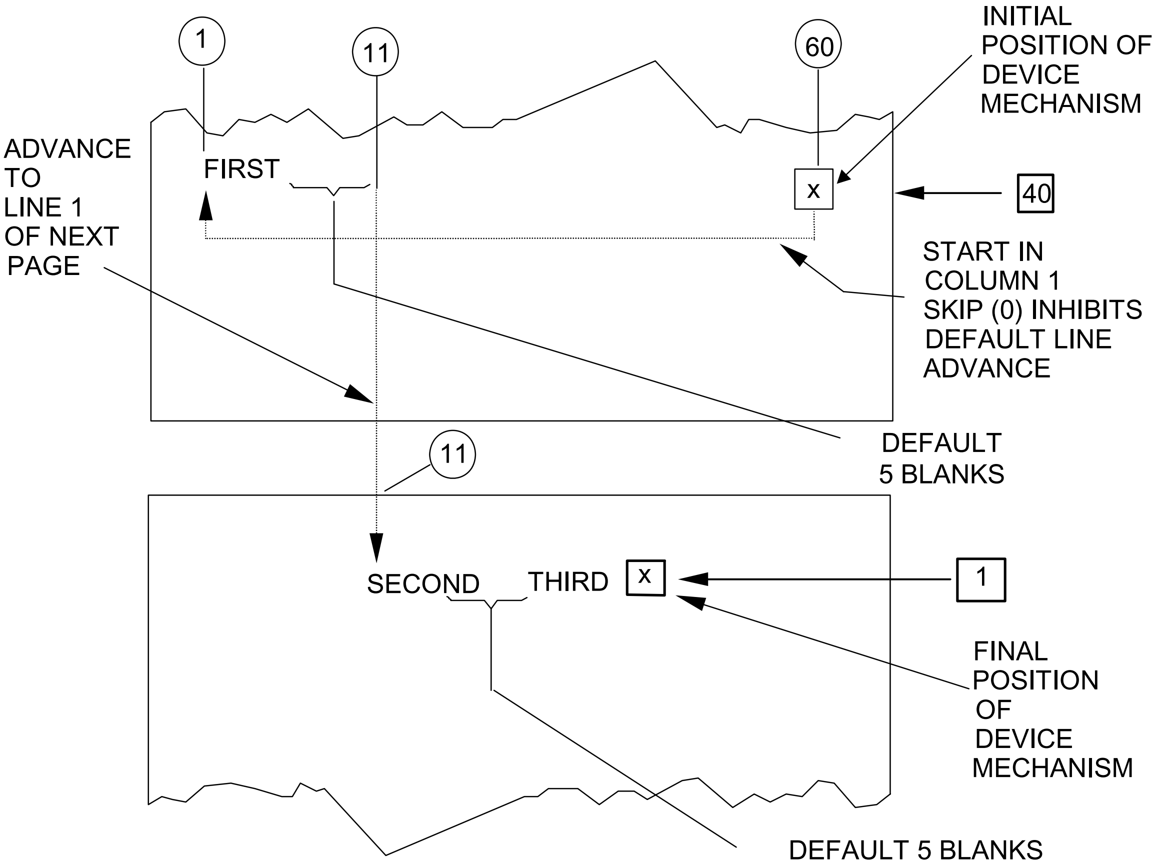

1
11
60
INITIAL POSITION OF DEVICE MECHANISM
ADVANCE TO LINE 1 OF NEXT PAGE
FIRST
x
40
START IN COLUMN 1 SKIP (0) INHIBITS DEFAULT LINE ADVANCE
DEFAULT 5 BLANKS
11
SECOND
THIRD
x
1
FINAL POSITION OF DEVICE MECHANISM
DEFAULT 5 BLANKS

Figure 12-6

Note: If channel 6 were unpaged, the WRITE statement would be illegal since it would be calling for an upwards movement from line 40 to line 1.

Further,

```
WRITE(6) C1,PAGE(1),C2;
```

produces the output of the form:

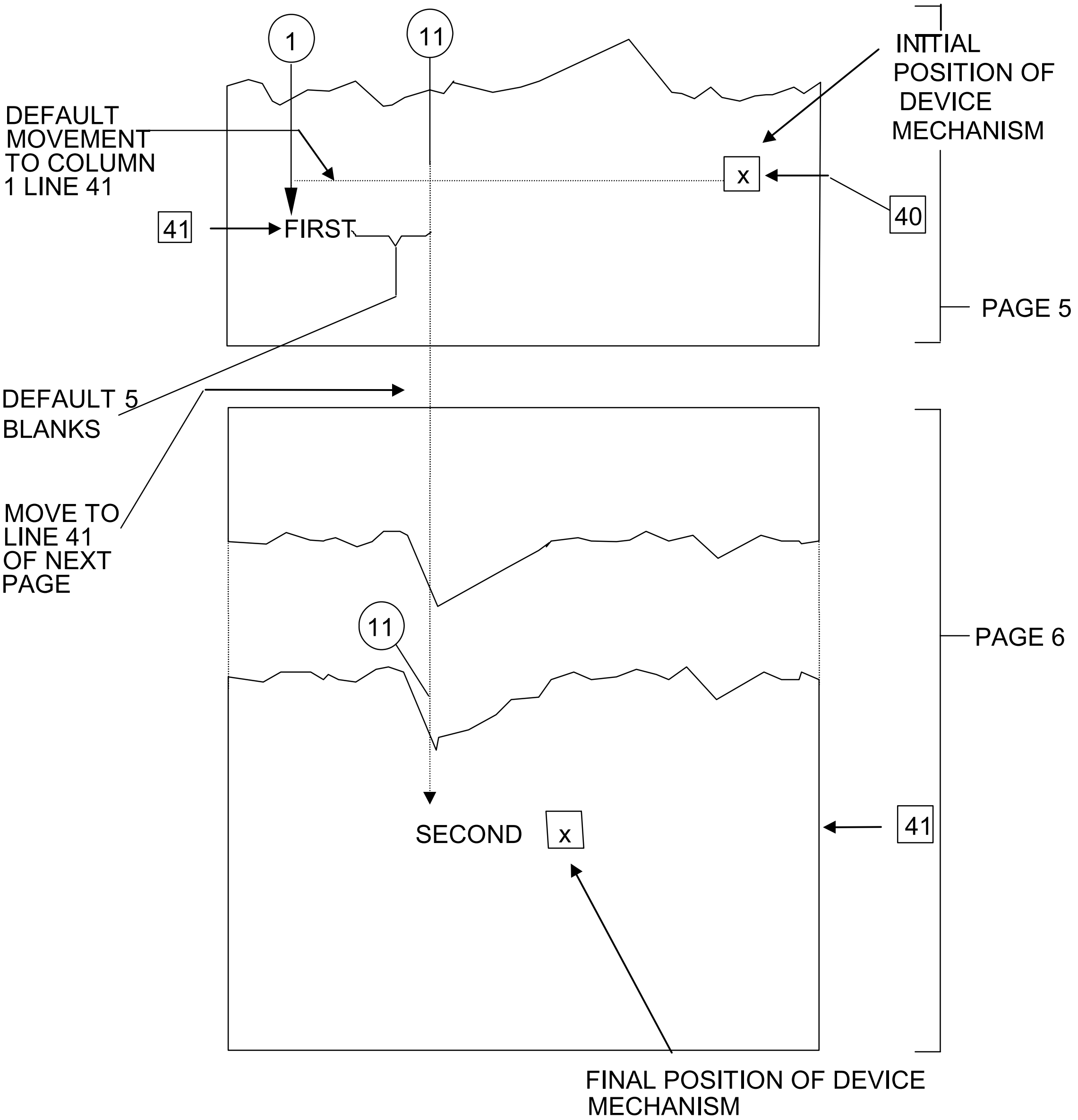

1
11
DEFAULT MOVEMENT TO COLUMN 1 LINE 41
INITIAL POSITION OF DEVICE MECHANISM
x
40
41
FIRST
PAGE 5
DEFAULT 5 BLANKS
MOVE TO LINE 41 OF NEXT PAGE
PAGE 6
11
SECOND
x
41
FINAL POSITION OF DEVICE MECHANISM

Figure 12-7

12.4.1 I/O WITH FORMATS

The shape of data fields can be described using FORMAT items. These consist of a character describing the field type followed by a description of allowable field contents. FORMAT items are associated with READ and WRITE statements via the keyword IN, as in:

```
WRITE(6) A IN 'I4';
```

which specifies that the variable A be output as an integer four characters in length.

IN is followed by a FORMAT character expression that may be a single FORMAT item, as shown above, or a list of FORMAT items separated by commas or slashes (/). Slash is equivalent to SKIP(1), COLUMN(1).

Several elements can be output according to a single format item, as in:

```
WRITE(6) (VAR1, VAR2, VAR3) IN 'I4';
```

A list of elements can also be associated with a list of format items, as in:

```
READ(5) (V1, V2, CM1) IN 'F8.2, F10.3/A6';
```

which is equivalent to:

```
READ(5) V1 IN 'F8.2', V2 IN 'F10.3',SKIP(1),COLUMN(1),CM1 IN
'A6';
```

HAL/S FORMAT items and their meanings are as follows.

Item	Meaning
I	INTEGER
F	SCALAR
E	SCALAR (with exponents)
U	INTEGER, SCALAR, or CHARACTER
A	CHARACTER
X	blanks on output, skips on input
P	INTEGER and SCALAR
Quote string	CHARACTER on output, skips on input

Each of these is described in more detail later.

When a data item is processed, the FORMAT character expression is processed until an I, F, E, U, A or P item is found. Slashes, I/O control, X items, and quote strings are processed as they occur. The next data item is processed similarly except that scanning of the FORMAT character expression resumes where it last stopped. Array items are treated element by element.

A repeat factor can be used to abbreviate FORMAT expressions.

For Example:

```
'5I4'
```

is equivalent to:

```
'I.4, I.4, I.4, I.4, I.4'
```

Parentheses may be placed around several FORMAT items and the repeat factor made to apply to all.

For Example:

```
    '2(I4/)'
```

is equivalent to:

```
    'I4/I4/'
```

If the end of a FORMAT character expression is reached before an associated list of data items has been exhausted, one of two actions is taken.

1. If the format character expression contains no parentheses, scanning resumes from the beginning.
2. Otherwise, scanning resumes from the open parenthesis corresponding to the last closed parenthesis. A repetition factor, if present, is taken into account.

For Example:

```
        WRITE(6) ARRAY_X IN '10F8.2/';
                    (1 to 100)
```

produces 10 rows of 10 figures each.

```
IN
```

1. n is an unsigned positive integer giving the field length.
2. Implicit INTEGER/SCALAR conversion is allowed.
3. Variables of type CHARACTER or BIT cannot be used with I format.

For WRITE statements, a sign is printed only for negative quantities. The number is right-justified in the output field. If the output field is too small, an error message is issued and asterisks printed.

For Example:

```
   DECLARE A INTEGER INITIAL(3)
   WRITE(6) (A,A+8,A-4,A+99) IN 'I2'
```

produces:

```
   ƀ311-1**
```

with an overflow error.

```
   Fn Fn.d
   En En.d
```

1. n is an unsigned positive integer giving the field length.
2. d is an unsigned positive integer giving the number of decimal places.
3. Only INTEGER or SCALAR variables or expressions can be read or written with F and E FORMATS.

For READ statements, there is no difference between the E and F FORMAT items. The input may be signed. If it contains a decimal, this overrides the d specification. Otherwise, d gives the number of decimal digits.

An exponent of the form E±K may be supplied; either E or ± may be omitted. Blanks are allowed preceding the sign, the first digit, E, ±, and the first digit of the exponent.

For WRITE statements with F FORMAT, the string printed is:

$$\underbrace{\texttt{-aaaa}}_{m}\underbrace{\texttt{.bbb}}_{n}$$

n is the second number in the FORMAT. m is determined by the magnitude of the quantity to be printed. The minus sign is printed only if the quantity is negative.

If there is room enough, a zero is added to the left of the decimal if there are no other digits there. Any additional positions are filled with blanks from the left.

For WRITE statements with E FORMAT, the quantity printed is:

$$\texttt{-}\underbrace{\texttt{a.bbb}}_{n}\texttt{E±cc}$$

The minus is printed only if the quantity is negative. One significant digit is printed to the left of the decimal point. This is zero if the quantity is zero. n is taken from the FORMAT item.

For both F and E FORMAT items, an insufficient field length results in an error message and the printing of asterisks.

Examples:

```
READ(5) A IN 'F6.3'
```

interprets:

ƀ12.34 as 12.34
ƀ ƀ1234 as 1.234
ƀ.1234 as .1234

```
READ(5) A IN 'E9.1'
```

interprets:

ƀ ƀ 246E+14 as 24.6E+14

```
WRITE(6) A IN 'F6.2'
```

writes:

98.672 as ƀ98.67

```
WRITE(6) A IN 'E9.1'
```

writes:

24.6E+14 as ƀ ƀ 2.5E+15

`An`

1. n is an unsigned positive integer giving the field length.
2. A format items are used for CHARACTER data only.

For READ statements, if n exceeds the declared maximum length of the variable, the leftmost characters of the field are retained. Otherwise, the current length of the CHARACTER variable is set to n.

For WRITE statements, if n exceeds the current length of the variable, the string is padded with blanks on the left. Otherwise, the leftmost characters are written to fill the field.

Examples:

```
DECLARE A CHARACTER(5)
     READ(5) A IN ‘A4’
```

reads the input ABCDEF as ABCD and set the current length of A to 4.

```
READ(5) A IN ‘A6’
```

reads the input ABCDEF as ABCDE since the maximum length of A is 5.

```
WRITE(6) A IN ‘A8’
```

writes the value ABCD as ƀ ƀ ƀ ƀ ABCD.

```
WRITE(6) A IN ‘A3’
```

would write the value ABC.

`Un`

1. n is interpreted as follows.

Data Type	Interpretation of Un
CHARACTER	An
INTEGER	In
SCALAR	En.d where d=n-7

2. U (undefined) is used for any of the three data types given above.

Examples:

```
DECLARE ARRAY(10,2) INTEGER, HEIGHT_AND_WEIGHT;
WRITE(6) (‘HEIGHT’, ‘WEIGHT’, HEIGHT_AND_WEIGHT IN ‘2U7/’;
```

would produce a table such as:

```
HEIGHT    WEIGHT
 61        120
 70        152
 .          .
 .          .
 .          .
 56        108
```

```
Xn
```

1. n is an unsigned positive integer giving the number of blanks on output, skips on input.
2. Xn is equivalent to TAB(n).

```
    “ccccc” or ‘ccccc’
```

1. c is a character.

Example:

```
   WRITE(6) ANS IN ‘ “ANSWER=”,I2’;
```

produces output of the form

```
   ANSWER = 21
```

```
   P ‘picture’
```

1. Picture establishes a format that mixes character and numeric data.
2. Numeric data fields (INTEGER and SCALAR) are indicated by forms such as:

   ```
              P$$$
              P$$$.
   P ANS =    $$$.$$$
              P$$$.$$$*$
   ```

   where ‘.’ places the decimal and * places an exponent.
3. The P format item runs from the P to the first ’,’ or ’/’ encountered or the end of the FORMAT character string. All characters are printed except for $ and *.
4. For READ statements, consecutive ‘$’, ‘.’, and ‘*’ characters define a field of the same length. Decimals in the input field take precedence over decimals in the FORMAT.

FORMAT I/O WITH BIT VARIABLES

There is no FORMAT Item specifically for BIT variables. Instead, the BIT and CHARACTER conversion functions (see 21.3 and 21.4) may be used with CHARACTER variables.

For Example:

```
    DECLARE BITS BIT(8) INITIAL HEX ‘1F’;
    WRITE(6) CHARACTER    (BITS) IN ‘A8’;
                     @BIT
```

produces:

```
      00011111
```

For READ statements, BIT values must be read into CHARACTER variables and the BIT conversion function applied.

12.5 DEVICE ATTRIBUTES

In HAL/S, devices have been characterized as either paged or unpaged. In the absence of any specific direction on the part of a user, the following rules determine whether a device being used is paged or unpaged.

- If only WRITE statements appear in a compilation for a given channel, then the device on that channel will be paged.
- If only READ statements appear, or if both READ and WRITE statements appear for a given channel, then the device on that channel will be unpaged.

The user may specifically direct certain channels to be paged or unpaged, overriding these rules[15].

HAL/S contains a FILE statement by which random-access I/O may be effected.

See: Guide/22.2.

[15] See the User's Manual for a given implementation.

13.0 REAL TIME PROGRAMMING - I

So far the Guide has made no reference to the dynamic properties of HAL/S programs. Clearly, any program will take a finite time to execute but none of the constructs hitherto described depend on any sense of time for their operation.

However, the HAL/S language does contain constructs which depend on a sense of time for their operation. This is what is meant by the statement that HAL/S is a "real time programming language". In other words, HAL/S programs can be written which, when executed, cause operations to be carried out at desired points or during desired intervals in "real time".

In some implementations of HAL/S, "real time" may be just what the phrase implies, real clock time. In others, the "real time" may be simulated in some way by the operating environment of a HAL/S program: in this case, it can be referred to as "pseudo-real time".

This section of the Guide explains the basic HAL/S concepts of real time programming, and describes some of the more elementary real time programming language forms.

13.1 HAL/S REAL TIME CONCEPTS

The true HAL/S concept of a program at run time is an entity executing over some interval in "real time", directed and controlled by a Real Time Executive (RTE). At the outset, the RTE begins execution of the program. When program execution is completed, control is returned to the RTE. In HAL/S terminology, the dynamic counterpart of the static program block, which is executing under RTE control, is called a "real time process''.

MULTI-PROCESSING IN HAL/S

Multi-processing is the simultaneous handling of more than one "real time process". With most present-day machines, "simultaneous" really means interleaved, because most machines can at one time only support the execution of a single machine instruction sequence. However, this distinction has no significance at the higher level of the HAL/S language.

The RTE of HAL/S can simultaneously handle an arbitrary[1] number of processes created by the user. A number is attached by the user to each process, called its "priority". The RTE maintains processes in a "process queue" ordered by priority, and always endeavors to execute the processes in order of priority, highest first.

The HAL/S program itself, beginning execution under the RTE, constitutes the first or "primal process". All other processes are brought into existence by the execution of SCHEDULE statements coded into the program. Just as the primal process has a static counterpart, which is the program block coded by the user, so must the other processes have their static counterparts. These are so-called task blocks, which are coded inside

[1] See the User's Manual for the maximum number supported in any given implementation.

the program block in a very similar way to procedure blocks. Each time a task block is invoked by execution of a SCHEDULE statement, a new process is created and queued by the RTE.

> A number of programs, independently compiled, can be brought together at run time. One of them is chosen by the user to start execution as the primal process. Processes can be generated from the others by invoking them with the same form of SCHEDULE statement. Any of the programs are allowed to contain task blocks from which more processes in turn can be created.
>
> See: Guide/23.1-23.3**.**

STATES OF A PROCESS

It is now possible to represent the behavior of the RTE by a more formal description of the possible states[2] in which a process can exist. This in turn will introduce other HAL/S constructs for controlling the activities of the RTE.

A process can be in either of the following two major states at a given time:

- ACTIVE STATE: a process is in an active state when it exists in the RTE’s process queue. The state actually comprises three substates or minor states in any one of which an active process may be at a given time.
- INACTIVE STATE: a process is defined for completeness as being in the inactive state if it does not exist in the process queue.

The minor states of an active process are as follows:

- EXECUTING: an active process is “executing” when it has actually been put into execution by the RTE, operating on the priority principle already described. The number of processes which can be in this state simultaneously implementation dependent.[3]
- READY: an active process is “ready” if it is available for execution, but higher priority processes in execution are currently barring it. The occurrence of a process first entering the ready state will be called its “initiation’’.
- WAITING: an active process is “waiting” if it is neither ready nor executing. Some condition set up by the user prevents it being available for execution by the RTE.

When a process is created by invoking a task block by a SCHEDULE statement, it makes a transition from the inactive state to an active state. It is entered into the process queue in either the ready or the waiting state, depending on the form of the SCHEDULE

[2] The states to be defined do not correspond one-to-one with the RTE states described in the Language Specification document. The latter are defined for the convenience of the formal description of language constructs. The former are defined with user convenience in mind.

[3] In most implementations it is likely to be 1, but see the User's Manual for a given implementation.

statement. If it entered in the ready state, then depending on its priority, it may immediately be elevated to the executing state.

A process is caused to make a transition from an active state to the inactive state (or removed from the process queue) by a TERMINATE statement. The process is said to have been "terminated".

The priority of an active process may be changed by an UPDATE PRIORITY statement.

A process may be forced into the waiting state by execution of a WAIT statement.

The statements outlined above are among the real time programming language forms to be described later in this section.

PROCESS SWAPPING & BREAKPOINTS

A process swap is a pair of state transitions in which one process leaves the executing state, and a second enters it from the ready state. The process swap may occur because the first process has been forced into the inactive state or the waiting state, or because the second process has a higher priority than the first.

The HAL/S language itself makes no assumptions on where process swapping can occur. However, most implementations, depending on the object machine characteristics, limit process swapping to given places in the HAL/S code sequences under execution by the RTE. These places are called "breakpoints". The determination of breakpoints is a function of the HAL/S compiler for a given implementation, and no language construct exists to modify their existence[4].

The effect of breakpoints is to superimpose a kind of time granularity on the operation of the RTE.

PRIORITY SCALES

The number specifying the priority P of a process is an integer in the range:

$$0 \leq P \leq 255^{5}$$

The primal process is assigned a priority of 50[5] by the RTE on beginning execution.

PROCESS DEPENDENCY

Suppose that there are two processes, *A* and *B*, and that *A* creates process *B* during the course of its execution. At the time of creation, *B* may be specified to be either "dependent" on or "independent" of *A* . If *B* is dependent, it means that it depends for its existence on the existence of *A.* If *B* is independent, then *A* may cease to exist without affecting *B*'s existence.

However, an overriding rule is that all other processes are always dependent on the primal process for their existence.

[4] As an example, in the HAL/S-360 implementation, breakpoints may occur at the beginning, end, or middle of an executable statement.

[5] These values are, however, implementation dependent. See appropriate User's Manual.

The consequences of dependency will be seen when the flow of execution through program and task blocks is described in Section 13.3, and again when the TERMINATE statement is introduced in Section 13.5.

13.2 TASK BLOCK DEFINITIONS

A task block is a static block of code interior to a program, from whence processes can be created by means of the SCHEDULE statement. Task blocks may only be defined at the program level, and not nested inside procedure or function blocks defined in a program. This is illustrated as follows:

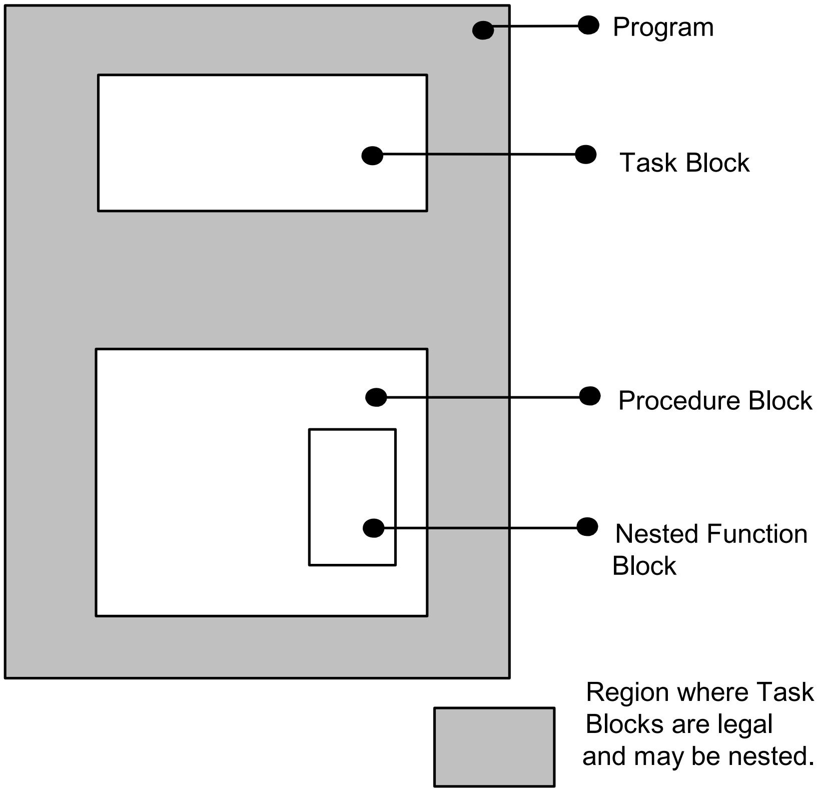


Figure 13-1

Task block definitions are similar to program block definitions as described in Section 3, and have similar opening and closing statements.

RELATIVE POSITION OF TASK DEFINITIONS

Statements invoking a task block should normally follow its block definition.

> This rule is not absolute - it can be circumvented by the use of a task declaration statement.
>
> See: Spec./4.6.

TASK OPENING

The statement opening a task block takes the form:

```
label: TASK;
```

label is any legal identifier name, and constitutes the name of the task block.

TASK CLOSING

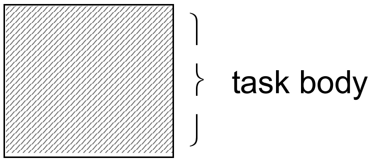
The statement closing a task block takes the form:

```
CLOSE label;
```

The identifier *label* is optional.
If supplied, it must be the name of the task block.

Example:

```
DISPLAY: TASK;
```



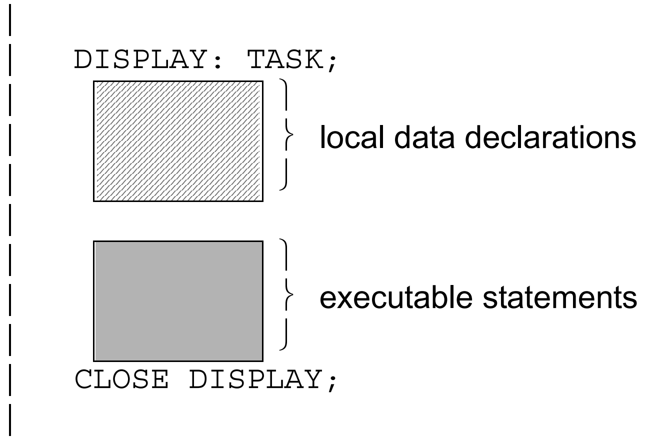
```
CLOSE DISPLAY;
```

LOCAL DATA DECLARATIONS

Local data can be declared in a task block in exactly the same way as it is declared in a procedure or function block. The declarations appear after the task opening statement, and before the first executable statement of the block. The forms of the declarations have been described in Section 4.

Examples:

general positioning -



```
DISPLAY: TASK;

CLOSE DISPLAY;
```

particular instance -

```
|
|   DISPLAY: TASK;
|    DECLARE S CHARACTER(10),   ⎫
|            I INTEGER;         ⎬——   local data
|                               ⎭
|    .
|    .
|    .
|   CLOSE DISPLAY;
|
```

13.3 FLOW OF EXECUTION IN PROGRAM AND TASK BLOCKS

The flow of execution through program and task blocks is subject to a new interpretation, based on the concepts of real time programming introduced in this section. Programs and tasks are treated together since their representations at run time are in both cases real time processes.

Execution of a process begins with the first executable statement in the corresponding static program or task block. It continues, and if not terminated by some other process, ends in one of the following ways:

- by execution of a TERMINATE statement terminating itself;
- by reaching the CLOSE statement of the block;
- by execution of a RETURN statement in the block.

If execution ends by self-termination, the process goes into the inactive state and is removed from the process queue. All dependents of the process are treated likewise.

If execution ends on a CLOSE or RETURN statement, the process goes into the inactive state directly only if it has no dependents. Otherwise, it goes into a waiting state until the dependents have in their turn terminated.

FORM OF RETURN STATEMENT

The form of RETURN statement for programs and tasks is the same as for procedures:

```
RETURN;
```

13.4 THE SCHEDULE STATEMENT

The SCHEDULE statement is an executable statement causing a new process to be placed in the process queue, or "initiated''. The SCHEDULE statement specifies a task block from which the process is to be created, and the priority which it is to be given. A condition for the initiation of the process can be supplied.

Only one process derived from a given task block may be active at any given time.

The form of the SCHEDULE statement varies, depending on whether it specifies immediate, or delayed initiation (transition to the ready state).

IMMEDIATE INITIATION

The following variant of the SCHEDULE statement is the simplest. It causes the creation of a process which is placed in the process queue in the ready state. The process is thus available for execution immediately.

```
SCHEDULE  label PRIORITY (α) DEPENDENT;
```

1. A process is created from the task block *label* and placed in the process queue in the ready state. The process created is also known by the name *label.*
2. α is an integer expression specifying the priority of the newly-created process. It must lie in the legal range for a given implementation.
3. The keyword DEPENDENT is optional. Its presence denotes the dependency of the process created on the process executing the SCHEDULE statement. In its absence, the processes are independent.

Examples:

```
SCHEDULE DISPLAY PRIORITY(100) DEPENDENT;
SCHEDULE RECOVER PRIORITY(255);
```

DELAYED INITIATION

The following form of the SCHEDULE statement causes a process to be placed in the process queue in the waiting state. The process is transferred to the ready state on a specified time criterion being met. There are two variants, each with a different time criterion.

- INITIATION after some duration.

```
SCHEDULE label IN interval PRIORITY(α) DEPENDENT;
```

1. A process called *label* is created from the corresponding task block and placed in the process queue in the waiting state.
2. PRIORITY(α) and DEPENDENT have the same meanings as described in the previous form of SCHEDULE statement.
3. The phrase IN *interval* indicates that the process is to be put in the ready state after a specified interval in the waiting state. *interval* is a scalar expression whose value specifies the duration in seconds.
4. If the value is negative or zero, the process is put in the ready state immediately.

- INITIATION at a given time.

```
SCHEDULE label AT time PRIORITY(α) DEPENDENT;
```

1. A process called *label* is created from the corresponding task block, and placed in the process queue in the waiting state.
2. PRIORITY (α) and DEPENDENT have the same meanings as described in the previous form of SCHEDULE statement.
3. The phrase AT *time* indicates that the process is to be put in the ready state at a specified real time. *time* is a scalar expression whose value specifies the time in seconds.†
4. If the indicated time is in the past, the process is placed in the ready state immediately.

Examples:

```
SCHEDULE  AT 1.25E4 PRIORITY(I+5);
SCHEDULE  IN S+15.5 PRIORITY(20);
```

SCHEDULE statements can also specify the cyclic execution of a process until a stopping criterion is met. An explicit specification of the interval between cycles can also be given.

See: Guide/ 23.4 & 23.5.

13.5 OTHER REAL TIME FEATURES OF HAL/S

Three other real time programming statements which have already been mentioned are now described. These are the TERMINATE, WAIT, and UPDATE PRIORITY statements. Certain other useful constructs are also introduced.

† The real time origin is not specified by the language. The origin is normally coincident with the initiation of the primal process. Some implementations allow its value to be preset at run time. See appropriate User's Manual.

TERMINATE STATEMENT

A process is forced to the inactive state (removed from the process queue) by means of the TERMINATE statement. Its form is shown below:

```
TERMINATE label;
```

1. The appearance of *label* is optional. If present, the statement terminates an active process called *label*.
2. If *label* is absent, then the process executing the TERMINATE statement is terminating itself.

In order to make independent processes truly independent, HAL/S places an added restriction on the operation of the TERMINATE statement. A process is only allowed to use it to terminate itself or its dependents.

Note that when a process is terminated by execution of a TERMINATE statement, all its dependents are automatically terminated at the same time.

Examples:

```
TERMINATE;            - self termination
TERMINATE BETA;       - termination of dependent
```

If a number of processes are to be terminated simultaneously, the TERMINATE statement can specify a list of process names:

```
TERMINATE ALPHA,BETA,GAMMA;
```

WAIT STATEMENT

The WAIT statement is used to force the process executing it into a waiting state until some condition is met, whereupon it returns to the ready state. Three forms, each with a different condition, are described below.

- WAIT for a duration.

```
WAIT interval;
```

1. The statement indicates that the process is to be placed in the waiting state for a specified duration.
2. *interval* is a scalar expression specifying the duration in seconds.
   A negative or zero value results in the process not leaving the ready state.

- WAIT until some time.

```
WAIT UNTIL time;
```

1. The statement indicates that the process is to be placed in the waiting state until some given time.
2. *time* is a scalar expression specifying the time of return to the ready state, in seconds[†].
3. Specification of a time in the past results in the process not leaving the ready state.

[†] See the discussion on the SCHEDULE statement in Section 13.4 for a footnote remarking on the real time origin.

- WAIT for dependents.

```
WAIT FOR DEPENDENT;
```

1. The statement indicates that the process is to be placed in the waiting until all its dependent processes have terminated.
2. If there are no dependents, the statement has no effect.

Examples:

```
WAIT UNTIL DELTA_T+15E2;
WAIT S/2;
WAIT FOR DEPENDENT;
```

UPDATE PRIORITY STATEMENT

The UPDATE PRIORITY statement is used to change the priority of an active process. Its form is:

```
UPDATE PRIORITY label TO α;
```

1. The process whose priority is to be changed is specified by *label*.
2. The name *label* is optional. If omitted, the process executing the statement is indicated.
3. α is an integer expression whose value indicates the new priority value to be assigned.

Examples:

```
UPDATE PRIORITY TO 16;
UPDATE PRIORITY  TO I+20;
```

Since the RTE operates on a basis of priority, apparently a user could control the execution of a desired set of processes by manipulating their relative priorities. Although this is entirely possible, it is not recommended since the behavior of such a priority-driven scheme would depend on how many processes an RTE could bring into the executing state simultaneously, which is an implement-dependent figure.

REAL TIME BUILT-IN FUNCTIONS

Two built-in or library functions are of utility in constructing real time programs:

Function	Comments
RUNTIME	returns the current value of real time as a scalar, in seconds.
PRIO	returns the priority of the process invoking the function as an integer.

MAJOR STATE INDICATION

There exists a way of finding out whether the current state of any process is either active or inactive (i.e. whether or not it exists).

The name of the process can be used as if it were a Boolean variable. The following tables shows the correspondence between state and truth value.

State	Value
ACTIVE	TRUE
INACTIVE	FALSE

Example:

to write a message if a process ALPHA exists -

```
IF ALPHA THEN WRITE(6) 'ALPHA IS ACTIVE';
```

13.6 A SIMPLE REAL TIME PROGRAM

The utility and importance of the constructs defined in this section can only be properly understood by presenting an actual example of a real time program.

The following example is given in the form of a problem and its solution.

PROBLEM

The problem is to write a program which, when run on a computer facility with remote interactive terminals, will aid users in electronic circuit design (to take an arbitrary example). A user begins each design session by logging onto the facility at a terminal, and invoking execution of the circuit design program.

The program is to be set up so that, at the outset, the user may specify the desired duration of his session. The program is then to interrupt the user's calculations every 10 minutes and remind him how much time he has used. At the expiration of the specified session duration, the program is to allow the user 10 minutes more and then terminate the session.

SOLUTION

Only the overall features of the program from the real time programming standpoint are illustrated here. The actual circuit design algorithms are of no consequence.

Execution of the circuit design program implies the existence of three real time processes.

- a SUPERVISOR process controlling the two others, which determines the session duration, and makes arrangements to terminate the session at its expiration. Most of the time this process will be in the waiting state.
- a TIMER process which informs the user how much time he has used every 10 minutes. This process is also mostly in the waiting state, temporarily being in execution every 10 minutes.
- a CALCULATOR process which actually interacts with the user in his design session. This process is executing most or all of the time.

The following diagram summarizes the activities of the three processes.

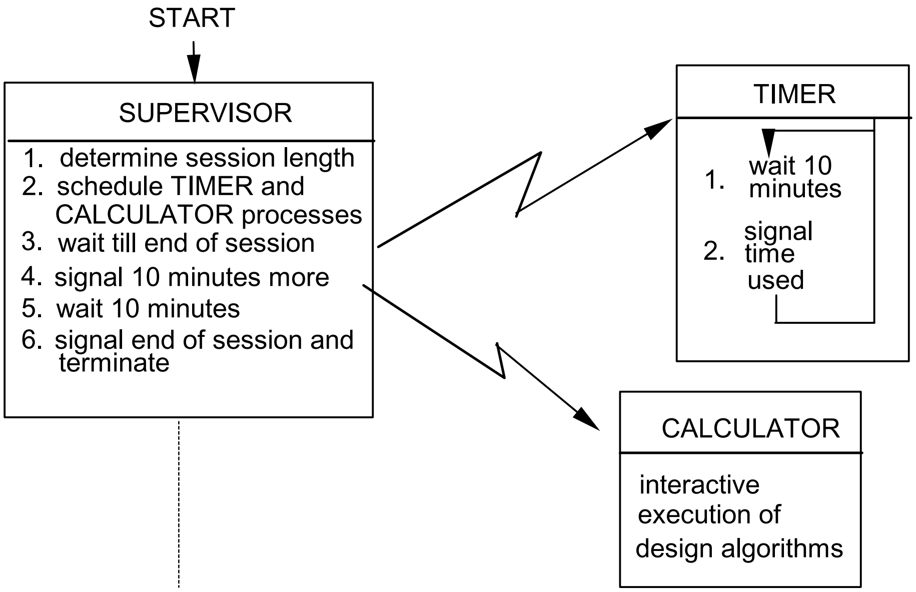
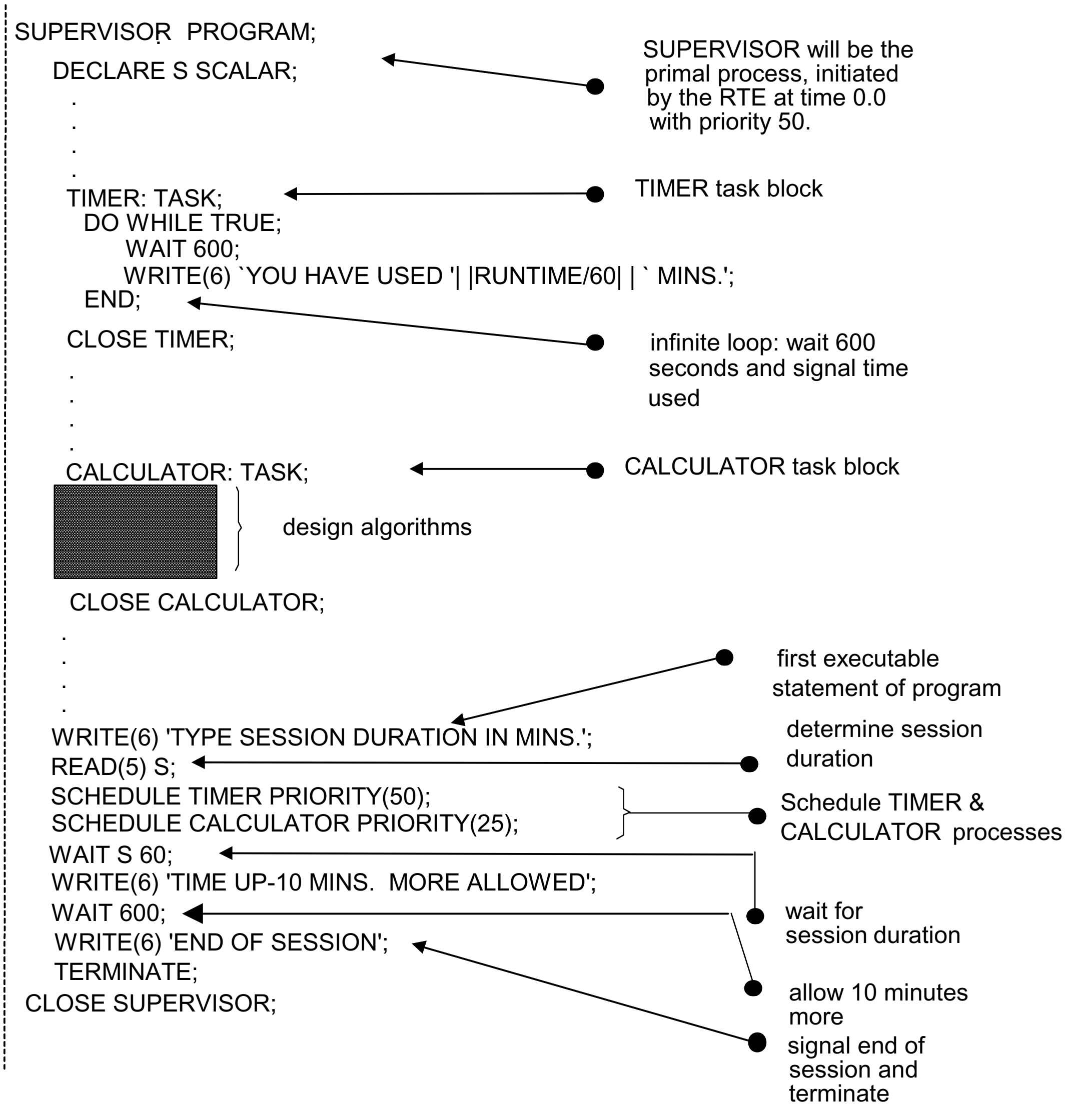

START
SUPERVISOR
1. determine session length
2. schedule TIMER and CALCULATOR processes
3. wait till end of session
4. signal 10 minutes more
5. wait 10 minutes
6. signal end of session and terminate
TIMER
1. wait 10 minutes
2. signal time used
CALCULATOR
interactive execution of design algorithms

Figure 13-2

Clearly, in order for TIMER to interrupt CALCULATOR reliably every 10 minutes, it must have a higher priority than CALCULATOR. Likewise, SUPERVISOR should be of higher priority than CALCULATOR. The relative priorities of SUPERVISOR and TIMER do not matter since TIMER is mostly in the waiting state anyway. The table below shows suitable priorities for each of the three processes.

process	priority
SUPERVISOR	50
TIMER	50
CALCULATOR	25

The HAL/S program corresponding to these processes is as shown below:



```
SUPERVISOR  PROGRAM;
   DECLARE S SCALAR;
     .
     .
     .
     .
    TIMER: TASK;
      DO WHILE TRUE;
           WAIT 600;
           WRITE(6) `YOU HAVE USED '| |RUNTIME/60| | ` MINS.';
      END;
    CLOSE TIMER;
     .
     .
     .
     .
    CALCULATOR: TASK;

    CLOSE CALCULATOR;
    .
    .
    .
    .
   WRITE(6) 'TYPE SESSION DURATION IN MINS.';
   READ(5) S;
   SCHEDULE TIMER PRIORITY(50);
   SCHEDULE CALCULATOR PRIORITY(25);
   WAIT S 60;
   WRITE(6) 'TIME UP-10 MINS.  MORE ALLOWED';
   WAIT 600;
   WRITE(6) 'END OF SESSION';
   TERMINATE;
 CLOSE SUPERVISOR;
```

SUPERVISOR will be the primal process, initiated by the RTE at time 0.0 with priority 50.

TIMER task block

infinite loop: wait 600 seconds and signal time used

CALCULATOR task block

design algorithms

first executable statement of program

determine session duration

Schedule TIMER & CALCULATOR processes

wait for session duration

allow 10 minutes more

signal end of session and terminate

Figure 13-3

The constructs described above enable real time processes to be manipulated according to time criteria. Other constructs enable their manipulation according to “event” criteria. HAL/S “events” are Boolean-like data types whose values are accessible to the RTE. Their values can be set by the user, thus indirectly controlling the real time process states.

See: Guide/24.

The problem of controlling the sharing of data by two or more processes is also important.

See: Guide/26.4.

14 (DELETED)

This page intentionally left blank.

15.0 COMPOOLS AND COMSUBS

The HAL/S program was represented in Part I of this Guide as a totally self-contained unit. In particular, the program was said to contain declarations of all its own data, and definitions of all the procedure and function blocks it needed to invoke.

However, a HAL/S program may also reference data declared externally, and invoke procedure or function blocks declared externally[1]. This ability is of considerable importance in the creation of large programs by teams of programmers because it facilitates the separate and parallel development of the programs' constituent algorithms. Further advantages will become apparent during the renewed discussion of real time multi-processing in Section 23.

In HAL/S, data external to a program is defined in a block called a COMPOOL. Externally defined procedures and functions are collectively called COMSUBs.

15.1 RELATIONS BETWEEN PROGRAMS, COMPOOLs AND COMSUBs

The compools and comsubs referenced by a program are themselves separately compoolable entities. For example, when a program invokes an external procedure, which shares with it the use of data in a single compool, then a total of three separate compilation units is involved[2]. This situation is shown below:

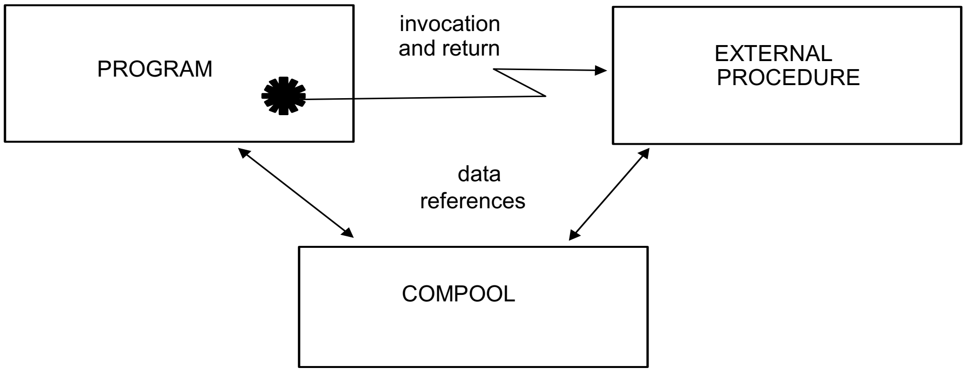


Figure 15-1

Section 3 of the Guide described one kind of compilation unit - the program block - but there are four kinds of compilation units in the HAL/S language:

- PROGRAM, the only independently <u>executable</u> compilation unit;
- EXTERNAL PROCEDURE, callable from a program or any other comsub;

[1] External procedures and functions in HAL/S are similar to the FUNCTIONS and SUBROUTINES of FORTRAN. External data roughly corresponds to the COMMON data of FORTRAN.

[2] The object modules resulting from their compilation have to be "link-edited" to produce a single executable load module.

- EXTERNAL FUNCTION, also callable from a program or any other comsub;
- COMPOOL, defining data shared by programs and comsubs, but containing no executable code.

The HAL/S language insists upon a full declaration of all data, and invariably checks the compatibility of function and procedure definitions with their invocations. These precautionary measures are specifically extended to compool data and comsubs through the use of so-called "block templates".

Every program or comsub which references compools or other comsubs must be provided with block templates of the compilation units referenced.

- COMPOOL TEMPLATE - contains data declarations identical with those of the compool itself, so that the referencing compilation unit possesses a complete description of the data.
- EXTERNAL FUNCTION TEMPLATE - contains an input parameter list identical with that of the external function itself, so that the compatibility of its invocations by the referencing compilation unit can be verified.
- EXTERNAL PROCEDURE TEMPLATE - contains input and assign parameter lists identical with those of the external procedure itself, so that the compatibility of its invocations by the referencing compilation unit can be verified.

The required block templates are included in the compilation units which reference the corresponding compools and comsubs. For example, in the case already described of a program invoking an external procedure and sharing data in a single compool, the situation is as shown:

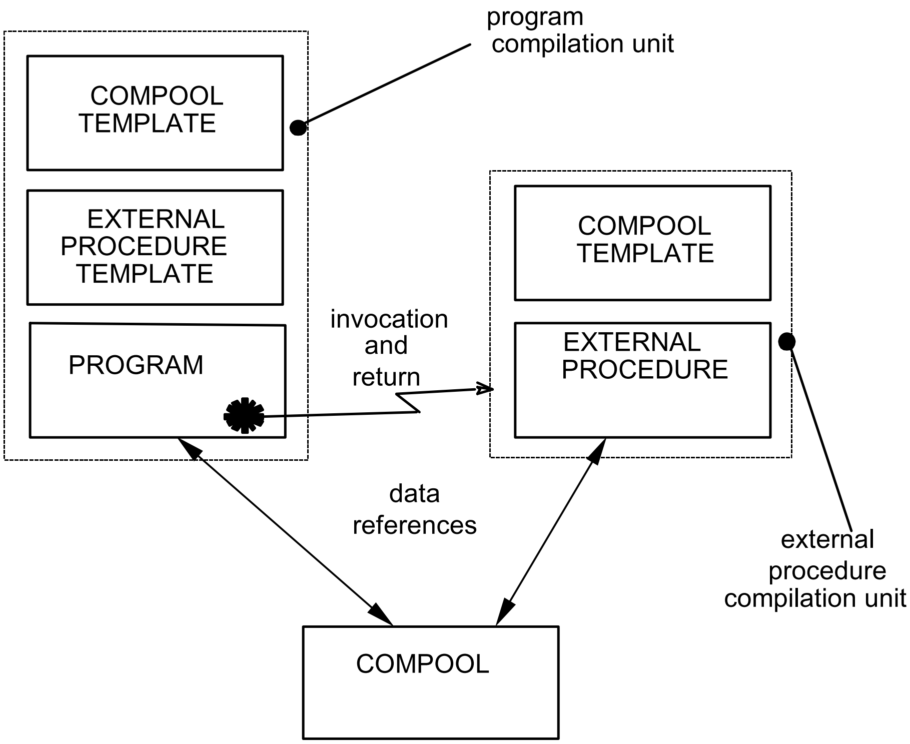


Figure 15-2

To summarize, when the term "compilation unit" was introduced in Section 3 of the Guide, its meaning was the same as "program block" because the existence of compools and comsubs had not been considered. Now it is apparent that a compilation unit does not necessarily contain executable code (it may be a compool), and neither is it necessarily just a single block of executable code (one or more templates may be included in it).

In HAL/S, block templates are designed to eliminate incompatibility between separately complied modules as a source of software unreliability. It may be objected however that no language construct can force the properties of a compool or comsub to be reflected correctly in the corresponding block template[3]. The use of correct templates is generally insured by use of a software management scheme. Part of such a scheme would be the automatic generation of block templates during compilation of the corresponding compools and comsubs.

[3] Neither can it ensure that the object modules "link-edited" together are the correct versions.

15.2 THE COMPOOL BLOCK

The compool block has been introduced as an external block of data accessible to programs and comsubs with which the appropriate block template is included. It consists of opening and closing statements delimiting a sequence of data declarations.

COMPOOL OPENING

The statement opening a compool block takes the form:

```
label: COMPOOL;
```

1. *label* is any legal identifier name, and constitutes the name of the block.

COMPOOL CLOSING

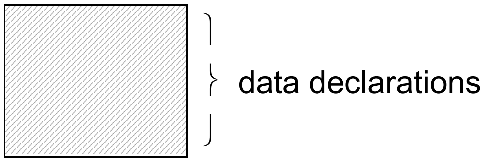
The compool block is closed with the statement:

```
CLOSE label;
```

1. The identifier *label* is optional.
2. If *label* is supplied, it must be the *label* supplied on the opening statement of the block.

Example:

```
COMMON: COMPOOL;
```



```
CLOSE COMMON;
```

COMPOOL DATA DECLARATIONS

Declaration of data in a compool differs in no respect from data declarations in a program as described in Section 4. In particular, there is no objection to the initialization of data in a compool.

Example:

```
POOL: COMPOOL;
  DECLARE VZERO VECTOR INITIAL(0);
  DECLARE INTEGER DOUBLE, I, J, K;
  DECLARE CC CHARACTER(10);
CLOSE POOL;
```

Note that REPLACE statements, which are placed together with declarations, can also appear in a compool, and thus affect any program or comsub with which the corresponding compool template is included. Simple REPLACE statements were described in Section 5.

15.3 EXTERNAL PROCEDURE AND FUNCTION BLOCKS

Comsubs have been introduced as external function and procedure blocks callable from programs or other comsubs.

The forms of external function and procedure blocks are identical with ordinary function and procedure blocks, whose definitions were described in Section 11. Likewise, they are invoked in a manner identical with that described in Section 11.

15.4 BLOCK TEMPLATES

Block templates indicate the properties of compools and comsubs to the program or comsub referencing them. Their form is similar to the corresponding compool or comsub.

COMPOOL TEMPLATES

A compool template is identical with its corresponding compool block except that the opening statement is modified by the keyword EXTERNAL:

```
label: EXTERNAL COMPOOL;
```

1. *label* is the name of the corresponding compool block.

Example:
compool block:

```
POOL: COMPOOL;
   DECLARE VZERO VECTOR INITIAL(0);
   DECLARE INTEGER DOUBLE, I, J, K;
   DECLARE CC CHARACTER(10);
CLOSE POOL;
```

corresponding template:

```
POOL: EXTERNAL COMPOOL;
   DECLARE VZERO VECTOR INITIAL(0);
   DECLARE INTEGER DOUBLE, I, J, K;
   DECLARE CC CHARACTER(10);
CLOSE POOL;
```

EXTERNAL PROCEDURE TEMPLATES

An external procedure template differs from its corresponding procedure block in the following respects:

- the body of the block is empty except for declarations describing the attributes of input and assign parameters;
- the opening statement is modified as shown below by the keyword EXTERNAL.

label: EXTERNAL PROCEDURE(i^1,i^2,...)ASSIGN(a^1,a^2,...);

1. *label* is the name of the corresponding procedure block.
2. i^1,i^2,...and a^1, a^2,... are lists of input and assign parameters respectively, identical with those in the corresponding procedure block.

Example:
external procedure:

```
FIXIT: PROCEDURE(INCR) ASSIGN(RESULT);
  DECLARE RESULT VECTOR(3),
   INCR VECTOR(3);
  DECLARE DELTA CONSTANT(1.5E-4);
  RESULT = RESULT + DELTA INCR;
CLOSE FIXIT;
```

corresponding procedure template:

```
FIXIT: EXTERNAL PROCEDURE(INCR) ASSIGN(RESULT);
  DECLARE RESULT VECTOR(3),
         INCR VECTOR(3);
             ←---
CLOSE FIXIT;     |
                 |
    no local data or executable code
```

Sometimes REPLACE statements (see Section 5), and structure template definitions (see Section 19) are required to fully define declarations of parameters. It is therefore legal for these to appear in procedure templates.

EXTERNAL FUNCTION TEMPLATES

An external function template differs from its corresponding function block in the following respects:

- the body of the block is empty except for declarations describing the attributes of input parameters;
- the opening statement is modified as shown below by the keyword EXTERNAL.

label: EXTERNAL FUNCTION(i^1,i^2,...) *attributes*;

1. *label* is the name of the corresponding function block.
2. i^1, i^2,... is a list of input parameters identical with those in the corresponding function block.
3. *attributes* defines type, precision, and size attributes, of the corresponding block.

Example:

- external function:

```
SWITCH:FUNCTION(ARG) BOOLEAN;
  DECLARE ARG SCALAR DOUBLE;
  IF ARG<0 THEN RETURN FALSE;
  RETURN TRUE;
CLOSE SWITCH;
```

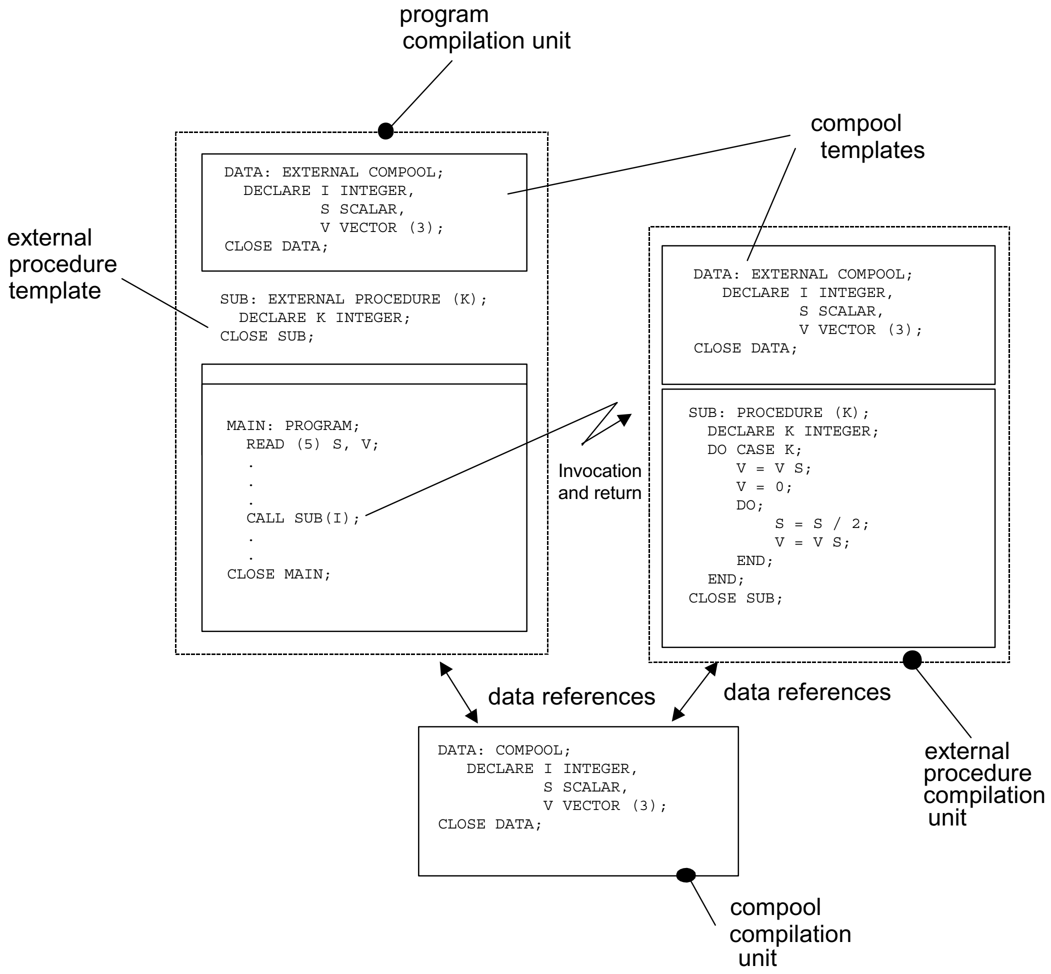
- corresponding function template:

```
SWITCH:EXTERNAL FUNCTION(ARG) BOOLEAN;
  DECLARE ARG SCALAR DOUBLE;
CLOSE SWITCH;
```

Function templates, like procedure templates, may also contain REPLACE statements and structure template definitions.

EXAMPLE OF USE

The example given below is a restatement of the example used twice in Section 15.1 in terms of actual HAL/S statements. It shows a program calling an external procedure, and sharing compool data with it.



```
DATA: EXTERNAL COMPOOL;
  DECLARE I INTEGER,
          S SCALAR,
          V VECTOR (3);
CLOSE DATA;

SUB: EXTERNAL PROCEDURE (K);
  DECLARE K INTEGER;
CLOSE SUB;

MAIN: PROGRAM;
  READ (5) S, V;
  .
  .
  .
  CALL SUB(I);
  .
  .
CLOSE MAIN;
```

```
DATA: EXTERNAL COMPOOL;
   DECLARE I INTEGER,
           S SCALAR,
           V VECTOR (3);
CLOSE DATA;

SUB: PROCEDURE (K);
  DECLARE K INTEGER;
  DO CASE K;
     V = V S;
     V = 0;
     DO;
         S = S / 2;
         V = V S;
     END;
  END;
CLOSE SUB;
```

```
DATA: COMPOOL;
   DECLARE I INTEGER,
           S SCALAR,
           V VECTOR (3);
CLOSE DATA;
```

Figure 15-3

16.0 ADDITIONAL DATA INITIALIZATION FORMS

This Section supplements the discussion in Section 4.3 on initialization by introducing the following topics:

- the implied repeated use of initial lists;
- other ways of reducing the length of an initial list;
- partial initialization of a data item;
- control of the actual occurrence of initialization.

16.1 IMPLIED INITIAL LIST REPETITION

Section 4.3 stated that for single-valued data items, only one literal value can be supplied in an INITIAL/CONSTANT specification. It stated that for multi-valued data items, two alternatives are possible:

- The number of literal values specified in the INITIAL/CONSTANT specification matches the total number of elements implied by the data declaration;
- only one literal value is supplied, in which case that same initial value is given to all elements implied by the data declaration.

When a data item is an arrayed vector or matrix, a third alternative exists. The initial list can consist of a sufficient number of literal values to satisfy the requirements of <u>one</u> array element. In this case, every array element of the data item is initialized to that same set of values. In effect, the initial list is being used repeatedly during the initialization process.

Example:

Consider the declaration:

```
DECLARE V ARRAY(4) VECTOR(3).....
```

the data item V can be initialized by 12 literal values:

```
INITIAL(1,2,3,4,5,6,7,8,9,10,11,12)
```

$$\text{WHEREUPON } V \equiv \left\{ \begin{bmatrix}1\\2\\3\end{bmatrix} \begin{bmatrix}4\\5\\6\end{bmatrix} \begin{bmatrix}7\\8\\9\end{bmatrix} \begin{bmatrix}10\\11\\12\end{bmatrix} \right\}$$

Alternatively, it can be initialized by 1 item:

```
INITIAL(4)
```

$$\text{WHEREUPON } V \equiv \left\{ \begin{bmatrix}4\\4\\4\end{bmatrix} \begin{bmatrix}4\\4\\4\end{bmatrix} \begin{bmatrix}4\\4\\4\end{bmatrix} \begin{bmatrix}4\\4\\4\end{bmatrix} \right\}$$

Thirdly, it can be initialized by 3 items, matching the number of components in each vector:

```
INITIAL(1,2,3)
```

$$\text{WHEREUPON V} \equiv \left\{ \begin{bmatrix}1\\2\\3\end{bmatrix} \begin{bmatrix}1\\2\\3\end{bmatrix} \quad \begin{bmatrix}1\\2\\3\end{bmatrix} \begin{bmatrix}1\\2\\3\end{bmatrix} \right\}$$

16.2 USE OF REPETITION FACTORS

If a number of consecutive values in an INITIAL/CONSTANT specification are identical, they may be replaced by one value and a repetition factor:

$$\ldots\ i^{r},\ i^{r+1},\ i^{r+2},\ \ldots i^{r+n}, \ldots$$

$$\ldots\ i^{r},\ n\#i^{r+1},\ \ldots\ldots.$$

1. In both forms, i represents a literal value in an INITIAL/CONSTANT specification.
2. In the first form $i^{r+1}, \ldots\ldots i^{r+n}$ are identical values.
3. The second form shows the replacement of $i^{r+1}, \ \ldots.\ i^{r+n}$ by $n\#i^{r+1}$, where n is a positive nonzero integer.

Example:

```
DECLARE V VECTOR(6) INITIAL(1,2,2,2,2,3);
```

may be replaced by

```
DECLARE V VECTOR(6) INITIAL(1,4#2,3);
```

If a sequence of values is repeated over and over, they may be treated in a similar way. The sequence is written once, enclosed in parentheses, and prefaced with a repetition factor.

Example:

```
DECLARE S ARRAY(10) INTEGER
     INITIAL(1,2,3,4,5,6,3,4,5,6);
```

may be replaced by:

```
DECLARE S ARRAY(10) INTEGER
     INITIAL(1,2,2#(3,4,5,6));
```

The factored form may be nested if necessary, and can be especially convenient in the initialization of multi-dimensional arrays, or arrays of matrices and vectors.

Example:

```
DECLARE V ARRAY(3,2,2)
    INITIAL(1,2,3,2,3,1,2,3,2,3,1,2);
```

may be replaced by:

```
DECLARE V ARRAY(3,2,2)
     INITIAL(2#(1,2,3,2,3),1,2);
```

which may in turn be replaced by:

```
DECLARE V ARRAY(3,2,2)
   INITIAL(2#(1,2#(2,3)),1,2);
```

16.3 PARTIAL INITIALIZATION

There are two forms of partial initialization of a data item. The first is similar to the repetition factor form of initialization already described.

.. i^{r}, n#, i^{r+n+1}, ...

1. i represents a literal value in an INITIAL/CONSTANT specification.
2. The form n# states that n elements are to remain uninitialized.
   n is a positive nonzero integer.

Example:

```
DECLARE I ARRAY(50)INTEGER
   INITIAL(1,2,3,45#,0,0);
```

leaves elements of I indexed 4 through 48 uninitialized.

The second kind of partial initialization construct signals that the remainder of the data item is to be uninitialized.

INITIAL(i^1,i^2,.....i^n,*)
CONSTANT(i^1,i^2,.....i^n,*)

1. In either form, the asterisk terminating the list signals that the remainder of the data item is to be uninitialized.
2. The number of literal values actually in the list (or implied by the use of repetition factors) must be less than the total number of elements in the data item.

Example:

```
DECLARE V ARRAY(2) VECTOR(3)
    INITIAL(1, 2, 3, 4,*);
```

results in $V \equiv \begin{pmatrix} \begin{bmatrix}1\\2\\3\end{bmatrix} \begin{bmatrix}4\\?\\?\end{bmatrix} \end{pmatrix}$

where ? stands for an uninitialized value.

Expressions computable at compile time may appear in a list of initial values.

See: Guide/Appendix D.

16.4 STATIC AND AUTOMATIC INITIALIZATION

Although initialization has been discussed at length, the circumstances under which it actually is effective have not been considered. In particular, it has not been stated whether initialization is effective only on the first entry of execution into a block, or on every such entry.

- STATIC initialization is initialization effective only on first entry into a block. It is called static because generally it results in the generation of initialized data areas by a compiler, rather than executable code.
- AUTOMATIC initialization is initialization on every entry into a block. It generally results in executable code being generated by a compiler.

The keywords STATIC or AUTOMATIC attached to the declaration of an initialized data item serve to distinguish between the two forms.

LEGAL USE OF SPECIFICATION

No STATIC/AUTOMATIC specification may be used in the declaration of initialized data items in a compool (see Section 15.2). A COMPOOL block is not executable, so the question of entry does not arise. Initialization is viewed as taking place before execution of a program begins.

No data item initialized by the CONSTANT specification may possess a STATIC/AUTOMATIC specification. Such data items are viewed as being similar to literal, so that the question of entry again does not arise.

STATIC/AUTOMATIC specifications can appear, then, in data declarations in any kind of block except for COMPOOL blocks. The utility in the case of PROCEDURE, FUNCTION or TASK blocks is obvious. The utility in the case of PROGRAM blocks will become clear when the discussion of real time processing is reviewed in Section 23.

FORM OF STATIC SPECIFICATION

In the absence of any explicit indication, static initialization is assumed. Alternatively, the keyword STATIC may be used, placed either before or after the INITIAL specification.

Examples:

```
DECLARE I INTEGER STATIC INITIAL(5),
        J INTEGER INITIAL(0) STATIC,
        K INTEGER INITIAL(1);
```

FORM OF AUTOMATIC SPECIFICATION

The keyword AUTOMATIC is used, placed either before or after the INITIAL specification.

Examples:

```
DECLARE I INTEGER AUTOMATIC INITIAL(5),
        J INTEGER INITIAL(0) AUTOMATIC;
```

This page intentionally left blank.

17.0 BIT STRINGS

The form and use of Boolean data was discussed at various points in the first thirteen chapters of the Guide. Their stated purpose was the manipulation of binary valued (logical) quantities. The ability to handle strings of binary values is often useful. In HAL/S, this ability is characteristic of the "bit string" data type, which is essentially a generalization of the Boolean data type already described.

17.1 BIT STRING LITERALS

Boolean literals were described in Section 2. There are corresponding literal forms for bit string quantities:

```
BIN‘bbbbbb’
OCT‘oooooo’
HEX‘hhhh’
DEC‘dddd’
```

1. In the above forms,

   b ~ binary digit
   h ~ hexadecimal digit
   d ~ decimal digit
   o ~ octal digit

2. The number of binary digits represented must not exceed 32[†].

[†] This number may vary between implementations. See appropriate User's Manual.

Examples:

```
BIN‘10110’
HEX‘FAC2’
OCT‘777’
```

Note that BIN '0' ≡ FALSE ≡ OFF and BIN '1' ≡ TRUE ≡ ON

A second form involving a repetition factor exists, reducing the effort of writing strings of identical digits.

See: Spec./2.3.3.

17.2 DECLARATION OF BIT STRING DATA ITEMS

The basic declaration statement for bit string data items is shown below:

```
DECLARE name BIT(n);
```

1. *name* is any legal identifier.
2. n specifies the length of the bit string (i.e. the number of binary digits in it). It must be in the range $1 \le n \le 32$.[†]

[†] This number may vary between implementations. See appropriate User’s Manual.

Examples:

```
DECLARE B1 BIT(16);
```

Note that the following two forms are equivalent:

```
DECLARE B2 BIT(1);
DECLARE B2 BOOLEAN;
```

Declarations of bit string data items can be integrated into compound declarations as described for other data types in Section 4.2.

INITIALIZATION

Initialization of bit string data items follows the rules given in Section 4.2, using bit string literals in the list of initial values.

Examples:

```
DECLARE B16 BIT(16) INITIAL(HEX‘FFFF’);
DECLARE B1 BIT(1) CONSTANT(TRUE);
DECLARE B ARRAY(2) BIT(3) INITIAL(OCT‘7’,OCT‘5’);
```

Literals are padded or truncated as required to fit the data item initialized:

```
DECLARE B8 BIT(8) INITIAL(OCT‘770’);
DECLARE B11 BIT(11) INITIAL(HEX‘FF’);
```

results in

$B8 \equiv 11111000_{2,}$ $B11 \equiv 00011111111_2$

17.3 BIT STRING SUBSCRIPTING

Subscripting forms for bit string data items are similar to those for character data items, as described in Section 6.

UNARRAYED BIT STRINGS

In bit strings, bit positions are indexed left to right starting from 1. In the subscript forms given below, STRING represents an unarrayed bit string data item of length L.

- To select the α^{th} bit from STRING:

$\texttt{STRING}_{\alpha}$

α is an integer expression in the range $1 \leq \alpha \leq L$.

- To select α bits from STRING, starting from the β^{th}:

$\texttt{STRING}_{\alpha \texttt{ AT } \beta}$

1. α is an integer literal value in the range $1 \leq \alpha \leq L$.
2. β is an integer expression in the range $1 \leq \beta \leq L - \alpha + 1$.

- To select a substring starting with the α^{th} bit of STRING, and ending with the β^{th}:

$\texttt{STRING}_{\alpha \texttt{ TO } \beta}$

1. α and β are integer literal values in the range $1 \leq (a,\beta) \leq L$.
2. $\beta > \alpha$

Examples:

If β is an 8-bit string with $B \equiv 11101111_2$ then:

$B_4 \equiv 0_2$

$B_{3 \text{ AT } 3} \equiv 101_2$

$B_{4 \text{ TO } 5} \equiv 01_2$

If a data item is declared to be Boolean, it is really defined as a 1-bit string. It may therefore possess component subscripting consistent with the above rules, even though in this case it performs no useful purpose.

ARRAYED BIT STRINGS

The subscripting forms for arrayed bit string data items are as described in Section 6.2. The colon following an array subscript is mandatory.

Examples:

Let B be a 4-array of 3-bit strings

with $B \equiv (110_2{'}\ 010_2{'}\ \ 000_2{'}\ 101_2)$

then some forms of array subscripting only are:

$B_{2:} \equiv 010_2$ (unarrayed)

$B_{3 \text{ TO } 4} \equiv (000_2{'}\ 101_2)$ (still arrayed)

Some forms of simultaneous array and component subscripting are:

$B_{4:1} \equiv 1_2$ (unarrayed)

$B_{2 \text{ AT } 1:\ 1 \text{ TO } 2} \equiv (11_2{'}\ 01_2)$ (still arrayed)

Some forms of component subscripting only are:

$B_{*:3} \equiv (0_2,\ 0_2,\ 0_2, 1_2)$

Note the mandatory asterisk.

Literal subscripts may alternatively be expressions computable at compile time.

See: Guide/Appendix D.

17.4 BIT STRING OPERATIONS

Section 7.3 of the Guide outlined the logical operations which could be performed on Boolean data. Operations on bit strings are an extension of these. HAL/S recognizes the following operations:

Symbol	Purpose
& AND	intersection
| OR	conjunction
¬ NOT	complement
|| CAT	complement

COMPLEMENT

The complement operation complements the logical value of every bit in the bit string.

Symbolic form: ¬ NOT R

1. The operand *R* is a bit string.

Example:

If B is an 8-bit string with $B \equiv 11000101_2$

then $\neg\ B \equiv 00111010_2$

CONJUNCTION

The conjunction operation causes the logical values of corresponding bit positions in the operands to be OR'ed together.

Symbolic form: $L \overset{|}{\text{OR}} R$

1. The *L* and *R* operands are bit strings.
2. If the operands are of unequal length, the shorter is padded on the left with binary zeroes before ORing.
3. The truth table for each bit position is as follows:

		L	
		1_2	0_2
R	1_2	1_2	1_2
	0_2	1_2	0_2

Example:

If B is a 3-bit string with B $\equiv 100_2$

and BB is a 5-bit string with BB $\equiv 10110_2$

then B|BB $\equiv 10110_2$

Note that a 5-bit result is obtained.

INTERSECTION

The intersection operation causes the logical values of corresponding bit positions in the operands to be AND'ed together.

Symbolic form: $L \overset{\&}{\text{AND}} R$

1. The *L* and *R* operands are bit strings.
2. If the operands are of unequal length, the shorter is padded on the left with binary zeroes before ANDing.
3. The truth table for each bit position is as follows:

		L	
		1_2	0_2
R	1_2	1_2	0_2
	0_2	0_2	0_2

Example:

If B is a 3-bit string with B ≡ 100_2

and BB is a 5-bit string with BB ≡ 10110_2

then B&BB ≡ 00100_2

Note that a 5-bit result is obtained.

CATENATION

The two operands are catenated to form one longer bit string.

Symbolic form: L $\underset{\text{CAT}}{||}$ R

1. The *L* and *R* operands are bit strings.
2. The *L* operand is catenated to the left of the *R* operand.
3. If the sum of the lengths exceeds 32[†] the *L operand is left truncated as required.*

[†] This number may vary between implementations. See appropriate User's Manual.

Example:

If B is a 12-bit string with B ≡ $7E0_{16}$

and BB is a 24-bit string with BB ≡ $42F50B_{16}$

then B||BB ≡ $E042F50B_{16}$,

the left-most 4 bits of B being truncated.

PRECEDENCE

The following table summarizes the precedence rules for bit string operations, and is an extension of the table for Boolean operations given in Section 7.4.

Symbol	Precedence	Purpose
	FIRST	
¬, NOT	1	complement
||, CAT	2	catenation
&, AND	3	intersection
|, OR	4	conjunction
	LAST	

Sequences of operations of the same precedence are evaluated left to right.

Example:

In the following expression, the numbered pointers show the order of execution of operations:

```
¬ B1   || B2 & B3 | ¬ B4
↑       ↑     ↑   ↑  ↑
①       ②     ③   ⑤  ④
```

17.5 BIT STRING ASSIGNMENT

Bit string assignment is an extension of Boolean assignment as described in Section 8.4.

1. The operand types are both bit string:

L-type	*R*-type
BIT STRING	BIT STRING

2. The logical value of each bit position of the *R*-operand is transferred to the receiving data item.
3. If the operand differs in length from the receiving data item, the former is truncated or padded with binary zeroes on the left as appropriate.

Examples:

If B is an 8-bit string,

and BB is a 6-bit string with BB ≡ 101101_2’

then

```
B = BIN‘1111010110’;
```

results in B ≡ 11010110_2’

and

```
B = BB;
```

results in B ≡ 00101101_2

17.6 BIT STRINGS IN CONDITIONAL CONSTRUCTS

Execution of the HAL/S IF statement described in Section 9.3, and of the DO WHILE statement described in Section 10.2, are controlled by the logical value of an expression which was stated to be either Boolean or relational in type. Bit string expressions may not be used directly in place of Boolean expressions. This section will explain the method in which bit strings can be used.

DIRECT USE OF BIT STRINGS

The only way one can make use of a bit string for a Boolean expression is to subscript the bit string down to one bit, thereby making it a Boolean expression.

Examples:

Let B be a 4-bit string with B ≡ 1101_2

Let BB be a 2-bit string with BB ≡ 10_2

```
 IF B  THEN X = 0;
S    2
 ELSE X = 1;
```

The expression $B_2 \equiv 1_2$: since this is logically true X will be set to zero

```
 IF BB THEN X = 1;
 ELSE X = 2;
```

is illegal since BB has not been subscripted down to one bit.

BIT STRINGS IN RELATIONAL EXPRESSIONS

Section 9.2 showed how data items of each type, including Boolean, could be combined into relational expressions which evaluated to either TRUE or FALSE. Using the same nomenclature as that section, bit strings can be used in Class II comparative operations only:

Symbol	Purpose	Class
=	equals	
NOT =		II
¬ =	not equals	

The rules for bit string comparisons are given below:

Symbolic form L: $\left\{\begin{matrix} = \\ \text{NOT} = \\ \neg = \end{matrix}\right\} R$

1. The only legal type combination for the *L* and *R* operands is:

L-type	*R*-type
BIT STRING	BIT STRING

2. If the operands are of unequal length, the shorter is padded on the left with binary zeroes before comparison.

Examples:

If B is a 4-bit string with B ≡ 1101_2,

and BB is a 3-bit string with BB = 101_2,

then

`B = BIN‘01101’` is TRUE

and

`B = BB` is FALSE

The above comparative operations can be combined as described in Section 9.2, using the given precedence rules. Note that the important rule that Boolean and relational expressions cannot be mixed extends to bit string expressions as well.

Following are some examples clarifying the use of bit string relations.

Examples:

Let B be a 3-bit string with $B \equiv 110_2$,
and I be an integer with $I \equiv 5$

```
IF (B=BIN‘00110’) & (I>4) THEN I = 0;
```

In the above IF statement, both comparative operations evaluate to TRUE so that the condition is itself TRUE and the assignment

```
I = 0;
```

is executed.

```
IF (B ¬=BIN‘01’) & BIN‘11’ THEN I = 0;
```

is illegal because a relational expression is being mixed with a bit string literal to form the condition of the IF statement.

Note that

```
IF B¬=BIN‘01’ & BIN‘11’ THEN I = 0;
```

is illegal because the syntax is ambiguous. Parentheses must be used to specify its only legal interpretation:

```
IF B ¬=(BIN‘01’ & BIN‘11’) THEN I = 0;
```

17.7 BIT STRING ARGUMENTS AND PARAMETERS

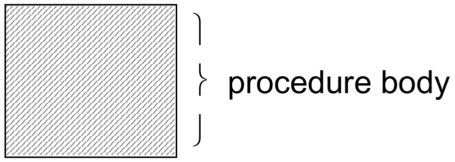
Section 11 described procedure and function blocks and how they were invoked. Procedures and functions may be defined with bit string parameters, and be passed bit string arguments.

FORM OF BIT STRING PARAMETERS

Any input parameter of a function, or any input or assign parameter of a procedure may be declared to be of bit string type, using the forms of declaration described in Section 17.2.

Example:

```
FLAGS: PROCEDURE(B1) ASSIGN(B2);
    DECLARE B1 BIT(16),
            B2 BIT(8);
```



```
CLOSE FLAGS;
```

ARGUMENT PASSAGE

An argument of a function or procedure invocation corresponding to a bit string parameter must conform to the following rules:

- INPUT PARAMETER. The transmission of the argument can be viewed as its assignment to the input parameter. The following rules apply:

1. The corresponding argument must be of bit string type.
2. If the input parameter is not of the same length as the argument, the latter is truncated or padded with binary zeroes on the left as necessary.

These rules apply to both procedures and functions.

- ASSIGN PARAMETER. The following rules apply for the matching of arguments to bit string assign parameters.

1. The assign argument must be a declared HAL/S bit string data item.
2. The length of the argument must be the same as that of the parameter.

   The argument may not possess subscripting.

These rules are only relevant to procedures.

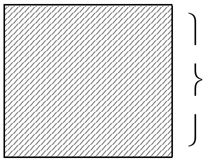
Examples:

Let the following data be declared:

```
DECLARE B1 BIT(16),
        B2 BIT(3);
```

and let the following procedure be defined:

```
 SWITCHES: PROCEDURE(D2) ASSIGN(D1);
     DECLARE D1 BIT(3),
             D2 BIT(8);
```



procedure body

```
 CLOSE SWITCHES;
```

Both legal and illegal invocations of this procedure are shown below:

```
CALL SWITCHES(B1|BIN‘1001’) ASSIGN B2;
              └─────┬─────┘
                    |
         this 16-bit quantity
          truncated to 8bits on passage
CALL SWITCHES (B2) ASSIGN(B1);
               ↑          ↑__ illegal length mismatch
 this 3-bit quantity padded
 to 8 bits on passage

CALL SWITCHES(BIN‘1’) ASSIGN(FALSE);
                                ↑
            illegal - not a declared bit string data item
```

17.8 BIT STRING FUNCTIONS

In Section 11.2 it was stated that functions of any legal HAL/S type could be created. Accordingly, it is legal to define functions of bit string type.

BLOCK DEFINITION

The opening statement of the function block takes the form:

label: FUNCTION(i^1,i^2,...) BIT(n);

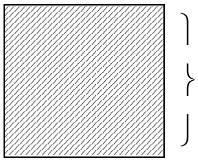
1. *label* is the name of the function.
2. i^1, i^2,... is the list of input parameters.
3. n indicates the number of bits, and lies in the range 1≤ n≤ 32[†].

[†] This number may vary between implementations. See appropriate User's Manual.

The closing statement is as described in Section 11.2.

Example:

```
  FUNCTION(B) BIT(5);
```



```
  CLOSE F1;
```

RETURN OF BIT STRING QUANTITIES

The RETURN statement of a bit string function follows the general function return form described in Section 11.6. The return is similar in nature to the transmission of input arguments of bit string type, the function itself playing the role of parameter. The relevant rules are the same as those described for argument passage in Section 17.7.

Examples:

```
  F1: FUNCTION(B) BIT(3);
      DECLARE B BIT(8);
        .
        .
        .
        .
      RETURN B;     ← truncation of 5 left-most bits occurs
        .
        .
        .
      RETURN B   ; ← result padded to 3 bits
S             4
        .
        .
        .
      RETURN 5.7E3;← illegal - bit string quantity not returned
        .
        .
        .
  CLOSE F1;
```

17.9 BIT STRINGS IN INPUT/OUTPUT

Bit strings may participate in input/output in the same way as other data types, as described in Section 12. The format of bit string data fields for input and output are described in Appendix F.

18.0 MULTI-DIMENSIONAL ARRAYS

Section 4.2 stated that it was possible to declare an array or table of any given data type. Section 4.2 showed the form of declaration for 1-dimensional arrays. HAL/S actually supports arrays of multiple dimensions.

First, the general form of declaration is presented. Then, some remarks on the order of initialization precede a discussion of the subscripting of multi-dimensional arrays.

18.1 DECLARATION

To declare an array of any data type and of any legal dimension, the following form of declaration is used:

DECLARE *name* ARRAY(n^1,n^2,...) *attributes;*

1. *name* is the name of the data item declared.
2. *attributes* are the attributes appropriate to the data type being declared.
3. n^i, $i = 1, 2$ ... are the sizes corresponding to each array dimension. The upper limit on i is 3[†]. The number of elements in any dimension must lie in the range $1 < n^i < 32768$[††].

[†] The limiting number of dimensions may vary between implementations: See appropriate User's manual.

[††] This value may vary between implementations. See the appropriate User's Manual. In some implementations, there may also be restrictions upon the contexts in which very large arrays may be used.

Examples:

```
DECLARE S ARRAY(5,5) INTEGER,
        V ARRAY(4) VECTOR(6),
        W ARRAY(2,2,1000) SCALAR;
```

18.2 ORDER OF INITIALIZATION

Section 4.3 stated the order of initialization of elements of 1-dimensional arrays of any data type. The order for multi-dimensional arrays is generated by the rules given in Appendix C.

The following examples illustrate the effect of these rules in the initialization of 2- and 3-dimensional arrays.

Example:

```
DECLARE I ARRAY(2,3) INTEGER INITIAL(1,2,3,4,5,6);
```

results in $I \equiv \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$

```
DECLARE J ARRAY(2,3,4) INTEGER
INITIAL(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,
        19,20,21,22,23,24);
```

$$\text{results in } J \equiv \begin{pmatrix} 1 & 2 & 3 & 4 & \quad & 13 & 14 & 15 & 16 \\ 5 & 6 & 7 & 8 & \quad & 17 & 18 & 19 & 20 \\ 9 & 10 & 11 & 12 & \quad & 21 & 22 & 23 & 24 \end{pmatrix}$$

18.3 SUBSCRIPTING

Section 6.2 gave the forms of array subscripting for 1-dimensional arrays. To summarize, the following kinds of subscript could be used:

- simple indexing, to select one array element;
- AT-partitioning, to select a sub-array of a given size starting from a given index value;
- TO-partitioning, to select a sub-array starting from one given index value and ending on a second.

In multi-dimensional arrays, such subscripting can be applied to each dimension of the array.

ARRAY SUBSCRIPTING ONLY

Let TABLE be an n-dimensional array. The general subscripting form is then:

$$\mathrm{TABLE}_{array^1, array^2, ...array^n:}$$

1. *arra*y stands for any array subscript of the form given in Section 6.2.
2. The colon is optional for integer and scalar data types only.
3. Any *arra*y may be replaced by an asterisk to denote specification of every element in that dimension.

Examples:

If I is a 2 x 3 array of integers

$$\text{with } I \equiv \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

then

$I_{1,2} \equiv 2$

$I_{2,1 \text{ TO } 2} \equiv (4 \;\; 5)$

$I_{*,3} \equiv (3 \;\; 6)$

$I_{*,*:}$ with I $\equiv$ $\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$ note redundant colon

ARRAY AND COMPONENT SUBSCRIPTING

If TABLE represents an n-dimensional array of vector, matrix, character or bit string type, then the general form when component and array subscripting is present is:

$\mathtt{TABLE}_{\mathit{array}^1,\mathit{array}^2,\ldots\mathit{array}^n:\mathtt{component}}$

1. *arra*y stands for any array subscript of the form given in Section 6.2.
2. *component* represents any form of component subscripting legal for the data type of TABLE, as described in Section 6.1 and 17.3.
3. Any *arra*y may be replaced by an asterisk to denote specification of every element in that dimension.

Examples:

If C is a 2 x 3 array of characters

with C $\equiv$ $\begin{pmatrix} \text{`ALPHA'} & \text{`BETA'} & \text{`GAMMA'} \\ \text{`DELTA'} & \text{`EPSILON'} & \text{`ZETA'} \end{pmatrix}$

then

$C_{1,2:}$ ≡ 'BETA' note mandatory colon

$C_{1,\ 2:\ 1\ \mathrm{TO}\ 3}$ ≡ ‘BET’

$C_{*,\ 3:4}$ ≡ (‘M’ ‘A’)

COMPONENT SUBSCRIPTING ONLY

When only component subscripting is required, array subscripting cannot be totally omitted, but must rather be replaced with asterisks. If, as before, TABLE represents an n-dimensional array of vector, matrix, character or bit string type, then the general form is:

$\mathtt{TABLE}_{*,\ldots *:}\ \mathit{component}$

1. n asterisks correspond to n dimensions of absent array subscripting.
2. *component* represents any form of component subscripting legal for the data type of TABLE.

Example:

If C is a 2 x 3 array of characters

with C $\equiv$ $\begin{pmatrix} \text{`ALPHA'} & \text{`BETA'} & \text{`GAMMA'} \\ \text{`DELTA'} & \text{`EPSILON'} & \text{`ZETA'} \end{pmatrix}$

then

with $C_{*,*:1}$ $\equiv$ $\begin{pmatrix} \text{`A'} & \text{`B'} & \text{`G'} \\ \text{`D'} & \text{`E'} & \text{`Z'} \end{pmatrix}$

Literal subscripts may alternatively be expressions computable at compile time.

See: Guide/Appendix D.

For a complete description of all subscript forms see Spec./5.3.

19.0 STRUCTURES

Section 4.1 of the Guide introduced some of the types of data definable in the HAL/S language. It further made reference to the fact that “hierarchical organizations of data items’’ exist in the language. It is the purpose of this Section to describe the form and use of these so-called “structures” data.

The HAL/S array feature is a useful construct for forming aggregates of data items, if they are homogeneous in attributes. Frequently, however, it is of great convenience to be able to form aggregates of data items with heterogeneous attributes. In addition, requirements may exist to reference not only the aggregate as an entity, but also subsets of it, or subsets of subsets of it. The HAL/S STRUCTURE data type fulfills both of these requirements.

19.1 HAL/S STRUCTURE CONCEPTS

HAL/S data structures have two characteristic properties:

- Data items or arrays of almost any type can be combined to form a structure.
- The data items can be organized into a tree-like hierarchy (similar in concept to a genealogical tree, for example).

The following diagram illustrates in concept the form of a typical structure tree.

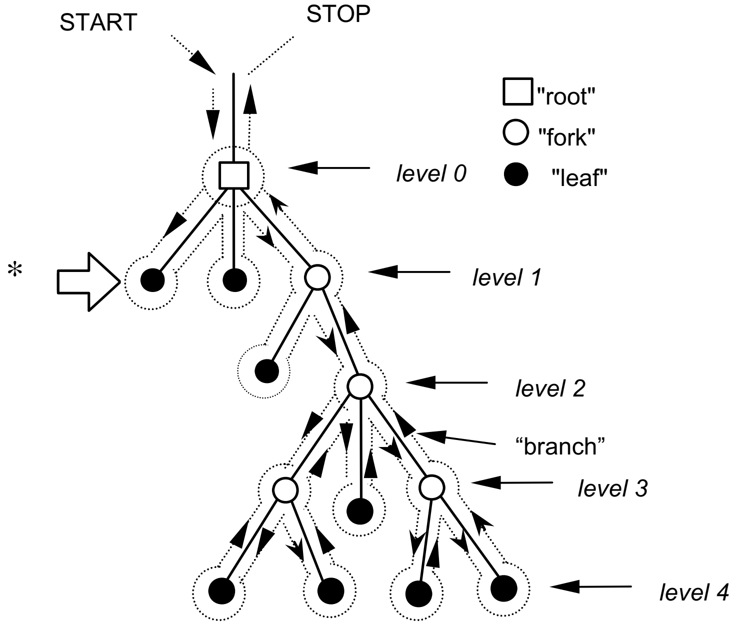


Figure 19-1

The tree consists of nodes connected by "branches". Every "leaf" node of the tree corresponds to one of the actual data items making up the aggregate. The whole tree can be referenced by using the name of the "root" node. Subsets of the tree can be referenced by using the name of the appropriate "fork" node. The dotted line is a "tree walk" which forms the basis for converting this tree representation into a linear list representation which the HAL/S language itself has to use.

The conversion consists of recording the name of each node (root, fork or leaf) and its level when the tree walk passes it in the direction shown by the arrow at *.

Example:

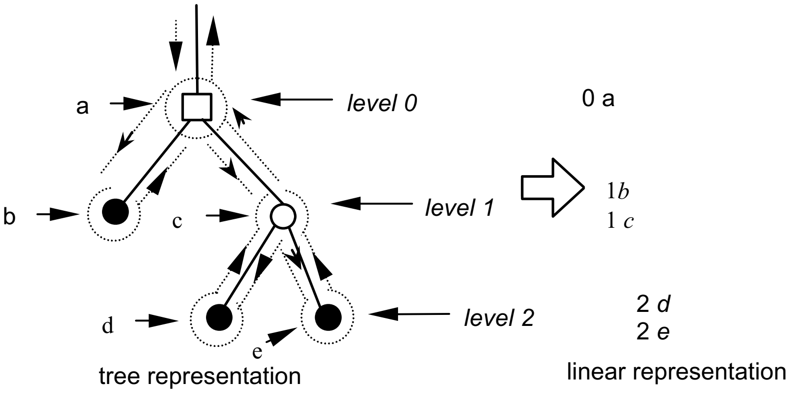


Figure 19-2

The reverse conversion consists of the following steps. First draw the "root" node appearing at the top of the list. Then, treat each of the remaining nodes in order as follows.

- Draw the node to the right of previous node with the same level number (if any), and under nodes with smaller level numbers.
- Connect it by a "branch" to the last-connected node with a level number one smaller.

Example:

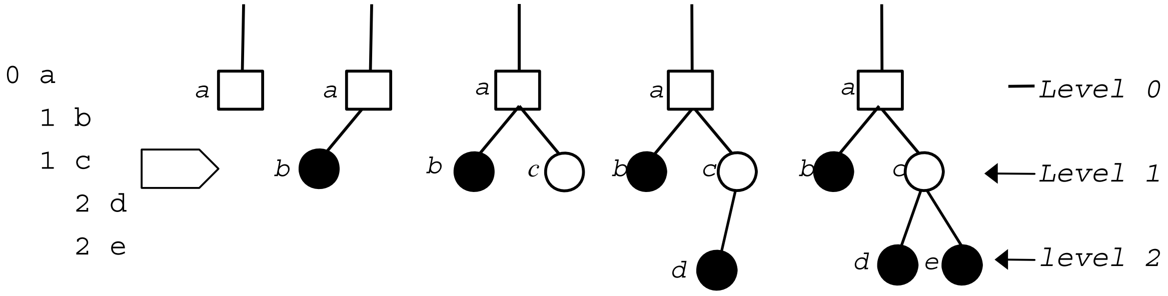


Figure 19-3

In the HAL/S language, the specification of a structure tree organization is separated from the declaration of the structure or structures possessing that organization.

- STRUCTURE TEMPLATES are used to specify structure tree organizations in a linear list representation. A structure template specifies all nodes in a tree from level 1 downwards.
- STRUCTURE DECLARATIONS are used to declare structures possessing predefined templates. For reasons which will become apparent when the referencing of structures is considered, the declared name of the structure is assigned as the "root" node name of the tree organization.

In the remainder of the section, structures will be referred to as data items, since even though they are aggregates of data items, they can be manipulated as entities in themselves.

19.2 STRUCTURE TEMPLATES

The structure template is the HAL/S construct which defines the structure tree organization in the form of a linear list. It defines by name and level all "fork" and "leaf" nodes in a tree from level 1 downwards.

In the HAL/S implementation of structure trees, the following nomenclature is used.

- TEMPLATE NAMES are names identifying structure templates. They appear as part of the template specification, and also in structure declarations.
- MINOR STRUCTURE NODES are the "fork" nodes of a structure template.
- STRUCTURE TERMINALS are the "leaf" nodes of a structure template. Every structure terminal is one of the data items comprising the structure aggregate.

GENERAL FORM OF A TEMPLATE

The form of a structure template consists of its name followed by a specification of all its minor structure and structure terminal nodes.

- OVERALL FORM

The overall form is as follows:

STRUCTURE *name*:
  node^1, *node*^{2}, ...
... *node*^{n};

1. *name* is the structure template name, and is any legal HAL/S identifier name.
2. *node*^{1}, *node*^{2},...*node*^{n} is a list of nodes forming the tree organization.

- MINOR STRUCTURE NODES

  The form of a minor structure node of a template is as follows:

n *name*

1. *n* is the level number of the node.
2. *name* is the name of the minor structure node, and may be any legal identifier name.

- STRUCTURE TERMINAL NODES

  The form of a structure terminal node of a template is as follows:

n *name attributes*

1. n is the level number of the node.
2. *name* is the name of the structure terminal node, and may be any legal identifier name.
3. *attributes* consists of array, type, size and other attributes applicable to data items.
4. The following data types are legal as structure terminals:

INTEGER
SCALAR
VECTOR
MATRIX
BOOLEAN
BIT STRING
CHARACTER
STRUCTURE

Note that in the case of a scalar structure terminal no attributes need appear[24]. However, there is no confusion as to whether the node is a structure terminal or a minor structure since the level number sequence is sufficient to distinguish the two cases. Structure terminals of structure type are a special case which is discussed later.

RESTRICTIONS

The attributes attached to the specification of a structure terminal node are written in the same form and order as in a declaration statement (as described in Section 4 and expanded in Sections 16, 17.2, and 18.1). The following restrictions are however made.

- No INITIAL/CONSTANT specification can be applied to a structure terminal.
- No STATIC/AUTOMATIC specification can be applied to a structure terminal.

[24] See the comment on the declaration of scalar data items in Section 4.

Example:

```
STRUCTURE Q:
        1 QT CHARACTER(80),
        1 QN1,
          2 QI INTEGER,
          2 QV VECTOR(3) DOUBLE,
          2 QS ARRAY(100) SCALAR,
        1 QN2,
          2 QV VECTOR(3) DOUBLE,
          2 QS ARRAY(100) SCALAR,
          2 QM MATRIX(3,3),
          2 QB BOOLEAN;
```

The above structure template corresponds to the following tree organization:

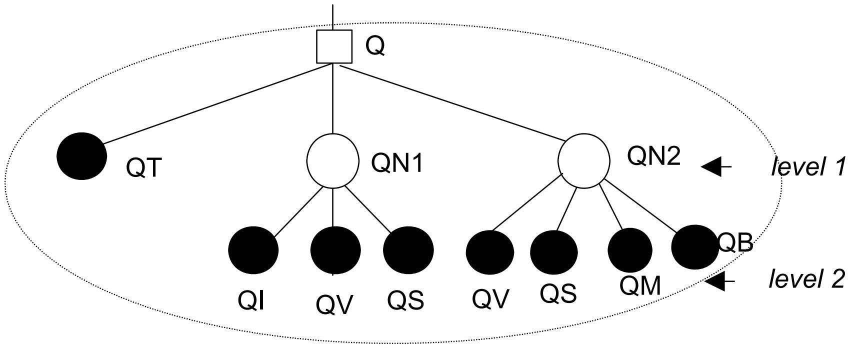


Figure 19-4

LOCATION OF STRUCTURE TEMPLATES

Structure templates are essentially parts of data declarations and therefore must appear before the first executable statement of the program or other block in which they are coded.

19.3 STRUCTURE DECLARATIONS

Structure declarations are used to declare structure data with a tree organization defined by a pre-existing structure template. Structure declarations are in the same general form as declarations of other kinds of data items, as described in Section 4.

BASIC FORM OF DECLARATION

The basic form of structure declaration is shown below:

`DECLARE` *name* α `STRUCTURE;`

1. *name* is the name of the structure data item, and may be any legal identifier name.
2. α is the name given to a pre-existing structure template which specifies the tree organization of the structure being declared.

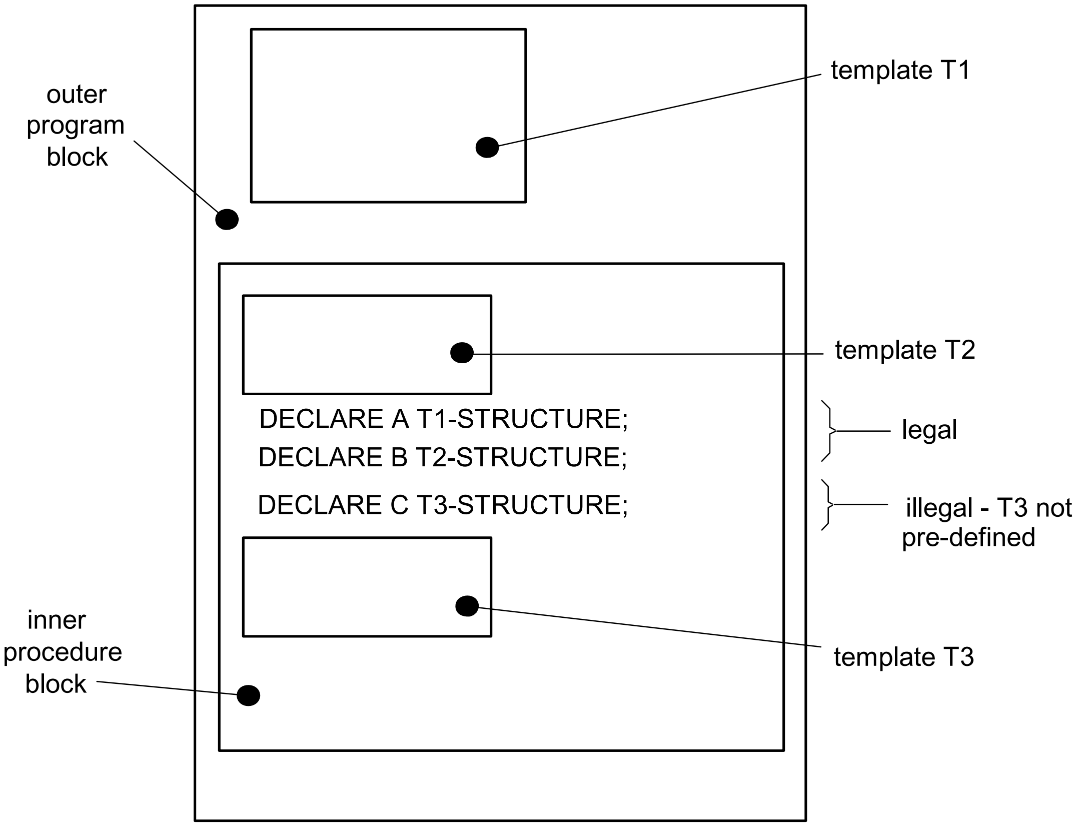
Note that the structure template referenced by a structure declaration must have been defined previously in the same block, or have been declared in a block enclosing the block containing the STRUCTURE declaration.

Examples:

form of declaration -

```
STRUCTURE Q:
        1 QA SCALAR,
        1 QB CHARACTER(80),
        1 QC BOOLEAN;
.
.
.
DECLARE ZZ1 Q-STRUCTURE;
DECLARE ZZ2 Q-STRUCTURE;
```

legal and illegal placing of templates -



Structure declarations can be integrated into compound declarations of the kind described in Section 4.2.

Figure 19-5

Example:

```
DECLARE A SCALAR,
        B Q-STRUCTURE,
        C CHARACTER(80);
```

MULTIPLE COPY STRUCTURES

Structures can be declared to have multiple copies of the data specified by the tree organization. Although the form of specification is different from HAL/S arrays, they can in some contexts be viewed as arrays of structures.

The data declaration for a multiple-copy structure takes the following modified form:

DECLARE *name* α-STRUCTURE(n);

1. *name* is the name of the structure.
2. α is the name of the predefined structure template.
3. n is the number of copies of the data required. It must lie in the range
   $1 < n < 32768$[†].

[†] This value may vary between implementations. See appropriate User's Manual.

INITIALIZATION OF STRUCTURES

Structures are initialized by supplying an INITIAL/CONSTANT specification with the structure declaration, rather than with the template. The specification is added to the declaration as described in Section 4.3.

Example:

```
STRUCTURE Q:
        1 QI INTEGER,
        1 QS SCALAR;
.
.
.
DECLARE Z Q-STRUCTURE INITIAL(5,4.3);
```

The order of initialization for structures is as follows.

- SINGLE-COPY STRUCTURES. The number of literal values in the initial list (or implied by the use of repetition factors) must equal the total number of elements summed over all the structure terminal nodes. Each structure terminal is initialized in the order it appears in the structure template, according to the rules given in Section 4.3 and further expanded in Sections 16 and 18.2.
- MULTIPLE-COPY STRUCTURES. The number of literal values in the initial list may either match the total number of elements summed over all copies, or match the number in one copy, in which case all copies are identically initialized. Each copy is initialized in turn in order of increasing index, according to rules for single-copy structures.

These ordering rules are a restatement of those given in Appendix C.

Example:

```
STRUCTURE Q:
        1 QV VECTOR(3),
        1 QM,
           2 QI INTEGER,
           2 QC CHARACTER(80);
 .
 .
 .
 DECLARE Z1 Q-STRUCTURE INITIAL(1.5,2.5,3.5,-2,'ALPHA');
 DECLARE Z2 Q-STRUCTURE(2) INITIAL(4.5,5.5,6.5,-4,'BETA');
 DECLARE Z3 Q-STRUCTURE(2) INITIAL(3#1.5,1,'GAMMA',
                                   3#2.5,2,'DELTA');
```

The above declarations result in initialization as follows:

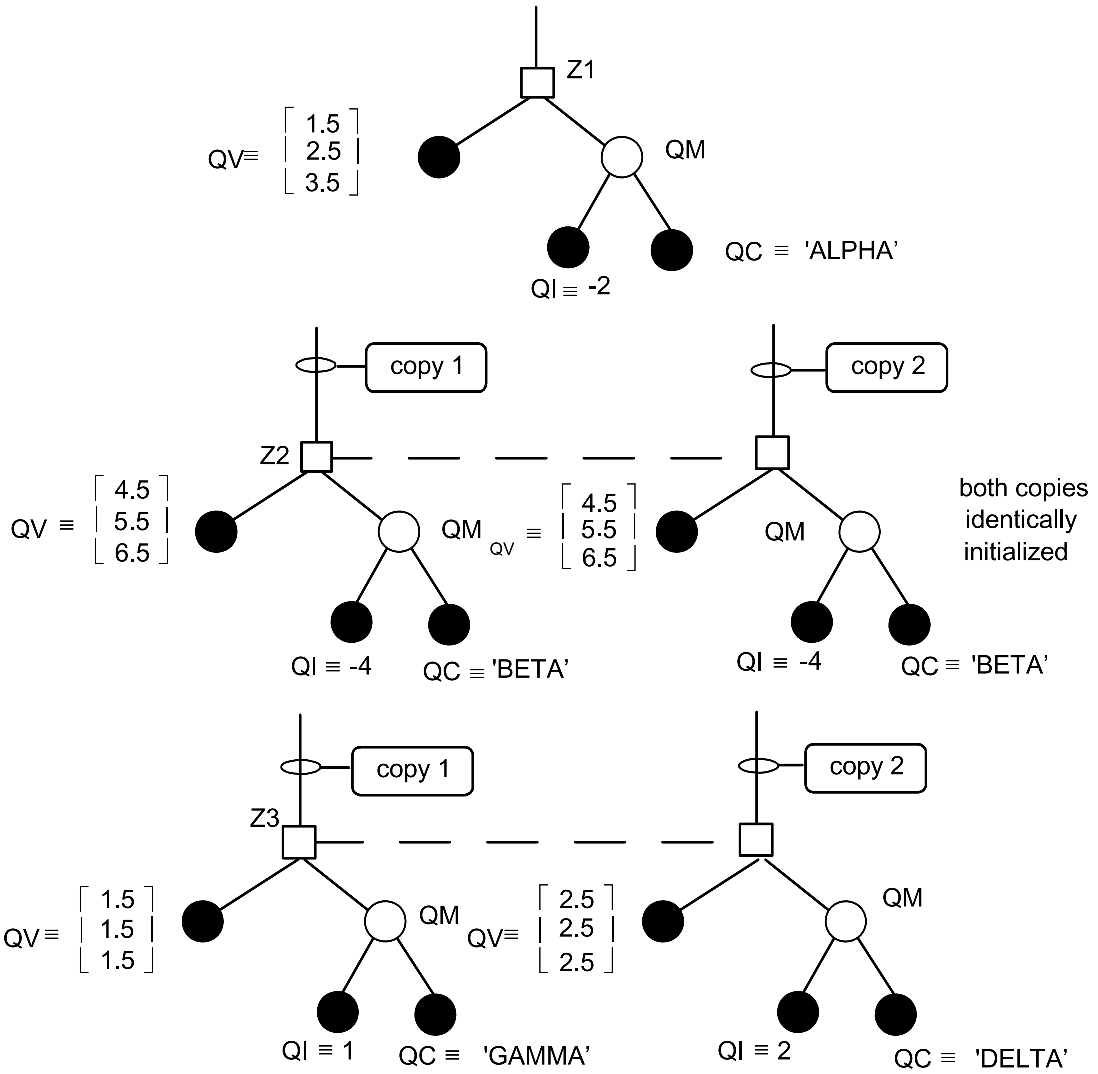


Figure 19-6

The supplementary initialization forms described in Section 16 are fully applicable to structure data types.

19.4 NESTED STRUCTURES

Section 19.2 stated that structure terminal nodes could themselves be of structure type. The effect of this is to nest a second template into the first, thus expanding the tree organization of the former.

Example:

```
STRUCTURE A:
    1 AI INTEGER,
    1 A1,
      2 AC CHARACTER(80),
      2 AB BOOLEAN;
STRUCTURE B:
    1 BS SCALAR,
    1 B1,
      2 BV VECTOR(3),
      2 BA A-STRUCTURE;
```

In a tree representation, this expressible is as:

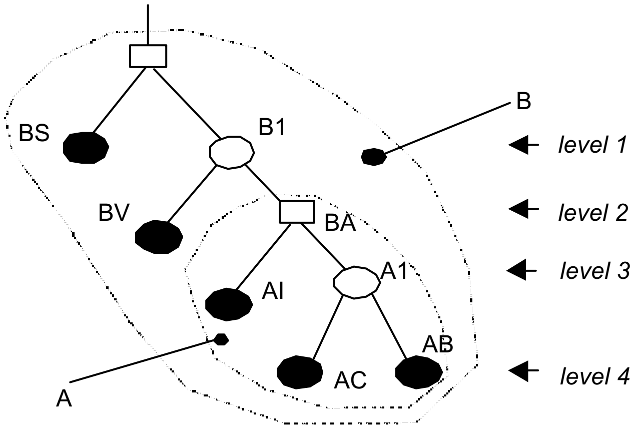


Figure 19-7

The structure template B is in many aspects like a template C given by:

```
STRUCTURE C:
        1 BS SCALAR,
        1 B1,
           2 BV VECTOR(3),
           2 BA,
              3 AI INTEGER,
              3 A1,
                4 AC CHARACTER(80),
                4 AB BOOLEAN;
```

which has superficially the same tree organization.

RESTRICTION

A structure terminal of structure type may not possess multiple copies.

Example:

The following template is illegal:

```
STRUCTURE Q:
        1 QI INTEGER,
        1 QS T-STRUCTURE(20);
```

19.5 QUALIFICATION AND STRUCTURE REFERENCING

The basic types of data item introduced in Section 4 are referenced merely by stating their names in the desired context. A structure in its entirety can be referred to in the same way. Referring to part of a structure is more complex, however, because in general more than one structure may possess the tree organization expressed by a particular template.

THE QUALIFIED REFERENCE CONCEPT

Any node of a structure other than the "root'' node is referred to by a composite or "qualified" name which is generated conceptually in the following way. Consider the tree organization:

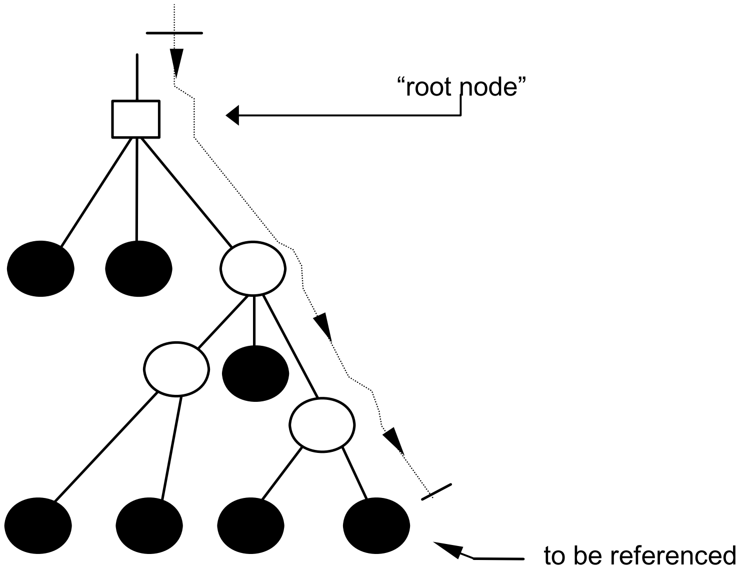


Figure 19-8

A tree walk is started at the "root" node, and continued down to the node to be referenced. The names of all the nodes traversed, including the "root" and final nodes, are listed. The resulting composite or "qualified" name is an unambiguous reference to the desired "leaf" node (given certain restrictions on duplicate naming which are to be described).

REFERENCING STRUCTURE TERMINALS

The qualified name of a structure terminal is generated by catenating the names of all nodes between the “root” node and the desired “leaf” node of the tree organization.

name^1•*name*^{2}•.....*name*^{n}

1. *name*^{1} is the name of the structure as declared.
2. *name*^{n} is the name of the structure terminal to be referenced.
3. *name*^{2},.....*name*$^{n-1}$ are the names of intervening minor structure nodes, if any.

Examples:

```
STRUCTURE Q:
        1 QI INTEGER,
        1 Q1,
           2 QS SCALAR,
           2 QC CHARACTER(80);
  .
  .
  .
DECLARE ZQ Q-STRUCTURE;
```

To reference QI and QC in ZQ, the following tree walks are required:

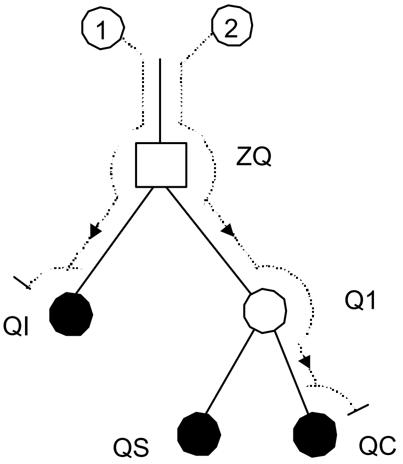


Figure 19-9

They generate the following names respectively:

```
ZQ.QI      ... ①
ZQ.Q1.QC   ... ②
```

REFERENCING MINOR STRUCTURE NODES

If it is required to perform an operation on a sub-tree of a structure (i.e. all parts of the tree beneath a certain "fork" node), the occasion arises to refer to a minor structure node name. The qualified name is generated by catenating the names of nodes between the "root'' node and the desired "fork" node.

$name^1 \bullet name^2 \bullet ... name^n$

1. *name*^{1} is the name of the structure as declared.
2. *name*^{n} is the name of the minor structure node to be referenced.
3. *name*^{2},...*name*$_{n-1}$ are the names of intervening minor structure nodes, if any.

Example:

```
STRUCTURE Q:
        1 QI INTEGER,
        1 Q1,
           2 QS SCALAR,
           2 QC CHARACTER(80);
  .
  .
  .
DECLARE ZQ Q-STRUCTURE;
```

To reference Q1 in ZQ, the following tree walk is required:

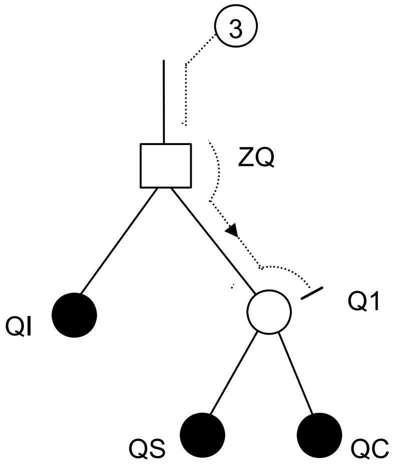


Figure 19-10

It generates the following name:

```
ZQ.Q1     … ③
```

NAMING UNIQUENESS

The node names used in a structure tree specification need only be unique in so far as all tree walks used to generate qualified names must be distinguishable. This means that some node names may actually duplicate others without error.

Examples:

```
STRUCTURE Q:
       1 Q1,
         2 QS SCALAR,
                    ↑
                    |---------   legal duplicate names
       1 Q2,        ↓
         2 QS SCALAR,
.
.
.
 DECLARE ZR Q-STRUCTURE;
```

The above duplicate names are legal because qualified references to each are distinguishable:

```
ZR.Q1.QS
ZR.Q2.QS
```

```
STRUCTURE R:
       1 R1,        ←------------┐
        2 RS SCALAR,             |  ----  illegal duplicate names
       1 R1 CHARACTER(80);←──────┘
.
.
.
 DECLARE ZR R-STRUCTURE;
```

The above duplicate names are illegal. ZR.R1 might be referring to a minor structure node or a structure terminal of character type.

The following situations are also permitted:

- The name of minor structure or terminal node may duplicate the name of any minor structure or terminal node in a different structure template.
- The name of a minor structure or terminal node may duplicate the name of any ordinary data item.

UNQUALIFIED REFERENCES

Qualified referencing of parts of structures can become laborious if the node names assigned are long, or there are many levels in the structure. By accepting certain restrictions, unqualified, or direct naming of minor structure or terminal nodes is permissible.

To be able to refer to a structure in an unqualified manner the following must apply:

> Unqualified reference may be made only to a structure whose name is the same as the template defining its tree organization.

It follows that only one unqualified structure may be declared for any template.

Examples:

```
STRUCTURE Q:
        1 QI INTEGER,
        1 Q1,
           2 QS SCALAR,
           2 QC CHARACTER(80);
 .
 .
 .
DECLARE ZQ Q-STRUCTURE;
DECLARE Q Q-STRUCTURE;
```

QC in ZQ must be referred to as:

```
ZQ.Q1.QC
```

QC in Q may be referred to simply as:

```
QC
```

More restrictive rules apply to the construction of a structure template used to declare an unqualified structure.

- The name of each node in the template must be unique to the block in which the template is defined.
- The template must be defined in the same block as the unqualified structure is itself declared.
- The template may contain no structure terminals of structure type (i.e. nested structures).

19.6 SUBSCRIPTING IN STRUCTURES

A structure terminal may possess "terminal" subscripts as a result of its type (vector, matrix, character, bit string) or its array property. In addition, any reference to the whole or part of a structure with multiple copies can introduce a level of "structure" subscripting.

The discussion on subscripting is divided into two parts:

- subscripting on references to the entire structure or to minor structure nodes;
- subscripting on references to terminal data items.

SUBSCRIPTING DATA ITEMS OF STRUCTURE TYPE

A reference to an entire structure or to one of its minor structure nodes may possess subscripting only if the structure is declared to possess multiple copies.

In the subscripting forms below, TREE represents any data item of structure type (i.e., either a "root" or "fork" node of the structure tree), the reference being unqualified or qualified. It is assumed that the entire structure is declared to possess L copies.

- To select the α^{th} copy from TREE:

$$\texttt{TREE}_{\alpha;}$$

1. α is an integer expression in the range $1 \leq \alpha \leq L$.
2. The semicolon is optional.

- To select a subset of α copies starting from the β^{th} copy of TREE:

$$\texttt{TREE}_{\alpha\ \texttt{AT}\ \beta;}$$

1. α is an integer literal value in the range $1 \leq \alpha \leq L$.
2. β is an integer expression in the range $1 \leq \beta \leq L - \alpha + 1$.
3. The semicolon is optional.

- To select a subset of copies starting from the α^{th} copy and ending with the β^{th} copy of TREE:

$$\texttt{TREE}_{\alpha\ \texttt{TO}\ \beta;}$$

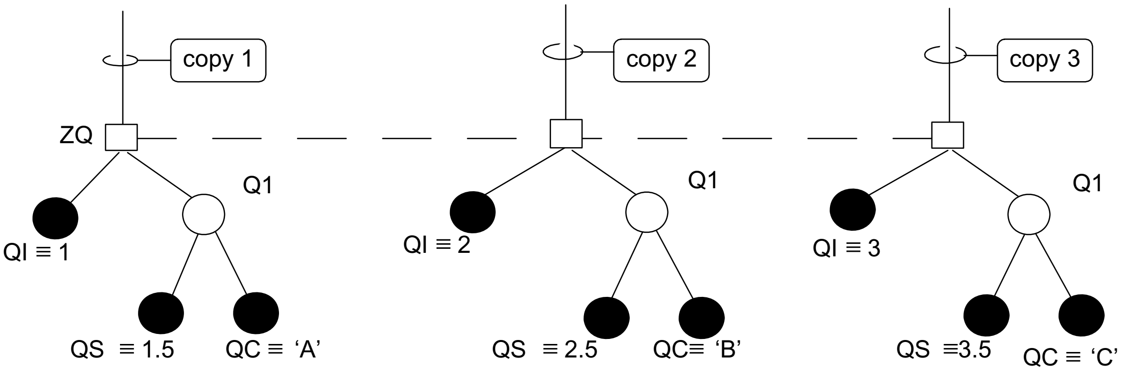
1. α and β are integer literal values in the range $1 \leq (\alpha,\beta) \leq L$.
2. $\beta \underline{>} \alpha$
3. The semicolon is optional.

Examples:

Given:

```
STRUCTURE Q:
        1 QI INTEGER,
        1 Q1,
          2 QS SCALAR,
          2 QC CHARACTER(80);
 .
 .
 .
DECLARE ZQ Q-STRUCTURE(3);
```

with the following values:



then ZQ$_{2;}$ selects copy 2 with values:

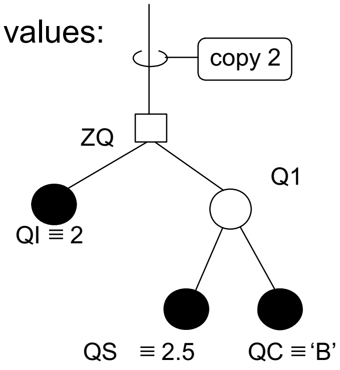
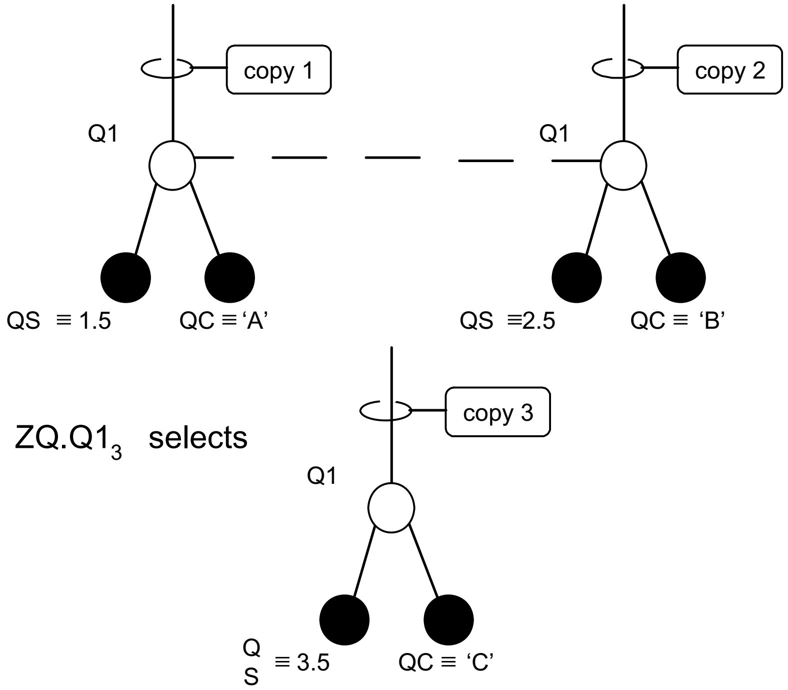


Figure 19-11

ZQ.Q1$_{1\ TO\ 2;}$ selects copies 1 and 2 of the sub-tree under Q1



ZQ.Q1$_3$ selects

Note the omission of the semicolon.

Figure 19-12

SUBSCRIPTING OF TERMINALS OF MULTIPLE-COPY STRUCTURES

If a structure terminal is part of a single copy structure, then it can possess subscripting only by virtue of its type or array property. Such subscripting is the same as for ordinary data items, and has been described in Sections 6, 17.3, and 18.3.

If, on the other hand, a structure terminal is part of a multiple copy structure then it may possess subscripting by virtue of its type or array property, and by virtue of the multiple copy property. Three cases of subscripting thus arise:

- STRUCTURE SUBSCRIPTING ONLY. The form of subscripting is the same as for structure data items, as described above. The only difference is that the terminating semicolon is optional only if the structure terminal is of integer or scalar type, and unarrayed.
- STRUCTURE AND TERMINAL SUBSCRIPTING. The structure subscripting takes the same form as before. Terminal subscripting (consequent on type or arrayness) follows the mandatory semicolon, and takes the forms described in Sections 6, 17.3 and 18.3.
- TERMINAL SUBSCRIPTING ONLY. The subscript forms are the same as in the previous case except that the structure subscript is replaced by an asterisk.

Examples:

Given the single-copy structure

```
STRUCTURE Q:
        1 QV VECTOR(3),
        1 Q1,
           2 QB ARRAY(2) BIT(4),
           2 QM MATRIX(3,3);
 .
 .
 .
DECLARE ZQ Q-STRUCTURE;
```

with the following values:

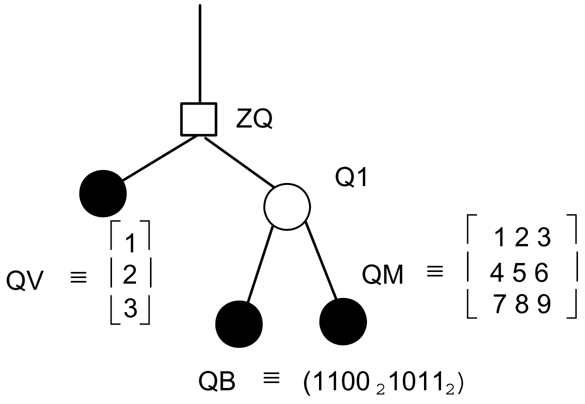


Figure 19-13

then

$$ZQ.QV_1 \equiv 1$$

$$ZQ.Q1.QB_{*:3} \equiv (0_2\ 1_2)$$

$$\text{and } ZQ.Q1.QM_{2\ TO\ 3,\ 2\ TO\ 3} \equiv \begin{bmatrix} 5 & 6 \\ 8 & 9 \end{bmatrix}$$

Note: The asterisk for the structure subscript is absent because there is only one copy.

Further, given the multiple-copy structure

```
DECLARE YQ Q-STRUCTURE(3);
```

with the following values:

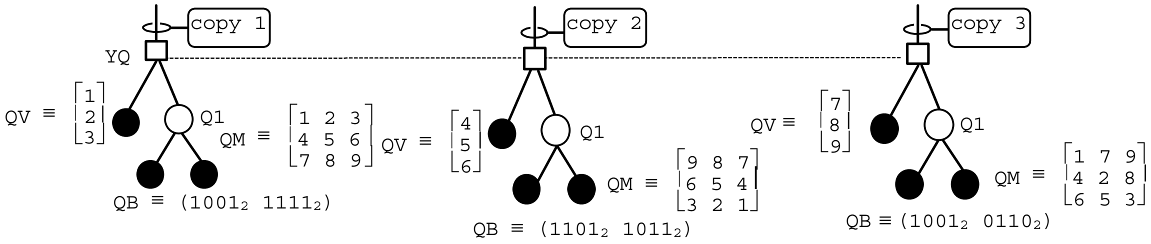


Figure 19-14

then

copy 1 copy 3

$YQ.QV_{*;3} \equiv (3 \quad 6 \quad 9)$ result is scalar type

copy 2

$YQ.Q1.QB_{2;*:1\ TO\ 2} \equiv (11_2 10_2)$

array property unmodified

$YQ.Q1.QM_{2;3,3} \equiv 1$

Figure 19-15

Literal subscripts may alternatively be expressions computable at compile time.

See: Guide/Appendix D.

19.7 TREE EQUIVALENCE OF STRUCTURES

Most operations involving more than one operand of structure type require their operands to possess tree organizations which are in most respects identical. Two structures which are compatible in this sense are said to be "tree-equivalent". Two basic requirements have to be satisfied to establish tree-equivalence:

- the actual shape of the trees must be equivalent;
- the attributes of corresponding structure terminal nodes must be the same.

EQUIVALENCE OF TREE SHAPE

The equivalence of tree shape can be achieved in a number of different ways:

- USE OF SAME TEMPLATE - If two structures are declared using the same template, they cannot avoid meeting both requirements for tree-equivalence.

Example:

```
STRUCTURE Q:
        1 QI INTEGER,
        1 Q1,
          2 QS SCALAR,
          2 QC CHARACTER(80);
DECLARE ZQ1 Q-STRUCTURE,
       ZQ2 Q-STRUCTURE(20);
```

ZQ1 and ZQ2 are tree-equivalent, (notwithstanding the mismatch in number of copies).

- USE OF TEMPLATE OF SAME SHAPE - If two structures are declared using distinct templates which do, however, have the same shape, then the first requirement of tree-equivalence is met.

Example:

```
STRUCTURE Q:
        1 QI INTEGER,
        1 Q1,
          2 QS SCALAR,
          2 QC CHARACTER(80);
DECLARE ZQ Q-STRUCTURE;
 .
 .
 .
STRUCTURE R:
        1 RI INTEGER,
        1 R1,
          2 RS SCALAR,
          2 RC CHARACTER(80);
DECLARE ZR R-STRUCTURE;
```

The tree shapes of ZR and ZQ are the same:

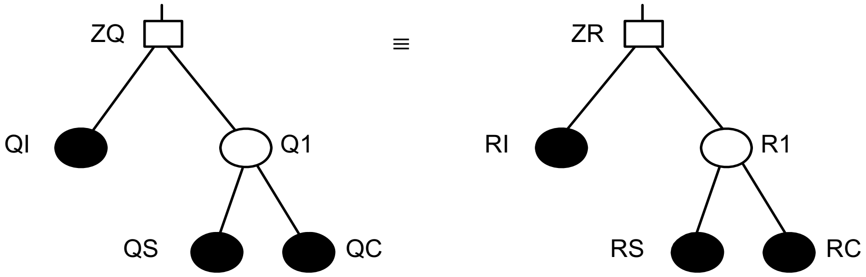


Figure 19-16

- MATCHING OF SUB-TREES - If the tree shape of a sub-tree of one structure matches the same of another structure, or sub-tree thereof, then the first requirement of tree-equivalence is met.

Example:

```
STRUCTURE Q:
        1 QI INTEGER,
        1 Q1,
          2 QS SCALAR,
          2 QC CHARACTER(80);
DECLARE ZQ Q-STRUCTURE;
  .
  .
  .
STRUCTURE R
          1 RS SCALAR,
          1 RC CHARACTER(80);
DECLARE ZR R-STRUCTURE;
```

- The tree shapes of ZQ and ZR clearly are not the same. However, the tree shapes of ZQ.Q1 and ZR <u>are</u> the same:

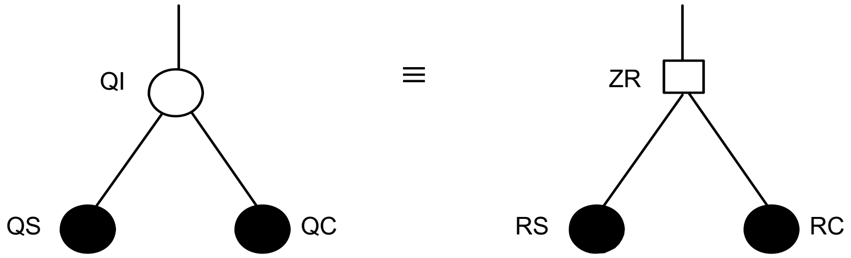


Figure 19-17

MATCHING OF TERMINAL NODE ATTRIBUTES

Once matching of tree shape has been established, to obtain tree-equivalence, corresponding structure terminal nodes of each tree must be verified as having identical attributes. Generally, terminal nodes must match exactly in their type and array property (if any). Additionally, for each type the following matching requirements must be met:

TYPE	MATCHING REQUIREMENTS
BIT STRING	Number of bits (BOOLEAN is equivalent to BIT(1))
CHARACTER	Maximum declared length
INTEGER	Precision
SCALAR	Precision
VECTOR	Precision, length
MATRIX	Precision, row and column dimensions
STRUCTURE	Specified structure template

Examples:

```
STRUCTURE Q:
        1 QI INTEGER,
        1 Q1,
          2 QM MATRIX(3,3),
          2 QC CHARACTER(80);
DECLARE ZQ Q-STRUCTURE;
.
.
.
STRUCTURE R:
        1 RI INTEGER DOUBLE,
        1 R1,
           2 RM MATRIX(3,3),
           2 RC CHARACTER(80);
DECLARE ZR R-STRUCTURE;
```

- ZQ fails to be tree-equivalent to ZR solely due to one precision mismatch: ZQ.QI is single precision, while ZR.RI is double precision.
- However, ZQ.Q1 is completely tree-equivalent to ZR.R1 since the offending terminal node is not present.

Note that the matching requirement for terminal nodes of structure type preclude tree-equivalence in cases typified by the following example:

Example:

```
 STRUCTURE Q:
         1 QS SCALAR,
         1 QC CHARACTER(80);
 STRUCTURE R:
        1 RI INTEGER,
        1 RQ Q-STRUCTURE;
 DECLARE ZR R-STRUCTURE;
 STRUCTURE S:
     1 SI INTEGER,
1 S1,
        2 SS SCALAR,
        2 SC CHARACTER(80);
 DECLARE ZS-S-STRUCTURE;
```

ZS is not tree-equivalent to ZR although their tree organizations are superficially alike (see Section 19.4). ZS would be tree-equivalent to ZR only if the template S had been specified as:

```
STRUCTURE S:
        1 SI INTEGER,
        1 SQ Q-STRUCTURE;
```

Where structure templates are declared with additional attributes such as RIGID, DENSE, LOCK, etc., matching extends to these also.

See Spec./4.3 and 4.5.

19.8 STRUCTURE ASSIGNMENTS

Values of one structure data item[25] may be transferred to another in a body using a structure assignment. Structure assignments have the same general form as other assignments: this form has been described in Section 8.1.

BASIC FORM

As applied to structures, the rules become:

Symbolic form: *L = R;*

1. *L* is the receiving structure data item. It may possess structure subscripting.
2. *R* is either a second structure data item, subscripted or not, or alternatively a structure function (see Section 19.11).
3. *L, R* must be tree-equivalent in the sense described in Section 19.7.

Examples:

Given:

```
STRUCTURE Q:
        1 QI INTEGER,
        1 Q1,
         2 QS SCALAR,
         2 QC CHARACTER(80);
DECLARE ZQ1 Q-STRUCTURE;
DECLARE ZQ2 Q-STRUCTURE(2);
```

where ZQ2 has the values:

[25] Unless specifically stated in Sections 19.8 through 19.12, a structure data item may either be a declared structure, or a minor structure node.

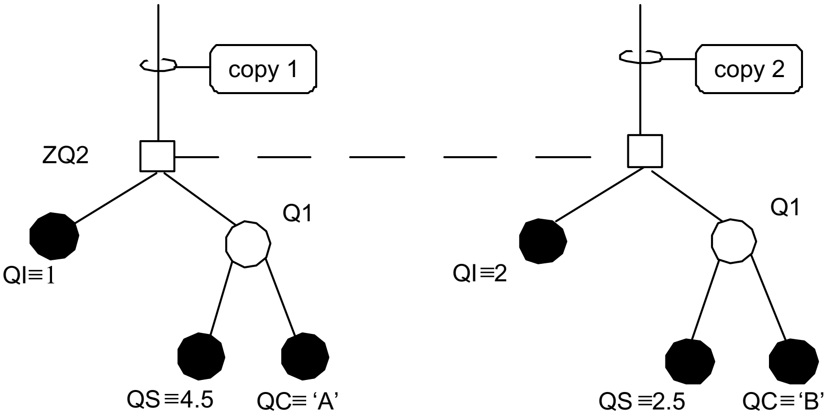


Figure 19-18

then

```
  ZQ1 = ZQ2  ;
S          2
```

results in ZQ1 having the values:

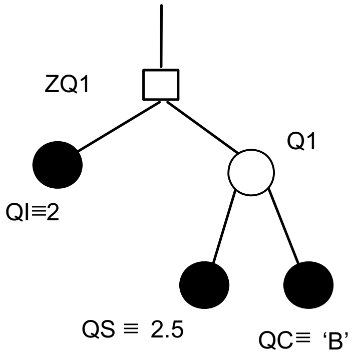


Figure 19-19

and if then the following is executed

```
  ZQ1.Q1= ZQ2.Q1 ;
S               1
```

the values of ZQ1 are modified to:

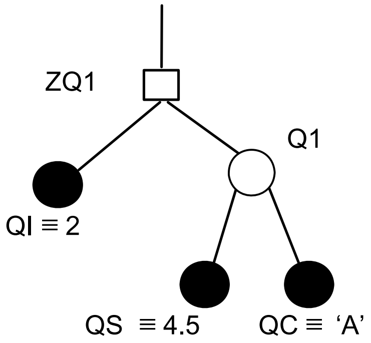


Figure 19-20

MULTIPLE ASSIGNMENTS

Several structure data items may be assigned values at one assignment by the following construction first presented in Section 8.5:

Symbolic form: $L^1,L^2,L^3,....L^n = R;$

1. L^1, ...L^n are receiving structure data items.
2. Any *L* must be tree-equivalent to the *R* structure operand.
3. No particular order of assignment is assumed.

Examples:

Given:

```
STRUCTURE Q:
   1 QI INTEGER,
   1 QS SCALAR;
DECLARE Q-STRUCTURE ZQ1, ZQ2, ZQ3;
```

then

```
ZQ1, ZQ2 = ZQ3;
```

assigns the values of ZQ3 to ZQ1 and ZQ2.

19.9 STRUCTURES IN CONDITIONAL CONSTRUCTS

Relational expressions appear in the IF statement described in Section 9.1 and the DO WHILE statement described in Section 10.2. Such expressions may contain comparative operations with structure operands.

Using the same nomenclature as in Section 9.2, structures can be used in Class II comparative operations only:

Symbol	Purpose	Class
=	equals	II
NOT =	not equals	
¬ =		

The rules for structure comparisons are:

=
Symbolic form: *L* NOT = *R*
¬ =

1. The *L* and *R* operands are either structure data items or structure functions (see Section 19.11).
2. The operands must be tree-equivalent.
3. Two structures are equal if, and only if, all corresponding terminals have equal values.

Example:

Given:

```
STRUCTURE Q:
          1 QI INTEGER,
          1 QS SCALAR;
DECLARE ZQ1 Q-STRUCTURE,
          ZQ2 Q-STRUCTURE;
```

with values of ZQ1 and ZQ2 given by

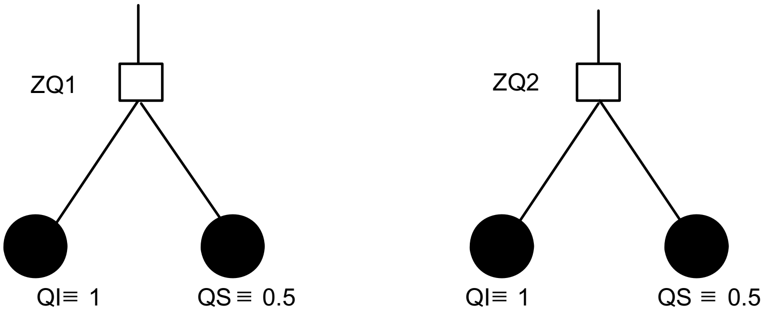


Figure 19-21

then ZQ1 = ZQ2 is TRUE.

19.10 STRUCTURE ARGUMENTS AND PARAMETERS

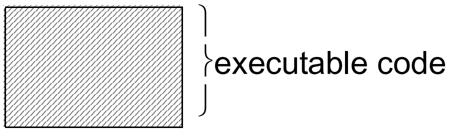
HAL/S procedures and functions may be defined with structure parameters, and be passed structure arguments.

FORM OF STRUCTURE PARAMETERS

Any parameter of a function, or any input or assign parameter of a procedure, may be declared to be a structure using the forms of declaration described in Section 19.3.

Example:

```
ANALYZE: PROCEDURE(S1) ASSIGN(S2);
  STRUCTURE S:
       1 SI INTEGER,
       1 SN,
           2 SS SCALAR,
           2 SC CHARACTER(80);
  DECLARE S1 S-STRUCTURE,
              S2 S-STRUCTURE;
```



```
  CLOSE ANALYZE;
```

Observe the position of the structure template.

ARGUMENT PASSAGE

Any argument of a function or procedure invocation corresponding to a structure parameter must conform to the following rules:

- INPUT PARAMETER. The transmission of the argument can be viewed as its assignment to the input parameter. The following rules apply:

1. The corresponding argument must be a structure data item or a structure function.
2. The argument and parameter must be tree-equivalent.

These rules apply to both procedures and functions.

- ASSIGN PARAMETER. The following rules apply for matching of arguments to structure assign parameters.

1. The assign argument must be a structure data item.
2. The argument and parameter must be tree-equivalent.
3. The argument may only be subscripted if it is a declared structure as opposed to a minor structure, and only then if the subscript reduces the number of copies to one.

These rules are only relevant to procedures.

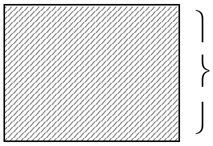
Examples:

Let the following be declared:

```
STRUCTURE Q:
        1 QI INTEGER,
        1 Q1,
          2 QS SCALAR,
          2 QC CHARACTER(80);
STRUCTURE R:
        1 RS SCALAR,
        1 RC CHARACTER(80);
DECLARE ZQ Q-STRUCTURE,
         ZR R-STRUCTURE,
         YQ Q-STRUCTURE(10);
```

and let the following procedure be defined:

```
 TREE: PROCEDURE(D1) ASSIGN(D2);
     DECLARE D1 R-STRUCTURE,
             D2 Q-STRUCTURE;
```



procedure body

```
 CLOSE TREE;
```

Both legal and illegal invocations of this procedure are shown below.

```
 CALL TREE(ZR) ASSIGN(ZQ);
 CALL TREE(ZR) ASSIGN(YQ );
S                       4
 CALL TREE(ZQ.Q1) ASSIGN(ZQ);
 CALL TREE(ZR) ASSIGN(ZR);
```

illegal - no tree-equivalence

19.11 STRUCTURE FUNCTIONS

In HAL/S, user functions may return a structure result type. Such functions can be used instead of structure data items in many of the structure operations described above.

Structure functions follow similar patterns for their block definitions and invocations as given in Section 11 for other data types.

BLOCK DEFINITION

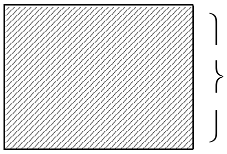
As usual, the block is opened with a characteristic opening statement, of the form:

label: FUNCTION(i^1,i^2,...)α-STRUCTURE;

1. *label* is the name of the function.
2. i^1, i^2, ... is the list of input parameters. The entire parenthesized list may of course be omitted.
3. α is the name of the template describing the tree organization of the function. The template must be defined in a block visible (according to usual HAL/S scoping rules) to the opening statement. Note in particular that the template cannot be defined in a group of declaration statements inside the function.

Example:

```
STRUCTURE Q;
    1 Q1 INTEGER,
    1 Q1,
      QS SCALAR,
      QS CHARACTER(80);
.
.
.
TREE: FUNCTION(I,J) Q-STRUCTURE;
```



function body

```
CLOSE TREE;
```

RETURN OF STRUCTURE QUANTITIES

The RETURN statement of a structure function follows the general form described in Section 11.6. The return is similar to the transmission of structure input arguments, the function itself playing the role of parameter. The relevant rules are the same as those described for the passage of input arguments, as given in Section 19.10.

Examples:

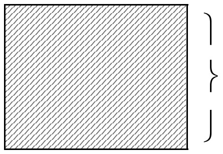
```
STRUCTURE S:
  1 SS SCALAR,
  1 SC CHARACTER(80);
STRUCTURE Q:
  1 QI INTEGER,
  1 Q1 S-STRUCTURE;
.
.
.
TREE: FUNCTION(D1) S-STRUCTURE;
   DECLARE D1 Q-STRUCTURE;
   .
   .
   .
   RETURN D1.Q1;
   .
   .
   .
   RETURN D1;

          illegal - lack of tree-equivalence
   .
   .
   .
CLOSE TREE;
```

INVOCATION OF STRUCTURE FUNCTIONS

A structure function is invoked in the same way as a function of any other data type, as described in Section 11.4. It should be noted, however, that the function may only be referenced as a whole. No reference, qualified or unqualified may be made to minor structure or terminal nodes of its tree.

Example:

```
STRUCTURE Q:
       1 QI INTEGER,
       1 Q1,
          2 QS SCALAR,
          2 QC CHARACTER(80);
DECLARE ZQ Q-STRUCTURE;
TREE: FUNCTION Q-STRUCTURE;
```



function body

```
CLOSE TREE;
.
.
.
ZQ = TREE;          legal invocation
ZQ.Q1 = TREE.Q1;    illegal invocation
```

19.12 STRUCTURES IN INPUT/OUTPUT

Structures may participate in input/output in the same way as other data types, as described in Section 12.

- In single-copy structures, values of the terminal nodes are transmitted in the order they are given in the structure template.
- In multiple-copy structures, values for one copy are completely transmitted before proceeding to the next, each copy being treated as for a single-copy structure.

These ordering rules are a restatement of the rules given in Appendix C.

The formats of the data fields are typically as given in Appendix F for each data type.

OUTPUT

Values of any structure data item (either declared structure or minor structure node), or of any structure function may be output.

INPUT

Values of any structure data item may be input.

Example:
Given:

```
STRUCTURE Q:
   1 QI INTEGER,
   1 Q1,
     2 QS SCALAR,
     2 QC CHARACTER(80);
DECLARE ZQ Q-STRUCTURE;
```

and the input data stream:

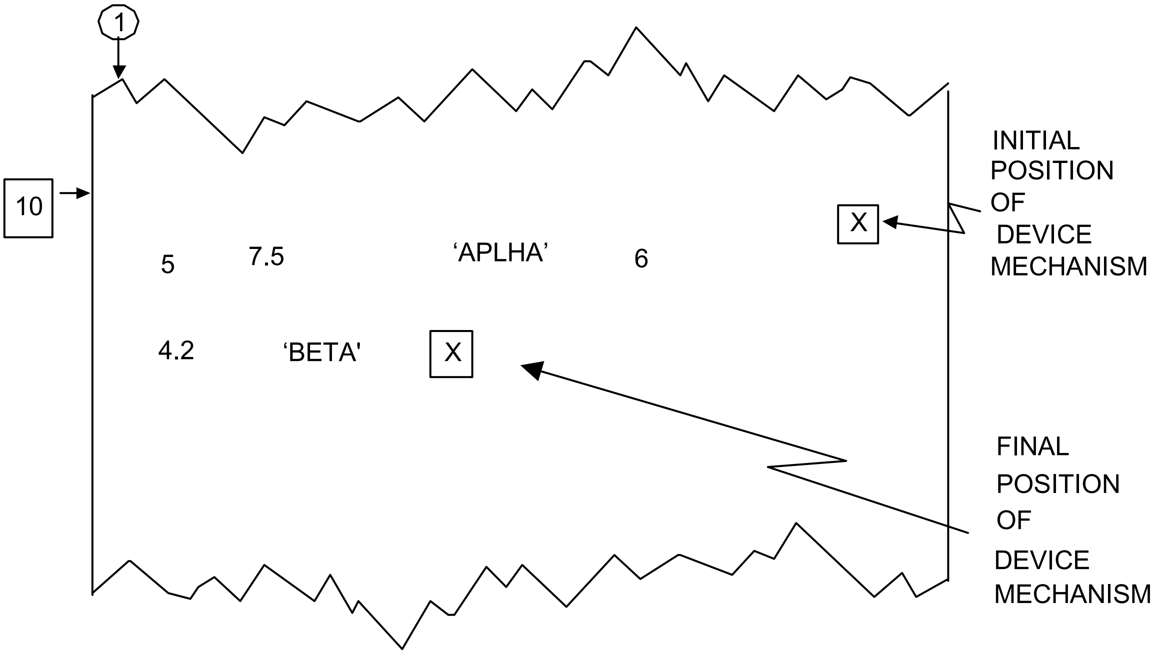

1
10
5
7.5
'APLHA'
6
X
4.2
'BETA'
X
INITIAL POSITION OF DEVICE MECHANISM
FINAL POSITION OF DEVICE MECHANISM

Figure 19-22

then:

```
READ(5) ZQ;
```

results in ZQ being given the following values:

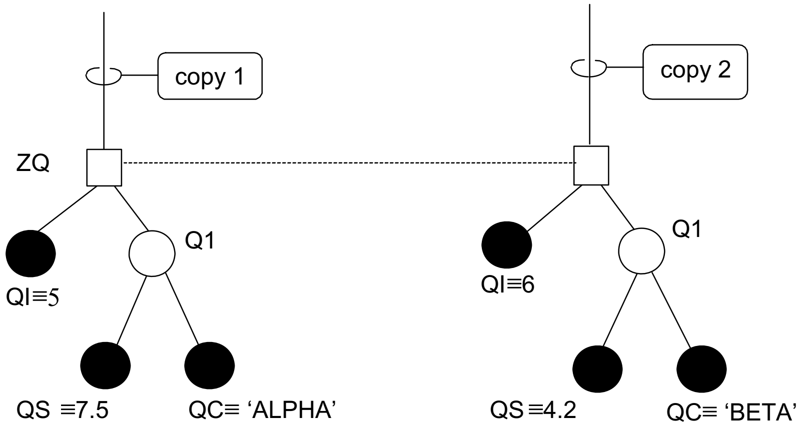


Figure 19-23

20.0 HAL/S ARRAY PROCESSING FEATURE

The concept of a HAL/S array of any data type has already arisen. Section 4 introduced one-dimensional arrays, and Section 18, multi-dimensional arrays. However, other than describing the subscript forms relevant to arrayed data items, no attempt has yet been made to catalog the ways they can be used.

There are occasions where a programming requirement exists to perform some transformation on all elements of an array. One realization of the algorithm for implementing the transformation consists of using the HAL/S DO FOR statement to index through the array, carrying out the transformation element by element. A much more compact and elegant HAL/S realization[26] consists of using "arrayed" expressions and assignments in which the operands are generally arrays, and which look representational as if the operations were being carried out in parallel on all array elements, rather than serially element by element.

Of course, with the current generation of machines, in most implementations the parallelism is only an illusion created at the source language level. Because of this, it is not necessarily more efficient to use arrayed expressions and assignments rather than DO FOR statements.

This section states the rules governing the construction of arrayed expressions, and describes how they are used in various language constructs. Understanding of Sections 18 and 19 is a prerequisite for this section.

20.1 THE ARRAYNESS OF OPERANDS

An operand is said to be arrayed if either or both of the following statements are true:

- it is a declared array item;
- it is a node (root, fork or leaf) of a structure with multiple copies
- (see Section 19).

QUANTITATIVE ARRAYNESS

The "arrayness" of an operand is a quantitative description of its array property:

- the number of dimensions;
- the size of each dimension in order.

In the remainder of this section, the arrayness of any operand will be expressed in the following general form:

$\{N: n^1, ...n^N\}$

where N is the number of dimensions,

n^i is the size of the i^{th} dimension for $1 \leq i \leq N$.

Arrayness arising from either of the sources stated above are indistinguishable as far as the constructs to be described are concerned.

[26] In FORTRAN, a sequential realization, using the DO statement is the best possible realization.

The following examples illustrate how the arrayness of various kinds of operands are derived.

Examples:

```
STRUCTURE Q:
        1 QI INTEGER,
        1 QS ARRAY(3) SCALAR;
DECLARE ZQ Q-STRUCTURE(4);
  .
  .
  .
DECLARE I ARRAY(4) INTEGER;
DECLARE S ARRAY(4,3) SCALAR;
```

The arrayness of S is {2:4,3}, i.e. 2 dimensions of size 4 and 3 respectively.

The arrayness of I is {1:4}

The arrayness of ZQ and ZQ.QI are both {1:4}.

The arrayness of ZQ.QS is {2:4,3} -

> Note that the part of the arrayness due to multiple structure copies is placed <u>before</u> that due to the ARRAY specification.

The arrayness of $S_{1,*}$ is {1:3}

The arrayness of $ZQ.QS_{\underbrace{1\ TO\ 2}_{\text{2 copies selected}};\ \underbrace{2\ AT\ 2}_{\text{2 array elements selected}}}$ is {2:2,2}

$ZQ.QS_{1;1}$ has no arrayness - one element in one copy has been selected by the subscript.

MATCHING OF ARRAYNESSES

Two operands have matching arrayness if, and only if, the quantitative arraynesses are identical in <u>all</u> respects.

Example:

```
DECLARE A1 ARRAY(2,3,4),
        A2 ARRAY(2,3,2);
```

The arrayness of A1 and A2 are {3:2,3,4} and {3:2,3,2} respectively. They differ because the sizes of the rightmost dimension differ. The arrayness of $A1_{*,\ *,\ 1\ TO\ 2}$ is {3:2,3,2} which <u>does</u> match that of A2.

20.2 ARRAYED EXPRESSIONS

In HAL/S, an arrayed expression is one whose result is an array. It may be one-dimensional or multi-dimensional, and of any type - arithmetic, character, or bit string.

Arrayed expressions are constructed precisely according to the rules given in Section 7, and expanded for bit strings in Section 17.4. The sole difference is that one or more of the operands possess arrayness.

The following rules govern the usage of arrays in expressions:

1. Any operand may either possess arrayness or not.
2. All operands with arrayness must match in their arrayness in the sense described in Section 20.1.

Evaluation of the expression can be viewed as a set of elemental evaluations, each proceeding in parallel with the others. Each elemental evaluation selects a unique combination of array index values out of the total possible given the arrayness common to all arrayed operands, and uses it throughout the entire evaluation. Unarrayed operands take part in all of the elemental evaluations.

Pictorially, the evaluation of an unarrayed expression may be represented typically thus:

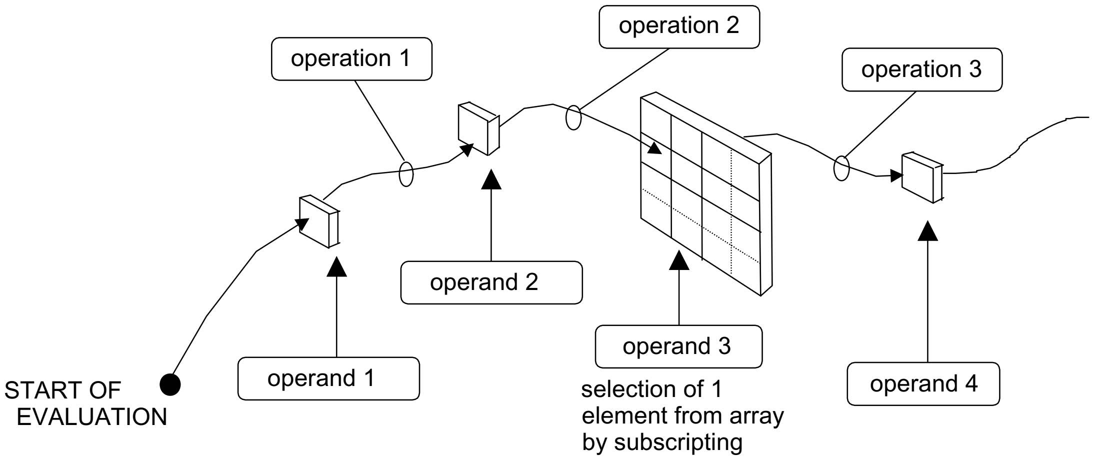


Figure 20-1

By comparison, the parallel evaluation of a typical arrayed expression can be represented pictorially (for two dimensions) thus:

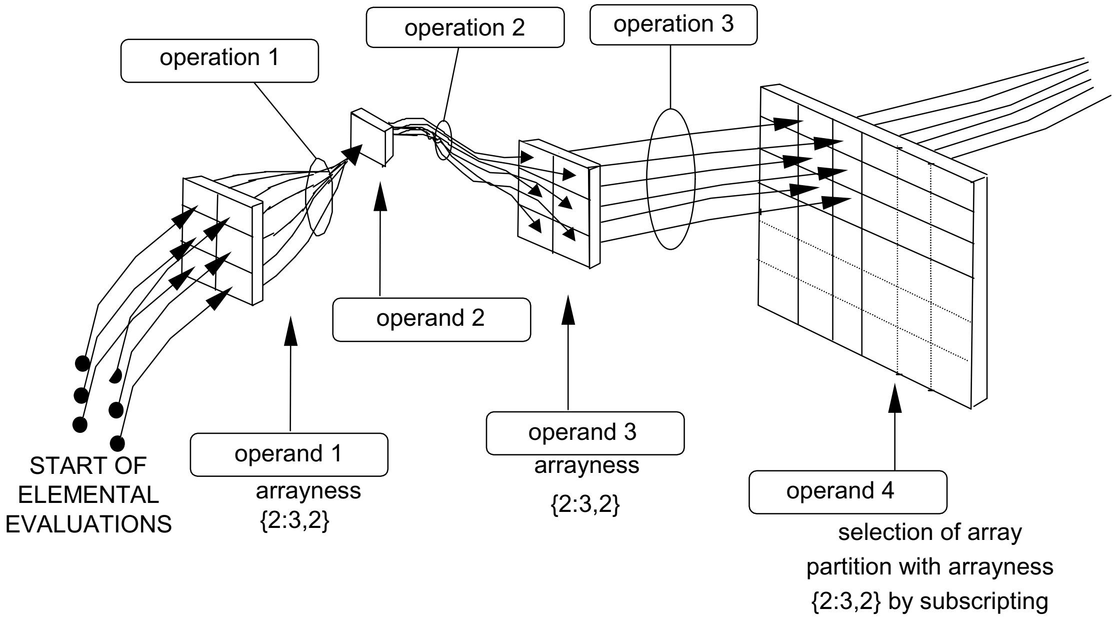


Figure 20-2

Examples:

Given:

```
DECLARE INTEGER,
        I1 ARRAY(2,3),
        I2 ARRAY(4);
```

with I1 ≡ $\begin{pmatrix}5 & 3 & 4\\ 6 & 2 & 1\end{pmatrix}$ and I2 ≡ (7 3 1 5)

then `I1 + I1 - 2` is an operand expression equivalent in effect to

$\texttt{I1}_{i,j} + \texttt{I1}_{i,j}\ \texttt{-2}$ for $1 \leq i \leq 2,\ 1 \leq_j \leq 3$

and its resulting value is $\begin{pmatrix}8 & 4 & 6\\ 10 & 2 & 0\end{pmatrix}$

Further `I1 + I2 - 1` is an illegal arrayed expression since the arraynesses of I1 and I2 are {2:2,3} and {1:4} respectively.

However, $\texttt{I1}_{1,*} + \texttt{I2}_{2\ TO\ 4}\ \texttt{-1}$ is legal since subscripting has caused the arraynesses of both operands to be {1:3}. Its result is (7 3 8).

If, in addition:

```
STRUCTURE Q:
   Q1 INTEGER ARRAY(3),
   QS SCALAR;
DECLARE ZQ Q - STRUCTURE(2);
```

with

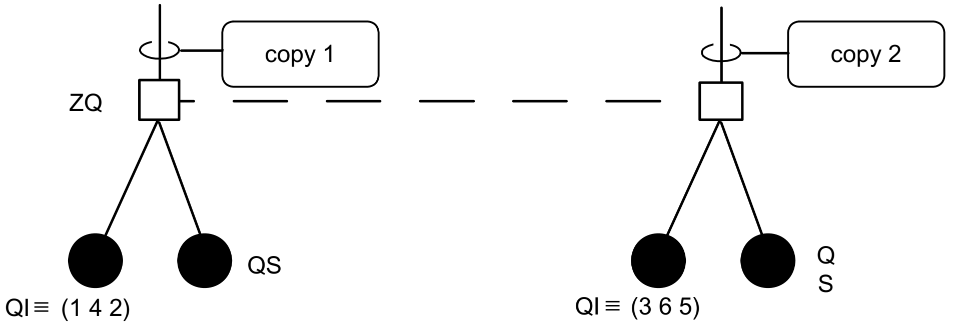


Figure 20-3

then ZQ.Q1 - I1 + 2 is legal since both arrayed operands have arrayness{2:2,3} and the result is a 2 by 3 array with values:

$$\begin{pmatrix}-2 & 3 & 0\\ -1 & 6 & 6\end{pmatrix}$$

BEHAVIOR OF BUILT-IN FUNCTIONS

Library or "built-in" functions may appear in arrayed expressions, with arguments with or without arrayness. Most of the built-in functions described in Appendix B, unless stated there to the contrary, are subject to the following rules in such contexts.

- A function with no arguments or with unarrayed arguments may either be evaluated once only, or once during every elemental evaluation of the expression in which the function is invoked, depending on the implementation. For most built-in functions, the difference is immaterial since repeated invocations with the same argument usually cause the same result to be returned. Some exceptions are[27]:

```
  RANDOM       CLOCKTIME
 RANDOMG       RUNTIME
```

- A function with one argument is treated as if it were a prefix operation upon its argument. When the argument is arrayed, the function is evaluated once during every elemental evaluation of the expression containing it.

[27] See Appendix B for a complete list of functions anomalous in this respect.

Example:

Given:

```
DECLARE X ARRAY(4) SCALAR;
```

then

`X + SIN(X)`

is equivalent to

X_i + SIN(X_i) for $1 \leq i \leq 4$,

the expression, including the sine function being evaluated 4 times.

- A function with two arguments is treated as if it were an infix operation upon its arguments. When one or both arguments are arrayed, the function is evaluated once during every elemental evaluation of the expression containing it.

Examples:

Given:

```
DECLARE INTEGER,
        I1 ARRAY(3,4),
        I2 ARRAY(3,4),
        I3 ARRAY(4);
```

then

`I1 + DIV(I2,5)`

is equivalent to

$I1_{i,j}$ + DIV($I2_{i,j}$, 5) for $1 \leq i \leq 3$, $1 \leq j \leq 4$

Note that

`DIV(I1,I2) - I3`

is not a legal expression because the arrayness of I3 is {1:4} which does not match those of I1 and I2, which are {2:3,4}.

20.3 ARRAYED ASSIGNMENTS

HAL/S permits the receiving data item of an assignment to be arrayed. The expression to be assigned to such a data item may be arrayed or unarrayed. The rules applicable to each of these cases are as follows:

- UNARRAYED EXPRESSION. The assignment can be viewed as a set of elemental assignments proceeding in parallel, each one selecting a different element of the receiving data item into which to place the single result of the expression. Pictorially, this may be represented typically (in two dimensions) thus:

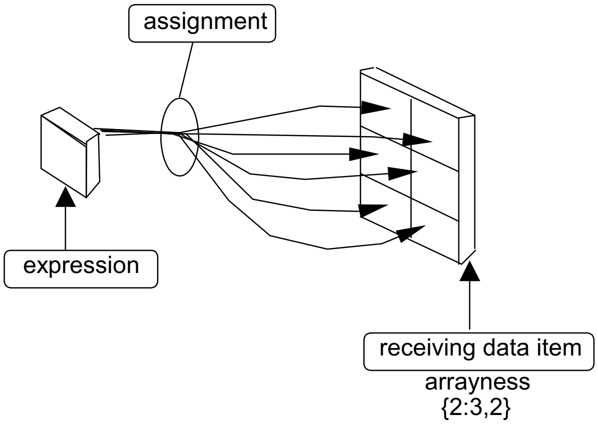


Figure 20-4

- ARRAYED EXPRESSION. The assignment can be viewed as a set of elemental assignments proceeding in parallel, each one selecting a different element of the receiving data item into which to place the result of the corresponding elemental expression evaluation. Pictorially, this may be represented typically (in two dimensions) thus:

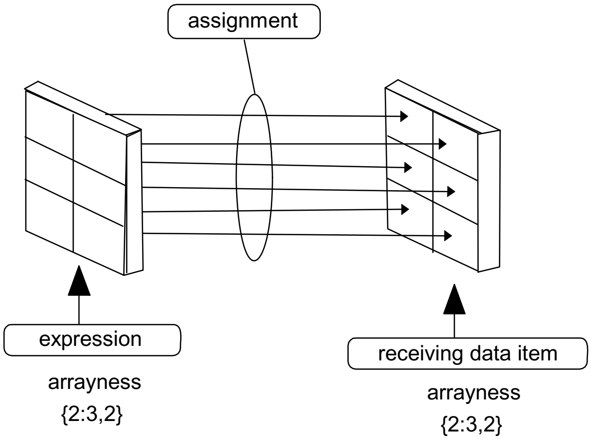


Figure 20-5

The following rules therefore govern a simple arrayed assignment:

1. The expression to be assigned may be arrayed if and only if the receiving data item is arrayed.
2. If the expression is arrayed, its arrayness must match that of the receiving data item.

Examples:

Given:

```
DECLARE INTEGER,
        I1 ARRAY(2,3),
        I2,
        I3 ARRAY(2,3),
        I4 ARRAY(4);
```

then

```
I2 = I1;
```

is an arrayed assignment in which all elements of I1 are assigned the value of I2.

```
I1 = I3;
```

assigns each element of I3 to the corresponding element of I1.

```
I1 = I4;
```

is illegal because the arrayness of the receiving data item is {2:2,3} while that of the right hand side is {1:4}.

```
I2 = I1;
```

is illegal because the right hand side has arrayness while the receiving data item has none.

Further given:

```
STRUCTURE Q:
          1 QI INTEGER,
          1 Q1,
            2 QS ARRAY(4) SCALAR,
            2 QC CHARACTER(80);
DECLARE ZQ1 Q-STRUCTURE(2);
DECLARE ZQ2 Q-STRUCTURE(2);
DECLARE S ARRAY(2,4) SCALAR;
```

the following assignments are legal:

```
ZQ1 = ZQ2;
ZQ1.Q1 = ZQ2.Q1;
ZQ1.Q1.QS = ZQ2.QS;
ZQ1.Q1.QS = S;
```

MULTIPLE ASSIGNMENTS

In assignments which have multiple receiving data items, the following extra rule is required:

3. If one receiving data item possesses arrayness, then all must possess matching arrayness.

Examples:

Given:

```
DECLARE INTEGER,
        I1 ARRAY(2,3),
        I2,
        I3 ARRAY(4),
        I4 ARRAY(2,3);
```

then

```
I1, I4 = I2;
```

is legal since the arrayness of I1 and I4 match.

However, both of the following are illegal:

```
I1, I2 = I2;
I1, I3 = I4;
```

20.4 ARRAYED SUBSCRIPTING

Variables and expressions may appear in subscripts only if all arrayness has been subscripted away.

Examples:

```
DECLARE  INTEGER,
         I1,
         I2,
         A1 ARRAY(10),
         A2 ARRAY(5);
```

$\mathtt{I1 = A1}_{\mathtt{I2}}\mathtt{;}$	legal
$\mathtt{I1 = A1}_{\mathtt{(A2}_{\mathtt{1}}\mathtt{ + 2)}}\mathtt{;}$	legal (all arrayness subscripted away from expression)
$\mathtt{A2 = A1}_{\mathtt{(3\ TO\ 8)}}\mathtt{ + A1}_{\mathtt{(5\ AT\ 2)}}\mathtt{;}$	legal
$\mathtt{A2 = A1}_{\mathtt{A2}}\mathtt{;}$	illegal

20.5 ARRAYED COMPARISONS

Relational expressions have been described in Sections 9.2, 17.6, and **18.9**. The comparisons which comprise relational expressions may possess arrayed operands. If one or both operands in a comparison are arrayed, then only the Class II comparative operators may be used, irrespective of the <u>types</u> of the operands:

Symbol	Purpose	Class
=	equals	II
NOT = ¬ =	not equals	

The additional rule applicable to arrayed comparisons is:

1. If both operands possess arrayness, their arraynesses must match.

The comparison is viewed as a set of elemental comparisons. The outcome of all elemental comparisons is combined to form a <u>single</u> TRUE or FALSE logical result. The following table shows the conditions necessary for arriving at TRUE or FALSE results.

Operation	Result	Conditions for Result
=	True	Equality in all elemental comparisons is obtained.
	False	Equality in one or more elemental comparisons is lacking.
Not = ¬=	True	Equality in one or more elemental comparisons is lacking.
	False	Equality in all elemental comparisons is obtained.

Examples:

If I1, I2 are 2 by 3 arrays of integers

with $I1 \equiv \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$ and $I2 \equiv \begin{pmatrix} 1 & 2 & 6 \\ 4 & 5 & 3 \end{pmatrix}$

then I1 = I2 is FALSE.

However, $I1_{*,1 \text{ TO } 2} = I2_{*,1 \text{ TO } 2}$ is TRUE.

$I1 = I2_{*,1}$ is illegal since the arraynesses of the operands no longer match.

If further I3 is a 2 by 3 array of integers with

$I3 \equiv \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$

then

$I3 \equiv I1_{1,1}$ is TRUE.

In each elemental comparison, an element of I3 is compared against $I1_{1,1}$ which is unarrayed, and in this case, equality is obtained.

20.6 ARRAYED ARGUMENTS IN PROCEDURES AND FUNCTIONS

The arguments of procedures and functions may possess arrayness when the corresponding formal parameters meet certain conditions. Parameters of procedures and functions may be declared as arrays of indefinite form, thus allowing the passage of arguments whose arrayness varies from invocation to invocation.

INDEFINITELY ARRAYED PARAMETERS

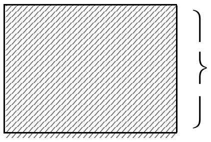
The parameters of functions and the input or assign parameters of procedures may be declared to be indefinite arrays. The form of array specification is:

```
ARRAY(*)
```

1. The array specification is placed as shown in Section 4.2.

Examples:

```
TWICE: PROCEDURE(A) ASSIGN(B);
   DECLARE A ARRAY(*) VECTOR(3);
   DECLARE B ARRAY(*) BIT(16);
```



procedure body

```
CLOSE TWICE;
```

The asterisk implies that although the arrayness of the corresponding argument is always 1-dimensional, its size can vary from invocation to invocation.

The number of multiple copies in a structure parameter may also be made indefinite using the following form:

```
α-STRUCTURE(*)
```

1. α is the name of the template describing the tree organization of the structure.
2. Section 19.3 describes the location of the construct in a structure declaration.

Example:

```
FUN: FUNCTION(C) SCALAR;
      STRUCTURE Q:
         1 QI INTEGER,
         1 QS SCALAR;
      DECLARE C Q-STRUCTURE(*);
 ▨▨▨▨▨▨▨▨ ⎫
 ▨▨▨▨▨▨▨▨ ⎬ function body
 ▨▨▨▨▨▨▨▨ ⎭
CLOSE FUN;
```

Note that the ability to define an indefinite array does not extend to an arrayed structure terminal.

Example:

```
 BAD: FUNCTION(C) SCALAR;
      STRUCTURE Q:
        1 QI INTEGER,
        1 QS ARRAY(*) SCALAR; ← illegal
      DECLARE C Q-STRUCTURE;
        ▨▨▨▨▨▨▨▨ ⎫
        ▨▨▨▨▨▨▨▨ ⎬ function body
        ▨▨▨▨▨▨▨▨ ⎭
 CLOSE BAD;
```

ARRAYED PROCEDURE ARGUMENTS

Both input and assign arguments of a procedure invocation may possess arrayness. The parameters corresponding to such arguments must be arrayed. The rules for passage of such arguments are as follows.

- INPUT ARGUMENTS. The transmission of the argument may be viewed as its assignment to the corresponding input parameter. However, the rules for arrayness matching are more severe than for arrayed assignments.

1. The arrayness of the argument must match that of the corresponding parameter.
2. If the parameter is an indefinite array, arrayness matching is ensured if the corresponding argument is a 1-dimensional array.

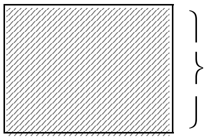
- ASSIGN ARGUMENTS. If an assign argument possesses arrayness, it is either because it is an arrayed data item (see Section 11.5) or because it is part or whole of a structure with multiple copies (see Section 19.3). In these cases the rules for arrayness matching are as follows:

1. The arrayness of the argument must match that of the corresponding parameter.
2. If the parameter is an indefinite array, arrayness matching is ensured if the corresponding argument is a 1-dimensional array.
3. If the argument is part of a structure which has multiple copies, structure subscripting must be used to limit the number of copies in the argument to one.
4. If array subscripting is present it must be such as to select one array element only.
5. If component subscripting is present, where necessary array subscripting must be used to limit the number of array elements in the argument to one.

Examples:

Given the following procedure:

```
  ONE: PROCEDURE(A) ASSIGN(B);
       DECLARE  A ARRAY(2,3) SCALAR,
                B ARRAY(4) BIT(16);
```



```
 CLOSE ONE;
```

and the following data declarations:

```
DECLARE P1 ARRAY(2,3) SCALAR,
         P2 ARRAY(2,5) SCALAR,
         P3 ARRAY(4) BIT(16),
         P4 SCALAR,
         P5 ARRAY(2,5) BIT(16),
         P6 BIT(16);
```

then some legal and illegal invocations of the procedure are as follows:

```
  CALL ONE(P1) ASSIGN(P3);
  CALL ONE(P2        ) ASSIGN(P3);
S           *,1 to 3
  CALL ONE(P2        + P1 - P4) ASSIGN(P6);
S           *,3 to 5
  CALL ONE(P4) ASSIGN(P5       );
S                       1,1 TO 4
```

illegal - not arrayed (P4)

legal arrayness but illegal subscript (1,1 TO 4)

illegal - not arrayed (P6)

If a second procedure is given:

```
 TWO: PROCEDURE(A) ASSIGN(B);
      DECLARE   A ARRAY(*) SCALAR,
                B ARRAY(*) BIT(16);
      ▨▨▨▨▨▨▨▨ ⎫
      ▨▨▨▨▨▨▨▨ ⎬ procedure body
      ▨▨▨▨▨▨▨▨ ⎭
CLOSE TWO;
```

Then some invocations to it are:

```
  CALL TWO(P1    ) ASSIGN(P3);
S           1,*
  CALL TWO(P2    ) ASSIGN(P3);
S           1,*
  CALL TWO(P1) ASSIGN(P6);
```

Illegal - not arrayed

Illegal -
wrong number of array dimensions

ARRAYED FUNCTION ARGUMENTS

If a function has one or more arrayed arguments, one of two situations can arise, depending on the form of the corresponding parameters.

- ARRAYED PARAMETER. If the parameter corresponding to an argument with arrayness is itself arrayed, then the whole of the argument is transmitted in a single invocation of the function. The same rules apply to this situation as to the input arguments of procedures.
- UNARRAYED PARAMETER. If the parameter corresponding to an argument with arrayness is itself unarrayed, then the arrayness of the argument must match other arraynesses in the expression in which the function is invoked. In this situation, the function is repeatedly invoked, once during every elemental evaluation of the expression containing it. During each invocation, the appropriate elemental argument transmittal takes place.

Examples:

Given the function:

```
 ONE: FUNCTION(A,B) SCALAR;
      DECLARE A SCALAR;
      DECLARE B ARRAY(2,5) SCALAR;
      ▨▨▨▨▨▨▨▨ ⎫
      ▨▨▨▨▨▨▨▨ ⎬ function body
      ▨▨▨▨▨▨▨▨ ⎭
CLOSE ONE;
```

and the declarations

```
DECLARE P1 ARRAY(2,5),
        P2 ARRAY(2,5),
        P3 ARRAY(3),
        P4;
```

then some legal and illegal invocations of the function are as follows:

$$P4 + ONE(P4,\underbrace{P2/2}_{\text{arrayness matches that of parameter - function invoked once}})\ldots$$

$$P2 + ONE(\underbrace{P2}_{\substack{\text{parameter unarrayed - one element of P2 transmitted every elemental}\\ \text{elevation.}}},\underbrace{P1}_{\substack{\text{arrayness matches that of parameter - all}\\ \text{of P1 transmitted every elemental elevation}}})\ldots$$

The second invocation is equivalent to:

$P2_{i,j}$ + ONE($P2_{i,j}$, P1) . . . for $1 \le i \le 2, 1 \le j \le 5$

The function can in fact generate the only arrayness of the expression:

`P4 + ONE(P2,P1) . . .`

is equivalent to

P4 + ONE($P2_{i,j}$,P1) . . . for $1 \le i \le 2, 1 \le j \le 5$

`P3 + ONE(P2,P1)`... is illegal because the arrayness of P3 does not match that of P2.

`P2 + ONE(P2,P3)`... is illegal because the arrayness of P3 does not match that of the corresponding parameter.

This example would become legal if the declaration for the parameter B were:

```
DECLARE B ARRAY(*) SCALAR;
```

Restrictions are placed upon the array processing of arguments of functions if the function definition follows any of its invocations.

See: Spec./4.6.

20.7 ARRAYS IN INPUT/OUTPUT

The values of an arrayed expression can be output by using the WRITE statement. The values of an arrayed data item can be input by using the READ statement.

Section 12.2 described the data formats for one-dimensional arrays on output: Section 12.2 described the data formats for one-dimensional arrays on input.

The order of input and output is generated by application of the rules given in Appendix C.

Example:

IF I is a 2 x 3 array of integers with

$$I \equiv \begin{pmatrix}1\ 2\ 3\\4\ 5\ 6\end{pmatrix}$$

then execution of

```
WRITE(6) I;
```

results in the following output being generated:

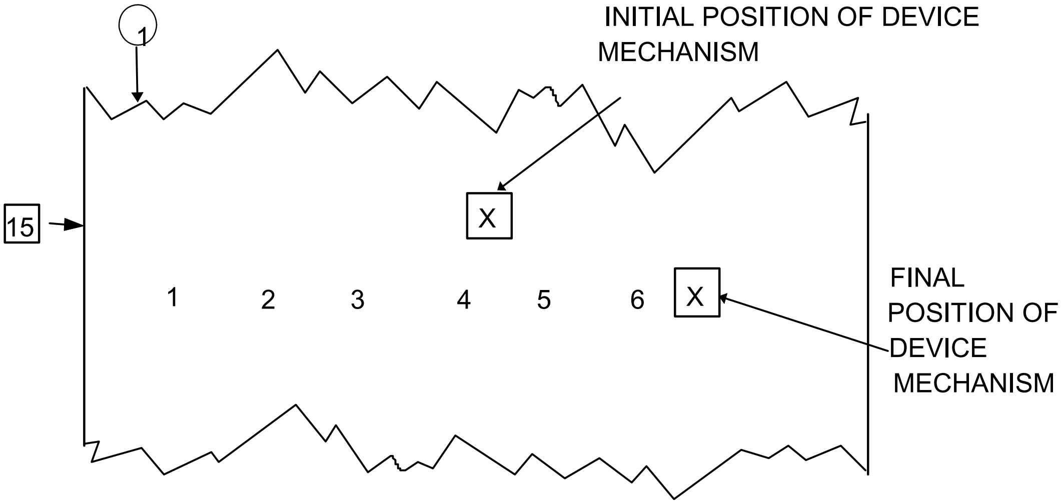

1
INITIAL POSITION OF DEVICE MECHANISM
15
X
1 2 3 4 5 6
X
FINAL POSITION OF DEVICE MECHANISM

Figure 20-6

21.0 EXPLICIT CONVERSIONS

Section 7.5 in Part I of the Guide introduced some of the commoner explicit conversions of HAL/S. Explicit precision conversion, and the VECTOR and MATRIX conversion functions were described. The language contains many more kinds of explicit conversions, however, which provide a controlled and highly visible interface between the various data types.

This section deals with conversion functions, classifying them according to the data type of their results.

21.1 VECTOR AND MATRIX CONVERSIONS

The forms of VECTOR and MATRIX conversion functions have been given in Section 7.5. It remains in this section to present the general forms of argument list they may possess.

The argument list of a VECTOR or MATRIX conversion may take the following general form:

(exp^1, exp^2......)

1. Each *exp* is an expression of any of the following types:

   ```
   MATRIX     INTEGER
   VECTOR     SCALAR
   ```

2. Any expression may possess arrayness in the sense described in Section 20.2.
3. The total number of values summed over all expressions must match the length of the vector result, or the product of the row and column dimensions of the result, as appropriate.

The ordering of the values of the expression list in the resulting vector or matrix is specified by the following.

- The values of each expression in turn are converted to a linear list by applying the rules of Appendix C.
- The lists are catenated from left to right forming a single linear list of values.
- For a VECTOR conversion, the resulting vector is formed directly from the linear list. For a MATRIX conversion, the resulting matrix is formed by a row-by-row assembly from the linear list.

Example:

If V is a 4-vector with $V \equiv \begin{bmatrix} 3 \\ 1 \\ 2 \\ 0 \end{bmatrix}$

and M is a 2 x 2 matrix with $M \equiv \begin{bmatrix} 4 & 5 \\ 2 & 4 \end{bmatrix}$

then

$$\text{MATRIX}_{2,4}(\text{M,V}) \equiv \begin{bmatrix} 4 & 5 & 2 & 3 \\ 3 & 1 & 2 & 0 \end{bmatrix}$$

EXPRESSION REPETITION

Any expression in the argument list of a MATRIX or VECTOR conversion can be repeated by prefacing it with a repetition factor with the following form:

...n# exp^{i} ,...

1. n is a positive non-zero integer literal specifying the number of times the value or values of the expression are to be repeated.

Example:

If V is a 3-vector with $V \equiv \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$

then

$$\text{MATRIX}_{2,3}(2\#\text{V}) \equiv \begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix}$$

21.2 INTEGER AND SCALAR CONVERSIONS

The INTEGER and SCALAR conversion functions convert to integer and scalar type respectively. The behavior of these functions varies, depending on whether they possess a single expression as argument, or a list of expressions.

SIMPLE FORM

The simple form of the INTEGER and SCALAR conversion functions is:

```
INTEGER(exp)
SCALAR(exp)
```

1. *exp* is an expression of any of the following types:

   ```
   BIT STRING (and BOOLEAN)    INTEGER
   CHARACTER                   SCALAR
   ```

2. *exp* may possess arrayness, in which case the arrayness must match that of the expression of which the conversion forms a part. The result is to cause an elemental conversion for every elemental evaluation of the outer expression (See Section 20.2).
3. Conversions to integer or scalar type proceed according to the rules given in Appendix A.

Examples:

If C is a character with C ≡ '123.5' and B is a 3-array of bitstrings of length 8

$$\text{with B} \equiv \begin{bmatrix} F_{16} \\ 10_{16} \\ 1C_{16} \end{bmatrix}$$

then

SCALAR(C) ≡ 123.5

and

$$\text{INTEGER(B)} \equiv \begin{bmatrix} 15 \\ 16 \\ 28 \end{bmatrix}$$

$$\text{If I is a 3-array with I} \equiv \begin{bmatrix} -10 \\ -10 \\ -20 \end{bmatrix}$$

$$\text{then I + INTEGER(B)} \equiv \begin{bmatrix} 5 \\ 6 \\ 8 \end{bmatrix}$$

LIST FORM

The list form of the INTEGER and SCALAR conversion functions creates an array result, in addition to type converting the list of expressions constituting its arguments. Its form is as follows:

$$\mathtt{INTEGER}_{n^1,n^2,...}(\mathit{exp}^1,\mathit{exp}^2,...)$$
$$\mathtt{SCALAR}_{n^1,n^2,...}(\mathit{exp}^1,\mathit{exp}^2,...)$$

1. The subscripts n^i for i = 1, 2,... are positive integers specifying the number and size of dimensions of the resulting array. The total number of values summed over all the expressions in the list must be consistent with the number of array elements implied. The upper limit on i is 3[†].
2. The subscripts may be omitted entirely, in which case a linear 1-dimensional array is created, whose length is equal to the total number of values summed over all the expressions.
3. Each *exp* is an expression of any of the following types:

   INTEGER MATRIX
   SCALAR BIT STRING(and BOOLEAN)
   VECTOR CHARACTER

   and may optionally possess arrayness.
4. Conversions to integer or scalar type proceed according to the rules given in Appendix A.

[†] This number may vary between implementations. See the appropriate User's Manual.

Note that the list form can only have one expression in the list without reverting to the simple form if either of the following statements are true:

- the type of the expression is matrix or vector;
- explicit subscripting of the function is present.

The ordering of values of the expression list in the resulting array is specified by the following:

- The values of each expression in turn are converted to a linear list by applying the rules of Appendix C.
- The lists are catenated from left to right forming a single linear list of values.
- The linear list is regenerated to an array of the given dimensions by applying the rules of Appendix C.

Examples:

If V is a 4-vector with $V \equiv \begin{bmatrix} 4 \\ 1 \\ 3 \\ 2 \end{bmatrix}$

and M is a 2 x 2 matrix with $M \equiv \begin{bmatrix} 1 & 0 \\ 5 & 4 \end{bmatrix}$

$$\text{then INTEGER}_{2,4}(\text{V,M}) \equiv \begin{bmatrix} 4 & 1 & 3 & 2 \\ 1 & 0 & 5 & 4 \end{bmatrix}$$

$$\text{and INTEGER(V,M)} \equiv \begin{bmatrix} 4 \\ 1 \\ 3 \\ 2 \\ 1 \\ 0 \\ 5 \\ 4 \end{bmatrix}$$

$$\text{INTEGER(V)} \equiv \begin{bmatrix} 4 \\ 1 \\ 3 \\ 2 \end{bmatrix} \text{ and is of list form}$$

If I is a 4-array of integers

$$\text{with I} \equiv \begin{bmatrix} 2 \\ 4 \\ 1 \\ 1 \end{bmatrix}$$

$$\text{then I + INTEGER(V)} = \begin{bmatrix} 6 \\ 5 \\ 4 \\ 3 \end{bmatrix}$$

Note that even though the function appears in an arrayed expression, in this and all other cases involving the list form, the implementation is generally to precompute the entire array result, and then evaluate the expression containing the conversion on an element-by-element basis.

EXPRESSION REPETITION

As with the VECTOR and MATRIX conversions, the expressions in the list of an INTEGER or SCALAR conversion may be repeated using the form:

$$\texttt{....n\# } exp^{i}\texttt{,....}$$

1. n is a positive non-zero integer literal specifying the number of times the values or values of the expression are to be repeated.

Example:

If S is a scalar with S≡1.5

$$\text{then INTEGER(5\#S)} \equiv \begin{bmatrix}1\\1\\1\\1\\1\end{bmatrix} \text{ and is of list form.}$$

SIMULTANEOUS PRECISION SPECIFICATION

In the absence of any explicit indication, the result of an INTEGER or SCALAR conversion is always single precision. The precision can be explicitly stated in same way as in a VECTOR or MATRIX conversion.

If no subscripting is present, the forms are:

$$\texttt{INTEGER}_{\texttt{@SINGLE}}\texttt{(....}$$
$$\texttt{SCALAR}_{\texttt{@SINGLE}}\texttt{(....}$$
$$\texttt{INTEGER}_{\texttt{@DOUBLE}}\texttt{(....}$$
$$\texttt{SCALAR}_{\texttt{@DOUBLE}}\texttt{(....}$$

1. The first two forms force a single precision result; the second two, double precision.
2. Precision conversion is carried out for each expression in turn before assembly of the result.

If subscripting is present, the corresponding forms are:

$$\texttt{INTEGER}_{\texttt{@SINGLE, } n^1,n^2\ ...}\texttt{(....}$$
$$\texttt{SCALAR}_{\texttt{@SINGLE, } n^1\ ,n^2\ ...}\texttt{(....}$$
$$\texttt{INTEGER}_{\texttt{@DOUBLE, } n^1\ ,n^2\ ...}\texttt{(....}$$
$$\texttt{SCALAR}_{\texttt{@DOUBLE, } n^1\ ,n^2\ ...}\texttt{(....}$$

Examples:

$\texttt{INTEGER}_{\texttt{@DOUBLE}}\texttt{(X)}$ simple form

$\texttt{INTEGER}_{\texttt{@DOUBLE, 2,2}}\texttt{(5.0,‘15’,BIN‘1011’,-7.5)}$ list form

21.3 BIT CONVERSION

Conversions to bit string type are carried out by the BIT conversion function. There are two forms: the simple form converts other data types to bit string type using the standard conversion rules; the radix form can only convert character data type to bit string type, and uses different conversion rules.

Both forms are similar to the simple form of INTEGER and SCALAR functions, in that they have one expression only.

SIMPLE FORM

The simple form of BIT conversion is as follows:

$\text{BIT}_{\text{subscript}}$ (exp)

1. *exp* is an expression of any of the following types:
```
INTEGER   BIT STRING(and BOOLEAN)
SCALAR    CHARACTER
```
2. *exp* may possess arrayness in which case the arrayness must match that of the expression of which the conversion forms a part. The result is to cause an elemental conversion for every elemental evaluation of the outer expression (see Section 20.2).
3. Conversion to bit string type proceeds according to the rules given in Appendix A.
4. *subscript* represents component subscripting on the result of the conversion. It possesses the same forms as component subscripting on bit string data items as described in Section 17.3. The result is always a 32-bit string[†].
5. If *subscript* is absent, the result of the function is the entire bit string generated by the conversion.

The length of the result depends on the length of the argument[†].

type		Length of the result
INTEGER	*DOUBLE*	32
	SINGLE	16
SCALAR	*DOUBLE*	32
	SINGLE	32
CHARACTER		32
BIT STRING		Number of bits in argument

[†] This value may vary between implementations. See the appropriate User's Manual.

Examples:

If I is a halfword integer with I≡5

then BIT(I) ≡0005_{16}

If C is a character data item with C = ‘10110011101’

then BIT(C) ≡ $00000000000000000000010110011101_2$

$BIT_{17\ TO\ 32}(C)$ ≡ 0000010110011101_2

and $BIT_{28\ TO\ 32}(C) = 11101_2$

RADIX FORM

The radix form of BIT conversion is used when a character value is to be converted by an explicit rule to a bit string. A radix specifying the conversion rule is supplied in place of a subscript. The possible forms are as follows:

$\mathtt{BIT}_{\mathtt{@BIN}}(\mathit{exp})$

$\mathtt{BIT}_{\mathtt{@OCT}}(\mathit{exp})$

$\mathtt{BIT}_{\mathtt{@DEC}}(\mathit{exp})$

$\mathtt{BIT}_{\mathtt{@HEX}}(\mathit{exp})$

1. *exp* is an expression of character type whose value must consist entirely of a string of digits legal for the specified radix.
2. The radices have the following meanings:

radix	digit string
@BIN	binary
@OCT	octal
@DEC	decimal
@HEX	hexadecimal

3. *exp* may possess arrayness with the same implications as in the simple form of BIT conversion.
4. The conversion generates the binary representation of the input digit string. The binary representation is truncated or padded with binary zeroes on the left to create a 32-bit string[†] .

[†] This value may vary between implementations. See the appropriate User’s Manual.

Examples:

BIT$_{@HEX}$(‘FA0’) ≡ $\mathrm{00000FA0}_{16}$

BIT$_{@DEC}$(‘1024’) ≡ 00000400_{16}

BIT$_{@OCT}$(‘177777’) ≡ $\mathrm{0000FFFF}_{16}$

BIT$_{@HEX}$(‘F0F1F2F3F4’) ≡ $\mathrm{F1F2F3F4}_{16}$

21.4 CHARACTER CONVERSION

Conversions to character type are carried out by the CHARACTER conversion function. As with the BIT conversion, there are two forms: the simple form converts other data types to character form using the standard conversion rules; the radix form can only convert bit string data to character type, and uses different conversion rules.

SIMPLE FORM

The simple form of CHARACTER conversion is as follows:

$\texttt{CHARACTER}_{\texttt{subscript}}\texttt{(exp)}$

1.*exp* is an expression of any of the following types:

```
INTEGER   BIT STRING (and BOOLEAN)
SCALAR    CHARACTER
```

2. exp may possess arrayness, with the same implications as in the BIT conversion function (See Section 21.3).
3. Conversion to character type proceeds according to the rules given in Appendix A. The length of the result of conversion depends on the type of the input data.
4. *subscript* represents component subscripting on the result of the conversion. It possesses the same forms as component subscripting on character data items as described in Section 6.1.
5. If *subscript* is absent, then the result of the function is the entire string of characters generated by the conversion.

Examples:

If I is a halfword integer with I = 173

then CHARACTER(I) ≡ '173'

$\text{CHARACTER}_{1\ \text{TO}\ 2}(\text{I}) \equiv$ '17'

$\text{CHARACTER}_{1\ \text{TO}\ 3}(\text{I}) \equiv$ '173'

If B is a bit string of length 4 with

$\text{B} \equiv 0101_2$

then

CHARACTER(B) ≡ '0101'
(note that number of characters is the same as the number of bits.)

RADIX FORM

The radix form of CHARACTER conversion is used when a bit string value is to be converted by an explicit rule to a character string. Analogous to the radix form of BIT function, a radix specifying the conversion rule is supplied in place of a subscript. The possible forms are as follows:

CHARACTER$_{@BIN}$(*exp*)

CHARACTER$_{@OCT}$(*exp*)

CHARACTER$_{@DEC}$(*exp*)

CHARACTER$_{@HEX}$(*exp*)

1. *exp* is an expression of bit string type, and possibly possessing arrayness, with the same implications as in the BIT conversion function.
2. The value of the bit string is converted to a string of digits as specified by the radix.
3. The radices have the following meanings:

radix	digit string
@BIN	binary
@OCT	octal
@DEC	decimal
@HEX	hexadecimal

4. The length of the resulting string varies depending on the value of *exp.*

Examples:

CHARACTER$_{@BIN}$(BIN ‘001010’) ≡ ‘001010’

CHARACTER$_{@OCT}$(BIN ‘001010’) ≡ ‘12’

CHARACTER$_{@DEC}$(BIN ‘001010’) ≡ ‘10’

CHARACTER$_{@HEX}$(BIN ‘001010’) ≡ ‘OA’

21.5 SUBBIT PSEUDO-CONVERSION

The SUBBIT pseudo-conversion function provides a way of transferring a value from one data type another without conversion. Effectively it is used as a method of circumventing HAL/S type compatibility rules in a limited and controlled way. The value transferal takes place using the bit string type as an intermediary:

old type → *bit string* → *new type*

This transferal requires the use of the SUBBIT conversion both in reference and in assignment context.

- In reference context, SUBBIT causes a data type to be referenced as if it were a bit string.
- In assignment context, SUBBIT causes a data item to be assigned into as if it were a bit string.

There are, of course, other contexts where it is convenient to use pseudo-conversion other than those described above.

The form of the SUBBIT pseudo-conversion in either context is as follows:

SUBBIT$_{\texttt{subscript}}$(argument)

1. In assignment context, *argument* is a data item, either subscripted or not. In reference context, *argument* is an expression. The following types are legal:

INTEGER	BIT STRING (and BOOLEAN)
SCALAR	CHARACTER

2. *argument* may possess arrayness. In reference context it causes a data type to appear to be a bit string expression with arrayness; in assignment to appear as a bit string data with arrayness.
3. The *subscript* is optional. If it is absent, only the N leftmost bits in the bit pattern of the argument are visible. For different types of argument, the values of N are:[†]

type		N
INTEGER	DOUBLE	32
	SINGLE	16
SCALAR	DOUBLE	32
	SINGLE	32
CHARACTER		
BIT STRING		Number of bits in argument

4. If *subscript* is present, it specifies what range of bits in the bit pattern of the argument are to be made visible. It must conform to the rules for subscripting of bit string data items as described in Section 17.3 in all respects save that the index values are confined to the range 1-N given below, rather than 32:[††]

type		N
INTEGER	DOUBLE	32
	SINGLE	16
SCALAR	DOUBLE	64
	SINGLE	32
CHARACTER		Current working length of argument
BIT STRING		Number of bits in argument

[†] The values may vary between implementations. See appropriate User's Manual.

[††] The maximum length of bit strings is an implementation. See appropriate User's Manual for variations.

Examples:

If I is a double precision (fullword) integer

then

```
| SUBBIT(I) = SUBBIT('1234');
```

causes the following to occur:

The SUBBIT in reference context causes the creation of a 32-bit string with value F1F2F3F4$_{16}$[29]. When this value is assigned into I its bit pattern becomes F1F2F3F4$_{16.}$

Hence, I ≡ 4059231220 in decimal notation, if unsigned.

Given that I now has this value

```
|
| SUBBIT       (I) = SUBBIT       (333);
|S      25 TO 32           4 TO 11
```

has the following effect:

The bit pattern of 333 is 014D$_{16}$ (it is assumed to be a halfword integer). The SUBBIT in reference context selects from this the 8-bit string 0A$_{16}$. This overlays bits 25-32 of the bit pattern of I:

$$\text{F1F2F3}\overset{\uparrow}{\overbrace{\text{F4}}}_{16}$$

$$\overset{\uparrow}{\overbrace{\text{0A}}}_{16}$$

The final bit pattern is thus F1F2F30A$_{16}$.

Hence, I = 4059230986 in decimal notation.

DETAILED BEHAVIOR OF SUBBIT

SUBBIT has the effect of opening a "window" on the bit pattern of its argument, the width and position being determined by the subscript, if the conversion possesses one, or implicitly otherwise. Various side effects may occur depending on the width and position of the window relative to the bit pattern of the argument. These effects may differ depending on whether the SUBBIT pseudo-conversion is in reference or assignment context.

[29] Using EBCDIC character codes.

The examples given above have avoided these side effects, which may include padding, truncation, or error conditions. These phenomena are summarized below for each context in turn.

- **REFERENCE CONTEXT**

  The following diagrams summarize the behavior of SUBBIT in reference context. In general, padding and truncation do not occur but an error condition may arise.

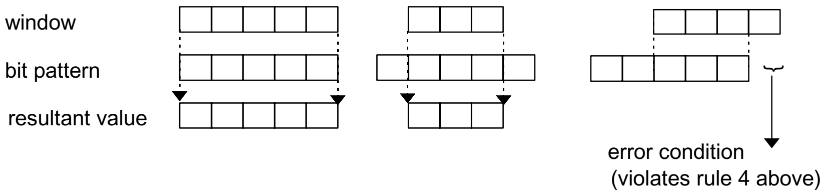


Figure 21-1

Examples:

If I is a single precision (halfword) integer

with I ≡ 32767

then it has the bit pattern $7FFF_{16}$

Thus,

SUBBIT(I) ≡ $7FFF_{16}$ (length 16 result)

$SUBBIT_{1\ TO\ 8}(I)$ ≡ $7F_{16}$ (length 8 result)

$SUBBIT_{1\ TO\ 32}(I)$ is illegal

ASSIGNMENT CONTEXT

In contrast to SUBBIT in reference context there are two steps to its operation in assignment context. The first is the fitting of the value to be assigned to the window; the second is the assignment through the window into the bit pattern of the argument. The first step may involve padding or truncation, the second may cause an error condition to arise. The following diagrams illustrate this.

STEP 1

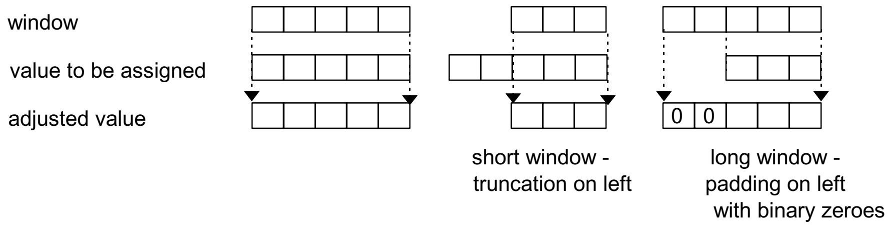


Figure 21-2

STEP 2

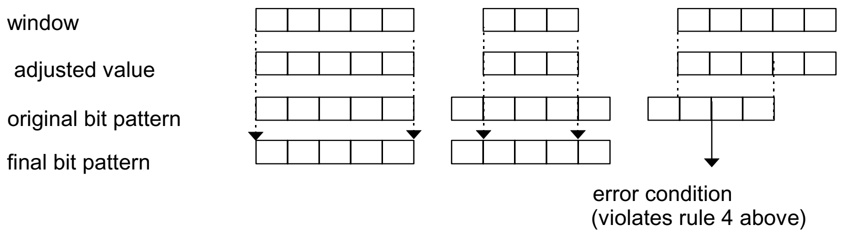


Figure 21-3

Examples:

If I is a single precision (halfword) integer

with I ≡ 32767

then it has the bit pattern $7FFF_{16}$

```
|
| SUBBIT (I) = HEX'FF';
|
```

causes the following to occur:

The window of the SUBBIT is 16 bits wide opening all of I. The bit string to be assigned into I is expanded to 16 bits:

$00FF_{16}$

The value of I thus becomes FF_{16} or I ≡ 255 in decimal notation.

```
|
| SUBBIT (I) = HEX'FFFFFFFF';
|
```

causes the following to occur:

The bitstring to be assigned into I must now be truncated to $FFFF_{16}$ to match the width of the window. The value of I thus becomes $FFFF_{16}$ or I ≡ -1.

If I initially has its original value

```
|
|  SUBBIT        (I) = HEX'7E';
|S       13 TO 16
|
```

then the following occurs:

The SUBBIT window is 4 bits wide. The bit string is thus truncated from $7E_{16}$ to E_{16}. This value is assigned through the window into I which thus becomes $7FFE_{16}$, or I ≡ 32766.

SUBBIT WITH CHARACTER ARGUMENT

Instances of SUBBIT pseudo-conversions with arguments of character type are even more complex than the foregoing, since a SUBBIT with no subscript has a different behavior than one with a subscript. Again the rules are summarized separately for reference and assignment contexts.

- REFERENCE CONTEXT

  If an unsubscripted SUBBIT pseudo-conversion has a character argument whose working length is less than the implied constant window width, then the bit pattern is left padded with zeroes to fill the gap. If a subscripted SUBBIT pseudo-conversion has a character argument, and the specified window lies partly or wholly outside the range of the current value of the argument, then an error condition arises.

Examples:

If C is a character string of maximum length 4
and C≡ 'AB'
then

$\text{SUBBIT (C)} \equiv \text{0000C1C2}_{16}$[30]

$\text{SUBBIT}_{\text{1 TO 16}}\text{(C)} \equiv \text{C1C2}_{16}$

$\text{SUBBIT}_{\text{1 TO 24}}$ (C) is illegal because the working length of C is too short.

- ASSIGNMENT CONTEXT

  If an unsubscripted SUBBIT pseudo-conversion has a character argument whose working length is less than the implied constant window width, then the window is shortened to the working length, causing truncation of the value being assigned into the argument. If a subscripted SUBBIT pseudo-conversion has a character argument, and the specified window lies partly or wholly outside the range of the current value of the argument, then an error condition arises.

Examples:

If C is a character string of length 4 with

C ≡ 'AB' initially,

then

```
SUBBIT (C)= HEX‘F1F2F3F4’;
```

causes shortening of the window to 16 bits.

The bit string to be assigned into C is therefore truncated to F3F4_{16} and hence finally C '34'[30]

[30] Using EBCDIC character codes.

```
  SUBBIT      (C) = HEX‘F1F2F3F4’;
S       1 TO 8
```

causes the first 8 bits of C to be replaced by F4.

Hence finally, C ≡ ‘AB’.

```
 SUBBIT       (C) = HEX ‘F1F2F3F4’;  is illegal
S      1 TO 24
```

RESTRICTIONS ON USE OF SUBBIT

The <u>only</u> assignment context in which SUBBIT may appear is an assignment statement. Other such contexts where SUBBIT is illegal include READ statements, and ASSIGN argument lists.

In reference context, SUBBIT may be used anywhere that a bit string expression is legal.

This page intentionally left blank.

22.0 ADDITIONAL INPUT/OUTPUT FEATURES

This section is supplementary to Section 12 of Part I of the Guide, which introduced sequential I/O and the HAL/S READ and WRITE statements. Two topics are covered here.

- Discussion of sequential I/O is concluded by a description of a second mode of input implemented by the READALL statement.
- Random-access I/O using FILE statements is described.

22.1 THE READALL STATEMENT

Section 12.3 described how data could be read from an I/O device into specified data items. It was stated that the input stream is considered to be divided sequentially into data fields each containing a value to be input. Input of each value is accompanied by a conversion appropriate to the data item receiving the value.

It is often important to be able to read the input data stream as a continuous sequence of characters, without division into fields, and without type conversion, thus leaving it to the programmer to decode the information any desired way. This ability is provided by the HAL/S READALL statement.

The READALL statement is an executable statement causing input of data from an unpaged[31] I/O device. Its form is as follows:

READALL(n)var^1,var^2,.....var^n;

1. n is the channel code number, and lies in the range 0 ≤ n ≤ 9†.
2. Each *var* is a data item of character type, or a structure whose terminal nodes are exclusively of structure type, and optionally subscripted.
3. The list of data items may be arbitrarily long. Alternatively, no list need be supplied.
4. The specified device reads values into each data item in turn from left to right.

† This value may vary between implementations. See appropriate User's Manual.

In execution the sequence of events is as follows:

- If the READALL statement is the first to be executed for the specified device, the device mechanism positions itself at column 1 of line 1. Otherwise, it moves down one line from its current position and repositions itself at column 1.
- The device begins reading into the first element of data specified by the list, stopping when the character string reaches its defined maximum length, or when the end of the line is reached, whichever happens sooner.

[31] See Section 12.1 for definition.

- The device then begins reading into the second element specified by the list, resuming from the next column of the line, or if the end of it was previously reached, from column 1 of the next line. The stopping condition is as before.
- The device continues reading as described above until all the data items in the list have been filled.
- This behavior is unaffected by the contents of the input stream.
- If no list of data items is supplied in the READALL statement, the device merely performs its initial positioning.

DATA FORMATS

No conversions occur during input, the input stream appearing unaltered in the data items constituting the READALL list.

The order of reaching into arrayed character items, and into structures, is in conformity with the rules given in Appendix C.

Examples:

Let C1 be a character string of maximum length 70

and C2 be a character string of length 20

Then

```
READALL(5) C1, C2;
```

using the following data:

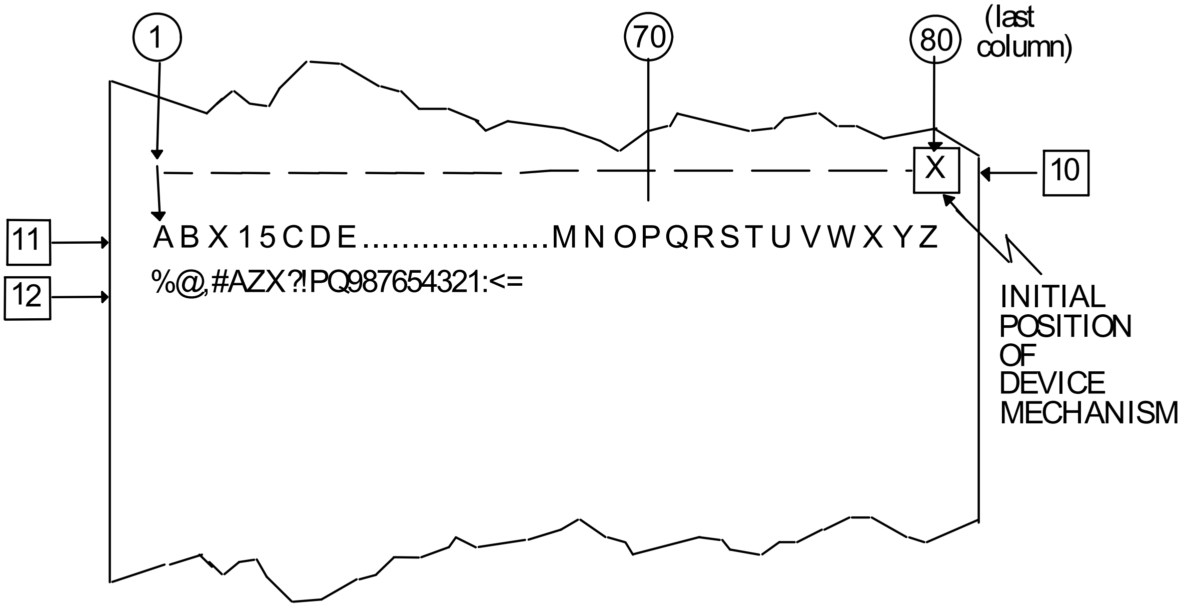

1
70
80
(last column)
X
10
11
12
A B X 1 5 C D E ....M N O P Q R S T U V W X Y Z
%@,#AZX?!PQ987654321:<=
INITIAL POSITION OF DEVICE MECHANISM

Figure 22-1

would result in

C1 ≡ 'ABX15CDE......MNOP' (maximum length reached)

C2 ≡ 'QRSTUVWXYZ' (end of line reached)

If C1 ≡ had a maximum length of 80

then the same data would have resulted in

C1 ≡ 'ABXI5CDE..........MNOPQRSTUVWXYZ'

C2 ≡ '%@,#AZX?!PQ987654321'

DEVICE MECHANISM POSITIONING

Section 12.4 described how the pseudo-functions SKIP, LINE, COLUMN, and TAB could be used to position a device mechanism of an unpaged device explicitly during input. The pseudo-functions can be used in the READALL statement in exactly the same way with identical effect.

DEVICE ATTRIBUTES

In determining whether by default a device is characterized as paged or unpaged according to the rules of Section 12.5, a READALL statement is equivalent to a READ statement.

22.2 RANDOM ACCESS INPUT/OUTPUT

Random access input/output consists of writing records on a device, or reading them from a device, in arbitrary or random order, rather than sequentially.

HAL/S implements random-access I/O by means of the FILE statement, which has the form of an assignment, and handles either input or output, depending on the form in which it is written.

This section introduces the HAL/S concept of random-access I/O and describes the form and use of the FILE statement.

HAL/S RANDOM-ACCESS CONCEPTS

Random access I/O is thought of as taking place via a number of "channels" each connected to a random-access device, and identified by an integer code (The channels and devices are taken to be conceptually and physically separate from the corresponding sequential I/O channels and devices).

Each device "saves" and "retrieves" data divided into records, which depending on the implementation may be of fixed or varying length. Each record possesses a unique "record address" assigned to it when the record is saved on the device. The record is retrieved by accessing the device with the same record address.

The format of data saved on the device is implementation dependent, but is generally taken to be in a binary core image form.

Execution of a HAL/S FILE statement causes the specified device to save or retrieve one record[32] of data whose record address is also specified.

- The saving of a record is caused by executing a “write-mode” FILE statement.
- The retrieving of a record is caused by executing a “read-mode” FILE statement.

WRITE-MODE FILE STATEMENT

The write-mode FILE statement is used to save a record with a given record address on a specified random-access device. Its form is as follows:

```
FILE(n, address) = exp;
```

1. n is the channel code number of the specified device, and is an integer in the range 0 ≤ n ≤ 9†.
2. *address* is an unarrayed integer expression whose resultant value is the record address, and must be a legal address for any given implementation.
3. *exp* is an expression of any of the following types:

```
INTEGER      BIT STRING(or BOOLEAN)
SCALAR       CHARACTER
VECTOR       STRUCTURE
MATRIX
```

and possibly possessing arrayness in the sense of Section20.2.

† This value may vary between implementations. See appropriate User’s Manual.

The action of the write-mode FILE statement is to save the value or values of the expression on the right of the assignment as one record with the specified address, on the specified device.

Examples:

```
  STRUCTURE Q:
     1 QA CHARACTER(80),
     1 QB SCALAR,
     1 QV VECTOR(9);
  DECLARE ZQ Q-STRUCTURE(20);
  DECLARE P ARRAY(1000) SCALAR;
  .
  .
  .
  .
  .
  .
  FILE(1,I) = ZQ        ;
S              10 TO 20
  FILE(2, I+2) = P;
  FILE(1, 160) = SIN(ZQ.QB  );
S                         5
```

[32] A conceptual HAL/S record may or may not be equivalent to a “logical record” of a particular operating system on a particular machine.

READ-MODE FILE STATEMENT

The read-mode FILE statement is used to retrieve a record with a given record address from a specified random-access device. Its form is as follows:

```
var =FILE(n, address);
```

1. n is the channel code number of the specified device, and is an integer in the range $0 \leq n \leq 9$[†].
2. *address* is an unarrayed integer expression whose resultant value constitutes a record address of an existing record on the device.
3. *var* is a subscripted or unsubscripted data item of any of the following types:

```
INTEGER   BIT STRING(or BOOLEAN)
SCALAR    CHARACTER
VECTOR    STRUCTURE
MATRIX
```

If it is of structure type, it may possess multiple copies, else it may be arrayed.

[†] This value may vary between implementations. See appropriate User's Manual.

The action of the read-mode FILE statement is to retrieve the value or values in the record with the specified address and assign them into the data item on the left of the assignment.

Various restrictions are placed on the kind of data item appearing in a read-mode FILE statement.

- Input parameters are excluded.
- Bit string and character types may not possess component subscripting.
- Partitioning component subscripting on vector and matrix data items is illegal, unless subscripting reduces to a single item.
- Partitioning array subscripting on arrayed data items is illegal.
- Partitioning structure subscripting on structure terminals or minor structures is illegal.

Examples:

```
 STRUCTURE Q:
   1 QA CHARACTER(80),
   1 QB SCALAR,
   1 QV VECTOR(9);
 DECLARE ZQ Q-STRUCTURE(20);
 DECLARE P ARRAY(1000) SCALAR;
 .
 .
 .
 ZQ = FILE(1,I);
 ZQ        = FILE(2, J+1);
S  1 TO 10
  P         = FILE(1, I+5); ← illegal array subscripting
S  1 TO 500
 ZQ.QA   = FILE(1, 140);     ← illegal component subscripting
S     1;1
 ZQ.QV        = FILE(1, 150);
S     10 TO 2
  ZQ.QV          = FILE(1, 160);← illegal component  subscripting
S      10;1 TO 2
 ZQ.QB         = FILE(1, I-1); ← illegal structure  subscripting
S     10 TO 15
```

Other restrictions due to the DENSE attribute of a data item are enforced.

See: Spec./10.2

COMPATIBILITY OF FILE OPERATIONS

It has been stated that when a write-mode FILE statement causes a device to save a record, the data saved is in general a binary image of the value specified in the statement. Because of this there is inherently no protection against retrieving the data by a read-mode FILE statement and assigning it to a completely different type of data item. It is the user’s responsibility to ensure that the intelligibility of data is maintained.

The behavior of a FILE statement when the size of the binary image of the data item or expression is different from the length of the record accessed, is implementation dependent.

Examples:

```
 DECLARE A ARRAY(1000) SCALAR,
         A1 ARRAY(1000) SCALAR,
         B ARRAY(1000) INTEGER DOUBLE,
         C ARRAY(1000) BIT(32),
         D ARRAY(10) CHARACTER(4);
.
.
.
  FILE(1,1) = A; ←———┐
                     │ --- compatibility assured since A and A1 are alike.
  A1 = FILE(1, 1); ←─┘
.
.
.
.
.
  FILE(1,2) = B; ←───┐
                     ├-- C will contain the bit pattern of B in an
  C = FILE(1,2); ←───┘   implementation where double precision
                         integers occupy 32-bits.
  FILE(1,3) = A;
  D = FILE(1,3); ←———    D will contain implementation dependent garbage:
                         error condition might occur if the binary range of
                         record 3 is too long to be contained in D.
```

This page intentionally left blank.

23.0 REAL TIME PROGRAMMING - II

Section 13 introduced some of the simpler aspects of HAL/S real time programming concepts. Real time processes, and their creation and execution by means of the SCHEDULE statement were described.

This section explains how real time processes can be created by invoking program blocks instead of task blocks, and the resulting implications when referencing one program from another.

This section also describes how real time processes can be scheduled to execute cyclically, and how cycling of execution can be arrested by means other than executing a TERMINATE statement.

23.1 PROGRAM PROCESSES

Section 13.1 explained that at run time, the dynamic counterpart of a HAL/S program is a real time process executing under control of a Real Time Executive (RTE). It stated that this “primal process” could create other processes whose static counterparts are task blocks embedded in the program block. However, it is also possible to create processes whose static counterparts, rather than being task blocks, are other program blocks. In order to avoid confusion, in the remainder of this Section the program block corresponding to the primal process will be called the “primal program”.

The program blocks which are invoked by SCHEDULE statements causing the creation of new processes, are the same in every respect as the primal program block: they are separately compiled blocks of code. The scheduling of program processes therefore requires the bringing together of a number of compilation units at run time.[33]

This situation is analogous to the invocation of external procedures and functions as described in Section 15, and is shown pictorially below:

[33] The object modules resulting from their compilation have to be “link-edited" to produce a single executable load module. The way in which the primal program is distinguished from the others in such a load module is extra-lingual and implementation dependent.

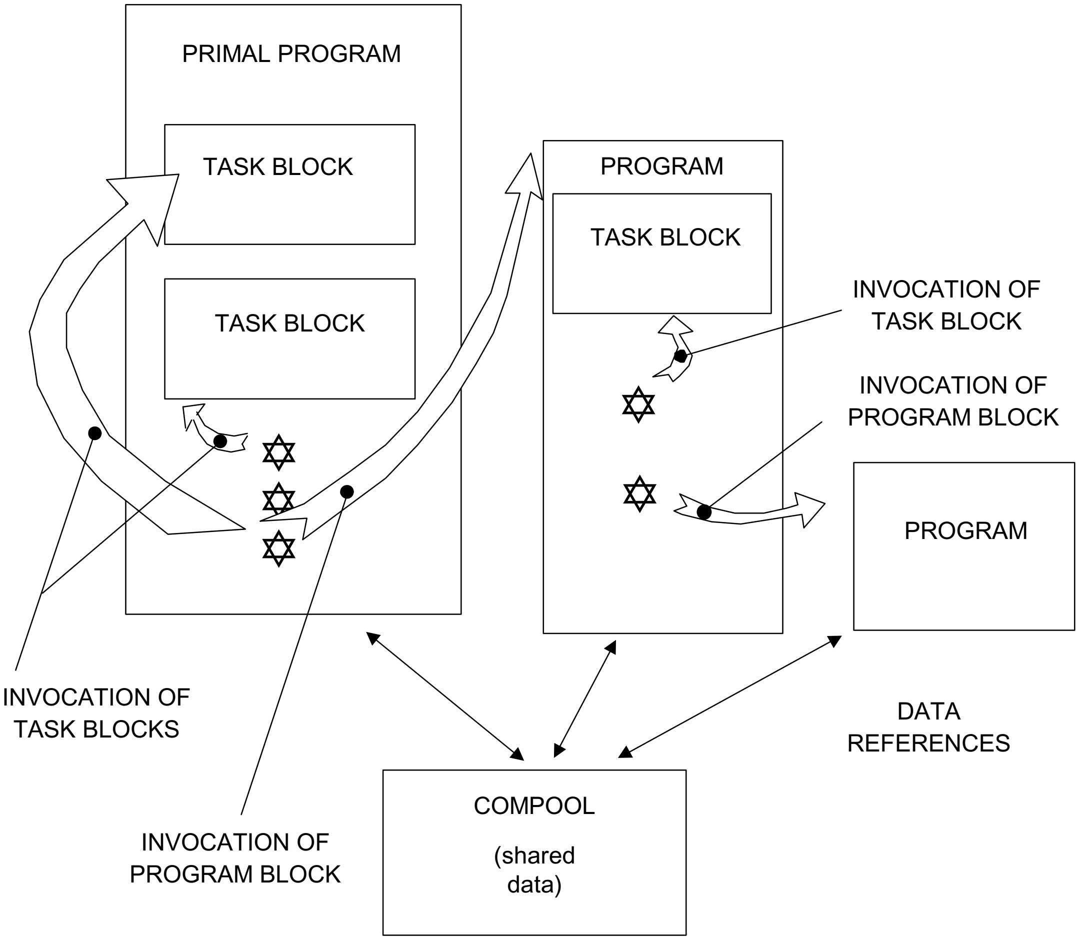


Figure 23-1

A program may invoke any other program in the same assemblage of compilation units, or invoke any task block within itself, in order to create a new process. The programs will probably need to share data in one or more compools, and may also share the use of comsubs (not shown)[34].

Any program which creates a program process, or otherwise controls its execution, perforce contains references to the program block which is the process' static counterpart. The first program must, under these circumstances, be provided with a block template of the program block referenced. The program template is included in the compilation unit of the first program, in the same way as if it were a compool or comsub template.

[34] Interfaces with compools and comsubs have been described in Section 15.

Expanding the example given above, the location of program templates is as follows:

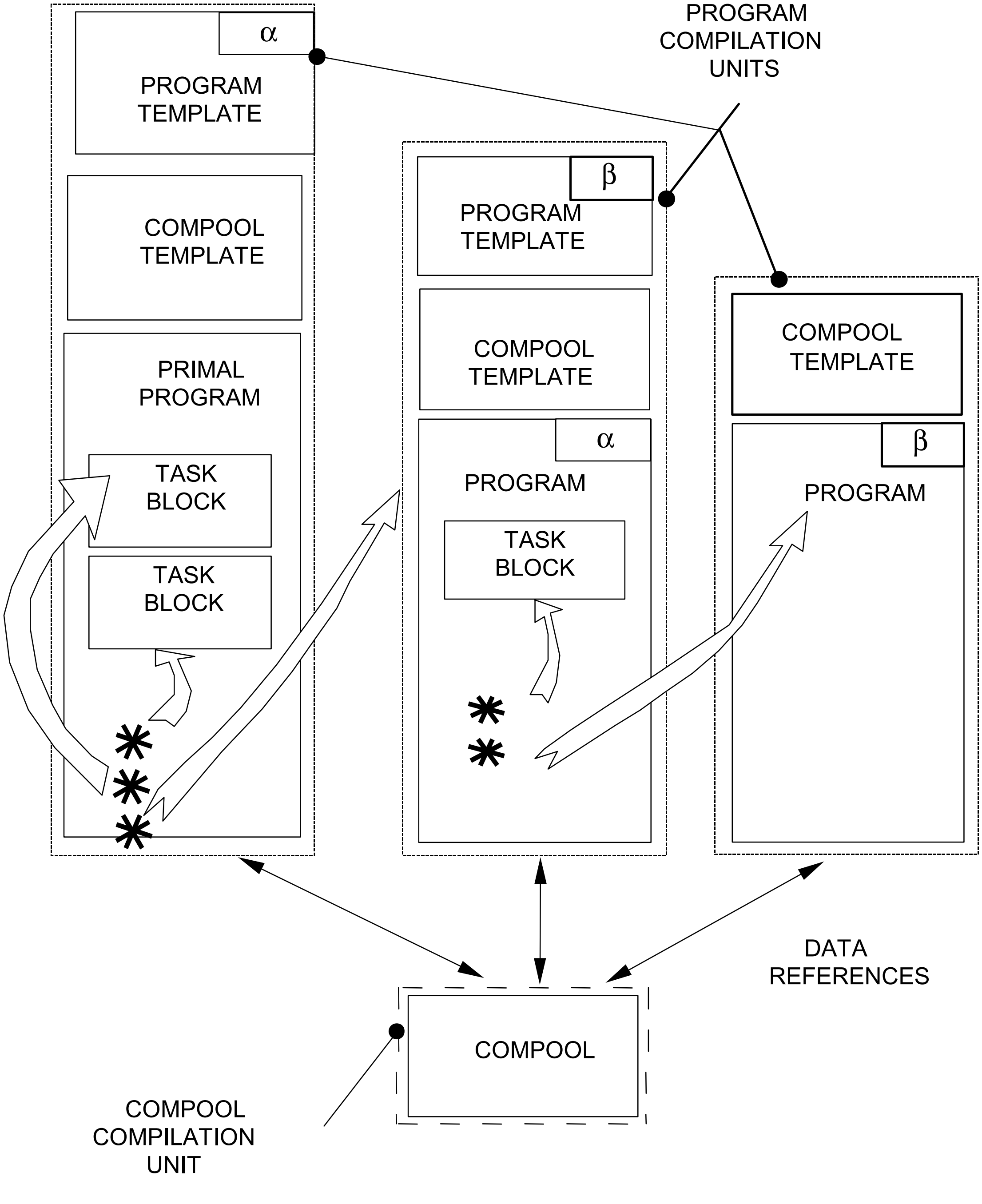


Figure 23-2

External procedure and function blocks, as well as program blocks, may contain SCHEDULE statements for creating processes. However, because external procedure and function blocks may not contain task block definitions, only program processes may be created thereby.

To ensure correctness of version, program templates would be subject to the same implementation dependent software management scheme as for compool and comsub templates (see Section 15.1).

23.2 PROGRAM TEMPLATES

If a program template is included with a compilation unit, then that compilation unit may invoke the corresponding program to create a new real time process.

A program template differs in the following respects from its corresponding program:

- the body of the block is empty;
- the opening statement is modified as shown by the keyword EXTERNAL.

```
label: EXTERNAL PROGRAM;
```

1. *label* is the name of the corresponding program.

Example:

program block:

```
           .
     ONE:  PROGRAM;
           DECLARE I INTEGER;
           .
           .
           .
           .
           I = I + 1;
           .
           .
           .
           .
           CLOSE ONE;
           .
```

corresponding program template;

```
           .
     ONE:  EXTERNAL PROGRAM;
           CLOSE ONE;
           .
```

23.3 CREATING AND CONTROLLING PROGRAM PROCESSES

Process created by invocation of a program differ very little from processes created by invocation of a task block.

- A program process is created by a SCHEDULE statement precisely as described in Section 13.4.
- A program process is forced into the inactive state and removed from the process queue by means of the TERMINATE statement as described in Section 13.5.
- A program process may be forced into the waiting state by execution of a WAIT statement, as described in Section 13.5.
- The priority of a program process may be updated by the UPDATE PRIORITY statement, as described in Section 13.5.
- The major state of a program process may be ascertained by using the name of the process as a Boolean variable, also as described in Section 13.5.

Only the notion of process dependency as used in the constructs mentioned, need be updated to allow for the existence of program processes.

PROGRAM PROCESSES AND PROCESS DEPENDENCY

Section 13.1 introduced the concepts of the dependency of one process upon another. The basic notion of dependency still stands:

When a process *A* creates process *B*, the latter may be specified as "dependent" on the former, or "independent" of it. If *B* is dependent on *A*, then it depends for its existence on the existence of *A*. If *B is* independent of *A*, then *A* may cease to exist without affecting the existence of *B*.

If *B* is a program process, this is always unequivocally true. However, if *B* is a task process, as stated in Section 13.1, there exists an overriding rule. Reinterpreted, this rule states that a task process *C*, however created, is always dependent on the program process whose static counterpart contains the task block whose invocation caused *C* to be created.

Example:

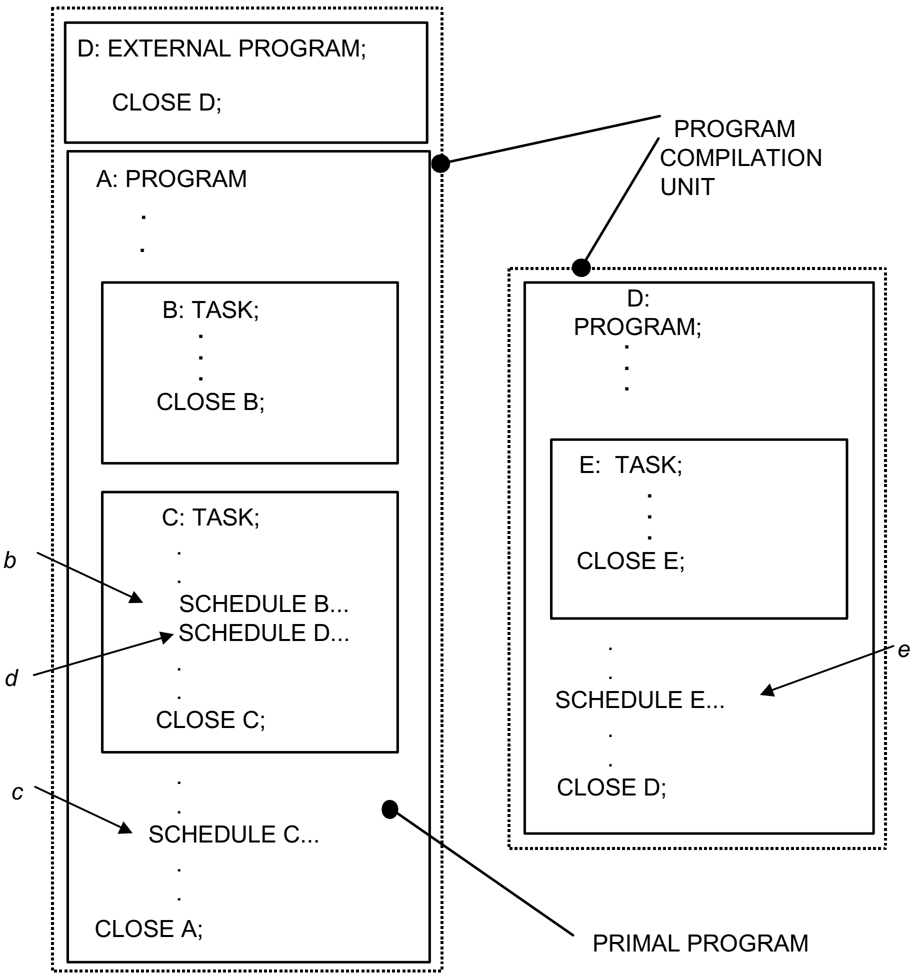


Figure 23-3

A is the primal process; execution of SCHEDULE statements at *b , c , d ,* and *e* cause the creation of other processes designated INDEPENDENT by default as follows:

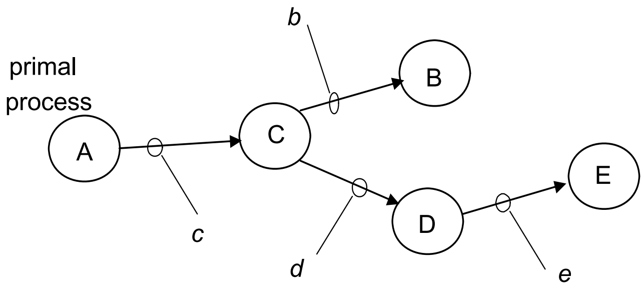


Figure 23-4

By the rules given above, only D is a truly independent process (apart from the primal process, to which the concept does not apply).

Although B is independent of C, both B and C are in actuality dependent on A. E is in actuality dependent on D.

23.4 CYCLIC PROCESSES

Hitherto, a real time process has been characterized as being in the active state for some duration, wherein it is either ready, executing, or waiting. As described in Section 13.3, such a process finally returns to the inactive state when one of two conditions are met:

- the process is terminated by a TERMINATE statement.
- execution reaches a RETURN or CLOSE of the related static program or task block.

In either circumstance, the process makes only one pass through the HAL/S code contained in the related program or task block. Subsequent passes through the same code would thus require the scheduling of a new process for each pass. Because of the uniqueness requirement stated in Section 13.4, each new process could only be created when the previous one returned to the inactive state.

To avoid the burden of continual intervention otherwise required to maintain cyclic execution of a program or task block, HAL/S supports cyclic real time processes. Cyclic real time processes are created by an extension of the SCHEDULE statement described in Section 13.4. Without further intervention, the process will, during execution, make an arbitrary number of passes through the code in the related program or task block until some predetermined condition is met.

STATES OF A CYCLIC PROCESS

The possible states of a cyclic process are the same as those of a non-cyclic process, as described in Section 13.1

When a cyclic process is created invoking a program or task block from a SCHEDULE statement, the process makes a transition from the inactive state to the active state. It is entered on the process queue in the ready or waiting state, according to the same criterion as for a non-cyclic process.

When the cyclic process is first elevated to the executing state by the RTE, it begins the first pass through the code of the related program or task block. Unless otherwise prevented, execution will eventually reach a RETURN or CLOSE statement in the block, whereupon the process will go into a waiting state until predetermined conditions for the beginning of the next cycle are met. At the expiration of this waiting period, the process is returned to the ready state. The relative priority of the cyclic process then determines when the next cycle of execution begins.

A cyclic process can return to the inactive state in one of two ways:

- by being terminated through execution of a TERMINATE statement;
- by being "canceled" at the end of the current cycle of execution, either because some prespecified condition is met, or through the execution of a CANCEL statement.

The implications of "cancellation" as opposed to termination will be examined in Section 23.6.

23.5 SCHEDULE STATEMENT FOR CYCLIC PROCESSES

The form of a SCHEDULE statement for creating cyclic processes is an extension of that for creating non-cyclic processes. The cyclic SCHEDULE statement conveys two additional items of information:

- a condition for starting each new cycle of execution;
- a cancellation condition.

There are several versions, depending on the way in which the above conditions are specified.

IMMEDIATE RECYCLING

The simplest conversion of cyclic SCHEDULE statement is one in which a new cycle of execution of the process is specified to start immediately after the end of the previous cycle. This form is shown below:

```
SCHEDULE label initiation, REPEAT UNTIL time;
```

1. A process called *label* is created from the corresponding program or task block.
2. *initiation* specifies a priority, and optionally an initiation condition and dependency of the new process, as described in Section 13.4.
3. The keyword REPEAT signifies that the process is to be cyclic. By default one cycle is to follow another with no interval in the waiting state.
4. UNTIL *time* specifies a cancellation condition. *time* is a scalar expression which when evaluated at the time of scheduling gives the time in seconds† at which the process is to be canceled.
5. If the UNTIL phrase is absent, execution cycles indefinitely until inhibited by other means.

† After the real time origin.

Cancellation actually takes place at the end of the first cycle which finishes later than the specified time.

Example:

```
SCHEDULE A AT 1600 PRIORITY(50);
```

a non-cyclic schedule statement creating a process A to be initiated 1600 seconds after the real time origin.

```
SCHEDULE B AT 1600 PRIORITY(50), REPEAT UNTIL 3200;
```

a cyclic schedule statement creating a cyclic process B to be initiated 1600 seconds after the real time origin, and to cease cycling at the end of the first cycle completed after 3200 seconds.

The state transitions of these processes are illustrated diagrammatically below:

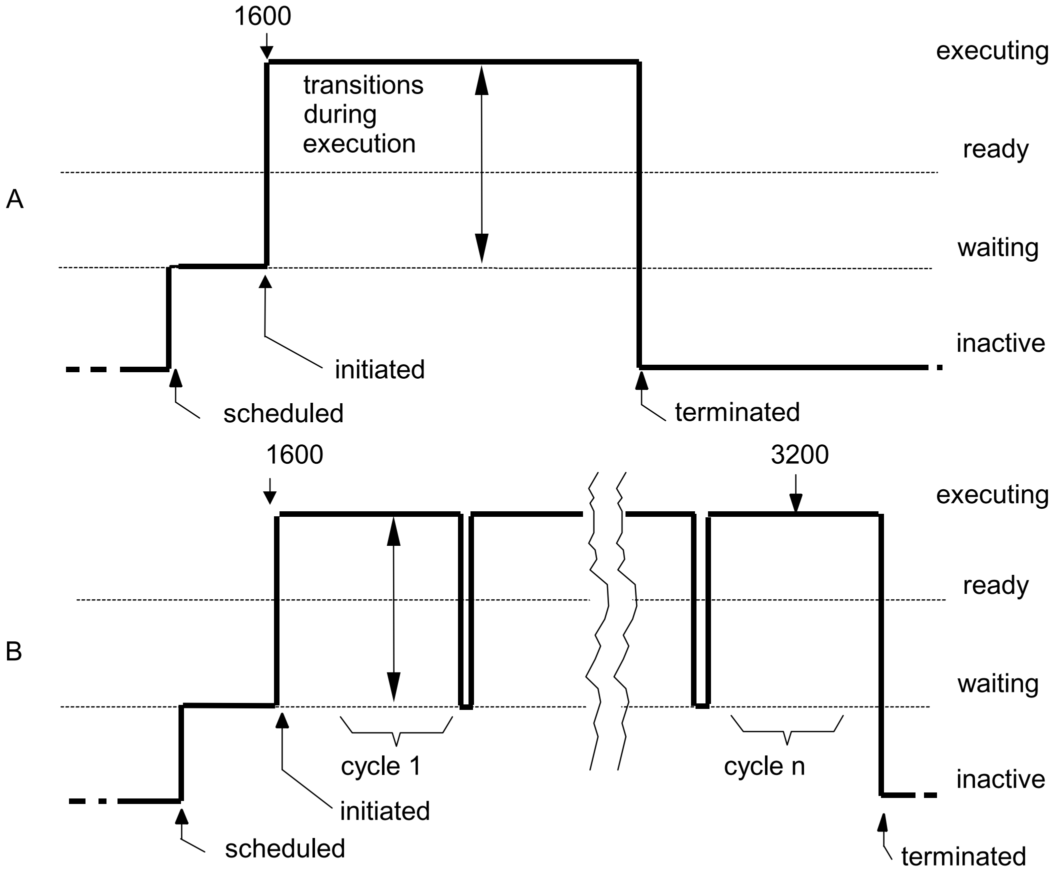


Figure 23-5

Note that the following case causes a run time error:

```
SCHEDULE C AT 1600 PRIORITY(50), REPEAT UNTIL 1000;
```

because the initiation time is later than the time at which cycling is to cease.

CONSTANT INTERCYCLE DELAY

The second version of cyclic SCHEDULE statement specifies a constant delay between cycles of execution. This form is shown below:

```
SCHEDULE label initiation, REPEAT AFTER delay UNTIL time;
```

1. A process called *label* is created from the corresponding program or task block.
2. The meaning of *initiation* and *time* are the same as for the previous version of cyclic SCHEDULE statement.
3. AFTER *delay* specifies a constant delay between the end of one cycle of execution and the start of the next. *delay* is a scalar expression whose value at the time of scheduling specifies the delay in seconds.

Cancellation takes place in the same way as before, with the provision that if the cancellation condition is met in the interval between cycles, cancellation takes place immediately.

Example:

```
SCHEDULE A AT 1600 PRIORITY(50), REPEAT AFTER 100 UNTIL 3200;
```

A cyclic process A is scheduled, specifying a delay of 100 seconds between cycles of execution. The state transitions of this process may be illustrated diagrammatically as follows:

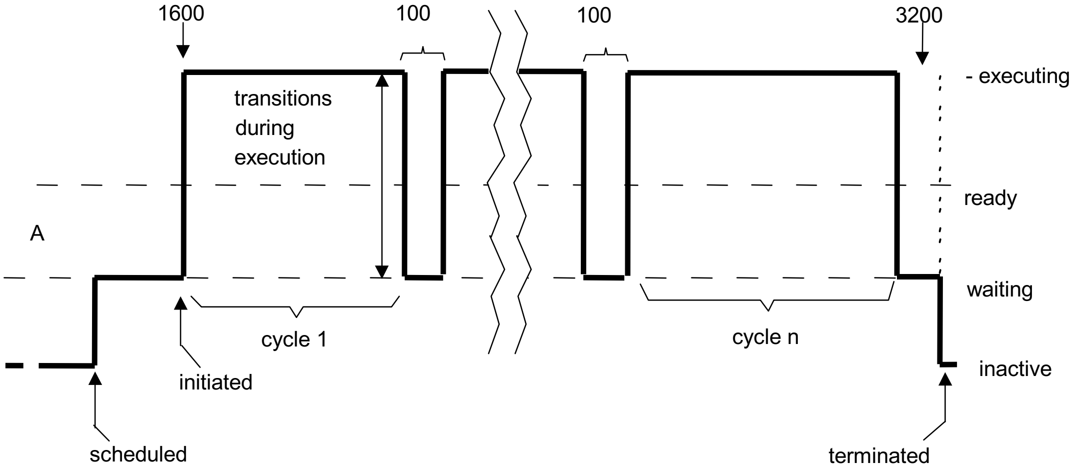


Figure 23-6

RECYCLING AT SPECIFIED INTERVALS

The third and last version of cyclic SCHEDULE statement specifies that each new cycle is to start a fixed interval of time after the start of the previous cycle. This form is shown below:

```
SCHEDULE label initiation, REPEAT EVERY interval UNTIL time;
```

1. A process called *label* is created from the corresponding program or task block.
2. The meaning of *initiation* and *time* are the same as for the previous two versions of cyclic SCHEDULE statement.
3. EVERY *interval* specifies that each cycle is to start a given interval after the start of the previous cycle. *interval* is a scalar expression whose value at the time of scheduling specifies the interval in seconds.

Cancellation takes place in exactly the same manner as with the previous version of the SCHEDULE statement.

Example:

```
SCHEDULE A AT 1600 PRIORITY(50), REPEAT EVERY 200 UNTIL 3200;
```

A cyclic process A is schedule, specifying that cycles are to succeed each other at intervals of 200 seconds. State transitions may be illustrated diagrammatically as follows:

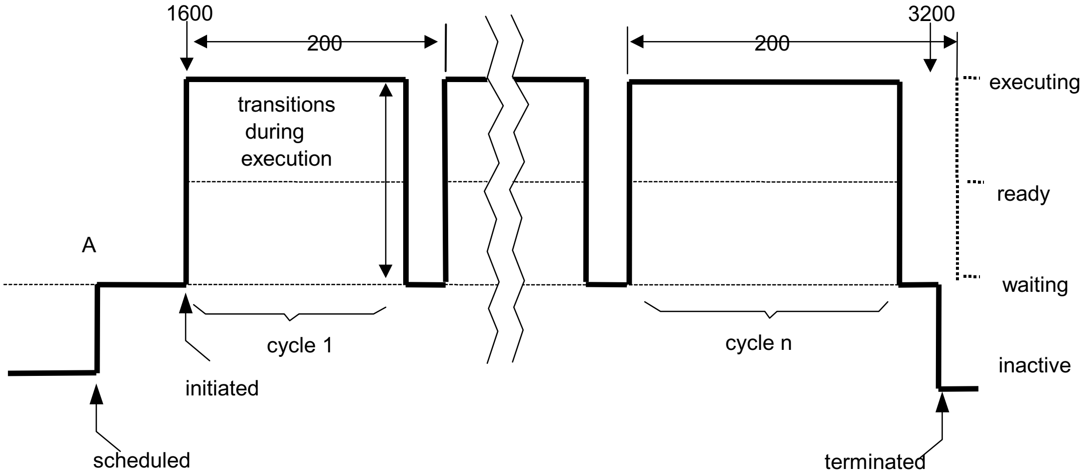


Figure 23-7

Note that if a cycle takes longer than 200 seconds to execute, the next cycle cannot start on time and a run time error occurs.

An UNTIL phrase can also be used in a non-cyclic SCHEDULE statement.

See: Spec./8.3.

23.6 TERMINATING AND CANCELING CYCLIC PROCESSES

When a cyclic statement is terminated by execution of the TERMINATE statement described in Section 13.5, both the process and its dependents are terminated, possibly in mid-cycle.

Cancellation is a more graceful way of termination. It cannot occur when a process is in mid-cycle. Further, when a process is canceled, its dependents are not terminated immediately: the following happens instead:

- non-cyclic dependents are allowed to execute until their normal termination;
- cyclic dependents are allowed to finish their own current cycle of execution.

The process being canceled is put in a waiting state until all its dependents have become inactive; it then becomes inactive itself.

Cancellation conditions in SCHEDULE statements cannot be dynamically modified. To cancel a cyclic process arbitrarily, the CANCEL statement must therefore be used.

CANCEL STATEMENT

A CANCEL statement specifies the cancellation of a process. Its form is as shown below:

```
CANCEL label;
```

1. The appearance of *label* is optional. If present, the statement causes cancellation of the active process called *label*.
2. If *label* is absent, the process executing the CANCEL statement is itself canceled.

The effect of a CANCEL statement is as follows:

- If the process has not yet been initiated, it is terminated and removed from the process queue.
- If the process is in a cycle of execution, it is canceled at the end of the cycle.
- If the process is waiting between cycles, it is canceled immediately.

CANCEL statements can actually be applied to non-cyclic processes, but unless the process has not yet initiated they have no effect. If the process has not been initiated, the process is removed from the process queue, just as if it were cyclic.

Examples:

```
CANCEL;  self cancellation
CANCEL BETA;
```

If a number of processes are to be canceled simultaneously, the CANCEL statement can specify a list of process names:

```
CANCEL ALPHA, BETA, GAMMA;
```

This page intentionally left blank.

24.0 REAL TIME PROGRAMMING - III

This section concludes the description of HAL/S constructs for real time programming, which was begun in Section 13 and continued in Section 23. The remaining topic of discussion is a HAL/S construct called the "event", and its use in real time programming.

The original idea behind the HAL/S "event" was that it should serve as an interface between HAL/S software and hardware interrupts; that is, the medium through which the arrival of interrupts would be signaled to the HAL/S program. Hence, the HAL/S "event" was conceived as a Boolean-valued data item, normally FALSE in value, but either becomes transiently TRUE, or latching TRUE, on the arrival of the interrupt[38]. The assumption was that the values of "events" at any given time could control the execution of real time processes by the RTE.

An extension of this idea was the definition of the ability to simulate the arrival of interrupts by changing the values of "events" within the HAL/S software itself.

However, the underlying operating systems of most machines do not allow for interfaces with interrupts of the above nature. Hence, the simulation property of "events" has become their major role: the ability to signal a software condition in one real time process synchronously to other processes by use of HAL/S "events" has become a real time programming tool of considerable importance.

24.1 HAL/S EVENTS

A HAL/S event is a Boolean-valued data item whose value is visible at any instant to the RTE. Except for this latter qualification, whose importance will be appreciated later, an event differs little from the Boolean data item first introduced in Section 4 of Part I.

A HAL/S event may optionally possess a "latching" property:

- An event with the latching property may be set in value to either TRUE or FALSE.
- An event without the latching property is normally FALSE in value, but may transiently become TRUE (for an "infinitesimal" time) when so specified.

The values of events can be changed only by special HAL/S statements, not by simple assignments.

Event expressions consisting of logical operations on event data items can be synthesized: the instantaneous values of such event expressions can be used to modify the activity of the RTE in controlling real time processes. Event expressions can be used in the following circumstances:

- in a SCHEDULE statement, to specify a condition for initiating a process;
- in a cyclic SCHEDULE statement, to specify a cancellation condition;

[38] Clearly, there would need to be some extra-lingual, implementation dependent way of relating particular "events" to particular hardware interrupts.

- in a WAIT statement, to specify a condition for ending the period a process is to remain in the waiting state.

In addition, in most contexts events can be used in Boolean or bit string expressions as if they were Boolean data items.

24.2 DECLARATION OF EVENT DATA ITEMS

The declaration of event data items is similar to the declaration of Boolean data items as described in Section 4.2 of the Guide. The basic forms are as follows:

```
DECLARE name EVENT;
DECLARE name EVENT LATCHED;
```

1. In each form, *name* is any legal HAL/S identifier.
2. The keyword LATCHED signifies that the event is to possess the “latching” property. Its absence signifies that it is not to possess it.

Examples:

```
DECLARE EV1 EVENT;
DECLARE EV2 EVENT LATCHED;
```

COMPOUND DECLARATIONS

Declaration of events may be mixed with declarations of other data types in compound declarations:

```
DECLARE A SCALAR,
        I INTEGER DOUBLE,
        E EVENT LATCHED;
```

The keyword LATCHED is an attribute which may be factored.

Example:

```
DECLARE El EVENT LATCHED,
        E2 EVENT LATCHED,
        E3 EVENT LATCHED;
```

may be rewritten more compactly as

```
DECLARE EVENT LATCHED, E1,E2,E3;
```

INITIALIZATION

All declared event data items are implicitly initialized to a FALSE value[39]. Only an event data item with the latching property may possess explicit initialization. It is initialized as if it were a Boolean data item, as described in Section 4.3.

Examples:

```
DECLARE EV1 EVENT LATCHED INITIAL(TRUE);
DECLARE EV2 EVENT LATCHED CONSTANT(OFF);
```

(Note: a constant event is of little use even though legal in HAL/S).

```
DECLARE EV3 EVENT INITIAL(TRUE);
```

- illegal since EV3 is not LATCHED.

ARRAYS OF EVENTS

An event data item may be arrayed, its array property being specified in the same way as described in Sections 4.2 and 18.1. Event arrays with the latching property may be initialized as described in Sections 4.3 and 18.2.

Examples:

```
DECLARE El ARRAY(5) EVENT;
DECLARE E2 ARRAY(2,2) EVENT LATCHED INITIAL(4#TRUE);
```

EVENTS IN STRUCTURES

A terminal node in a structure may not be an event (See Section 19).

24.3 EVENT EXPRESSIONS

It was stated that event data items could appear in Boolean or bit string expressions as if they were Boolean data items. It is possible that the operands of a Boolean expression could solely be event data items. It is stressed that even in this circumstance the expression is in general still taken to be a Boolean expression. The term "event expression" is reserved for a special purpose.

An event expression is an expression composed in general of a series of logical operations upon event operands in the context of a SCHEDULE or WAIT statement. The simplest case of an event expression is a lone event operand.

An event expression has the curious property that its evaluation is under control of the RTE and may take place more than once at times other than that of execution of the SCHEDULE or WAIT statement it appears in.

[39] This is the only HAL/S data type which is implicitly initialized.

OPERATIONS AND OPERANDS

The operations legal in an event expressions are the Boolean operations described in Section 7.3.

Symbol	Purpose
& AND	logical intersection
| OR	logical conjunction
¬ NOT	logical complement

The behavior of the operations is exactly as if the operands were of Boolean data type rather than event.

The operands in an event expression are solely event data items. Operands which are event arrays must possess array subscripting which selects one, and only one array element. Such array subscripting is the same as used for the selection of array elements from Boolean arrays, and has been described in Section 6.2 and 18.3, with the exception that the ending colon is optional rather than mandatory.

Examples:

Given the following declarations

```
DECLARE EV1 EVENT,
        EV2 EVENT LATCHED,
        EV3 ARRAY(2,4) EVENT,
        EV4 EVENT;
```

in the contexts of SCHEDULE or WAIT statements, the following are legal event expressions:

(EV1&EV2)& EV4
(EV2|EV3$_{2,2:}$) ←colon optional

The following is illegal:

EV1|EV3$_{*,1}$
↑_________ subscripting does not select one element of EV3

EV1||EV2 illegal operator
EV1&TRUE
EV1|BIT$_1$(125) illegal operands

Note however, that the above are legal bit string expressions in the appropriate contexts.

EXECUTION OF EVENT EXPRESSIONS

It was stated earlier that event expressions are evaluated under direct control of the RTE, and not necessarily only at the time of execution of the SCHEDULE or WAIT statement in which they appear. The reason for this can now be explained.

Event expressions are placed in SCHEDULE and WAIT statements to provide dynamic conditions for controlling the execution of processes. On a basic level the conditions control the transition of processes from state to state, and thus the activity of the RTE in swapping processes.

Hence, it is appropriate to evaluate an event expression, not only at the time of execution of the SCHEDULE or WAIT statement it appears in, but subsequently whenever the value of any of its event operands is modified. This is why the values of events are visible to the RTE. Not only each event operand, but the entire event expression has to be accessible to the RTE so that it can perform re-evaluations when required.

If an event expression contains subscripting which has to be evaluated at run time, then the subscript calculation takes place only once, when the event expression itself is first evaluated upon the execution of the SCHEDULE or WAIT statement it appears in.

Example:

```
 DECLARE EV ARRAY(5) EVENT;
 DECLARE I INTEGER INITIAL(1);
 .
 .
 .
 .
 .
 .
 WAIT FOR EV  ;
S           I
 I=I+1;
```

The RTE first evaluates EV_I when the WAIT statement is executed, and thus is interested in the value of EV_1 since $I \equiv 1$. Whenever the expression is re-evaluated, it is the value of EV_1 which is examined, even though the value of I may since have changed.

24.4 CHANGING VALUES OF EVENTS

HAL/S uses a special terminology for the operation of changing event values.

- An event with the latching property is said to be "set" when its value is forced TRUE, and "reset" when its value is forced FALSE.
- An event without the latching property is said to be "signaled" when its value is transiently forced TRUE.

These operations are carried out by the HAL/S SET, RESET, and SIGNAL statements respectively. Changes in value of an event data item as a result of one of these statements is visible to the RTE for the reason outlined in Section 24.3.

SET AND RESET

The SET and RESET statements only apply to latched events, and force their values to TRUE, and FALSE respectively. The forms of the two statements are shown below:

```
SET var;
RESET var;
```

1. In either form, *var* is a latched event data item. If it is arrayed, it must possess array subscripting causing the selection of one and one only array element (See Sections 6.2 and 18.3).
2. SET causes the value of *var* to be forced TRUE; RESET causes it to be forced FALSE.

Examples:

Given:

```
 DECLARE EV1 EVENT LATCHED,
         EV2 EVENT,
         EV3 ARRAY(3) EVENT LATCHED;
```

the following are legal:

```
 SET EV1;
 RESET EV3 ;
S          3
```

whereas the following are illegal

```
 SET EV2;    event not latched
 SET EV3;    more than one element specified
```

Note that the SET statement does not cause an event which is already TRUE to change in value. Neither does the RESET statement cause an event which is already FALSE to change in value. Hence, the RTE does not necessarily always sense an event change when such a statement is executed.

SIGNAL

The primary purpose of the SIGNAL statement is to cause the value of an event without the latching property to become transiently TRUE. However, it also has an effect on latched events, which will be described. The form of the SIGNAL statement is as follows:

```
SIGNAL var;
```

1. *var* is any event data item. If it is arrayed, it must possess array subscripting causing the selection of one and one only array element (See Sections 6.2 and 18.3).
2. If *var* does not have the latching property, SIGNAL causes its value to become transiently TRUE.
3. If *var* has the latching property, SIGNAL causes its value to be transiently complemented.

Examples:

Given:

```
 DECLARE EV1 EVENT LATCHED,
         EV2 EVENT,
         EV3 ARRAY(3) EVENT;
```

the following are legal:

```
 SIGNAL EV1;
 SIGNAL EV2;
 SIGNAL EV3 ;
S          3
```

whereas the following is illegal:

```
 SIGNAL EV3      ;
S          2 TO 3
```

because more than one array element is selected.

The SIGNAL statement always causes a change in value of an event, so that the RTE always senses an event change when it is executed. However, the RTE only senses the leading edge of the transient, not the trailing edge.

Example:

If EV1 and EV2 are declared thus:

```
 DECLARE EV1 EVENT,
         EV2 EVENT LATCHED INITIAL(TRUE);
```

then when

```
SIGNAL EV1;
```

is executed, EV1 changes in value thus:

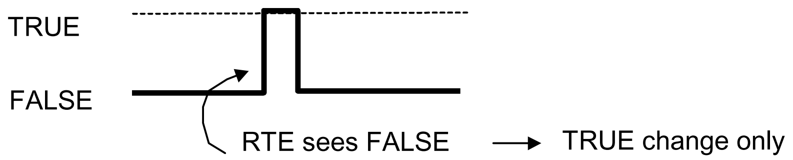


Figure 24-1

When

```
SIGNAL EV2;
```

is executed, EV2 changes in value thus:

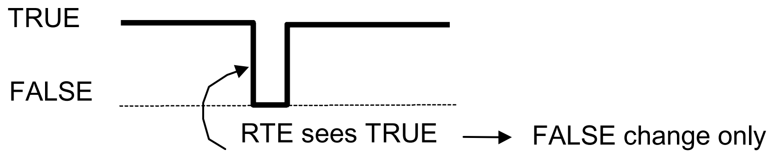


Figure 24-2

SUMMARY

The following table summarizes the effects of SET, RESET, and SIGNAL statements.

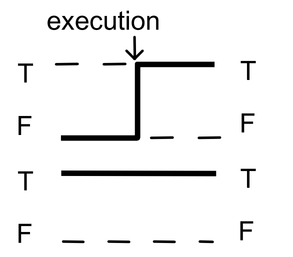
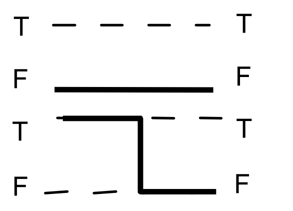
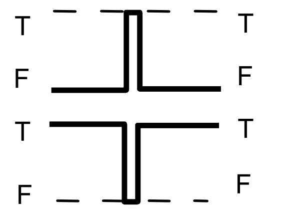
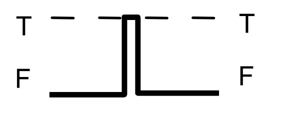
Statement	Event	T = TRUE, F = FALSE	
		Actual Value	Change sensed by RTE
SET	latched		F → T none
RESET	latched		none T → F
SIGNAL	latched		F → T T → F
	unlatched		F → T

Figure 24-3

24.5 EVENT EXPRESSIONS IN SCHEDULE STATEMENT

Event expressions may appear in a SCHEDULE statement for two reasons:

- to specify a condition for initiating a process;
- to specify a condition for ceasing to cycle a process.

INITIATION ON AN EVENT CONDITION

Section 13.3 described two time conditions under which the initialization of a process created by the SCHEDULE statement could be delayed. A third means of delaying initiation is to delay it pending the value of some event expression becoming TRUE. The basic form of SCHEDULE statement for this is shown below.

```
SCHEDULE label ON exp PRIORITY(α) DEPENDENT;
```

1. A process *label* is created from the corresponding program or task block and placed on the process queue.
2. PRIORITY(α) and DEPENDENT have the same meanings as described in Section 13.3 for other forms of SCHEDULE statement.
3. *exp* is any event expression. If its value is TRUE, when the SCHEDULE statement is executed, the process is placed in the ready state.
4. If its value is FALSE, the process is placed in a waiting state until its value becomes TRUE, whereupon it is transferred to the ready state.

Example:

Let EV1 and EV2 be latched events with EV1 ≡ EV2 ≡ FALSE.

After execution of

```
SCHEDULE ALPHA ON EV1 & EV2 PRIORITY(50);
```

let first EV1 then EV2 become TRUE. Then the state transitions of process are as shown below -

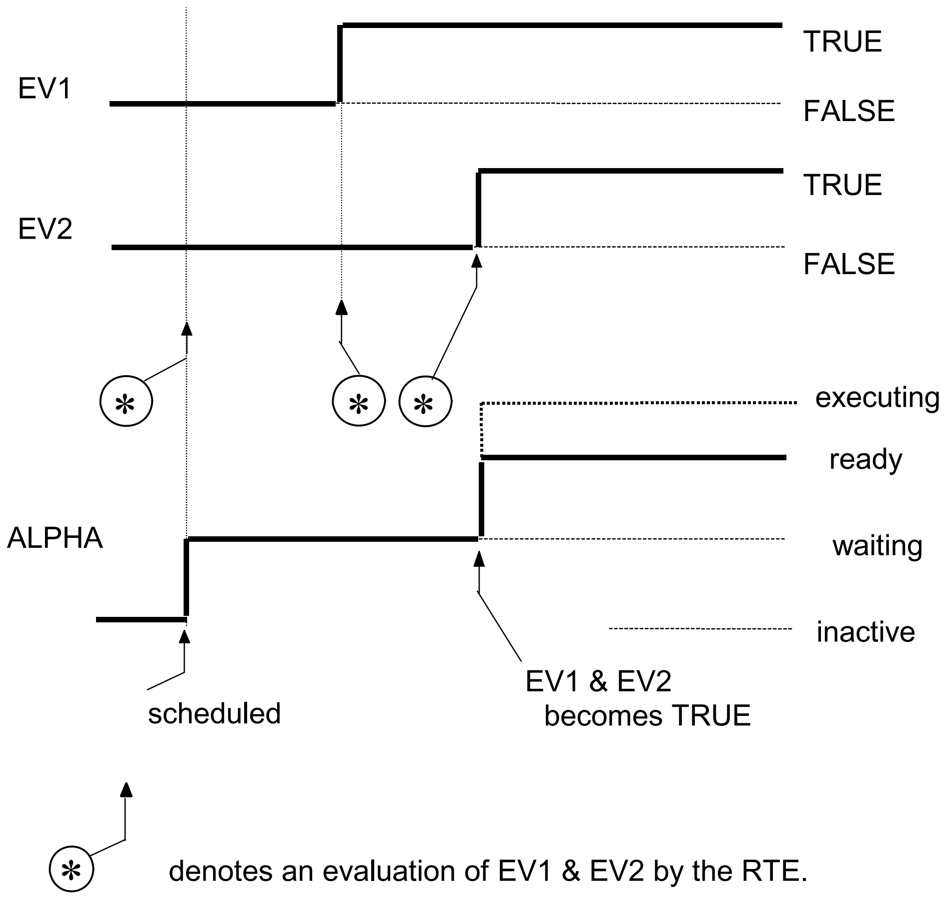


Figure 24-4

CANCELLATION OF AN EVENT CONDITION

Section 23.5 described three versions of cyclic SCHEDULE statement, in each of which the cancellation could be specified at a certain time. There are two ways of causing cancellation on an event condition:

- Cycling may be allowed to proceed while an event expression remains TRUE.
- Cycling may be allowed to proceed until an event expression becomes TRUE.
- CYCLING while TRUE

The following form of cyclic SCHEDULE statement causes cycling of execution to proceed while an event expression remains TRUE.

```
SCHEDULE label initiation , REPEAT cycle WHILE exp;
```

1. A process called label is created from the corresponding program or task block.
2. *initiation* specifies a priority, and optionally an initiation condition, and the dependency of the new process, as described in Section 13.4.
3. *cycle* optionally specifies a criterion for recycling execution as described in Section 23.5.
4. WHILE *exp* specifies that cycling is to continue while the value of *exp* remains TRUE. *exp* is any event expression.
5. If the value of *exp* becomes FALSE before the process is initiated, it is merely removed again from the process queue, and becomes inactive.

Cancellation of the process actually occurs at the end of the first cycle in which the event expression becomes FALSE[40]. If the event expression becomes FALSE in the interval between cycles, cancellation takes place immediately.

Example:

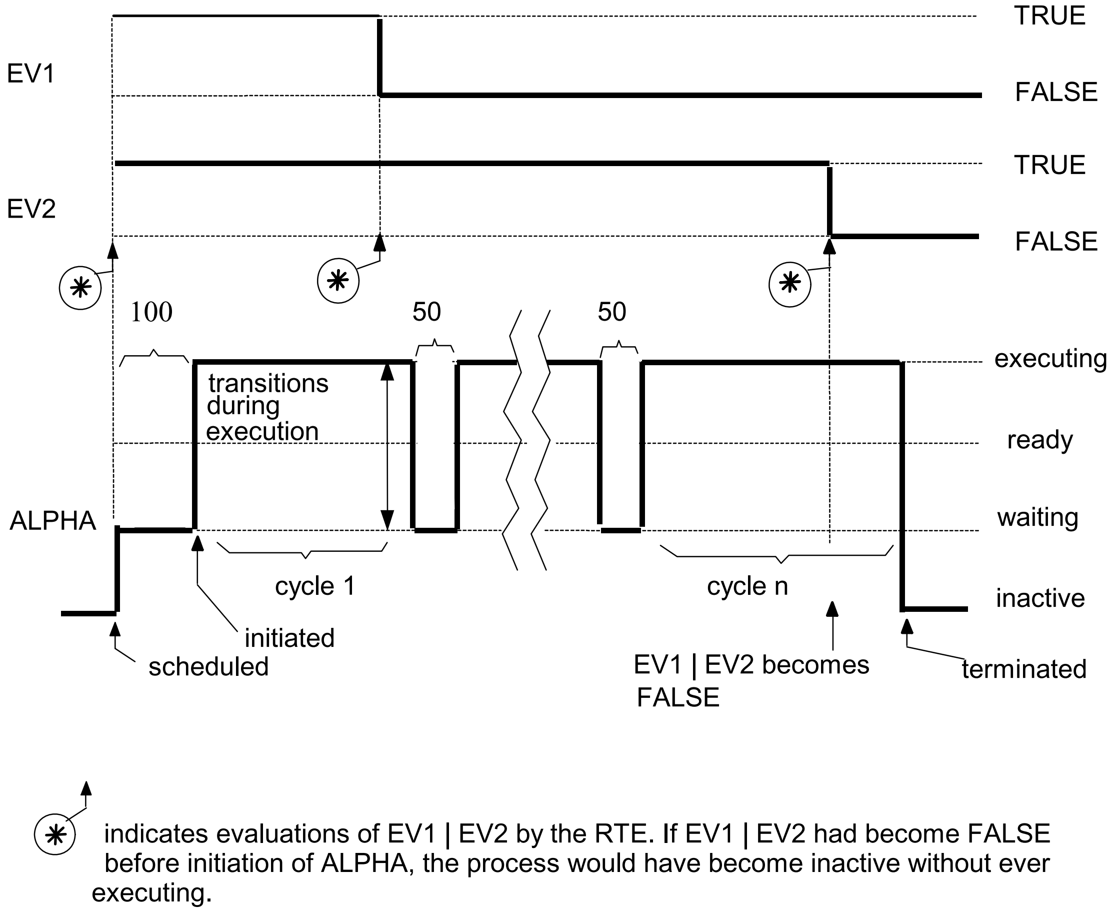
Given that EV1 and EV2 are latched events with EV1 ≡ EV2 ≡ TRUE

suppose that a cyclic process ALPHA is created by the following statement:

```
SCHEDULE ALPHA IN 100, PRIORITY(50), REPEAT AFTER 50 WHILE
EV1|EV2;
```

Let first EV1 and then EV2 become FALSE some time after initiation of ALPHA. The state transitions of ALPHA can then be illustrated diagrammatically as follows:

[40] Even if it subsequently becomes TRUE again during the same cycle.



indicates evaluations of EV1 | EV2 by the RTE. If EV1 | EV2 had become FALSE before initiation of ALPHA, the process would have become inactive without ever executing.

Figure 24-5

- CYCLING until TRUE

A modification of the above form allows cycling of execution to proceed until an event expression becomes TRUE. This is not merely a simple inversion of logic since the value of the event expression is not allowed to take effect until after the first cycle of execution of the process has started. In contrast to the above form, the following modification always allows at least one cycle of execution to be completed.

```
SCHEDULE label initiation , REPEAT cycle UNTIL exp;
```

1. A process called *label initiation* is created from the corresponding program or task block.
2. The meanings of *initiation* and *cycle* are as for the previous form of SCHEDULE statement.
3. UNTIL *exp* specifies that cycling is to continue until the value of *exp* becomes TRUE, with the provision that at least one cycle shall be executed. *exp* is any event expression.

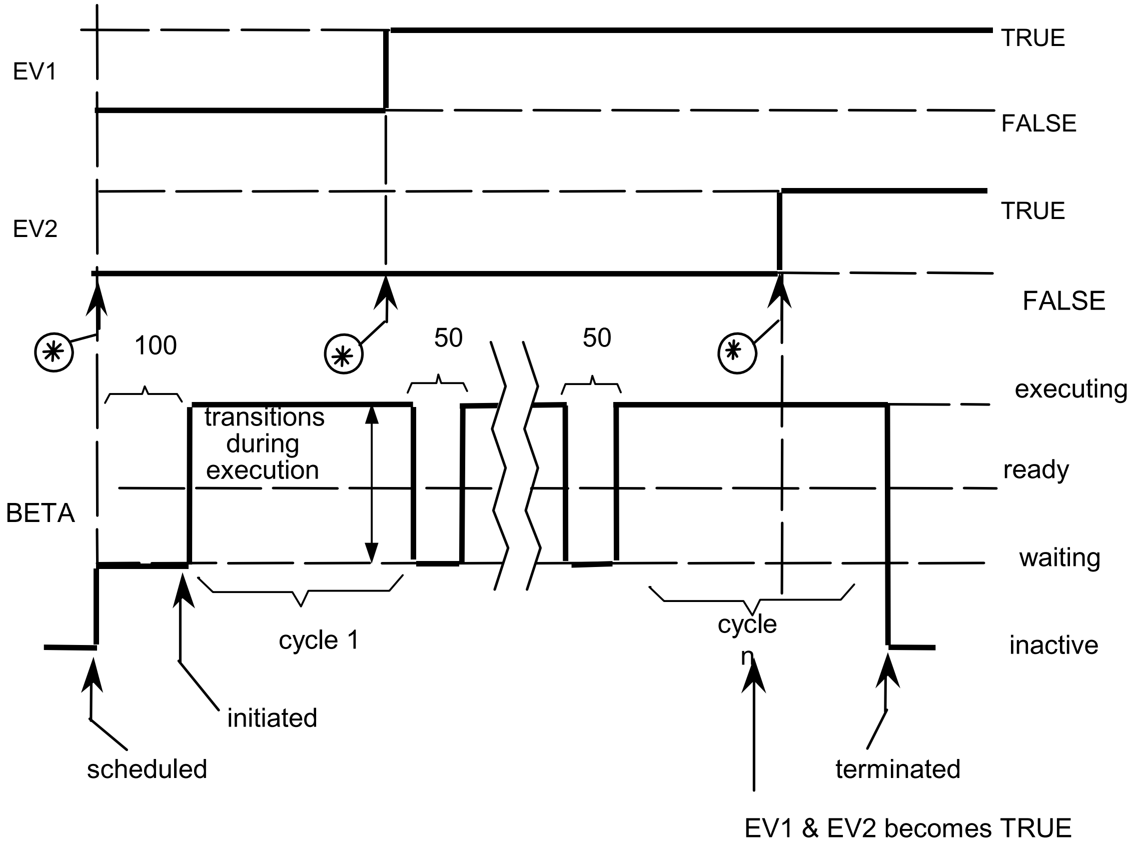
Cancellation of the process occurs at the end of the first cycle in which the event expression becomes TRUE[41]. If it becomes TRUE in the interval between cycles, cancellation takes place immediately.

Example:

Given that EV1 and EV2 are latched events with EV1 ≡ EV2 ≡ FALSE suppose that a cyclic process BETA is created by the following statement:

```
SCHEDULE BETA IN 100 PRIORITY(50), REPEAT AFTER 50 UNTIL EV1 & EV2;
```

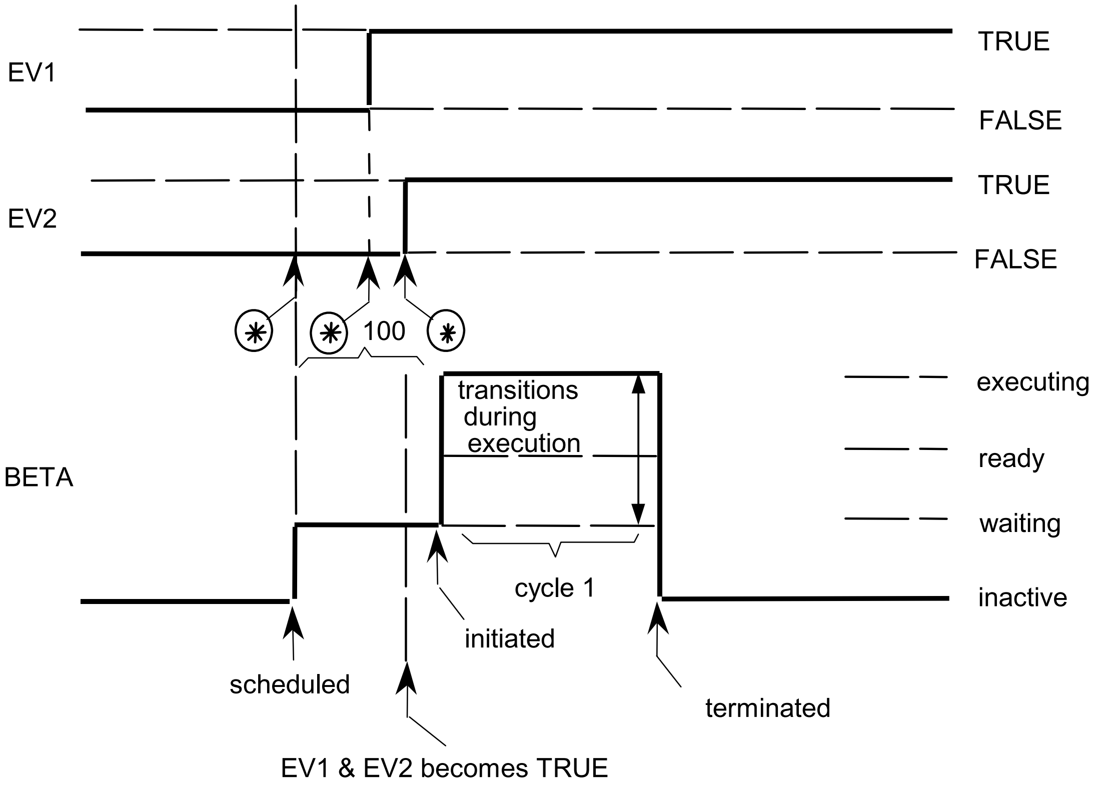
Let first EV1, and then EV2 become TRUE, some time after initiation of BETA. The state transitions of BETA can then be illustrated diagrammatically as follows:



✱ indicates evaluations of EV1&EV2 by the RTE. If EV1 and EV2 had both become TRUE before initiation of BETA, the following state transitions would have occurred:

Figure 24-6

[41] Even if it subsequently becomes FALSE again during the same cycle.



⊛ again indicates evaluations of EV1&EV2 by the RTE. Even though EV1 & EV2 becomes TRUE before initiation, the RTE postpones cancellation until the end of the first cycle.

Figure 24-7

The WHILE event expression is also allowed to appear in non-cyclic SCHEDULE statements.

See: Spec. /8.3.

24.6 EVENT EXPRESSIONS IN WAIT STATEMENT

Section 13.5 explained how the WAIT statement could be used to force a process into a waiting state until some timing condition is satisfied. The WAIT statement can alternatively specify an event condition. This causes a process to remain in a waiting state until some event expression becomes TRUE, whereupon the process returns to the ready state.

The form of this version of the WAIT statement is as follows:

```
WAIT FOR exp;
```

1. *exp* is any event expression.
2. The process executing the WAIT statement is placed in the waiting state until the value of *exp* becomes TRUE.
3. If *exp* is already TRUE when the WAIT statement is executed, the statement has no effect.

Example:

Given that EV1 and EV2 are latched events with EV1 ≡ EV2 ≡ FALSE

Suppose that

```
WAIT FOR EV1 & EV2;
```

is executed, and that some time later first EV1 and then EV2 become TRUE. Then the state transitions of the process executing the above statement are as shown below:

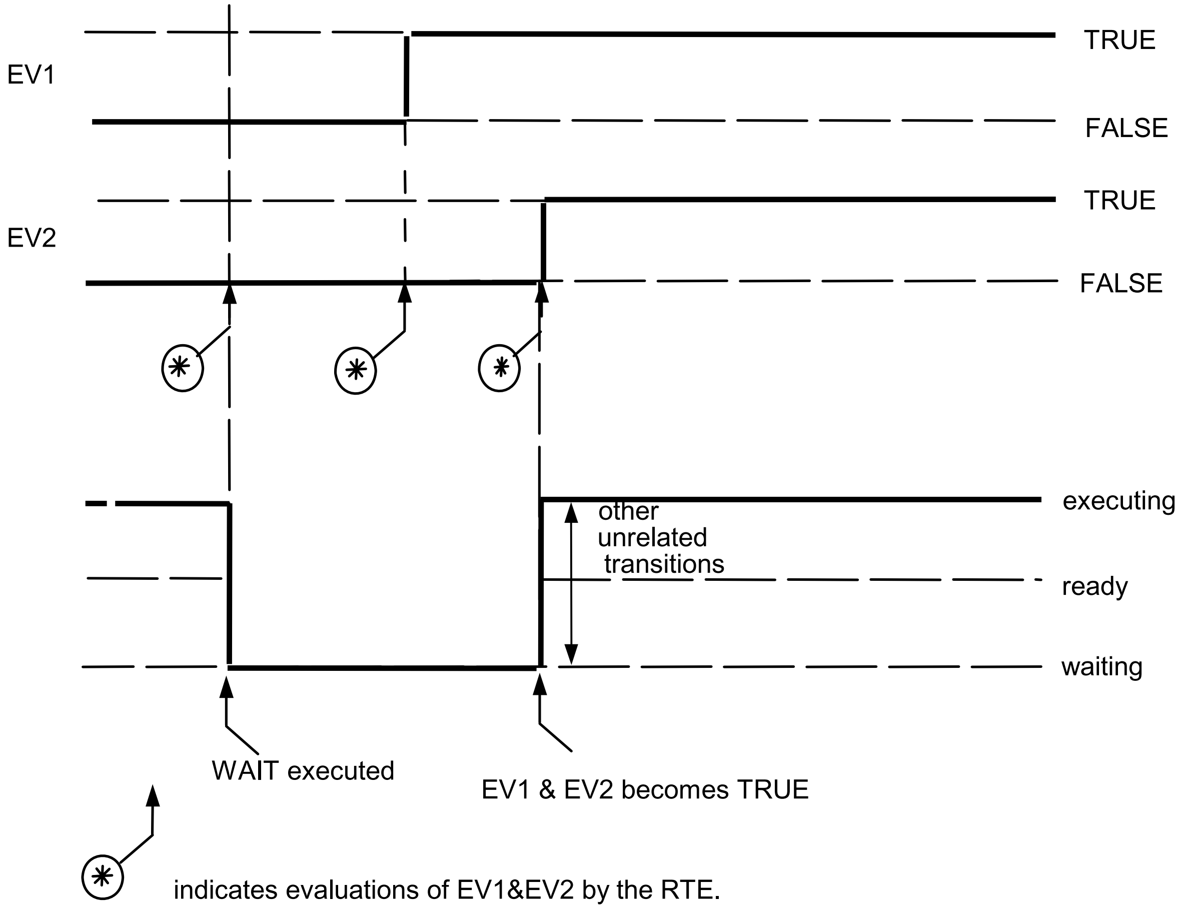


Figure 24-8

24.7 EVENTS IN BOOLEAN CONTEXT

This section presents some examples showing how events are used in Boolean or bit string context, reinforcing the remarks made in Section 24.3.

Examples:

Given the following declarations:

```
DECLARE B1 BOOLEAN,
        B8 BIT(8),
        EV1 EVENT,
        EV2 ARRAY(5) EVENT LATCHED,
        B16 ARRAY(5) BIT(16);
```

the following are legal bit string expressions:

BI||B8
EV1||B8
$EV2_2|(\neg EV1\&B16_{1:})$
$EV2_{3\ TO\ 5}||B16_{3\ TO\ 5:}$ (arrayed expression)

In

```
  IF EV1|EV2  THEN B1 = FALSE;;
S           1
```

EV1|EV2$_1$ is treated as a Boolean expression. It is only evaluated whenever the IF statement is executed, and is not under control of the RTE.

However, in

```
  WAIT FOR EV1|EV2  ;
S                 1
```

EV1 | EV2$_1$ is treated as an event expression as described in Section 24.6.

24.8 PROCESS EVENTS

Section 13.5 stated that the name of a process could be used as if it were a Boolean data item in order to determine the major state of the process. The names of processes can also be used in event expressions as if they were event data items. In this context they are called “process events”.

The truth table shows again the correspondence between logical value and major state.

State	Value
ACTIVE	TRUE
INACTIVE	FALSE

Example:

If EV1 is a latched event with EV1≡FALSE initially, and ALPHA is the name of an active process, then

```
WAIT FOR EV1 & (¬ALPHA);
```
[42]

causes the following state transitions in a process BETA executing the WAIT statement.

[42] Due to FCOS limitations this expression will generate an E102 error.

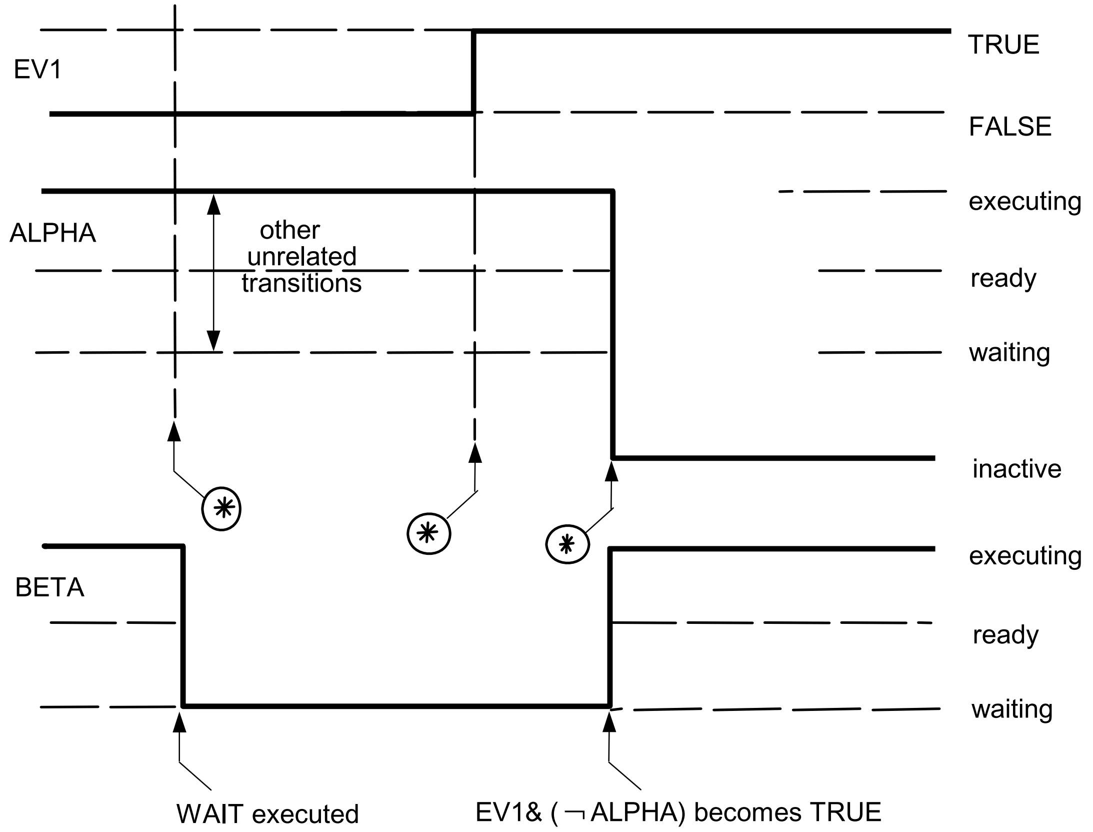

EV1
TRUE
FALSE
ALPHA
other unrelated transitions
executing
ready
waiting
inactive
BETA
executing
ready
waiting
WAIT executed
EV1& (¬ ALPHA) becomes TRUE

Figure 24-9

This page intentionally left blank.

25.0 ERROR RECOVERY AND SIMULATION

HAL/S compilations can be created which, although seen as legal at compile time, violate the rules of the language during execution[42]. Such violations give rise to "run time errors". Run time errors are also produced when abnormal hardware conditions are encountered during execution.

HAL/S has a comprehensive and flexible mechanism for detecting and recovering from run time errors. It also has the capability of simulating run time errors, which can be extremely useful for checkout purposes. Another feature of the language is the ability to specify and signal user-defined run time errors.

This section explains how run time errors are handled as part of the activity of the Real Time Executive (RTE) and describes statements by which HAL/S programmers can extend or modify this activity.

25.1 HAL/S RUN-TIME ERROR CONCEPTS

Each HAL/S implementation possesses a defined set of run time errors which are detectable during execution. These errors are called "system-defined" errors. The HAL/S user may, at will, create a certain limited number of supplementary "user-defined" errors for his own purposes. Each run time error, whether system-defined or user-defined, possesses a unique numerical "error code" by which it may be referenced in a HAL/S compilation. This error code consists of two parts:

- an error group number;
- an error member number[43].

ERROR DETECTION AND RECOVERY

The activity of detecting and recovering from run time errors is handled by an Error Recovery Executive (ERE) which in practice is part of the Real Time Executive (RTE). For every error group, an implementation-dependent, standard, system recovery action is defined[44]. On detecting an error belonging to a certain group, the ERE takes the appropriate system recovery action for the group, unless otherwise directed by the user.

Depending upon the kind of error, the system recovery action may be any one of the following:

- to execute a fix-up routine and continue;
- to terminate execution abnormally;
- to ignore the error.

[42] This fact is true for any language.

[43] The classification into groups, and the assignment of error codes is implementation dependent. See appropriate User's Manual.

[44] See appropriate User's Manual.

ERROR ENVIRONMENT OF A PROCESS

The behavior of the ERE in detecting and recovering from run time errors must be viewed from the standpoint of HAL/S as a real time programming language.

Every active real time process possesses its own so-called "error environment", which is essentially a description of the recovery actions in force for all possible run time errors the process could be subject to. On initiation of the process, the system recovery action is in force for all run time errors. During the life of a process, its error environment may be modified by the specification of a "user recovery action" for some error or error group. The user recovery action is enforced by the execution of specific HAL/S error control statements which will be described later.

A process may only modify its own error environment, never that of another process.

DYNAMIC SCOPING OF ERROR ENVIRONMENTS

During its execution, a process may invoke procedures and functions, which may in turn invoke further procedures and functions, and so on to an arbitrary depth of nesting. Modifications made to the error environment during execution of a procedure or function remain in force only until return from it. Thus, execution of HAL/S error control statements has an inherent dynamic scoping property.

To clarify this concept, consider the following diagram, showing a process *A* invoking procedures *B* during execution, which in turn invoke procedures *C*.

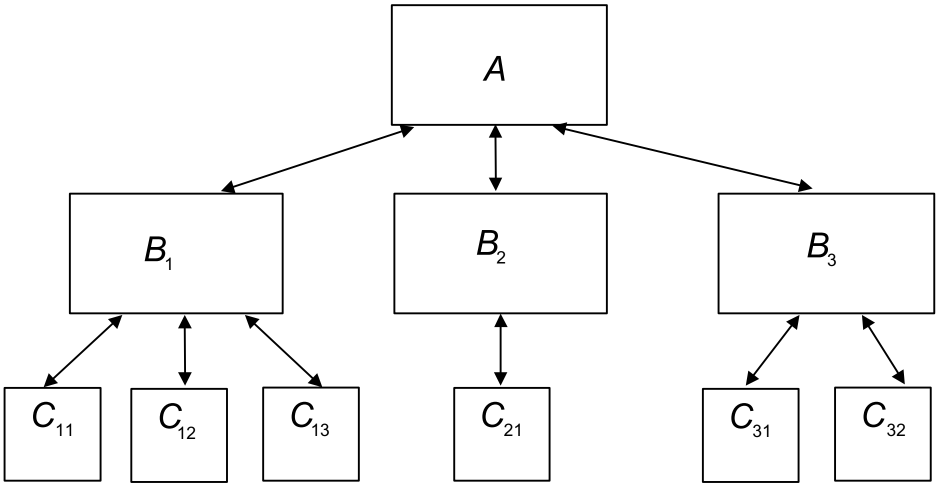


Figure 25-1

- Modifications to the error environment made in *A* remain in force for the remainder of *A*'s execution unless countermanded by removal or further modification.
- Modifications made in B_1 remain in force until return from B_1 unless countermanded by removal or further modification in B_1.

- Modifications made in C_{12} remain in force until return from C_{12} unless countermanded by removal or further modification in C_{12}.

It is stressed that this is a dynamic scoping property, that is not related to whether or not, for example, procedure block C_{12} is physically nested inside procedure block B_1.

Further clarification is required in cases where more than one process can invoke the same procedure or function. If two processes A1 and A2 both execute the same procedure B as shown below, then error control statements executed in B affect the error environment of whichever process is executing B.

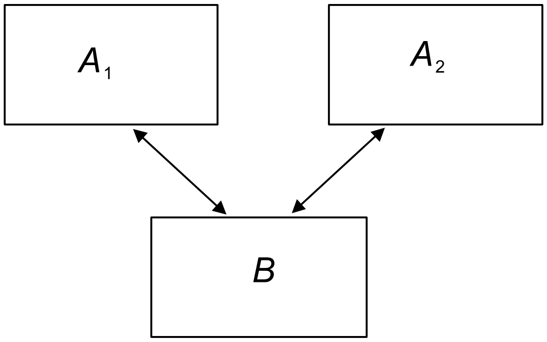


Figure 25-2

The error environment in force for each process on invocation of B is reinstated on return from B. There is no cross-coupling effect between the two error environments.

25.2 ERROR ENVIRONMENT MODIFICATION

HAL/S possesses two statements which can alter the error environment of the process which executes them.

- The ON ERROR statement modifies the error recovery action for a particular error or error group.
- The OFF ERROR statement causes the removal of a previously-applied modification for a particular error or error group.

Both statements have an identical construct for representing the error group and member numbers involved.

ERROR GROUP AND MEMBER NUMBER SPECIFICATION

Error group and member numbers appearing in the HAL/S ON ERROR and OFF ERROR statements are specified by appropriately subscripting the keyword "ERROR". Three basic forms exist. Each form is dealt with in order of decreasing generality.

- SPECIFICATION OF ALL ERRORS

To specify all errors, the keyword ERROR, without subscript, is used:

```
ERROR
```

1. Lack of subscript implies all members of all error groups.

- SPECIFICATION OF ALL ERRORS IN A GIVEN GROUP

To specify all members in a given error group the following form is used:

ERROR$_{m:}$

1. m is an unsigned integer literal.
2. All members in group m are specified.
3. The colon is optional.

- SPECIFICATION OF A GIVEN ERROR

To specify a given error member of an error group, the following form is used:

ERROR$_{m:n}$

1. m, n are unsigned integer literals.
2. Error member n in group m is specified.

ON ERROR STATEMENT

The ON ERROR statement is used to modify the error environment with respect to the error or errors specified. The statement can modify the error environment in the following ways:

- by causing the error or errors to be ignored; …CASE ①
- by causing the standard system recovery action to be taken; ..CASE ② …
- by causing execution to branch to specified HAL/S code on occurrence of the error. ….CASE ③

In addition, in the first two forms, the value of an event data item can be changed on occurrence of the error or errors.

An ON ERROR statement may specify system-defined or user-defined errors[45].

- CASES ① and ②: SYSTEM AND IGNORE ACTIONS

The basic form of the ON ERROR statement is as shown below:

[45] For reasons of software security, some implementations may prohibit the modification of the error environment with respect to certain errors. See appropriate User's Manual.

```
ON specification SYSTEM;
ON specification IGNORE;
```

1. *specification* is an error specification of the form previously described.
2. The keyword SYSTEM states that standard system recovery action is to take place.
3. The keyword IGNORE implies that errors specified in the *specification* are to be ignored.

Examples:

```
 ON ERROR SYSTEM;    ←———  revert to standard system recovery
                           action for all errors.

 ON ERROR    IGNORE;    ←———  ignore error member 4 in group 1.
S        1:4
 ON ERROR  SYSTEM;   ←———  revert to standard system recovery
S       3                  action for all errors in group 3.
```

If the value of an event is to be changed in addition to the actions specified above, one of the following clauses is added after the keyword SYSTEM or IGNORE.

```
...AND SET var…
...AND RESET var…
...AND SIGNAL var…
```

1. SET, RESET, and SIGNAL have the same actions as described in Section 24.4 of the Guide.
2. If *var* contains run time subscript evaluations, they are carried out at the time of execution of the ON ERROR statement rather than on the occurrence of the specified error or errors.

On the occurrence of an error covered by the error specification, the value of the specified event data item is modified before the SYSTEM or IGNORE is taken by the ERE.

Examples:

```
 ON ERROR IGNORE AND SET EV1;
 ON ERROR   SYSTEM AND SIGNAL EV2  ;
S        1:1                      5
 ON ERROR SYSTEM AND SIGNAL EV3 ;
S       5                      I
                               ↑
```

I is evaluated on execution of the ON ERROR statement, not on occurrence of an error in group 5

- CASE ③: USER-SUPPLIED ACTION

The user can supply the action to be performed on an error occurrence by means of the following form of ON ERROR statement.

```
| ON specification statement;
```

1. *specification* is an error specification in the form previously described.
2. *statement* is an executable HAL/S statement with which execution is resumed after occurrence of the specified error condition.
3. *statement* may possess a statement label but cannot be branched to from outside the ON ERROR statement.
4. This kind of ON ERROR statement may not form by itself the "true part" of an IF statement (See Section 9.1).

It is important to understand the flow of execution implied by the above form, both when the ON ERROR is executed, and on the occurrence of an indicated error. The following example shows this in detail.

Example:

```
|  ON ERROR   DO;
|  S       5:1
|                 ▨  }user-supplied error
|                 ▨  }recovery action is this
|                 ▨  }entire DO...END group.
|  END
|  I = I + 1;
|   .
|   .
|   .
|   .
|   .
|   .
   ✻  ←error 5:1 occurs.
```

The path of execution is shown by the following symbolic flow diagram:

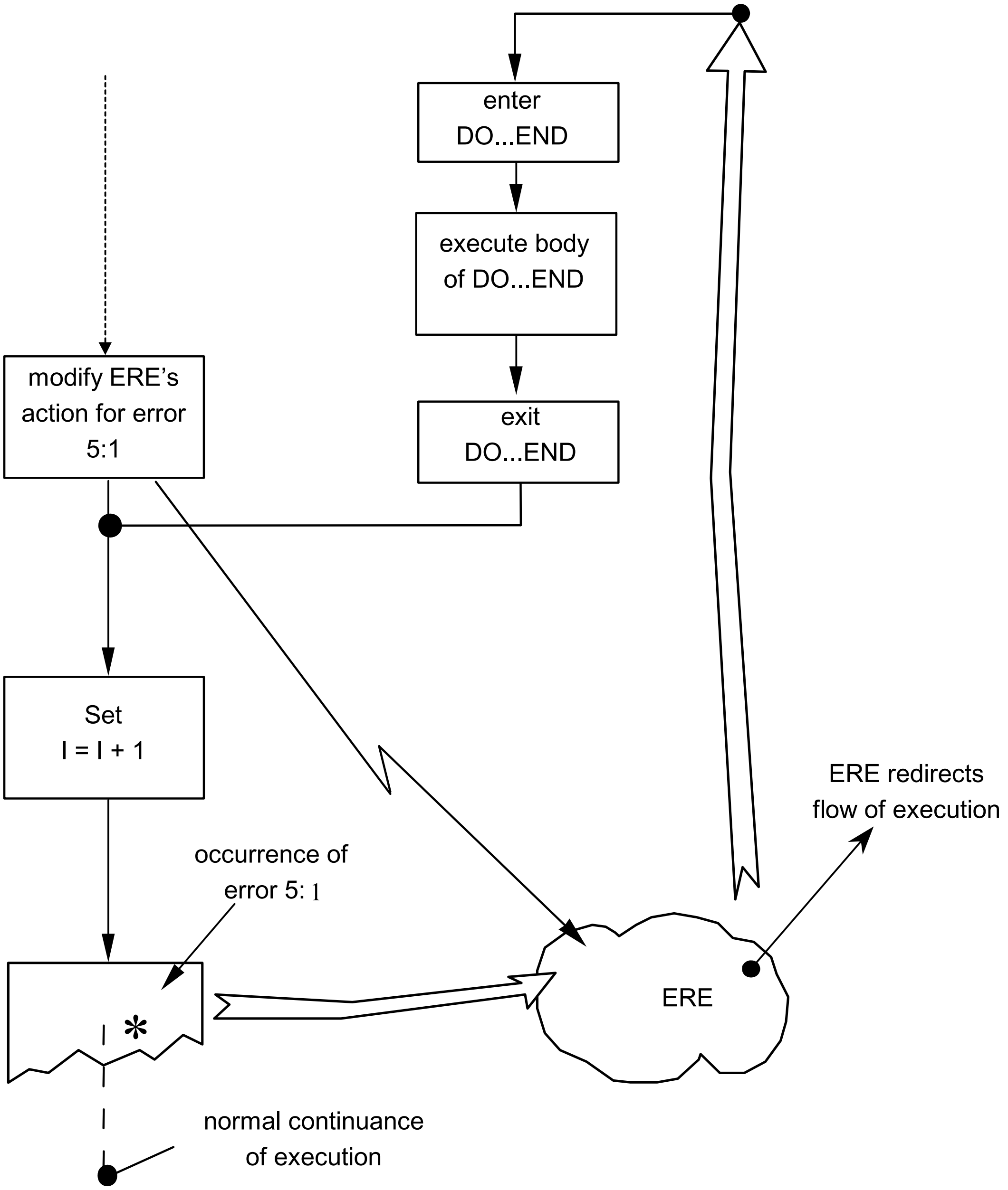


Figure 25-3

The above example assumes that there is no branch in the DO...END group to cause execution to be diverted.

OFF ERROR STATEMENT

The OFF ERROR statement is used to remove the effects of an ON ERROR statement with the same error specification, previously executed by the same process in the same block of HAL/S code. Its form is as follows:

```
OFF specification;
```

1. *specification* is an error specification in the form already described.
2. The statement nullifies the effect of an ON ERROR statement previously executed in the same code block by the same process, and with the same *specification.*
3. The statement has no effect if such an ON ERROR statement does not exist.

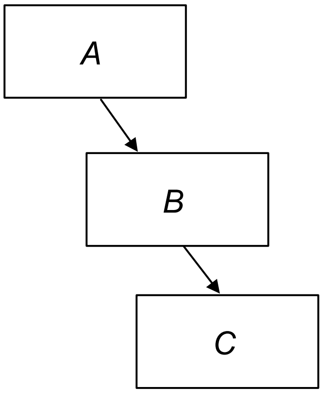
Example:

```
 ON ERROR   IGNORE;
S        5;6
 .
 .
 .
 OFF ERROR    ;  ←———   this nullifies action of previous ON ERROR
S         5:6           statement
```

PRECEDENCE OF ON AND OFF ERROR STATEMENTS

Some additional information needs to be supplied in order to understand in detail how successive execution of several ON and OFF ERROR statements modifies the error environment of a process.

In general, an executing process *A* is executing code in some block several nesting levels of invocation deep, as illustrated below:



A invokes *B* invokes *C*

Figure 25-4

The ERE keeps continuously updated lists of all error environment modifications in force at any instant of time[46]. When execution of the process A described above is in the body of block C, the ERE possesses three linked lists of ON ERROR modifications, each corresponding to a block not yet returned from:

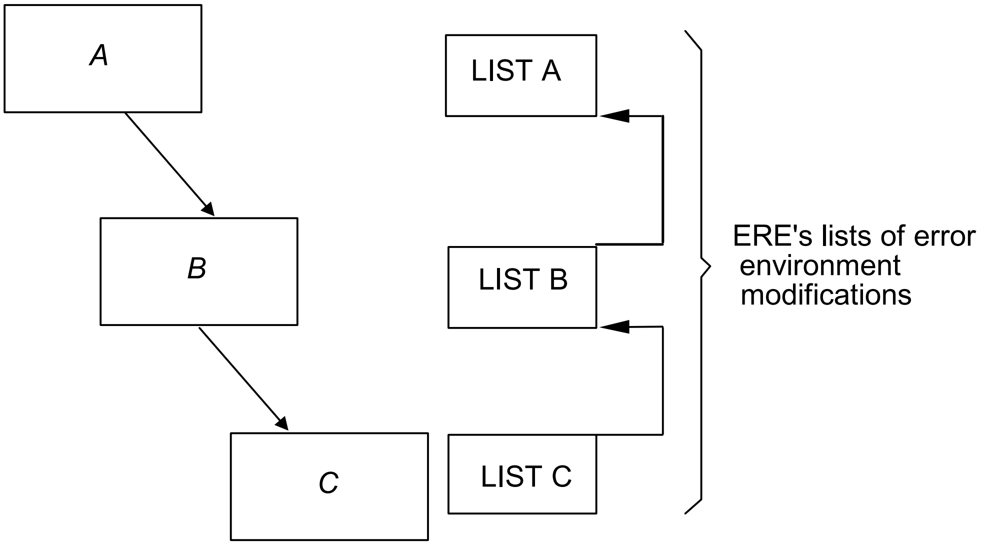


Figure 25-5

When block C is returned from, LIST C is deleted, leaving LIST A and LIST B in force. When block B is returned from, LIST B is deleted leaving only LIST A in force.

Each list is divided into three sublists as illustrated below for LIST C:

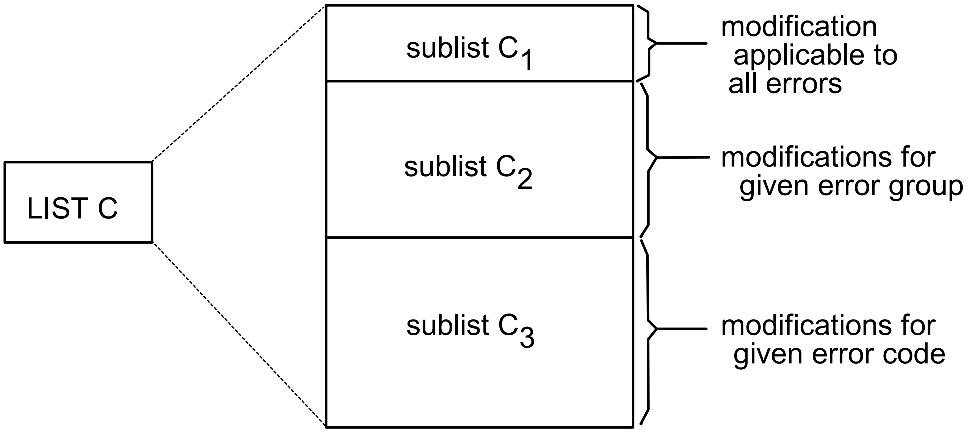

LIST C
sublist C1
sublist C2
sublist C3
modification applicable to all errors
modifications for given error group
modifications for given error code

Figure 25-6

[46] This description of the ERE's behavior is representational only: an actual implementation of the ERE may employ different algorithms producing the same result.

- Sublist C_3 contains modifications generated by ON ERROR statements of the form:

  ON ERROR$_{m:n}$ *.....*

- Sublist C_2 contains modifications generated by ON ERROR statements of the form:

  ON ERROR$_{m}$*.....*

- Sublist C_1 can contain at most one entry, the modification generated by an ON ERROR statement of the form:

  ON ERROR.....

If a new ON ERROR statement in block C is executed, then one of the following happens:

- if an entry in the appropriate sublist exists for the given error specification, the entry is replaced with the new information gained, thus erasing memory of the previous recovery action specified;
- otherwise a new entry is added at the end of the sublist.

With this background, the behavior of the ERE in recovering from a run time error can now be described in more detail. Suppose that a run time error occurs while execution is in block C. On detecting the error, the ERE gains control and scans backwards through the lists until it finds an entry applicable to the error which occurred. The ERE may find such an entry in any of the lists A, B, or C, in which case it takes the indicated recovery action; or it may find no such entry, in which case it takes the standard system recovery action.

Bearing in mind how entries are made into the sublists of error environment modifications, up to three entries may be applicable to a given run time error:

- an entry applicable only to the given error;
- an entry applicable to the whole group of which the given error is a member;
- an entry applicable to all errors.

Given the sublist scanning order described, it is clear that there is an inherent precedence order of ON ERROR statements.

Error Specification		Precedence
		FIRST
ERROR$_{m:n}$	error code specification	1
ERROR$_{m:}$	error group specification	2
ERROR	specification of all errors	3
		LAST

Example:

If the following statements have been executed in a block:

```
 ON ERROR   GO TO ALPHA;
S        5:1
 ON ERROR  GO TO BETA;
S        5:
 ON ERROR IGNORE;
```

Then if error 5:1 occurs, execution branches to ALPHA. If error 5:3 occurs, execution branches to BETA. If error 6:1 occurs, the error is ignored.

The above are true no matter in what order the ON ERROR statements have been executed.

The behavior of an OFF ERROR statement now also becomes clearer. On execution of an OFF ERROR statement in, say, block *C*, the ERE loops through the whole of LIST C and on finding an entry with the same error specification, removes it from its sublist. This may expose to the scanning process another modification in another sublist of LIST C or a modification in LIST A or LIST B.

Example:

If the following statements have been executed in a block:

```
 ON ERROR   GO TO ALPHA;
S        5:1
 ON ERROR  GO TO BETA;
S        5:
```

then if error 5:1 occurs, execution will branch to ALPHA. If now the following statement is executed:

```
 OFF ERROR   ;
S         5:1
```

and afterwards error 5:1 occurs, execution branches to BETA.

25.3 ERROR SIMULATION

At the beginning of Section 25 it was stated that run time errors could be simulated. In fact, the same HAL/S construct is used both to simulate "system-defined" errors, and to signal "user-defined" errors. This construct is the SEND ERROR statement, whose form is shown below:

```
|
| SEND ERROR   ;
| S         m:n
|
```

1. m and n are unsigned integers representing an error group number, and an error member number respectively.
2. If the error code m:n represents a system-defined error, that error is being simulated[†].
3. If the error code m:n represents a user-defined error, that error is being signaled.

[†] For reasons of software security, some implementations may prohibit certain system-defined errors from being simulated. See appropriate User's Manual.

The recovery action taking place on execution of a SEND ERROR statement is as if the corresponding run time error had really occurred.

Example:

```
|
| ON ERROR  GO TO ALPHA;
|S        5:
|  .
|  .
|  .
| SEND ERROR   ;
|S          5:2
|
```

Error 5:2 is simulated or signaled: a previous ON ERROR statement has modified the recovery action for error group 5, so that the result is a branch to ALPHA.

In this example, it is immaterial whether error 5:2 is system-defined or user-defined.

26.0 DATA STORAGE AND ACCESS

Given the purposes for which HAL/S is intended, the way in which declared data is physically located in the core of the object machine will often be an important concern. In particular, in the design of HAL/S software, the following questions must often be addressed:

- Does the declared data occupy as small an area of core as is practical?
- Is the data physically ordered as it was declared?
- Can some non-critical data be relegated to segments of core addressable by slower methods, to make more room for critical data?
- Can use be made of registers or temporary storage areas for some data?
- Does the data fall within a predefined set of values?

HAL/S contains constructs by whose means some degree of control over each one of these factors can be achieved. Necessarily the degree of control is implementation dependent.

In the context of HAL/S as a real time language, the access of data is another important concern. During execution, an arbitrary number of real time processes will in general be competing for access to shared data. Certain "sensitive" data may require protection to prevent modification by one process while a second is referencing it. HAL/S contains constructs through which the integrity of shared data may be assured.

26.1 PACKING DENSITY OF STORED DATA

The efforts that any HAL/S compiler makes to optimize the density of storage of HAL/S data items are implementation dependent. Generally speaking however, the default assumption is that optimization is relatively unimportant compared with speed of access.

The attribute DENSE when applied in the declaration of data items causes more emphasis to be placed on storage density optimization at the expense of speed of access. Potentially the attribute DENSE may be applied to data of any type, although it is a matter of implementation as to when it causes packing density to increase.

DENSE STRUCTURES

Packing density optimization is most commonly applied to HAL/S structures. If the packing density of a structure data item is to be optimized, the keyword DENSE must appear in the specification of the structure template defining its tree organization. The form of such a template is as follows:

```
STRUCTURE name DENSE:
        node1, node2,.......
        noden;
```

1. *name* is the structure template name.
2. *node*^{1}, *node*^{2},...*node*^{n} is a list of nodes forming the tree organization, as described in Section 19.2.
3. The keyword DENSE indicates that the storage packing density of all the structure terminals is to be optimized†.

† See appropriate User's Manual for packing algorithms.

Note that such optimization may cause the physical ordering of structure terminals to differ from that given in the template specification.

Example:

```
STRUCTURE A DENSE:
   1  A1,
      2 A11 BIT(16),
      2 A12 INTEGER,
      2 A13 ARRAY(10) BOOLEAN
   1  A2 CHARACTER(80);
 DECLARE ZA A-STRUCTURE;
```

All the structure terminals in ZA have their storage packing density optimized.

When the keyword DENSE is used as described above, storage packing density is optimized for the whole of a structure. If the DENSE keyword is used on a fork or leaf node of a structure template such optimization can be restricted to part of a structure. The way in which this works is illustrated by the following tree diagram:

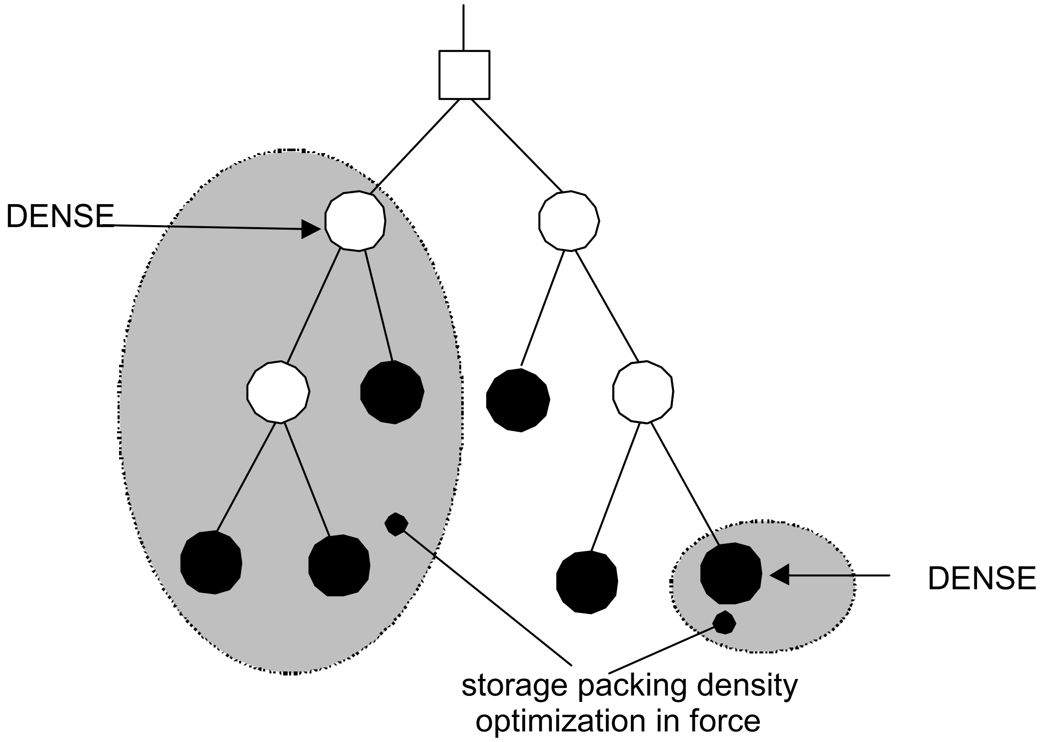


Figure 26-1

Nodes connected below a “fork” node on which the keyword DENSE appears inherit the property from it. The keyword ALIGNED can be used to prevent inheritance of the property:

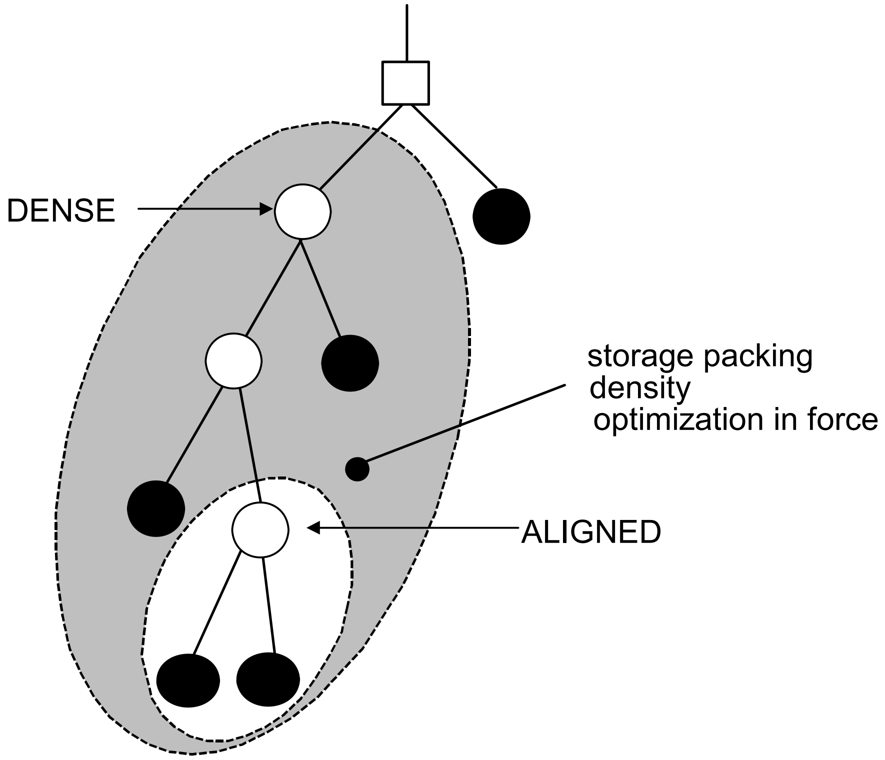


Figure 26-2

The following example shows how the keywords are actually specified in a structure template.

Example:

```
STRUCTURE A:
  1  A1 DENSE,
     2 A11 BIT(16),
     2 A12 INTEGER,
     2 A13 ARRAY(10) BOOLEAN ALIGNED
  1  A2 CHARACTER(80);
DECLARE ZA A-STRUCTURE;
```

The packing density is only optimized in those terminals of ZA shown in the shaded areas of the following structure tree:

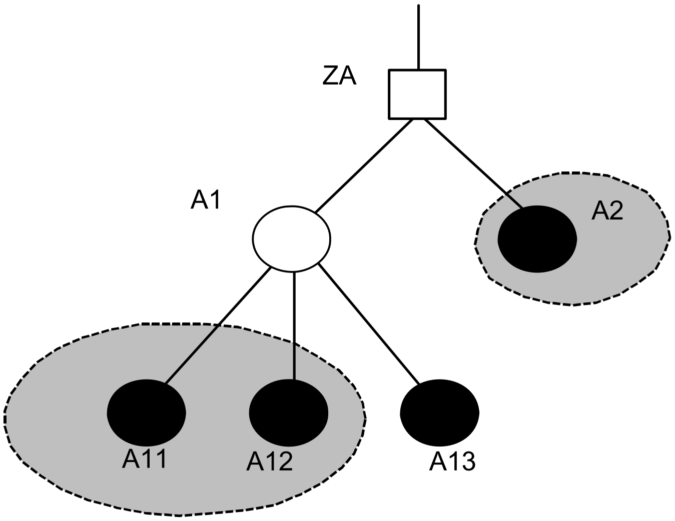


Figure 26-3

The ALIGNED keyword on A13 has prevented the inheritance of the DENSE property from A1.

Detailed rules for the appearance of DENSE and ALIGNED on fork and leaf nodes of structure templates, and on data items of other types are given in Spec./4.5.

26.2 ORDERING OF STORED DATA

The HAL/S language does not guarantee that the physical order in which data is stored is the same as the order of appearance of data items in a compilation, either globally or locally. Nor does HAL/S guarantee that the physical order of structure terminals in a structure data item is the same as the order of their definition in its structure template. Indeed, some implementations will deliberately reorder data so that access to it can be optimized.

In most cases such reordering is not of importance to the HAL/S programmer. However, since there are exceptions, HAL/S has a capability for specifying the non-reordering of data in storage.

Reordering may be inhibited in the following constructs:

- an entire compool;
- a structure template.

NON-REORDERING OF COMPOOLS

To prevent the reordering of data items in a compool, the keyword RIGID is placed in the opening statement of the compool block, as shown below.

```
label: COMPOOL RIGID;
```

1. *label* is the name of the compool.
2. The keyword RIGID denotes that the physical order of storage of data items is the same as the order of their appearance in the compool.

The corresponding compool template must possess the keyword RIGID also.

Example:

```
POOL: COMPOOL RIGID;
      DECLARE A ARRAY(1000) SCALAR,
              B BIT(16);
      DECLARE C CHARACTER(80);
CLOSE POOL;
```

The data in the compool are guaranteed to be stored in the following order:

A
B
C

The corresponding compool template is as shown below:

```
POOL: EXTERNAL COMPOOL RIGID;
      DECLARE A ARRAY(1000) SCALAR,
              B BIT(16);
      DECLARE C CHARACTER(80);
CLOSE POOL;
```

Use of the keyword RIGID in the above context does not of itself prevent the reordering of structure terminals within a structure.

Example:

```
DATA: COMPOOL RIGID;
      STRUCTURE Q:
              1 QI INTEGER,
              1 QS SCALAR,
              1 QB BIT(8);
      DECLARE A ARRAY(100) SCALAR,
              B Q-STRUCTURE,
              C CHARACTER(80);
CLOSE DATA;
```

The order of data items in storage is guaranteed to be:

A
B
C

However, the ordering of terminals of B is <u>not</u> guaranteed to be:

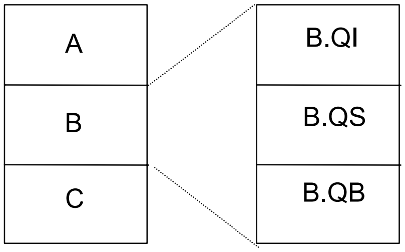


Figure 26-4

NON-REORDERING OF STRUCTURE TERMINALS

The potential reordering of structure terminals may be inhibited by use of the keyword RIGID on the structure template, as shown below:

STRUCTURE *name* RIGID:
    node^1, node^2,…
    … node^n;

1. *name* is the structure template name.
2. *node*^{1}, *node*^{2},...*node*^{n} is a list of nodes forming the tree organization, as described in Section 19.2.
3. The keyword RIGID denotes that the physical order of structure terminals is guaranteed to be the same as the order of appearance of the terminals in the template.

Example:

```
STRUCTURE Q RIGID:
        1 QI INTEGER,
        1 QS SCALAR,
        1 QB BIT(8);
DECLARE ZQ Q-STRUCTURE;
```

The order of storage of the structure terminals of ZQ is the same as the order of their appearance in template Q:

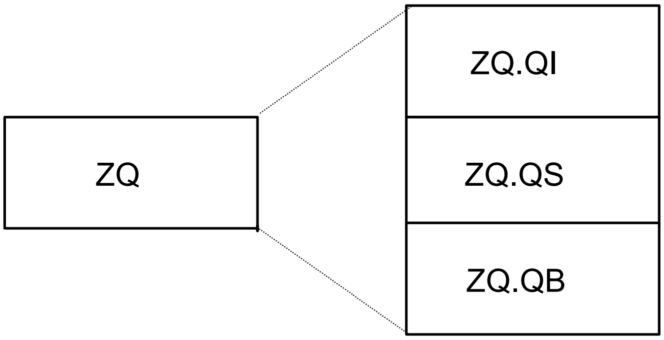


Figure 26-5

Both the keywords RIGID and DENSE may appear on a structure template (in any order). The effect of RIGID takes precedence over storage packing density optimization.

The keyword RIGID may appear on fork and leaf nodes of a template.

See: Spec./4.5.

26.3 TEMPORARY AND REMOTE STORAGE

The data accessing characteristics of some object machines are such that most efficient use of core is made by dividing data into two categories:

- data which needs to be accessed quickly and often;
- data which needs to be accessed seldom, and where speed is not critical.

Normally all declared data in a HAL/S compilation is treated alike, as falling into the first of these categories. However, by appropriate specification, a HAL/S data item can be relegated to the second category: such data items are termed "remote".

Sometimes in HAL/S code, data items are used only as temporary storage in an extremely localized sequence of statements, and have no significance as far as the algorithm implemented is concerned. If such data items were declared normally, then the core area they occupy would remain unused for a substantial part of the duration of execution of the code. This waste can be avoided by declaring them as "temporary" data items, whereupon the HAL/S compiler can be allowed to locate them in some reusable "scratch pad" area[44] .

[44] The nature and usage of such areas is implementation specific.

Control variables in repetitive DO groups are a particular instance of data items used for temporary storage purposes. However, in this instance a consideration is the speed with which the value of the control variable can be accessed, since it may be required for many subscript evaluations within the DO group. Here it is more appropriate to set aside a register than to locate the data item in a scratch pad area. Declaration of such variables as “temporary” can allow a HAL/S compiler to perform this kind of allocation also.

SPECIFICATION OF REMOTE DATA

A data item is declared to be remote by use of the keyword REMOTE in its declaration. Data items of any type except event may be designated REMOTE. The position of the keyword in a declaration is illustrated by the following examples:

Examples:

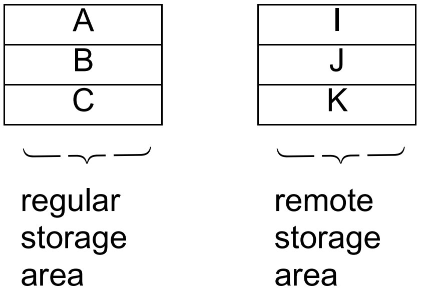
```
DECLARE I INTEGER REMOTE;
DECLARE V VECTOR(3) DOUBLE REMOTE;
DECLARE S SCALAR REMOTE INITIAL(1.5);
DECLARE B BOOLEAN INITIAL(TRUE) REMOTE AUTOMATIC;
DECLARE ARRAY(4) INTEGER REMOTE,I,K,L;
STRUCTURE Q:
        1 QI INTEGER,
        1 QS SCALAR,
        1 QB BIT(16);
DECLARE ZQ Q-STRUCTURE REMOTE;
```

If remote data items appear in a RIGID compool, then the remote data items appear in the remote storage area in the same order as they were declared; the other data items appear in the regular storage area in the same order as they were declared.

Example:

```
DATA: COMPOOL RIGID;
      DECLARE A SCALAR,
              B BIT(16);
      DECLARE ARRAY(100) INTEGER REMOTE,I,J,K;
      DECLARE C CHARACTER(80);
CLOSE DATA;
```

The physical ordering of data in the above compool is as shown below:



For more precise rules on positioning the keyword REMOTE, see Spec./ 4.5.

DECLARING AND TEMPORARY DATA

The HAL/S language enforces localized use of temporary data items by requiring them to be declared and used within DO...END statement groups (see Section 10). The END statement of a group signals to the HAL/S compiler that "scratch pad" storage allocated to temporary data defined in the group is available for other use.

Temporary data items are declared by TEMPORARY statements which are declaration statements in which the keyword DECLARE has been replaced by the keyword TEMPORARY. The basic form is thus:

```
TEMPORARY  name attributes;
```

1. *name* is a legal HAL/S identifier name.
2. *attributes* describe the type, array property, precision and other properties of the data item as in a declaration statement.

All TEMPORARY statements must appear immediately after the DO statement and before the first statement inside the group.

Examples:

```
DO;
 TEMPORARY  S SCALAR;
 TEMPORARY  I INTEGER DOUBLE;
 TEMPORARY  B BIT(16),          }compound statement - compare with
              ZQ Q-STRUCTURE    }compound  declarations in  Section 4.2
 .
 .
 .
 .
 END;
```

The structure template Q cannot be defined in the DO...END group. Its definition must appear at the beginning of the code block in which the DO...END group is imbedded.

The control variable in a DO FOR statement can also be designated a temporary data item by preceding its appearance in the DO FOR statement by the keyword TEMPORARY. In this context, the control variable is taken implicitly to be a single precision (halfword) integer.

Example:

```
DO FOR TEMPORARY I = 1 TO 18 BY 2;
.
.
.
END;
```

The declaration of temporary data items is subject to the following restrictions:

- they may not be initialized;
- they may not be declared remote;
- they may not be of event type;
- the name of a temporary data item may not duplicate the name of another temporary data item in the same DO...END group;
- the name of a temporary data item may not duplicate the name of an ordinary data item known by the scoping rules of Section 1.2 to the body of the DO...END group.

26.4 ACCESS TO SHARED DATA

Generally at run time, an arbitrary number of real time processes are able to share data defined in compools. Thus, it is entirely possible that one process may be in the act of modifying such data while another process is referencing it. It may be crucial to the integrity of the algorithm implemented in the second process that this be guaranteed not to take place.

To handle this situation, HAL/S has the capability to designate certain compool data items as protected, or "locked". Such data items can only be accessed from within areas of code called "update blocks". The boundaries of update blocks are visible to the Real Time Executive (RTE) which can therefore control entry into them and exit from them on a process-by-process basis.

LOCK GROUPS

The protection of data could be carried out on an individual basis data item by data item. Consider two processes *A* and *B*, each requiring to use protected data item *Z* as shown below:

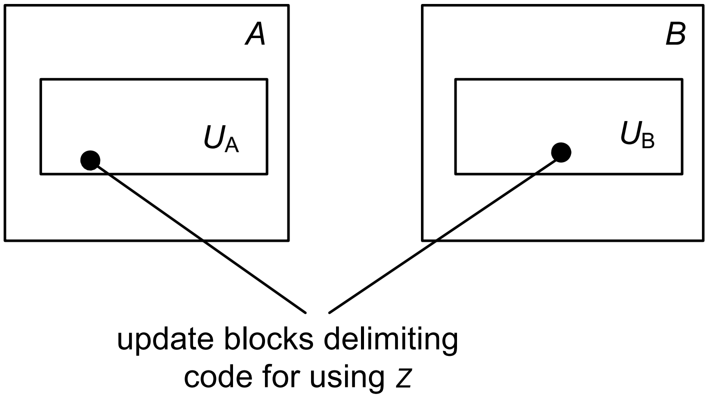


Figure 26-6

If process *A* began executing update block U_A first, and thus began using *Z*, then process *B* would be prevented from beginning execution of update block U_B until *A* had finished executing U_A.

Protection of data on an individual basis would impose an arbitrarily large burden on the RTE depending on the number of data items to be protected, and the number of processes requiring to share them.

In order to limit this overhead of effort, HAL/S applies protection on a group basis rather than an individual one. Each data item to be protected is designated as belonging to one of a limited number of "lock groups". The above illustration can be restated for HAL/S as follows.

Consider two processes *A* and *B*, each requiring to use protected data in lock group N:

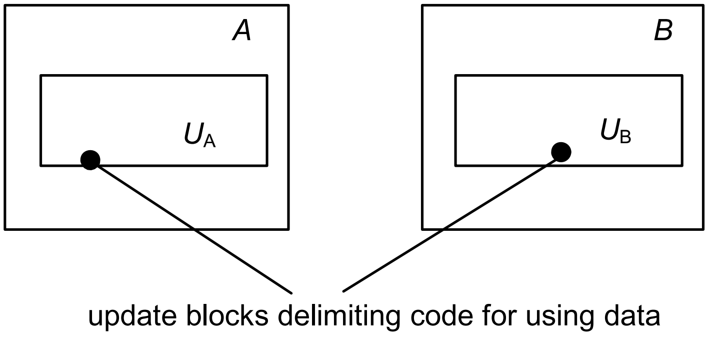


Figure 26-7

If process *A* begins executing U_A first, then all protected data in lock group N become unusable by process *B* which therefore cannot begin executing U_B until *A* finishes executing U_A.

For more global protection, some protected data items can be designated as belonging to all lock groups simultaneously.

If in the above illustration, for example, process *A* required to use a protected data item belonging to all groups, and execution reached U_A first, then process *B* could not enter U_B to use protected data from <u>any</u> lock group until *A* had finished executing U_A.

LOCK GROUP SPECIFICATION

A data item in a compool is designated as protected at the time of its declaration. The following construct is inserted in its declaration:

```
....LOCK(n)....
....LOCK(*)....
```

1. In either form, the keyword LOCK indicates that the data item is to be protected.
2. n is a positive integer denoting that the data item is to belong to lock group n, where 1 ≤ n ≤ 15†.
3. * denotes that the data item is to be considered as belonging to all lock groups simultaneously.

† This value may vary between implementations. See appropriate User's Manual.

The following examples illustrate the positioning of the construct within declarations:

Examples:

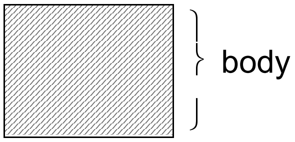
```
DECLARE I INTEGER DOUBLE LOCK(3);
DECLARE S SCALAR INITIAL(5.5) LOCK(*);
DECLARE V VECTOR(3) LOCK(1) INITIAL(0);
DECLARE B ARRAY(1000) BOOLEAN LOCK(*);
STRUCTURE Q DENSE:
        1 QI INTEGER,
        1 QS SCALAR,
        1 QB BIT(16);
DECLARE ZQ Q-STRUCTURE(20) LOCK(3);
```

For more precise rules concerning the location of the locking attribute see Spec./4.5.

UPDATE BLOCK DEFINITIONS

An update block is an explicitly delimited body of code wherein locked data may be referenced or modified. Superficially, an update block looks similar to any other kind of code block in the HAL/S language. Its delimiting statements are of the form shown below:

```
label: UPDATE;
```



```
CLOSE label;
```

1. On the opening statement *label is* any HAL/S identifier, and represents the name of the update block.
2. The update block may be unlabeled, in which case *label*: is omitted.
3. If the update block is labeled, the closing statement may optionally possess a matching *label.*

An update block is unique in that it is never invoked as are other kinds of code blocks: rather it is executed when it is encountered in the path of execution. Consistent with this, the label on the opening statement of the block may be treated as a statement label.

Example:

```
 I = I + 1;
 IF I < 0 THEN GO TO ENTER;
    J = J + 1;
 ENTER: UPDATE;
E              T
  M = M + U.U N N;
 CLOSE ENTER;
  .
  .
  .
```

The possible paths of execution in the above code are represented by the following flow diagram.

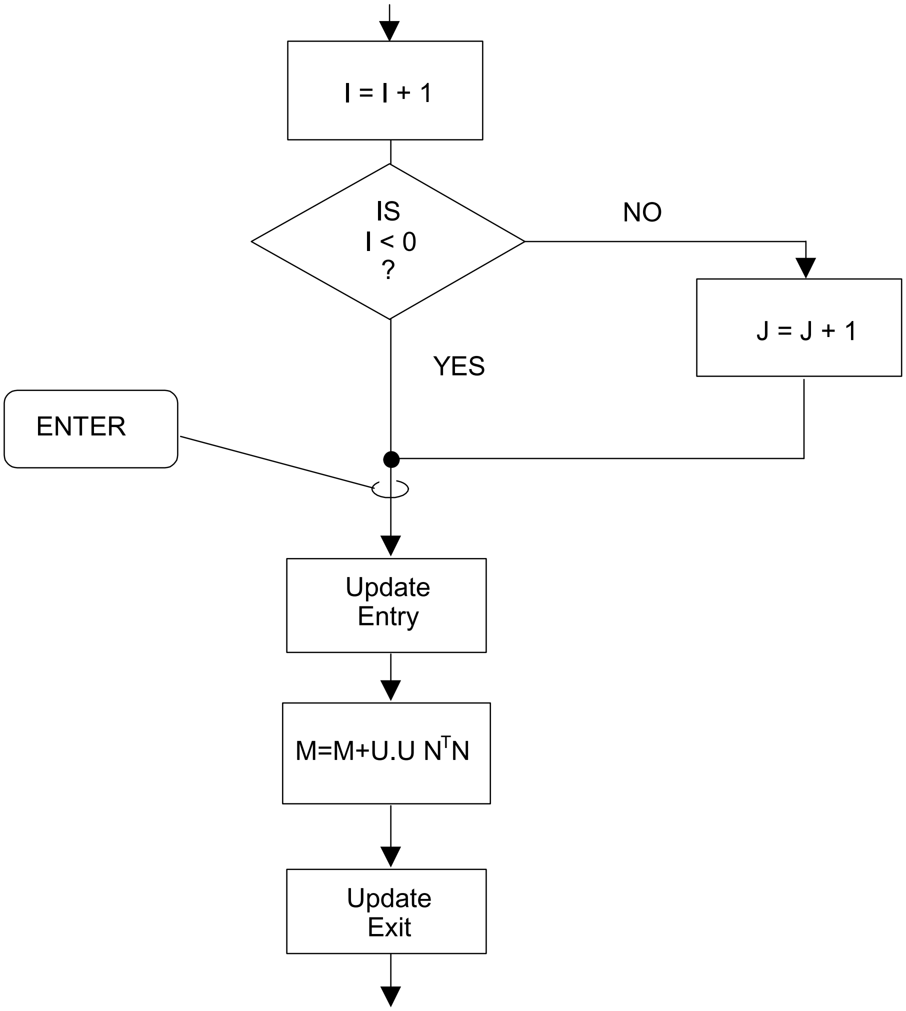


Figure 26-8

The following rules govern the contents of any update block.

- The opening statement may be immediately followed by the declaration of local data, as if it were a program block (see Section 3.2).
- Input/Output statements of any kind are illegal.
- SCHEDULE, WAIT, CANCEL, TERMINATE and UPDATE PRIORITY statements are illegal.
- Procedure and function blocks, but neither task nor other update blocks may be nested within it.
- The only procedure or function invocations which are legal are those referencing procedure or function blocks defined within it.

Example:

```
UPDATE;
  DECLARE I INTEGER,
          S SCALAR;
  V = V/S;
  .
  .
  WRITE(6)V;    ← illegal
  .
  .
INNER: PROCEDURE;
▨ } body of  procedure
CLOSE INNER;
    .
    .
    .
    CALL INNER;
    CALL OUTER; ← illegal ~ outer not defined in update block
    UPDATE PRIORITY ALPHA TO 50; ← illegal
    .
    .
    .
CLOSE;
```

EXECUTION OF UPDATE BLOCKS

The behavior of processes on encountering update blocks has already been described in this section, but only superficially by example. This behavior is now reexamined in more detail.

The simplest case is that of two processes wishing to use data items from the same lock group. Each process has to execute an update block to use the protected data items. The following activity takes place:

- If both of the processes require data items from the same lock group to be modified then the first process to enter its update block must complete execution of it before the second process can enter its own update block. The RTE places the second process in a waiting state for this period of time.
- If one or both of the processes only require to reference the data then in some implementations of HAL/S, the behavior of the RTE will be the same as before. Alternatively, in other implementations, to reduce the second process’ waiting time, the RTE may allow partial overlap in execution of the update blocks, consistent with exclusive use of data by the process modifying it[45].

If the two processes wish to use data from more than one lock group, the RTE tracks the

[45] This alternative entails more work by the RTE thus “stealing" time from the processes' productive work. The behavior of any implementation is therefore the result of a trade-off to achieve an acceptable RTE performance.

use of each lock group in the above way. If one or both processes use data protected by LOCK (*), then the situation is equivalent to one in which the process or processes wish to use data in every lock group.

If data is shared by more than two processes, then all processes except one are put in a waiting state by the RTE. The eventual order in which the processes complete execution of their update blocks will depend on the contents of the process queue and the relative priority of the processes.

Example:

> In some real time application, it is required that a process ALPHA print the values of a covariance matrix M once every 19 seconds. The values are updated once every 1.5 seconds by a second process BETA. The implementation must guarantee that a partially updated covariance matrix not be printed.

The covariance matrix M is declared thus:

```
 DECLARE M MATRIX(3,3) LOCK(1);
```

Two task blocks corresponding to ALPHA and BETA are shown below:

```
 ALPHA: TASK;
    DECLARE M_LOCAL MATRIX(3,3);
    U1: UPDATE;
            M_LOCAL = M;
    CLOSE U1;
    WRITE(6) ‘COVARIANCE=’, M_LOCAL;
 CLOSE ALPHA;
C
 BETA TASK;
    DECLARE VT VECTOR(3);
    U2: UPDATE ;
E                T
    V =(PHI M PHI + QA)Z;
    M = V V/(QB + Z.V);
    CLOSE U2;
 CLOSE BETA;
```

Processes ALPHA and BETA could be created by invoking these task blocks with cyclic SCHEDULE statements (see Section 23.5) of the following form:

```
 SCHEDULE ALPHA PRIORITY(10), REPEAT EVERY(l9);
 SCHEDULE BETA PRIORITY(20), REPEAT EVERY(1.5);
```

The following diagram shows the state transitions of the processes:

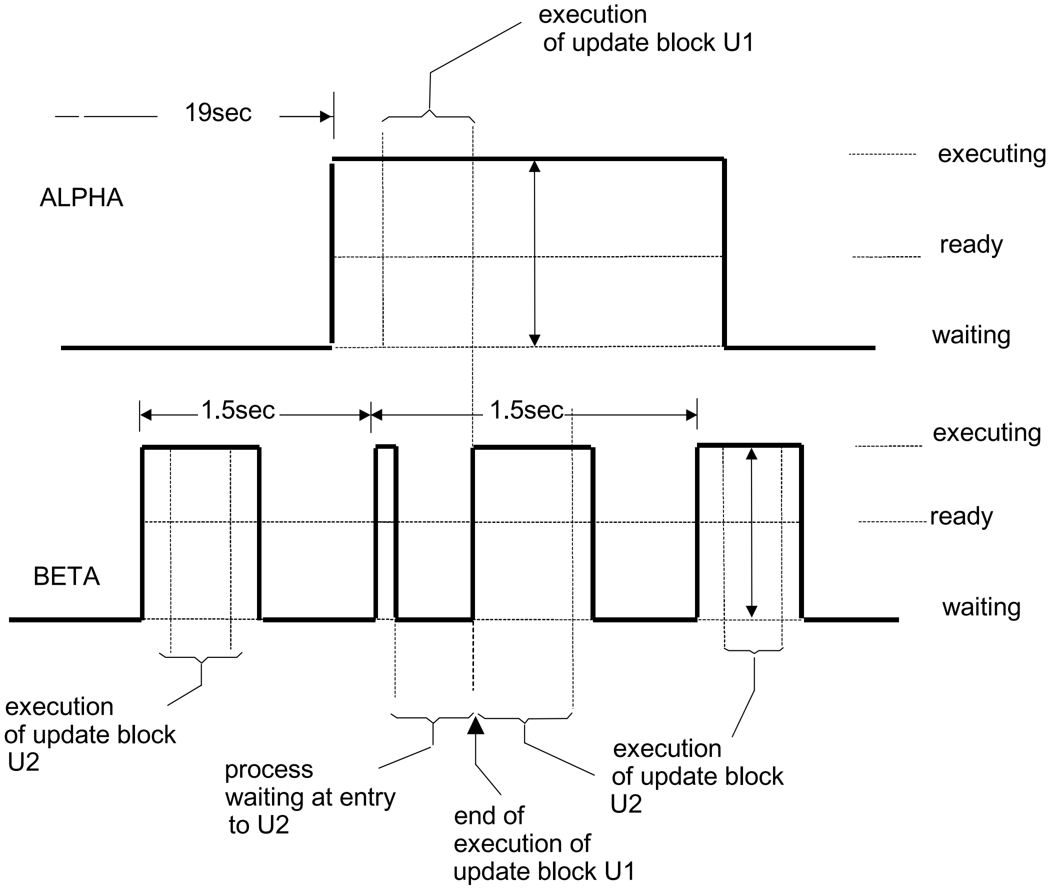


Figure 26-9

Note that if in this example process swaps occurred only on statement boundaries, update blocks would not be needed since ALPHA could not ever be brought into execution with covariance matrix M partly updated.

LOCKED ASSIGN ARGUMENTS

The rule that locked data items can only appear in update blocks has one sole exception: it is possible for locked data items to appear as assign arguments in procedure invocations. This provides the ability to "parameterize'' update blocks, as will be shown in an ensuing example.

The following rules govern the passage of locked assign arguments:

1. If the argument is a data item belonging to lock group n, then the corresponding parameter must be declared LOCK(n) or LOCK(*).
2. If the argument is a data item belonging to all lock groups, the corresponding parameter must be declared LOCK (*).
3. Argument and parameter must also match in the senses described in Sections 11.5, 17.7, or 19.10 as applicable.

Examples:

```
DECLARE A SCALAR LOCK(1),
        B SCALAR LOCK(2),
        C SCALAR LOCK(*);
PICK: PROCEDURE(P) ASSIGN(Q, R);
   DECLARE P SCALAR,
           Q SCALAR LOCK(1),
           R SCALAR LOCK(*);
▨ } body of procedure
CLOSE PICK;
```

For the above procedure definitions and declarations, the following invocations are legal:

```
CALL PICK(1.0) ASSIGN(A,B);
CALL PICK(2.0) ASSIGN(A,C);
```

The following are illegal:

```
       locked data item as input argument
          ↓                ______ unmatched lock group
                         ↓
CALL PICK(A) ASSIGN(B,C);__ /
                         ↓
CALL PICK(3.0) ASSIGN(C,B);
```

The procedure PICK may contain an update block changing the value of Q and R:

```
PICK: PROCEDURE(P) ASSIGN(Q,R);
      DECLARE P SCALAR,
              Q SCALAR LOCK(1),
              R SCALAR LOCK(*);
      .
      .
      .
U: UPDATE;
   Q = Q + P;
   R = R - P;
CLOSE U;
      .
      .
      .
CLOSE PICK;
```

PICK may be invoked with different locked assign arguments, thus effectively parameterizing the update block.

```
CALL PICK(l) ASSIGN(A,B); updates A and B
CALL PICK(2) ASSIGN(A,C); updates A and C
```

27.0 HAL/S AND REENTRANCY[46]

This section deals with another indirect implication of multi-processing in real time: reentrancy. In HAL/S, reentrancy arises because more than one real time process at a time can use a procedure or function. The HAL/S language possesses constructs by which reentrancy can be allowed or inhibited in procedures and functions.

27.1 DETERMINING REENTRANCY REQUIREMENTS

A HAL/S user intending to code a procedure or function (either internal or external) to be invoked in a real time context, should first determine which of the following two categories it falls into:

- The places where it is invoked are such that it can never be in use by more than one process at a time.
- The places where it is invoked are such that it can potentially be in use by more than one process at a time.

If the user determines that the procedure or function falls into the first category, then the procedure or function block is coded following the rules given in Section 11.

If, on the other hand, it falls into the second category, the user must make a choice between the following courses of action:

- to insure that during execution, the Real Time Executive (RTE) allows only one process at a time to use it;
- to insure that during execution, more than one process can use it at a time.

A procedure or function in whose respect the first course of action is taken, is called "exclusive". One in whose respect the second course of action is taken is called "reentrant". The opening statements of such procedures and functions must contain specific indication of their exclusive or reentrant property.

27.2 EXCLUSIVE PROCEDURES AND FUNCTIONS

An exclusive procedure or function is one which the RTE allows only one process to use at any given time. A procedure or function is designated exclusive by the presence of the keyword EXCLUSIVE in the opening statement of its block definition.

DEFINING AN EXCLUSIVE PROCEDURE

The form of the opening statement of an exclusive procedure is as shown below:

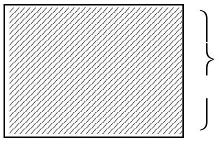
[46] The term "reentrancy" denotes the property of being reentrant.

label: PROCEDURE(i^1,i^2,...) ASSIGN(a^1,a^2,...) EXCLUSIVE;

1. *label* is a legal HAL/S identifier constituting the procedure name.
2. i^1, i^2,... and a^1, a^2,... are lists of input and assign parameters as described in Section 11.2.
3. The keyword EXCLUSIVE designates an exclusive procedure.

Example:

```
P: PROCEDURE(A) EXCLUSIVE;
DECLARE A SCALAR;
```



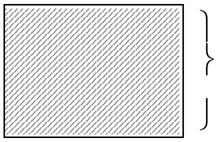
```
CLOSE P;
```

The template corresponding to an exclusive external procedure must also bear the keyword EXCLUSIVE.

Example:

The template corresponding to

```
P: PROCEDURE(A) EXCLUSIVE;
DECLARE A SCALAR;
```



```
CLOSE P;
```

would be:

```
P: PROCEDURE(A) EXCLUSIVE;
   DECLARE A SCALAR;
CLOSE P;
```

DEFINING AN EXCLUSIVE FUNCTION

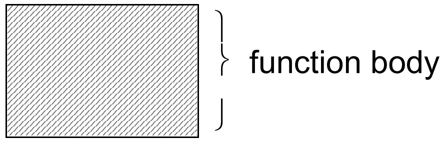
The form of the opening statement of an exclusive function is as shown below:

label: FUNCTION(i^1,i^2,...) *attributes* EXCLUSIVE;

1. *label* is a legal HAL/S identifier constituting the function name.
2. i^1,i^2 is a list of input parameters as described in Section 11.2.
3. *attributes* defines the type and, where applicable, precision of the function, as described in Section 11.2.
4. The keyword EXCLUSIVE designates an exclusive function.

Example:

```
F: FUNCTION BOOLEAN EXCLUSIVE;
```



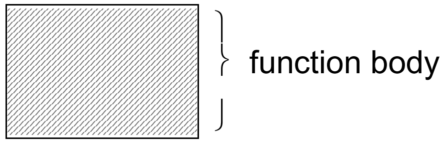
```
CLOSE F;
```

The template corresponding to an exclusive external function must also bear the keyword EXCLUSIVE.

Example:

The template corresponding to:

```
F: FUNCTION BOOLEAN EXCLUSIVE;
```



```
CLOSE F;
```

would be:

```
F: EXTERNAL FUNCTION BOOLEAN EXCLUSIVE;
CLOSE F;
```

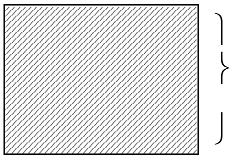
BEHAVIOR OF EXCLUSIVE PROCEDURES AND FUNCTIONS

If an exclusive procedure or function is in use by a process *A*, and a process *B* tries to invoke it, then the RTE places process *B* in the waiting state until process *A* returns from its use.

Example:

Two processes, ALPHA and BETA, can invoke the following procedure:

```
P: PROCEDURE(A) EXCLUSIVE;
```



```
CLOSE P;
```

Suppose that ALPHA invokes P first and during its execution, BETA tries to invoke it. The state transitions for this situation is shown below:

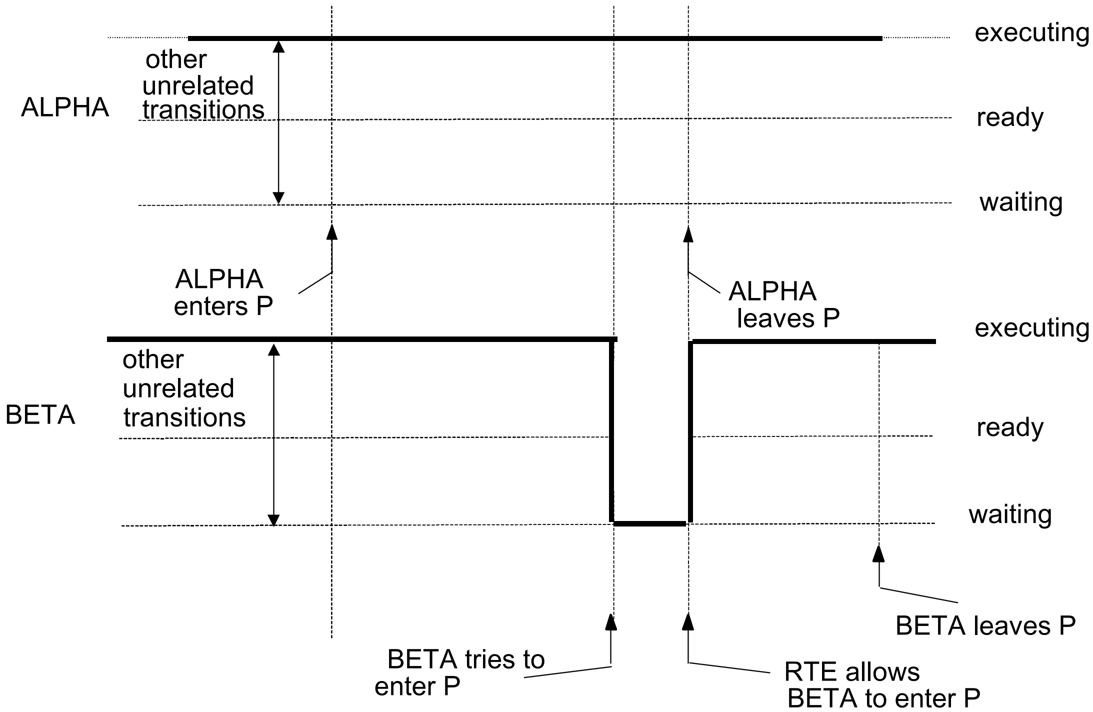


Figure 27-1

27.3 REENTRANT PROCEDURES AND FUNCTIONS

A reentrant procedure or function is one in which deliberate steps are taken by the programmer to ensure correct execution when the RTE allows more than one process to use it simultaneously. A procedure or function which is intended to be reentrant must possess the keyword REENTRANT in its opening statement.

This is a necessary <u>but not sufficient</u> condition to ensure reentrancy. The programmer must observe certain additional guidelines unenforceable by a HAL/S compiler to ensure that a procedure or function is truly reentrant in all relevant respects.

DEFINING A REENTRANT PROCEDURE

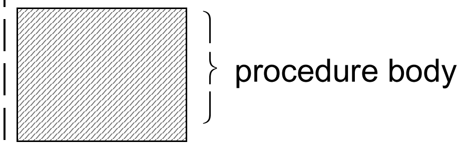
The form of the opening statement of a reentrant procedure is shown below:

label: PROCEDURE(i^1,i^2,...) ASSIGN(a^1,a^2,...) REENTRANT;

1. *label* is a legal HAL/S identifier constituting the procedure name.
2. i^1, i^2,... and a^1, a^2,... are lists of input and assign parameters as described in Section 11.2.
3. The keyword REENTRANT indicates that the procedure is to be considered reentrant.

Example:

```
P: PROCEDURE REENTRANT;
```



```
CLOSE P;
```

If P were an external procedure, the corresponding template would be:

```
P: EXTERNAL PROCEDURE REENTRANT;
CLOSE P;
```

DEFINING A REENTRANT FUNCTION

The form of an opening statement of a reentrant function is shown below:

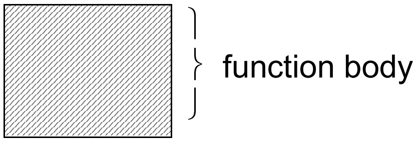
label: FUNCTION(i^1,i^2,...) *attributes* REENTRANT;

1. *label* is a legal HAL/S identifier constituting the function name.
2. i^1, i^2... is a list of input parameters as described in Section 11.2.
3. *attributes* defines the type and, where applicable, precision of the function, as described in Section 11.2.
4. The keyword REENTRANT indicates that the function is to be considered reentrant.

The template corresponding to an external reentrant function must also possess the keyword REENTRANT.

Example:

```
F: FUNCTION MATRIX(4,4) REENTRANT;
```



```
CLOSE F;
```

If F were an external function, the corresponding template would be:

```
F: EXTERNAL FUNCTION MATRIX(4,4) REENTRANT;
CLOSE F;
```

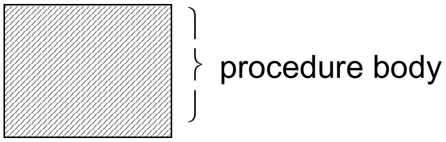
BEHAVIOR OF REENTRANT PROCEDURES AND FUNCTIONS

If a reentrant procedure or function is in use by a process *A*, and a process *B* tries to invoke it, the RTE allows the invocation to proceed without restriction.

Example:

Two processes, ALPHA and BETA, can invoke the following procedure:

```
P: PROCEDURE REENTRANT;
```



```
CLOSE P;
```

Suppose that ALPHA invokes P first and during its execution, BETA invokes it. The state transitions for this situation are as follows (compare corresponding example for exclusive procedure):

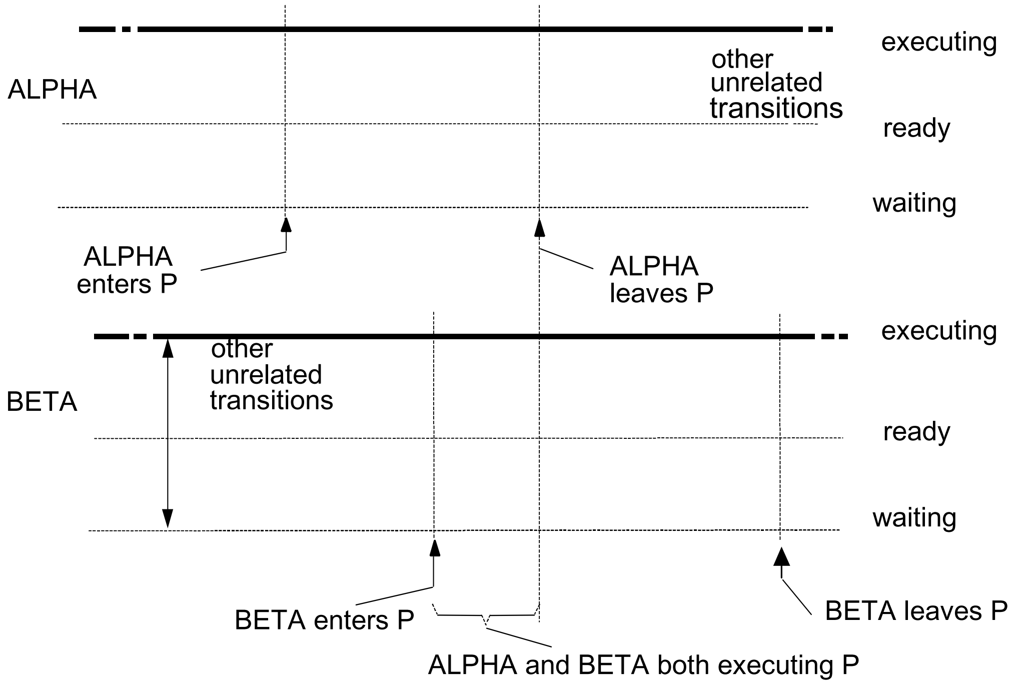


Figure 27-2

LOCAL DATA IN REENTRANT BLOCKS

The most important consideration in writing reentrant procedures and functions is that of declaring local data. The issue that confronts the programmer is whether for each local data item he merely wants one "copy" of it, to be shared by all processes concurrently executing the block; or whether a separate "copy" for each process is wanted. Normal reentrant procedures require that execution by one process be completely decoupled from execution by another. Hence, separate copies for each process are usually required.

Separate copies of a local data item for each process concurrently executing a reentrant block are generated by the RTE as a result of declaring the data item in a particular way. Specifically, the data item is declared using the keyword AUTOMATIC.

The keyword AUTOMATIC was introduced in Section 16.4 as a method for causing local data to be initialized upon every entry into a block, rather than only the first. Used in reentrant blocks, it causes allocation of storage on entry into the block, as well as initialization. The keyword may be used even though the data item is not to be initialized.

In contrast, by default, or by using the keyword STATIC, storage for a data item will be allocated at compile time, and only one shared copy will exist.

Examples:

In the reentrant procedure:

```
P:PROCEDURE(A) ASSIGN(B) REENTRANT;
  DECLARE A VECTOR;
  DECLARE B SCALAR;
  DECLARE V VECTOR(3) AUTOMATIC;
  .
  .
  .
  V = VECTOR(B,0,0);
  B = V.A;
CLOSE P;
```

V is used to store an intermediate result in the calculations. One copy for each process is required to insure that P contains completely reentrant code. Hence, V is declared AUTOMATIC.

In contrast, suppose the number of times a reentrant procedure is invoked is required to be known and printed every 10 invocations. In this unusual, and rather artificial case, it would be appropriate to use a local data item not declared AUTOMATIC:

```
P2: PROCEDURE(A,B) ASSIGN(C) REENTRANT;
   DECLARE VECTOR, A, B, C;
   DECLARE COUNT INTEGER INITIAL(0);
   COUNT = COUNT + 1;
   IF REMAINDER(COUNT, 10) = 0 THEN
       WRITE(6)’NUMBER OF ENTRIES=‘||COUNT;
       .
       .
       .
       .
       .
CLOSE P2;
```

In an implementation where process swaps can only occur at the end of every executable statement, the code shown would maintain a correct count of the number of invocations.

OTHER CONSIDERATIONS IN REENTRANT BLOCKS

To preserve complete reentrancy of the code inside a reentrant procedure or function, other guidelines must be adhered to:

- Any procedure or function invoked by the reentrant block should itself be reentrant.
- Update blocks and inline functions[47] should declare no local data, either STATIC or AUTOMATIC.

It should be noted that no update block in a reentrant procedure or function can itself be reentrant, because of an update block's inherent properties (see Section 26.4). However, the processes executing the reentrant procedure or function can only pass through the update block serially. Hence, it appears as if process swaps were inhibited pending passage through the update block by each process, and cross-coupling of computational results in different processes still cannot occur. Hence, complete reentrancy is still effectively being preserved.

[47] To be described in Section 29.4.

This page intentionally left blank.

28.0 THE HAL/S NAME FACILITY

Successful and efficient systems programming in a higher order language requires an ability to "point to" specified data items. This implies the existence of the following constructs:

- a class of data items whose values are pointers to other data items, (or in assembly language terms, data items whose values are addresses of other data items);
- a mechanism for referencing and modifying pointer values at run time;
- a mechanism for referencing and modifying ordinary data items indirectly through pointers to them.

The HAL/S NAME facility satisfies all three of these requirements. This section introduces the conceptual basis of the facility, and describes in detail the language constructs involved in its use.

A careful reading of Section 19 is a prerequisite for the complete understanding of this section.

28.1 HAL/S NAME CONCEPTS

In some higher order languages, there exist pointer data items which at run time can be made to point to other data items of any kind. Thus, sometimes they may point to scalar data items, and at others they may point to integer or character data items. Ordinary data items indirectly referenced through pointers cannot in such a language be checked for legality in their given context at compile time.

Software using such pointer data items is therefore inherently unreliable, unless run time checking is instituted, which may be an unacceptable alternative in real time flight software applications.

NAME DATA ITEMS

In HAL/S, pointer data items are called "NAME" data items. To substantially eliminate software unreliability a specific mechanism in HAL/S assures that any given NAME data item can point to only other data items of a single kind specified at compile time. The mechanism consists in declaring a NAME data item with properties of type, precision, and arrayness, just as if it were an ordinary data item. These properties, rather than actually belonging to the NAME data item, are the properties which must be possessed by data items to which the NAME data item can point.

This has two beneficial implications:

- Any construct causing the pointer value of a NAME data item to be modified can be subjected to compile time checkout insuring that at run time the NAME data item will always point to another <u>legal</u> data item.
- When a NAME data item is used to indirectly access an ordinary data item, type, precision, and arrayness properties of the data pointed to are known and can be checked <u>at compile time</u> for legality in their given context.

In HAL/S, NAME data items can also point to programs and tasks, enabling the use of pointers in conjunction with SCHEDULE, WAIT, TERMINATE and CANCEL statements.

INDIRECT ACCESS THROUGH POINTERS

The appearance of an ordinary data item in an executable statement causes its value to be referenced or modified at run time. If a NAME data item appears in an executable statement as if it were an ordinary data item, then at run time the ordinary data item it is currently pointing to is referenced or modified. This is what is meant by the indirect accessing of data.

If a NAME data item points to a program or task, and it appears in the same context as an ordinary program or task name, then the program or task pointed to is being indirectly accessed.

ACCESSING POINTER VALUES

If the value of a NAME data item is itself to be referenced or modified, a special construct called the "NAME pseudo-function" is required. This serves to distinguish direct accessing of pointer values from indirect accessing through pointer values.

28.2 DECLARATION OF NAME DATA ITEMS

HAL/S allows NAME data items to be defined which can point to the following data types:

```
INTEGER    BIT STRING(and BOOLEAN)
SCALAR     CHARACTER
VECTOR     STRUCTURE
MATRIX     EVENT
```

In addition, NAME data items can be defined which can point to the following kinds of code block:

```
PROGRAM    TASK
```

NAME DATA ITEMS POINTING TO DATA

Declarations of NAME data items for pointing to data have exactly the same form as declarations of ordinary data items, except that the keyword NAME immediately follows the identifier name declared.

Examples:

```
DECLARE A NAME ARRAY(100) SCALAR;
DECLARE MATRIX(3,3) DOUBLE, M1 NAME, M2 NAME;
DECLARE B NAME BIT(16),
        C NAME CHARACTER(80);
 STRUCTURE Q:
        1 QI INTEGER,
        1 QS SCALAR,
        1 Q1,
          2 QB BIT(16),
          2 QC CHARACTER(80);
DECLARE ZQ NAME Q-STRUCTURE;
```

Given the above declarations:

- A may only point to 1-dimensional single precision scalar arrays of size 100.
- M1, M2 may only point to 3x3 double precision matrices.
- B may only point to 16-bit strings.
- C may only point to character strings of maximum length 80.
- ZQ may only point to Q-STRUCTURES (or tree equivalent structures) with a single copy.

DATA ITEMS POINTING TO CODE BLOCKS

Declarations of NAME data items for pointing to programs and tasks have the following basic form:

```
DECLARE name NAME PROGRAM;
DECLARE name NAME TASK;
```

1. *name* is any legal HAL/S identifier name.

Such declarations can be part of a compound or factored declaration statement.

Examples:

```
DECLARE  P1 NAME PROGRAM;
DECLARE  T1 NAME TASK;
DECLARE  P2 NAME PROGRAM,
         T2 NAME TASK,
         S1 NAME SCALAR;
```

Given these declarations:

P1, P2 may only point to program blocks.

T1, T2 may only point to task blocks.

NAME DATA ITEMS AS STRUCTURE TERMINALS

NAME data items for pointing to both data and program or task blocks may appear as structure terminals in a structure template. The definition of a NAME data item in a structure terminal takes the form described in Section 19.2, except that the keyword NAME follows the name of the structure terminal.

Examples:

```
STRUCTURE Q:
        1 QS NAME SCALAR,
        1 QI,
           2 QC NAME CHARACTER(80),
           2 QR NAME PROGRAM,
           2 QB NAME BOOLEAN,
        1 Q2,
           2 QA ARRAY(4) BIT(16);
```

Note that NAME data items for pointing to events can appear in a structure template, even though events themselves cannot. Note also that NAME data items in a template *A* may point to structures, even those possessing *A* as template.

Examples:

The following are legal definitions:

```
STRUCTURE R:
        1 QR NAME R-STRUCTURE,
        1 QE NAME EVENT;
DECLARE ZR R-STRUCTURE;
DECLARE NZR NAME R-STRUCTURE;
```

In this example NZR may point to ZR. ZR.QR may also point to ZR. The implications of this ability will be investigated later.

PROPERTIES OF DECLARED NAME DATA ITEMS

It has already been stated that the properties of type, precision and arrayness appearing in the declaration of a NAME data item actually specify the kind of data item or code block to which it can point. Other attributes besides these can appear in such declarations. Most of them serve the same purpose as described, but in contrast others apply to the NAME data item itself. Most prominent in the latter category is initialization.

The following table summarizes the purpose of each attribute which can appear in the declaration of a NAME data item.

ATTRIBUTE OF NAME DATA ITEM	Applies to Data or Code Block Pointed To	Applies to NAME Data Item Itself	Comments
`ARRAY()`	✓		See note ②
`BIT()`	✓		
`BOOLEAN`	✓		
`CHARACTER()`	✓		See note ①
`EVENT`	✓		
`VECTOR()`	✓		
`MATRIX()`	✓		
`INTEGER`	✓		
`SCALAR`	✓		
α `STRUCTURE()`	✓		See note ①
`PROGRAM`	✓		
`TASK`	✓		
`SINGLE`	✓		
`DOUBLE`	✓		
`DENSE`		✓	Affects NAME data item as if it were an ordinary data item. See Sections 26.1 & 26.2.
`ALIGNED`		✓	
`RIGID`		✓	
`REMOTE`	✓		See note ③
`ACCESS`			Illegal, but see note ④
`INITIAL()`		✓	Cause initialization of pointer value. To be described in Section 26.8.
`CONSTANT()`		✓	
`STATIC`		✓	State the kind of initialization, as for ordinary data items. See Section 16.4.
`AUTOMATIC`		✓	

NOTES:

① The forms ARRAY(*) or α-STRUCTURE(*) are illegal.

② The form CHARACTER(*) when used for a NAME data item, enables it to point to a character data item of <u>any</u> maximum length.

③ The REMOTE attribute may appear on the declarations of NAME data items as structure terminals since they may be required to point to REMOTE data.

④ The illegality of the ACCESS attribute does not prevent protected data items from being pointed to.

NAME DATA ITEMS AND TEMPORARIES

The nature and purpose of temporary data items were described in Section 26.3. The following rule summarizes relationships between temporary data items and NAME data items.

	No NAME data item may point to a temporary data item.

NAME FORMAL PARAMETERS

Formal parameters may be declared with the NAME keyword as if they were NAME data items. The purpose of this will be described in Section 28.9.

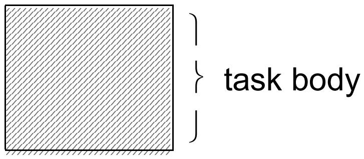
28.3 INDIRECT ACCESS THROUGH NAME DATA ITEMS

If a NAME data item appears in an executable statement as if it were an ordinary data item, then the data item it points to is taken to be accessed. Similarly, if a NAME data item appears as if it were the name of a program or task block, then the block it points to is taken to be accessed.

It might be said that the NAME data item has been substituted for the ordinary data item or block, so as to achieve indirect, rather than direct access.

Examples:

```
| DECLARE VECTOR(3), V, NV NAME;
| DECLARE SCALAR, S, NS NAME;
| DECLARE NT NAME TASK;
| .
| .
| .
| T: TASK;
```



```
| CLOSE T;
|
```

If NV → V, NS → S and NT → T[†], then

```
| NS = NV.NV;
| SCHEDULE NT IN NS PRIORITY(50);
```

effectively performs the operations:

```
|  S = V.V;
|  SCHEDULE T IN S PRIORITY(50);
```

The foregoing statements about appearances of NAME data items, while appearing

[†] In this and following examples "→" means "point to".

simple and unequivocal, contain a number of subtle implications arising from:

- interactions in structure data items;
- the effects of subscripting.

INDIRECT ACCESSING AND STRUCTURES

The subtleties of indirect accessing in conjunction with structures arise as a consequence of these two facts:

- Any structure may possess NAME structure terminals some of which may point to structure data items.
- Such a NAME structure terminal can actually point back to the structure containing it.

These subtleties are best illustrated by the extended examination of an apparently very simple example. By the rules given in Section 28.2, the following are legal structure declarations:

```
STRUCTURE A:
        1 C SCALAR,
        1 B NAME A-STRUCTURE;
DECLARE A-STRUCTURE, Z1, Z2, Z3;
DECLARE Z4 NAME A-STRUCTURE;
```

Z1.B is a NAME structure terminal of A-STRUCTURE type, which may therefore legally point to Z2. Pictorially:

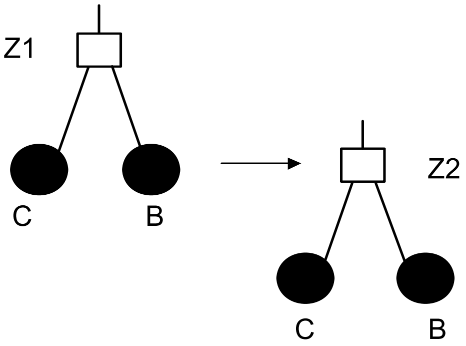


Figure 28-1

Because Z1.B points to Z2, any appearance of Z2 may be substituted by Z1.B, so achieving indirect access to Z2.

It is crucially important at this point to understand that because Z1.B points to Z2, parts of Z2 as well as Z2 itself may be indirectly accessed. For example, to achieve indirect access to Z2.C, the appearance of Z2 in the qualified name is substituted by Z1.B. That is, indirect access to Z2.C is achieved by the qualified form Z1.B.C.

To illustrate this substitution process further, if Z4 points to Z2, then Z2.C is indirectly accessed by the qualified form Z4.C, and if Z4 points to Z1, then Z2.C is indirectly accessed by the qualified form Z4.B.C.

Multiple levels of indirection are handled in the same way. Suppose for example that in addition Z2.B points to Z3. Then pictorially:

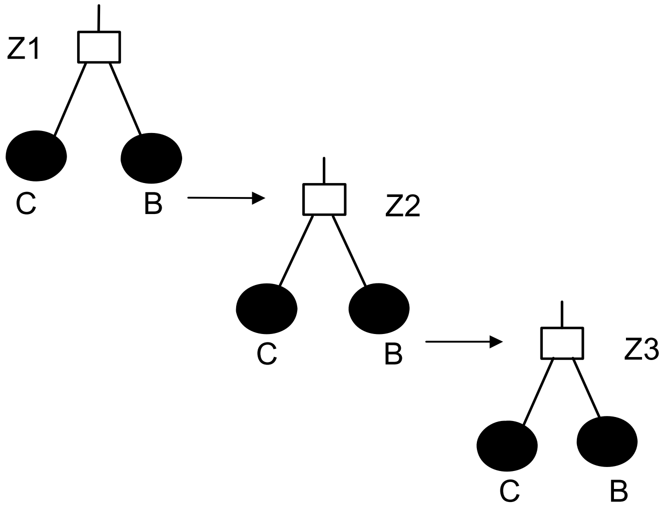


Figure 28-2

Using the same kind of substitution as before, Z3 may be indirectly accessed by the qualified form Z1.B.B, so that in its turn, structure terminal C in Z3 may be indirectly accessed by the qualified reference Z1.B.B.C.

Restating how the form Z1.B.B.C was arrived at, the following steps were taken:

- substitution of Z2.B.C for Z3.C (since Z2.B points to Z3);
- substitution of Z1.B.B.C for Z2.B.C (since Z1.B points to Z2).

There are other curious consequences arising from the interaction of indirect accessing with structures. Suppose now, for example, that Z2.B points to Z1 rather than Z3. Then, pictorially:

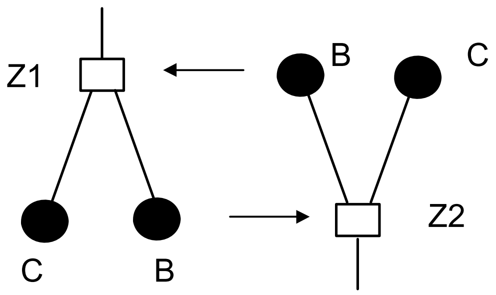


Figure 28-3

Now Z2.C can be indirectly accessed by the qualified form Z1.B.C, since Z1.B points to

Z2. Since Z2.B points to Z1, the following forms are also possible:

```
Z2.B.B.C
Z1.B.B.B.C
Z2.B.B.B.B.C
Z1.B.B.B.B.B.C
.
.
.
```

This example illustrates the logical consequence of a closed indirection loop between two structures.

INDIRECT ACCESS AND SUBSCRIPTING

In this discussion, for simplicity, subscripting in connection with structures or structure terminals will at first be excluded. With this restriction, subscripting on NAME data items is straightforward in its meaning.

> Subscripting is effective on the data item that is being indirectly accessed.

With this interpretation, it is clear that such subscripts must be legal for the data type pointed to. In particular, NAME data items pointing to programs and tasks may not be subscripted.

Examples:

```
DECLARE VECTOR(3), V, NV NAME;
DECLARE ARRAY(2) CHARACTER(4), C, NC NAME;
DECLARE BIT(4), B, NB NAME;
```

$$\text{Let } V \equiv \begin{bmatrix} 0.5 \\ 1.5 \\ 2.5 \end{bmatrix}, \quad C \equiv \quad (\text{'ABCD' 'EFGH'}), \quad B \equiv \quad 1010_2$$

Then if NV → V, NC → C, NB → B:

$NV_3 \equiv 2.5$ since V_3 is indirectly referenced,

$NC_{1:3} \equiv$ 'C' since $C_{1:3}$ is indirectly referenced,

$NB_{2\ TO\ 3} \equiv 01_2$ since $B_{2\ TO\ 3}$ is indirectly referenced.

NV_5, NB_9 are illegal since the subscripting is illegal for V and B respectively. Such subscripting is always illegal since NV can only point to 3-vectors, and B to 4-bit strings.

The complexities arising from structure subscripting are best studied by another apparently simple example. Suppose that the following declarations are made:

```
STRUCTURE A:
       1 C MATRIX(3,3),
       1 B NAME A-STRUCTURE;
DECLARE A-STRUCTURE(3), Z1, Z2, Z3 NAME;
```

Let copies 1, 2 and 3 of Z1.B point respectively to copies 2, 3 and 1 respectively of Z2. Pictorially:

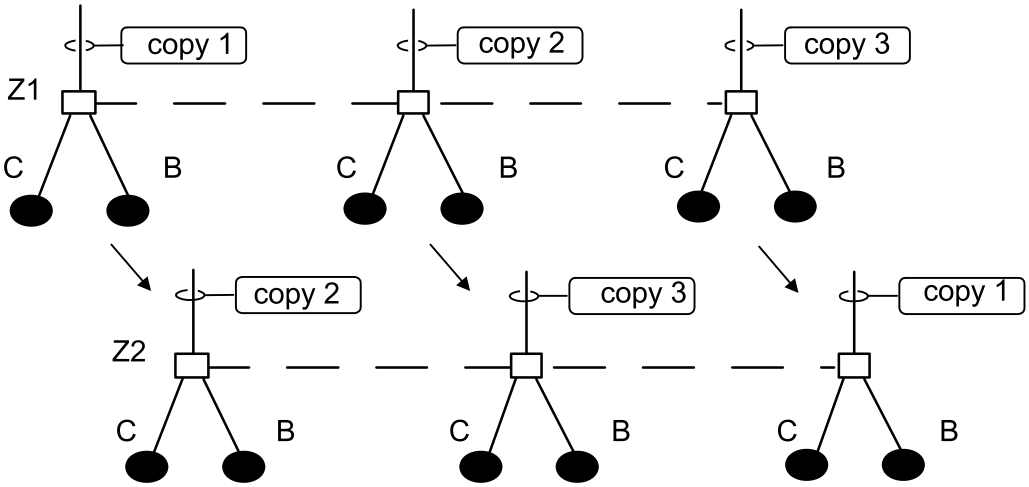


Figure 28-4

According to the substitution process previously described, the three copies of structure terminal C and Z2 can be indirectly accessed by specifying the three copies of Z1.B.C:

$Z1.B.C_{1;}$ indirectly accesses $Z2.C_{2;}$

$Z1.B.C_{2;}$ indirectly accesses $Z2.C_{3;}$

$Z1.B.C_{3;}$ indirectly accesses $Z2.C_{1;}$

Using the terminology of Section 20.1, Z2.C is an operand with arrayness {1:3}. Indirectly accessed as Z1.B.C, the operand still has arrayness {1:3} but the order of the individual elements is different. In general of course the three copies of Z1.B may point to three different structures (all with template A), resulting in operand Z1.B.C being synthesized from three different sources.

Note that the structure subscript is effective before indirection, not after. As a further illustration, in

$Z1.B.C_{1;\ 3,\ 3}$

the structure subscript selects copy 1 of the pointers Z1.B. Note, however, that in contrast the component subscript selects the component in row 3 and column 3 of C in the structure to which Z1.B points.

This is not always true for structure subscripts. For example, let Z3 point to Z2. Then in

$$\text{Z3.B.C}_{1;\ 3,\ 3}$$

the structure subscript selects copy 1 of Z2, which is pointed to by Z3.

These examples illustrate the following general rule:

> A structure subscript may either be effective on the data being indirectly accessed, or upon the NAME data item accessing it, depending on whether the data pointed to has copies, or whether the NAME data item itself has copies[†].

[†] Note that since a structure terminal which is itself a structure (or a NAME data item pointing to a structure) cannot possess copies, the two forms of structure subscripting are mutually exclusive.

28.4 THE NAME PSEUDO-FUNCTION

As briefly stated in Section 28.1, referencing or modifying pointer values requires use of a special construct called the NAME pseudo-function. This section states its form, and describes its properties and their implications.

BASIC FORM OF NAME PSEUDO-FUNCTION

The basic form of the NAME pseudo-function is that of a function with a single argument. The argument of the pseudo-function is a HAL/S data item or parameter of some description.

The arguments fall into two categories:

- NAME DATA ITEMS (including NAME formal parameters). The pointer value (or values) of the argument are accessed.
- ORDINARY DATA ITEMS, (including assign parameters, but not input parameters or temporary data items) and program or task block names. The pointer value (or values) to the argument are created.

The form of the NAME pseudo function is shown below:

```
NAME( item)
```

1. *item* is a NAME data item, ordinary data item, or program or task block name.
2. The legality of subscripting on the argument depends on the context in which the pseudo-function appears.

The appearance of a NAME pseudo-function in reference context causes one or more pointer values to be referenced or created:

- If the argument is an ordinary data item, one or more pointers to it are created;
- If the argument is a NAME data item, its current pointer value or values are referenced.

The appearance of a NAME pseudo-function in assignment context causes one or more pointer values to be modified. Consonant with this, in such a context, the argument may be only a NAME data item.

Instances of the NAME pseudo-function in both reference and assignment contexts will be described in Sections 28.5 through 28.9.

Examples:

Given:

```
DECLARE S SCALAR,
      NS NAME SCALAR,
      NT NAME TASK,
      NA NAME ARRAY(1000) INTEGER;
STRUCTURE Q:
      1 QS SCALAR,
      1 QN NAME Q-STRUCTURE;
DECLARE ZQ Q-STRUCTURE;
```

the following are legal:

```
NAME(S)       } reference only
NAME(ZQ.QS)   }
NAME(NS)
NAME(NT)
NAME(NA)
NAME(ZQ.QN)
```

the following are illegal:

```
NAME(1.5)
NAME(S/2)
```

INTERACTION WITH STRUCTURES

Section 28.3 described in detail how qualified structure naming forms which involve implicit indirect access could be constructed. Any such qualified form may appear as the argument of a NAME pseudo-function, with effects best summarized by example.

Take again the declarations:

```
STRUCTURE A:
        1 C SCALAR,
        1 B NAME A-STRUCTURE;
DECLARE A-STRUCTURE,Z1,Z2,Z3;
DECLARE Z4 NAME A-STRUCTURE;
```

Let Z1.B point to Z2, and Z2.B point to Z3, as shown pictorially below:

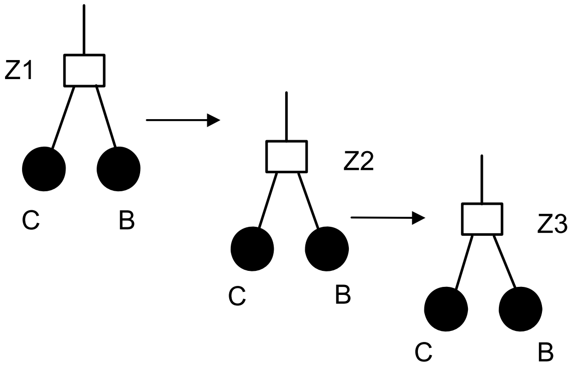


Figure 28-5

A pointer value to Z3.C can be created by the construct:

```
NAME(Z3.C)
```

Section 28.3 showed how Z3.C is indirectly accessed by the qualified form Z2.B.C because Z2.B points to Z3. Hence, a pointer value to Z3.C can also be created by:

```
NAME(Z2.B.C)
```

Now Z1.B points to Z2 so that Z3.C is accessed through two levels of indirection by Z1.B.B.C. A third way of creating a pointer value to Z3.C is therefore:

```
NAME(Z1.B.B.C)
```

If furthermore, Z4 points to Z1, then

```
NAME(Z4.B.B.C)
```

also has the same effect.

In each of the above cases, the argument of the NAME pseudo-function is Z3.C which is an ordinary data item, even though indirect access is used. Each of the above instances may therefore only be used in a reference context.

The pointer value of Z2.B can itself be set up by using

```
        NAME(Z2.B)
```

in an appropriate assignment context to be described. The NAME structure terminal Z2.B may be indirectly accessed by the qualified form Z1.B.B, since Z1.B points to Z2. Hence, the pointer value of Z2.B can also be set up by using:

```
        NAME(Z1.B.B)
```

in assignment context. With Z4 again pointing to Z1,

```
        NAME(Z4.B.B)
```

has the same effect, since Z2.B is again accessed, this time through two levels of indirection.

ARGUMENTS WITH SUBSCRIPTS

Depending on the context in which the NAME pseudo-function appears, subscripting of its argument may or may not be legal. Following the precedent set by Section 28.3, complications caused by structures will initially be ignored. With this restriction, two major rules apply:

1. Subscripting may only appear when the NAME pseudo-function is used to reference a pointer value, never when it is used to assign one.
2. Subscripting is effective on the ordinary data item specified or pointed to (possibly through several levels of indirection).

Additionally, only subscripts which perform the following specific operations are legal at all:

- selection of one scalar value from an unarrayed matrix or vector data item;
- selection of one array element from an array of any data type;
- selection of one scalar value from one array element of an array of matrices or vectors.

These restrictions are designed to ensure that in any implementation the resultant pointer is to an unfragmented area of physical storage[1].

[1] In particular, partitioning subscripts on matrices and arrays can cause the selection of fragmented areas of physical storage.

Examples:

Given the following declarations:

```
DECLARE V VECTOR(3),
       NV NAME VECTOR(3),
       S ARRAY(100) SCALAR,
       NS NAME ARRAY(100) SCALAR,
       M ARRAY(5) MATRIX(3,3),
       NM NAME ARRAY(5) MATRIX(3,3),
       C CHARACTER(80),
       NC NAME CHARACTER(80);
```

suppose that NV → V, NS → S, NM → M and NC → C.

The following are legal in contexts causing reference of pointer values:

NAME(V_3) creates pointer to scalar value which is 3rd element of vector V

NAME(NV_3) same as above since NV → V

NAME(S_5) creates pointer to 5th array element of array S

NAME(NS_5) same as above since NS → S

NAME($M_{3:1,1}$) creates pointer to scalar value in row 1, column 1 of 3rd array element of M

NAME($NM_{3:1,1}$) same as above since NM → M

NAME($M_{4:}$) creates pointer to 4th array element in M

The following are illegal:

NAME(C_1)
NAME(NC_1) } subscripting on character strings illegal

NAME($V_{1 \text{ TO } 2}$) more than one element of V selected

NAME($M_{*:1,1}$) one scalar value selected from more than one array element

The problem of structure subscripting on the argument of a NAME pseudo-function is now addressed. Section 28.3 showed how qualified names which indirectly access structure terminals could be formed. It also described how structure subscripting is either effective on the data indirectly accessed, or on the NAME data item accessing it, depending on which possesses multiple copies. Structure terminals accessed by such subscripted qualified forms can appear as arguments of the NAME pseudo-function.

The two rules previously stated for subscripted arguments of the NAME pseudo-function must be restated to allow for this. The modified rules are as follows:

1. When a NAME pseudo-function is used to assign pointer values, only structure subscripting effective on the pointer copies is legal.
2. For NAME pseudo-functions in reference context, array and component subscripting is always effective on the ordinary data item specified or indirectly accessed. Structure subscripting is effective in the ordinary data item specified or indirectly accessed, or upon the NAME data item indirectly accessing it, depending on which possesses the multiple copies.

Even when subscripts effective on the ordinary data item pointed to are legal, only restricted forms are allowed. Ordinary data items which are structure terminals are subject to the additional restrictions on array and component subscripts already described. Furthermore, where the data item is the whole or part of a structure with multiple copies, the following rules apply:

- Structure subscripting must select one copy only;
- such structure subscripting is mandatory unless the entire structure is being pointed to.

Application of the above set of rules is illustrated by the following example.

Example:

Given the following declarations

```
STRUCTURE A:
1 M ARRAY(5) MATRIX(3,3),
1 C CHARACTER(80),
1 V VECTOR(6),
1 B NAME A-STRUCTURE;
DECLARE Z A-STRUCTURE;
DECLARE A-STRUCTURE(3), Z1, Z2, Z3 NAME;
```

let $Z1.B_1$ → $Z2_2$
$Z1.B_2$ → $Z2_3$
$Z1.B_3$ → $Z2_1$
$Z3$ → $Z1$

Illustrations for NAME pseudo-functions in a reference context -

(a) Array and component subscripting:

NAME(Z.M$_{1:3,3}$) Creates a pointer to the scalar value in row 3, column 3 of the first array element of Z.M

NAME(Z.M$_{*:1,1}$) is illegal since the subscript selects a scalar value from more than one array element of Z.M

NAME(Z.C$_{10 \text{ TO } 15}$) is illegal since character strings may not possess component subscripts

NAME(Z.V$_{1}$) creates a pointer to the 1st element of vector Z.V

NAME(Z.V$_{1 \text{ TO } 3}$) is illegal since more than one element of Z.V is selected by the subscript

(b) Structure subscripting effective upon the data item pointed to or directly specified:

NAME(Z1$_{2}$) creates a pointer to the second copy of Z1 since the subscript acts directly on Z1

NAME(Z3$_{2}$) since Z3 is a single pointer, pointing to the whole of Z1, the subscript is effective on Z1 rather than Z3; hence a pointer to the second copy of Z1 is again created

NAME(Z1.M$_{2;}$) creates a pointer to the array of matrices M in the second copy of Z1

NAME(Z3.M$_{2;}$) as before, the structure subscript is effective on Z1 rather than Z3; hence as before a pointer to the array of matrices M in the second copy of Z1 is created

NAME(Z1.M$_{1 \text{ TO } 2;}$) is illegal since the subscript selects more than one copy of structure Z1

NAME(Z3.M$_{1 \text{ TO } 2;}$) is illegal for the same reason

NAME(Z1.M) is illegal since subscripting to select one copy of Z1.M must be used

NAME(Z3.M) is illegal for the same reason

(c) Structure subscripting effective on a pointer value:

The following examples use the fact that Z1.B$_1$ points to Z2$_2$

NAME(Z1.B$_{1}$) references the pointer value Z1.B$_1$, i.e. it creates the pointer to Z2$_2$

NAME(Z1.B.M$_{1;}$) the subscript is effective of Z1.B, so that a pointer to the array of matrices in the second copy of Z2 is created

NAME(Z1.B.V$_{1;1}$) the structure subscript is effective on Z1.B as before so that a pointer to the first component of the vector in the second copy of Z2 is created

Note that there is no restriction on the selection of one pointer only by a structure subscript effective on pointer data:

`NAME(Z1.B)` - “simultaneously” references three pointer values

The significance of generating more than one pointer at a time is discussed later.

Illustrations for structure subscripts in NAME pseudo-functions in assignment context are adequately covered by the examples given in part (c) above, since any other kind of subscripting is illegal.

ARRAYED SUBSCRIPTING

Section 20.3 discussed the phenomenon of arrayed subscripting. Subscripts appearing on the argument of a NAME pseudo-function may under no circumstances be arrayed.

POINTER ARRAYNESS

The preceding examples have made it apparent that a NAME pseudo-function could generate or assign more than one pointer value at a time. Such NAME pseudo-functions are said to have “pointer arrayness”.

Pointer arrayness can arise whenever a NAME structure terminal of a structure with multiple copies is used. The following example illustrates this explicitly.

Example:

```
|
|  STRUCTURE A:
|          1 D NAME SCALAR,
|          1 C SCALAR,
|          1 B NAME A-STRUCTURE;
|  DECLARE A-STRUCTURE(2), Z1, Z2;
|
```

Given the above declarations, NAME(Z1.D) in reference context simultaneously references the 2 pointer values of Z1.D. In assignment context, the two pointer values of Z1.B could be assigned simultaneously by NAME(Z1.B).

Further, if $Z1.B_1 \rightarrow Z2_2$ and $Z1.B_2 \rightarrow Z2_1$ then

NAME(Z1.B.C) in reference context can generate simultaneously pointer values to $Z2.C_2$ and $Z2.C_1$.

The previous discussion on subscripting has shown that appropriate structure subscripting can reduce the number of pointers generated or assigned, ultimately but not necessarily to one.

If the number of pointers simultaneously generated or assigned by a NAME pseudo-function is N then the pointer arrayness of the NAME pseudo-function is denoted by

`{N}`

The behavior of pointer arrayness in operations involving the NAME pseudo-function is similar to that of ordinary arrayness in regular expressions (as described in Section 20).

The importance of the concept of pointer arrayness will thus become clear when constructs using the NAME pseudo-function come to be described.

28.5 NULL POINTER VAUES

Generally the use of a pointer facility requires the definition of a "null" pointer value. In HAL/S a "null" pointer value is indicated by the keyword NULL.

There are two forms of specification:

```
NULL
NAME(NULL)
```

1. The above forms are equivalent and interchangeable.

It is used in reference contexts in the same way as instances of the NAME pseudo-function.

Example:

To generate a null pointer instead of a pointer to X, use NULL or NAME(NULL) instead of NAME(X).

28.6 INITIALIZATION OF NAME DATA ITEMS

Although Section 28.2 dealt with the declaration of NAME data items, discussion of initialization was deferred because the construct makes use of the NAME pseudo-function.

The form of initialization construct is as described in Section 4.3. However, for NAME data items, the values in the initial list are pointer values rather than literals. Pointer values are generated by use of the NAME pseudo-function, or are null.

This is an instance in which the appearance of the NAME pseudo-function is in a reference context. However, in this particular instance much more severe restrictions are placed on the argument of the pseudo-function:

1. It must be a previously-declared ordinary data item.
2. Its attributes must be such that it is legal for the NAME data item to point to it.
3. No implicit indirection by qualified structure references are allowed.

Examples:

The following are legal initializations of NAME data items:

```
DECLARE  S SCALAR,
         V ARRAY(4) VECTOR DOUBLE;
DECLARE  NS1 NAME SCALAR INITIAL(NAME(S));
DECLARE  NV1 NAME ARRAY(4) VECTOR DOUBLE
         INITIAL(NAME(V));
STRUCTURE A:
    1 C SCALAR,
    1 B NAME A-STRUCTURE;
DECLARE Z1 A-STRUCTURE;
DECLARE Z2 A-STRUCTURE INITIAL(1.5, NAME(Z1));
DECLARE NA NAME SCALAR INITIAL(NAME(Z1.C));
```

The following are illegal initializations of NAME data items:

```
  DECLARE  T SCALAR;
  DECLARE  NT NAME SCALAR DOUBLE
           INITIAL(NAME(T));
                        ↑— NT cannot legally point to T
  DECLARE NT1 NAME SCALAR INITIAL(NAME(T1));
  DECLARE T1 SCALAR;                     ↑
                        T1 is not previously defined
  DECLARE V VECTOR(4);
  DECLARE TV NAME SCALAR INITIAL(NAME(V  ));
S                                       3  ← subscripting
                                              illegal
  STRUCTURE X:
     1 Y SCALAR,
     1 Z NAME X-STRUCTURE;
  DECLARE XX1 X-STRUCTURE;
  DECLARE XX2 X-STRUCTURE INITIAL(1.5, NAME(XX1));
  DECLARE NX NAME SCALAR INITIAL(NAME(XX2.Z.Y));
                                         ↑
                  Contains implicit indirection since XX2.Z→XX1
                  through previous initialization
```

NULL INITIALIZATION

All NAME data items which are not explicitly initialized, are implicitly initialized with null pointer values. The following examples show the explicit initialization to null pointer values.

Examples:

```
DECLARE LV NAME VECTOR INITIAL(NULL);
STRUCTURE A:
    1 C SCALAR,
    1 B NAME A-STRUCTURE;
DECLARE Z A-STRUCTURE(20) INITIAL(20#(7.53, NULL));
                                              ↑
                       each copy of B initialized to a null pointer value
```

28.7 NAME ASSIGNMENTS

A primary use of the NAME pseudo-function is in the NAME assignment statement, where it is used to assign pointer values. The NAME assignment statement looks similar in form to the regular assignment statement described in Section 8, except that:

- the left hand or receiving operand is a NAME pseudo-function in an assignment context;
- the right hand operand is either a NAME pseudo-function in a reference context, or the specification of a null pointer value.

BASIC FORM

The basic form of the NAME assignment statement is as follows:

Symbolic form: L = R;

1. The receiving operand *L* is a NAME pseudo-function in assignment context.
2. The right hand operand *R* is either a NAME pseudo-function in reference context, or the specification of a null pointer value as described in Section 28.5.
3. Given the first alternative in Rule 2, the NAME data item specified in *L* must legally be able to point to the ordinary data item whose pointer value is generated by *R*.

Arguments of the NAME pseudo-functions in a NAME assignment follow the rules laid down in Section 28.4.

Examples:

Given the declarations

```
DECLARE S SCALAR,
        NS NAME SCALAR,
        NSD NAME SCALAR DOUBLE;
DECLARE V VECTOR(3),
        NV NAME VECTOR(3);
STRUCTURE A:
    1 C SCALAR,
    1 B NAME A-STRUCTURE;
DECLARE   Z1 A-STRUCTURE,
          Z2 A-STRUCTURE,
          NZ NAME A-STRUCTURE;
```

then

`NAME(NSD) = NULL;`	results in NSD→∅[†]
`NAME(NS) = NAME(S);`	results in NS→S
`NAME(NSD) = NAME(NS);`	is illegal since NS∏S and NSD may not legally point to S itself
`NAME( NV) = NAME(V);`	results in NV→V

`NAME(NS) = NAME(V  );`
`S                  2`
	results in NV→V2 - note that V2 is a scalar value, which is why NS may legally point to it

`NAME(NZ) = NAME(Z1);`	results in NZ→Z1
`NAME(NZ.B)= NAME(Z2);`	results in Z1.B∏Z2 because of implied indirection in qualified reference NZ.B, in which NZ→Z1
`NAME(NS)=NAME(NZ.B.C)`	results in NZ→Z2.C because of 2 levels of implied indirection in qualified form NZ.B.C, in which NZ→Z1 and Z1.B→Z2

[†] ∅ indicates a null pointer value.

MULTIPLE ASSIGNMENTS

Section 8.5 showed how regular assignment statements could possess multiple left hand operands. NAME assignments can also possess multiple left hand operands, so enabling one pointer value to be assigned to more than one NAME data item at a time.

The general form of a multiple NAME assignment statement is shown below:

Symbolic form: L^1, L^2,....L^n = *R;*

1. The receiving operand L^1...L^n is a NAME pseudo-function in assignment context.
2. The right hand operand *R* is the same as in the basic form of assignment.
3. If *R* is a NAME pseudo-function, the NAME data items specified in L^1... L^n must each legally be able to point to the ordinary data item whose pointer value is created by *R.*

Example:

Given

```
DECLARE   S SCALAR,
          NS NAME SCALAR,
          NT NAME SCALAR;
STRUCTURE U:
          1 US NAME SCALAR,
          1 UN NAME U-STRUCTURE;
DECLARE Z U-STRUCTURE;
```

The following is a legal multiple NAME assignment:

```
NAME(NS), NAME(NT), NAME(Z.US) = NAME(S);
```

POINTER ARRAYNESS IN NAME ASSIGNMENTS

Section 28.4 discussed the ability of a NAME pseudo-function to generate or assign more than one pointer at a time, calling this property "pointer arrayness".

Pointer arrayness in NAME assignments must conform to the following requirements:

1. In the basic form of NAME assignment, if the *R*-operand has a pointer arrayness {N}, then the *L*-operand must have pointer arrayness {N}. If the *R*-operand has no pointer arrayness, the *L*-operand may have arbitrary pointer arrayness or none.
2. In multiple NAME assignments, all *L*-operands are always required to have the same pointer arrayness.

Examples:

Given

```
|
|  STRUCTURE A:
|          1 B NAME SCALAR,
|          1 C SCALAR;
|  DECLARE Z1 A-STRUCTURE(3),
|           Z2 A-STRUCTURE(5);
|  DECLARE S SCALAR;
|
```

then 3 copies of Z1.B exist, and 5 copies of Z2.B exist. Hence in

```
|
| NAME(Z1.B) = NAME(S);
|
```

the pointer arrayness on the left is {3} while the right hand operand has none. The result of this assignment is:

$$\left.\begin{array}{l}Z1.B_{1;}\\ Z1.B_{2;}\\ Z1.B_{3;}\end{array}\right]\rightarrow S$$

Further,

```
|
| NAME(Z2.B) = NAME(Z1.B);
|
```

is illegal since the left and right hand pointer arraynesses are {5} and {3} respectively which do not match. However,

```
|
|   NAME(Z2.B),      ) = NAME(Z1.B);
|S             3 TO 5;
|
```

is legal since the left hand arrayness has been reduced to {3}. The result of the assignment is

$Z2.B_{3;} \equiv Z1.B_{1;}$ (i.e. they both have the same pointer value)
$Z2.B_{4;} \equiv Z1.B_{2;}$
$Z2.B_{5;} \equiv Z1.B_{3;}$

If pointer values of both Z1.B and $Z2.B_{3\ TO\ 5;}$ are to be assigned together, then

```
|
|   NAME(Z1.B), NAME(Z2.B        ) = NAME(S);
|S                         3 TO 5;
|
```

is the appropriate assignment.

Note that

```
|
|   NAME(Z1.B), NAME(Z2.B) = NAME(S);
|   NAME(Z1.B), NAME(Z2.B   ) = NAME(S);
|S                        1;
|
```

are both illegal because the pointer arrayness of the left hand sides in each case do not match.

28.8 NAME COMPARISONS

Section 9.2 of Part I showed how relational expressions could be built by combining comparisons with the operators &, |, and ¬. Such comparisons may include comparisons between pointer values. Pointer values are compared through use of the NAME pseudo-function and the null pointer specification. Only the Class II operations are legal in NAME comparisons:

Symbol	Purpose	Class
=	equals	II
NOT = ¬ =	not equals	

The rules for NAME comparisons are given below:

=
Symbolic form: L NOT = R
¬ =

1. The *L* and *R* operands are either NAME pseudo-functions in reference context or null pointer value specifications.
2. If both *L* and *R* operands are NAME pseudo-functions, the ordinary data items pointed to must have matching attributes (i. e. there must exist a NAME data item which can legally possess the pointer value generated either by *L* or by *R*).
3. Equality is achieved if both *L* and *R* generate the same pointer value.

Arguments of the NAME pseudo-functions in a NAME comparison follow the rules laid down in Section 28.4.

Examples:

Given

```
DECLARE  S SCALAR;
DECLARE  NS NAME SCALAR INITIAL(NAME(S)),
         NT NAME SCALAR INITIAL(NULL);
```

Then

```
NAME(NS) = NAME(S) is TRUE;
NAME(NS) = NAME(NULL) is FALSE;
NAME(NT) ¬= NAME(NULL) is FALSE;
NAME(NT) ¬= NAME(NS) is TRUE;
```

POINTER ARRAYNESS IN NAME COMPARISONS

Any NAME pseudo-function in a NAME comparison may have pointer arrayness. In such circumstances the following rules apply:

1. Either one or both *L* and *R* operands may possess pointer arrayness.
2. If both *L* and *R* operands possess pointer arrayness, they must possess the same pointer arrayness.

When the comparison possesses pointer arrayness in the above sense, it is viewed as a set of elemental comparisons proceeding in parallel. The outcome of all elemental comparisons are combined to form a single TRUE or FALSE result, in accordance with the following table:

Operation	Result	Conditions for Result
=	TRUE	Equality in all elemental comparisons is obtained
	FALSE	Equality in one or more elemental comparisons is lacking
NOT= ¬ =	TRUE	Equality in one or more elemental comparisons is lacking
	FALSE	Equality in all elemental comparisons is obtained

Examples:

Given

```
STRUCTURE A:
        1 D NAME SCALAR,
        1 C SCALAR;
DECLARE Z1 A-STRUCTURE(3),
        Z2 A-STRUCTURE(5);
DECLARE S SCALAR;
```

After execution of

```
NAME(Z1.D) = NAME(S);
```

then the result of the comparison

`NAME(Z1.D) = NAME(S) is TRUE`

since

$$\left.\begin{array}{l} \text{Z1.D}_1; \\ \text{Z1.D}_2; \\ \text{Z1.D}_1; \end{array}\right] \rightarrow \text{S}$$

After subsequent execution of

```
  NAME(Z1.D  ) = NULL;
S           1;
```

then the result of the comparison

`NAME(Z1.D) = NAME(S)` is FALSE

because

$$\begin{array}{l} \text{Z1.D}_1; \rightarrow \varnothing \\ \text{Z1.D}_2; \rightarrow \text{S} \\ \text{Z1.D}_1; \longrightarrow\!\uparrow \end{array}$$

The comparison

```
NAME(Z1.D) = NAME(Z2.D)
```

is illegal because the pointer arraynesses of the left and right operands are {3} and {5} respectively, which do not match. However, the comparison

$\text{NAME(Z1.D) = NAME(Z2.D}_{3\ \text{TO}\ 5})$

is legal since the pointer arrayness of the right hand operand has been reduced to {3}.

PRECEDENCE OF NAME COMPARISONS

The precedence of NAME comparisons in relational expressions is the same as that of any other kind of comparison. The relevant precedence rules have already been tabulated in Section 9.2.

28.9 ARGUMENT PASSAGE OF POINTER VALUES

Pointer values may appear as arguments in procedure or function invocations provided that the corresponding formal parameters are declared using the keyword NAME, as if they were NAME data items.

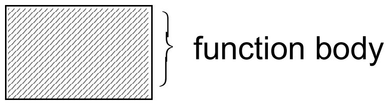
- INPUT ARGUMENTS - The NAME pseudo-function in reference context, or the null pointer value specification is used.
- ASSIGN ARGUMENTS - The NAME pseudo-function in assignment context is used.

INPUT ARGUMENTS

The effect of using a pointer value as an input argument of a procedure or function is as if the pointer value were being assigned to the corresponding NAME input parameter. The attributes of the NAME input parameter must therefore allow legal acceptance of that pointer value.

Examples:

```
DECLARE S SCALAR;
DECLARE NS NAME SCALAR;
DECLARE NT NAME TASK;
.
.
.
F:FUNCTION(A,B) SCALAR;
  DECLARE A NAME SCALAR,
          B BOOLEAN;
```



```
CLOSE F;
.
.
.
NAME(NS) = NAME(S);
S = F(NAME(S), TRUE); { invocation results in input parameter A pointing to S

S =F(NAME(NS),FALSE);has the same effect: A gets same pointer value as NS ,i.e.A→S

S = F(NAME(NT), TRUE);  is illegal since pointer values legal for NT are not legal for A

S = F(NULL, FALSE);     results in A →∅
```

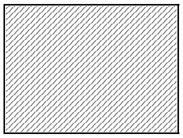
Note that although ordinary input parameters are prevented from appearing in NAME pseudo-functions, NAME input parameters are only prevented from appearing in NAME pseudo-functions in assignment context.

ASSIGN ARGUMENTS

A pointer value may be passed both into and out of a procedure by the appearance of a NAME pseudo-function in the assign argument list of the procedure's invocation. The class of data items which can be pointed to by the NAME data item appearing in the NAME pseudo-function must be the same as that which can be pointed to by the corresponding NAME assign parameter.

Examples:

```
DECLARE NAME SCALAR;
DECLARE NT NAME TASK;
  STRUCTURE A:
       1 B NAME A-STRUCTURE,
       1 C SCALAR;
DECLARE Z A-STRUCTURE;
  .
  .
  .
P: PROCEDURE ASSIGN(U,V);
     DECLARE U NAME TASK,
             V NAME A-STRUCTURE;
```



procedure body

```
CLOSE P;
 .
 .
 .
 .
 .
CALL P ASSIGN(NAME(NT), NAME(Z.B));
                                    causes passage of pointer values
                                    between NT and U, and between Z.B and V.

CALL P ASSIGN(NAME(NS), NAME(Z));
                     ↑          ↑__ illegal because Z is not a
                     |              NAME data item.
     Illegal because NS points to scalar
     data items but NT points to tasks
```

POINTER ARRAYNESS IN ARGUMENTS

No NAME formal parameter can possess multiple pointer copies, because a formal parameter cannot be declared as a NAME structure terminal of a structure with multiple copies. Hence, an appearance of a NAME pseudo-function as the argument of a procedure or function invocation may only give rise to the transmission of one pointer value per invocation.

The implications of this differ depending on whether the argument is in a procedure invocation or a function invocation.

- PROCEDURE INVOCATIONS: NAME pseudo-functions appearing as arguments of a procedure invocation may not possess pointer arrayness.
- FUNCTION INVOCATIONS: NAME pseudo-functions appearing as arguments of a function invocation may possess pointer arrayness. The pointer arrayness must match the ordinary arrayness of the expression in which the invocation is imbedded, as if it were itself an ordinary arrayness. The function is repeatedly invoked, once for every elemental evaluation of the outer expression; and, during each invocation, transmittal of one of the pointer values takes place. Reference to Section 20.6 will clarify this behavior.

Examples:

```
STRUCTURE A:
   1 B NAME SCALAR;
DECLARE Z A-STRUCTURE(20);
DECLARE S1 ARRAY(20) SCALAR,
         S2 ARRAY(10)  SCALAR;
.
.
.
P: PROCEDURE(U) ASSIGN(V);
   DECLARE U NAME SCALAR,
              V NAME SCALAR;
                          }
                          } procedure body
                          }
CLOSE P;
 .
 .
 .
F: FUNCTION(W) SCALAR;
     DECLARE W NAME SCALAR;
                          }
                          } function body
                          }
CLOSE F;
.
.
.
CALL P(NAME(Z.B$_{1;}$)) ASSIGN (NAME(Z.B));

.      legal because pointer            illegal because pointer arrayness exists
.     arrayness subscripted away
.
S1 = S1 + F(NAME(Z.B));

legal because pointer arrayness {20} matches arrayness {1:20} of S1
```

The above is equivalent to

$S1_i = S1_i + F(NAME(Z.B_i));$ for $1 \le i \le 20$

wherein each of 20 invocations of F cause transmission of a different pointer value.

Note that

```
S2 = S2 + F(NAME(Z.B));
```

is illegal because the pointer arrayness of Z.B does not match the regular arrayness {1:10} of S2.

28.10 POINTER VALUES IN INPUT/OUTPUT

No construct using the NAME pseudo-function exists to allow pointer values to be input or output. However, because structures containing NAME structure terminals may be input or output, rules must be laid down specifying their behavior in such circumstances.

SEQUENTIAL INPUT/OUTPUT

Sequential I/O statements were described primarily in Section 12 and 22.1. The sequential I/O of structure data items was described in Section 19.12.

The fundamental rule for NAME structure terminals is that they do not take part in the I/O operation: so far as input as output processing is concerned they do not exist.

Example:

So far as sequential I/O is concerned, structures Y and Z declared below are exactly equivalent.

```
STRUCTURE A:
  1 A1 SCALAR,
  1 N NAME VECTOR(3),
  1 A2 CHARACTER(80),
  1 A3 MATRIX(3,3);
STRUCTURE B:
  1 B1 SCALAR,
  1 B2 CHARACTER(80),
  1 B3 MATRIX(3,3);
DECLARE Z B-STRUCTURE,
        Y A-STRUCTURE;
```

The pointer value of Y.N would not be changed by any input operation.

RANDOM ACCESS INPUT/OUTPUT

Random access I/O has been described in Section 22.2. In contrast to sequential I/O, NAME structure terminals do take part in random access I/O. The pointer value involved is input or output along with the other parts of the structure.

Example:

```
STRUCTURE Q:
        1 A NAME SCALAR,
        1 B ARRAY(1000) BIT(16);
DECLARE Q-STRUCTURE, Q1, Q2;
.
.
.
.
.
FILE(1,10) = Q1;
Q2 = FILE(1,10);
```

The above FILE statements result in the pointer value originally in Q1.A being transferred to Q2.A, just as the values of Q1.B are transferred to Q2.B

This page intentionally left blank

29.0 REPLACE MACROS AND INLINE FUNCTIONS

The simple REPLACE statement which defines symbolic text substitutions was introduced in Section 5 of Part I. It was stated that the REPLACE statement is used to define a “replace name” symbolically representing arbitrary HAL/S text, and that subsequent appearances of the replace name cause the HAL/S compiler to substitute the text represented.

The utility of this feature is greatly extended by the ability to specify parametric replacements wherein the text to be substituted is modifiable from substitution to substitution. This section describes how such parametric replacements are defined and used.

The “inline function” is a HAL/S construct designed to enhance the versatility of parametric replacements. It takes the form of a parameterless function block whose definition is also its invocation. Inline functions and their use in conjunction with parametric replacements are also described in this section.

29.1 THE PARAMETRIC REPLACE STATEMENT

The REPLACE statement as defined in Section 5.1 allows only simple replacements to be specified. Replace names for specifying parametric replacements are define with a list of parameters which also appear in the text to be substituted, and are called “replace macros”. Every appearance of the replace macro is accompanied by a list of text string arguments that replace the parameters in the text to be substituted.

The form of REPLACE statement for defining replace macros is shown below:

REPLACE name(parm$_1$,…parm$_n$) BY “XXXXXXX”;

1. *name* is the name of the replace macro, and is a legal HAL/S identifier name.
2. *parm*^{1},…*parm*^{n} is an arbitrary number of parameters, each of which is a legal HAL/S identifier name.
3. XXXXXXX represents HAL/S source text to be substituted, Any of the parameters specified in the replace macro definition may appear in it any number of times. It conforms to the same rules as given in Section 5.1 for the simple REPLACE statement.

Examples:

```
REPLACE A(X,Y) BY “READ(X) Y”;
REPLACE B(Z) BY “Z = Z + 1”;
```

The text to be substituted may not begin or end in the middle of identifiers, reserved words, literals or imbedded comments. Effectively this means that the text must be complete in itself, and cannot result in the generation of any new symbols when substitution occurs.

This rule is equally applicable to the substitution of parameters in the replace text. The appearance of a parameter in a literal, identifier, reserved word, or imbedded comment is not seen by the HAL/S compiler as such, and substitution will not occur.

Example:

```
REPLACE P(A) BY “READ(5) ‘VALUES:’,AB, A”;

                these do not  ←——  appearance of A
                constitute
                appearance of
                parameter A
```

29.2 USE OF REPLACE MACROS

Every appearance of a replace macro is accompanied by a list of text string arguments that replace the parameters in the text to be substituted. The general form of such an appearance is given below:

name(*arg*^{1}, *arg*^{2},.... *arg*^{n})

1. *name* is the name of the replace macro.
2. Each *arg* is either a string of text which conforms to the same rules as the substitution text itself, as described earlier, or is empty.
3. Each argument replaces the corresponding parameter in the replace definition during substitution.

Each non-empty text string argument also has the following restrictions:

- It may contain only an even number of apostrophes (’), ensuring that bit string and character literals are completely contained within it.
- It may contain only an even number of double quote marks (”), ensuring that substitution text of any imbedded REPLACE statement is completely contained within it.
- It must contain a balanced number of left and right parentheses.
- Since commas (,) are used to separate text string arguments, commas can only appear in arguments if they are part of a character literal, or are imbedded in replace text, or nested within parentheses.

Note that, because the arguments consist of HAL/S source text, blanks are always considered potentially significant, and are included in the parameter replacement process.

Examples:
If the replace macro TEST is defined by:

```
REPLACE TEST(A,B,C) BY “IF A THEN B ELSE C";
```

then

```
TEST(P = 0, S = 1;, S = 2;)
```

expands into

```
IF P = 0 THEN S = 1; ELSE S = 2;
```

The instance

```
TEST(P = 0, S = 1;, S = 3**() P + Q);
```

although intended to expand into

```
IF P = 0 THEN S = 1; ELSE S = 3**(P + Q);
```

is illegal since the last argument has an unmatched parenthesis.
Note that

```
TEST(P = 0,S = 1;,);
```

expands into

```
IF P = 0 THEN S = 1; ELSE;
```

since the last argument is empty.

THE SUBSTITUTION PROCESS

The substitution process itself must be examined more closely if the full potential of the replace macro for nesting and recursion are to be understood.

The recognition of replace macros, and the resulting text substitutions, are carried out by a part of the HAL/S compiler which will, for convenience, be called the “macro scanner”.

On seeing the appearance of a replace name with or without arguments, the macro scanner first substitutes the replace text. It then begins scanning through the text from left to right, searching for appearances of parameters, or other replace names (with or without arguments). When one is found, the related text is substituted, and scanning resumes from the beginning of the inner text just substituted.

Therefore, parameters and replace names appearing in substituted text strings are always themselves replaced by more text, however deep the level of nesting may get.

Example:

Given

```
REPLACE C BY “(X+Y)/2”;
REPLACE B BY “SIN(C)”;
REPLACE A BY “X=B+1”;
```

Then the appearance

```
A;
```

is expanded by the macro scanner in the following stages.

```
 X =  B + 1;                        …1st substitution
 ↑        ↑_ embedded replace name found
scanning starts here

 X = SIN(C) + 1;                    …2nd substitution
 ↑          ↑_embedded replace name found
scanning resumes here

 X = SIN((X + Y)/2) + 1;            …3rd substitution
            ↑          ↑_no more replace names found - end of scan
scanning resumes here
```

It is important to remember that during the substitution of a replace macro that any appearance of a parameter, either in the text substituted or appearing in it as a result of an inner substitution, will be recognized and replaced by the corresponding argument text string. This may give rise to recursive substitution in some circumstances.

Examples:

(a) Given

```
REPLACE Z BY “SIN(X)”;
REPLACE A(Y,X) BY “WRITE(6) Y, Z”;
```

the appearance

```
A(‘VALUE=‘, 1.5);
```

is expanded by the macro scanner in the following stages:

```
 WRITE(6) ‘VALUE =’,Z;               …1st substitution
  ↑                  ↑______embedded replace name found
 scanning starts here
WRITE(6) ‘VALUE =’,SIN(X);           …2nd substitution
                    ↑   ↑______parameter found
   scanning resumes here
 WRITE(6) ‘VALUE =’,SIN(1.5);        …3rd substitution
                        ↑   ↑____end of scan
                 scanning resumes here
```

(b) Given

```
REPLACE A(X,Y) BY “Z = X + Y”;
```

then the appearance

```
A(1.5, X);
```

is expanded as follows:

```
 Z =  1.5 + X;     …1st substitution
 ↑           ↑______ parameter found
scanning starts here

 Z = 1.5 + 1.5;     …2nd substitution
          ↑ ↑______end of scan
scanning results here
```

(c) Given

```
REPLACE A(X,Y) BY “Z = SIN(X)”;
```

the appearance

```
A(Y,X);
```

is expanded as follows:

```
Z = SIN(Y);          …1st substitution
↑       ↑______ parameter found
scanning starts here

Z = SIN(X);          …2nd substitution
        ↑
 scanning resumes here and finds parameter

Z =  SIN(Y);           …3rd substitution
     …expansion continues indefinitely.
```

29.3 IDENTIFIER GENERATION

Text strings involved in simple or parametric replacements cannot begin or end in the middle of an identifier. However, there exist two extensions of the REPLACE facility by whose means identifier names can be generated during the substitution process.

Identifier names can be generated both during the process of substitution of a simple replace name, and during the replacement of a parameter. The normal replace mechanism is used, except that the appearance of the parameter or of the replace name is delimited by ¢ signs.

Examples:

(a) Given

```
REPLACE X BY "ALPHA";
```

X BET is expanded as the product ALPHA BET

XBET is merely a single identifier, X not being recognized as a replace name

but

¢X¢BET is expanded generating the identifier ALPHABET.

(b) Given

```
REPLACE P(A) BY "Z = A + X";
```

Z¢P(1)¢5 expands into ZZ = 1 + X5 generating two identifiers, ZZ and X5.

(c) Given

```
REPLACE P(A) BY "Z = T¢A¢6 + X";
```

P(1) expands into Z = T16 + X generating T16 and Z¢P(1)¢5 expands into
ZZ = T16 + 5X generating T16, ZZ, and X5.

29.4 PRINTING OF REPLACE MACROS

In the output compiler listing, the macro replacement text is usually not printed unless a cent (¢) sign is placed around a REPLACE label in the statement referencing it. In that case, the original text which was replaced will be printed in the compiler listing instead of just the replace macro itself.

Examples:

a) Given

```
REPLACE X BY "A B C";
```

The source statement Y=X; will be printed in the listing as follows:

```
  |
M | Y = X;
```

The source statement Y=¢X¢; will be printed as follows:

```
  |
M | Y = A B C;
```

b) Given

```
| REPLACE ZZ BY "XYZ";
| REPLACE FOO(A,B) BY "A B";
|
```

The source statement C=FOO(ZZ,5) will be printed as follows:

```
M | C = FOO(ZZ,5);
  |
```

- Without the ¢ sign, all replace macros are underlined.

The source statement C=¢FOO(ZZ,5)¢; will be printed as follows:

```
M | C = ZZ 5;
  |
```

- With ¢ signs around the outer-most replace macro (FOO), only the expansion of FOO macro is printed.

The source statement C=FOO(¢ZZ¢,5); will be printed as follows:

```
E |         _  _
M | C = FOO( ZZ ,5);
  |
```

- With ¢ signs only around the inner level of the nested macro (ZZ), no macro expansions are printed. The ¢ sign is translated into a blank and the overpunch character "_" is printed on the E-line.

The source statement C=¢FOO(¢ZZ¢,5)¢; will be printed as follows:

```
M | C = XYZ 5;
  |
```

- With ¢ signs around all replace macros, macros are fully expanded.

29.5 INLINE FUNCTIONS

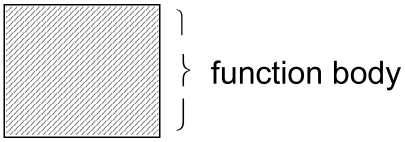
An inline function is a parameterless function block whose definition also constitutes its invocation. Hence, the block defining the function can actually appear embedded in an expression which forms part of some executable HAL/S statement. Since its definition is its sole invocation, nothing can reference it explicitly, and so it is given no name.

BASIC FORM

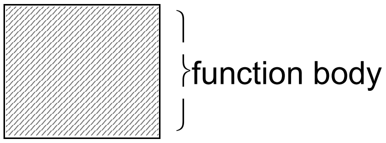
The form of an inline function block is exactly that of an ordinary function with no parameters, as described in Section 11.2 with one exception, namely that the opening statement does not bear a label indicating its name.

Examples:

```
FUNCTION SCALAR;
```



```
CLOSE;

FUNCTION CHARACTER(80);
```



```
CLOSE;
```

Local data may be declared within the function body, as in an ordinary function. The following constructs are however not allowed:

- procedure, function or task block definitions;
- any kind of I/O statement;
- invocations of any procedure or function;
- nested inline functions;
- SCHEDULE, WAIT, CANCEL, TERMINATE or UPDATE PRIORITY statements.

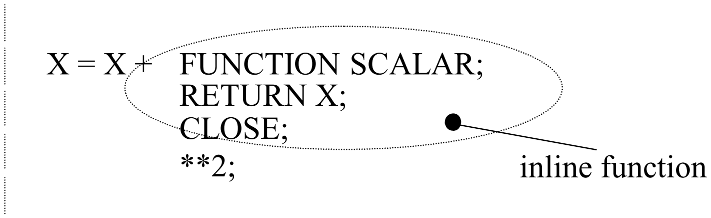
USE OF INLINE FUNCTIONS

Except as noted below, inline functions may appear wherever an ordinary function could be legally invoked. The exceptions are:

- exponent expressions;
- subscript expressions.

The following example illustrates how inline functions are used.

Examples:



```
X = X + FUNCTION SCALAR;
        RETURN X;
        CLOSE;
        **2;
```

This would give the same result as

```
X = X + X**2;
```

or as

```
F: FUNCTION SCALAR;
   RETURN X;
   CLOSE;
   .
   .
   .
  X = X +F**2;
```

The following usage is illegal:

```
X = 2** FUNCTION SCALAR;
           RETURN X;
        CLOSE;
           +1;
```

inline function

since the function appears as an exponent.

MOTIVATION FOR USE

Inline functions are potentially useful because, when generated by the substitution of replace macros, they increase the flexibility of parameterization. The fact that the appearance of a replace macro looks like a function invocation, and can expand into an inline function returning a value, is of particular interest.

Example:

Suppose that some algorithm requires finding the maximum element of a 1-dimensional array on a number of different occasions; and that in each occasion the array has a different size, and may be of integer or scalar type and of single or double precision.

On each occasion, it is supposed that the resultant index is used in some subsequent evaluation, so that a function invocation returning the result would be the most appropriate implementation. However, the differing attributes of the arrays preclude any regular HAL/S function being written to perform the operation.

The use of a replace macro which on each occasion expands into an inline function returning the resultant index is a feasible alternative.

The replace macro would be defined as follows:

```
REPLACE MAXIMUM(A) BY
  “FUNCTION INTEGER;
   DECLARE I INTEGER INITIAL(1) AUTOMATIC;
   DO FOR TEMPORARY J = 2 TO SIZE(A);
      IF A$J > A$I THEN I = J;
   END;
   RETURN I;
   CLOSE;”;
```

The text comprises an inline function returning the index of the maximum element of the array, which is represented by the parameter A.

Each appearance of replace macro MAXIMUM is accompanied by one argument, the desired array. The algorithm implemented in the inline function works for any 1-dimensional array of integer or scalar type, and of single or double precision.

Such an appearance might be:

```
 IF MAXIMUM(XTABLE) = 1 THEN
 WRITE(6) 'FIRST ELEMENT IS MAXIMUM';
```

And would expand into:

```
 IF FUNCTION INTEGER;
    DECLARE I INTEGER INITIAL(1) AUTOMATIC;
    DO FOR TEMPORARY J = 2 TO SIZE(XTABLE);
    IF XTABLE  > XTABLE  THEN I = J;
S            J         I
    END;
    RETURN I;
    CLOSE;
         = 1 THEN
    WRITE(6) `FIRST ELEMENT IS MAXIMUM';
```

The improved readability and ease of use of this implementation compared with direct in-line HAL/S code for each occurrence of the operation will readily be appreciated.

30.0 MANAGERIAL CONTROL OF ACCESS TO DATA AND CODE

In a large software project numerous compilation units each performing independent functions may be brought together in a single module at run time for execution. Commonly, these compilation units will require access to shared data contained in compools. If several teams of programmers cooperate in the production of this software, it is desirable to be able to place managerial restrictions upon who can access "sensitive" shared data. It may also be desirable to place managerial restrictions upon who can invoke "sensitive" programs or comsubs.

30.1 ACCESS CONTROL IN HAL/S

HAL/S contains a method for specifying which data, and which programs and comsubs are to be protected by managerial restriction.

If at compile time, a compilation unit written by an unauthorized programmer is found by a HAL/S compiler to contain:

- a construct causing modification of protected data in a compool; or
- a construct referring to any entity in a protected compool; or
- a construct invoking a protected program, or external procedure or function;

then an error will be signaled and the compiler will produce no executable object module.

To circumvent this error, the programmer's authorization to use protected data or code blocks must be signaled to the compiler, and in addition, the program itself must state that the data or code blocks are under protection. This latter restriction insures that an authorized programmer is aware that he is accessing protected data or code.

The mechanism by which the HAL/S compiler determines whether a given programmer is authorized to use a particular protected data item or code block is implementation dependent[48].

30.2 ACCESSING PROTECTED COMPOOL DATA

Compool data items which are protected must be declared using the keyword ACCESS. The following examples illustrate the position of the keyword in a declaration.

Examples:

```
POOL: COMPOOL;
   DECLARE A SCALAR DOUBLE ACCESS;
   DECLARE B ARRAY(1000) BIT(8) INITIAL(FALSE) ACCESS;
   DECLARE V VECTOR(3);
   DECLARE ZX ARRAY(100) ACCESS INITIAL(0) REMOTE;
CLOSE POOL;
```

[48] See appropriate User's Manual.

The keyword appears also in the declarations of the corresponding compool template.

An authorized programmer will be allowed to modify protected compool data provided it is declared as described above.

For more precise rules for locating the keyword in a declaration see Spec./4.5.

30.3 PROTECTION OF AN ENTIRE COMPOOL

It is possible to place an entire compool under protection in addition to placing its declared data items individually under protection. Every data item, structure template and replace name defined within such a compool is protected against <u>any</u> unauthorized use. A protected compool must contain the keyword ACCESS or part of its definition, the keyword being placed at the end of its opening statement.

Example:

```
POOL:COMPOOL ACCESS;
     REPLACE X BY”1000”;
     DECLARE S SCALAR,
             V VECTOR(3),
             M MATRIX(3,3),
             A ARRAY(1000) SCALAR;
     DECLARE C CHARACTER(80);
     STRUCTURE Q:
      1 QS SCALAR,
      1 QI INTEGER;
     DECLARE Z Q-STRUCTURE;
CLOSE POOL;
```

- Data items S, V, M, A, Z and C are protected against unauthorized modification or reference. Replace name X and structure template Q are also protected.

The template corresponding to a protected compool must also possess the keyword ACCESS,

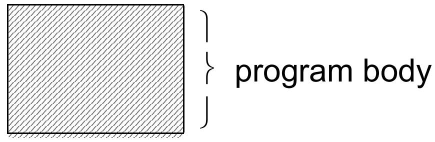
Any data item in a protected compool may itself also be individually protected, the keyword ACCESS appearing in its declaration as described before. A user requiring to modify the data item must be authorized in respect to the compool <u>and</u> to the individual data item itself.

30.4 ACCESSING PROTECTED PROGRAMS AND COMSUBS

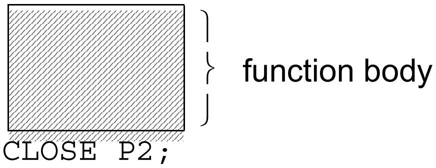
Programs, external procedures, and external functions which are protected must contain the keyword ACCESS as part of their block definition. The keyword is placed at the end of their opening statement.

Examples:

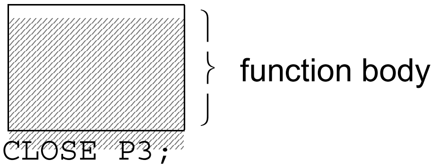
```
P1: PROGRAM ACCESS;
```



```
CLOSE P1;

P2: PROCEDURE(A) ASSIGN(B) ACCESS;
```



```
CLOSE P2;

P3: FUNCTION(I) CHARACTER(80) ACCESS;
```



```
CLOSE P3;
```

The corresponding block templates must also possess the keyword ACCESS in the corresponding place.

An authorized programmer will be allowed to invoke or otherwise reference the protected block provided it is defined as described above.

For a complete description of using the ACCESS keyword on the opening statements of programs and comsubs;
See Spec./3.7.1-3.7.3.

This page intentionally left blank.

31.0 INTERFACES WITH NON-HAL/S CODE

The HAL/S language has been expressly designed so that nearly all flight software can be written in it. However, it is realized that sometimes legitimate reasons exist for implementing some segments of the software in languages other than HAL/S (notably in assembly language, for example). Hence, HAL/S must be able to provide interfaces to the resulting code.

Two approaches are possible in interfacing with non-HAL/S code.

- Sometimes the purpose of the non-HAL/S software segments is to provide a set of "utility functions" which are standard for a particular implementation of the HAL/S compiler system. In such circumstances it is desirable for the HAL/S compiler to recognize references to these utility functions automatically, as if they were in some sense extensions to its known list of HAL/S built-in functions. This is done by defining the segments to be "%macros", which may be referenced in appropriate contexts in the HAL/S software.
- Where the use of the non-HAL/S segments is specific and localized, rather than global, the alternative is to create the segments as externally defined procedures or functions. The linkage between the segments and the HAL/S software may either conform to the standard HAL/S conventions for a particular implementation, or may take one of a number of alternative forms which have been predefined in the implementation.

31.1 %MACROS

The "%macros" defined in any implementation of a HAL/S compiler system, effectively constitute implementation dependent extensions to the list of HAL/S built-in functions. Their reference may cause a HAL/S compiler either to emit in-line object code for their execution, or to emit linkages to external routines, depending on the particular macro and implementation.

%macros fall into two classes:

- TYPED %MACROS, which are known by an implementation to be of a particular HAL/S data type, and which are invoked as if they were built-in functions, returning a value of the specified type;
- TYPELESS %MACROS, which are known by an implementation not to be of any HAL/S data type, and which are invoked by executing a specific "%macro call" statement, as if they were procedures.

Either class of invocation uses the same construct to reference the %macro.

FORM OF %MACRO REFERENCE

The construct referencing any %macro (typed or typeless) specifies the name of the %macro, and a list of arguments to be transmitted. Its form is as shown below:

%name(arg^1,..... arg^n)

1. *%name* is the name of the %macro. The HAL/S compiler knows it as a %macro name because its first character is always the % symbol.
2. *arg*^{1},..... *arg*^{n} is a list of arguments by which values may be passed potentially both into and back from the %macro. The number of arguments and their form is dependent on the functional specification of the %macro.
3. The entire parenthesized argument list may be omitted.

Examples:

```
%NAMECOPY(A,B)
%SVC(5)
```

INVOKING TYPED %MACROS

A typed %macro may possess any one of the HAL/S data types given below:

```
INTEGER    CHARACTER
SCALAR     BIT STRING (including BOOLEAN)
VECTOR     STRUCTURE
MATRIX
```

Such a %macro is expected to return one or more values consistent with its type, and must therefore only appear in contexts legal for that data type.

Examples:

If S is a scalar, V is a 3-vector and %UCALC is a 3-vector %macro, then

```
|
| S = S + V.%UCALC(5);
|
```

contains a legal invocation, but

```
|
| DO WHILE %UCALC(5);
|
```

contains an illegal invocation because %UCALC does not return a bit string.

INVOKING TYPELESS %MACROS

Typeless %macros do not return values, except through their argument lists. They are invoked by ”%macro call” statements, whose form is shown below:

```
%macro-reference;
```

1. *%macro-reference* is a %macro reference as described above.

Examples:

```
%SWAP;
IF A = B THEN %EXIT(FALSE);
```

Currently-defined %macros are given in Spec./Appendix I.

31.2 REFERENCING NON-HAL/S PROCEDURES AND FUNCTIONS

Non-HAL/S code segments can be designed so that they can be invoked from HAL/S software as if they were HAL/S procedures or functions. In designing such segments a choice of linkages to the HAL/S software is available:

- the standard HAL/S linkage for the particular implementation;
- one of a number of alternate linkages predefined for the particular implementation.

How these non-HAL/S code segments are indicated as such in the HAL/S software invoking them, depends on the forms of linkage chosen.

STANDARD HAL/S LINKAGE

Standard HAL/S linkage to an external procedure or function will be assumed by a HAL/S compiler if it believes the external block to have been written in HAL/S. It will believe this if a suitable template for the external block is included in the compilation unit invoking it. External procedure and function blocks and block templates have been described in Section 15.

The relationship between the block name appearing in the template and the actual name of the object module of the non-HAL/S code segment is implementation dependent[48].

[48] The relationship must be correct, otherwise the “link editing” of object modules will be unsuccessful. See appropriate User's Manual for naming conventions.

Example:

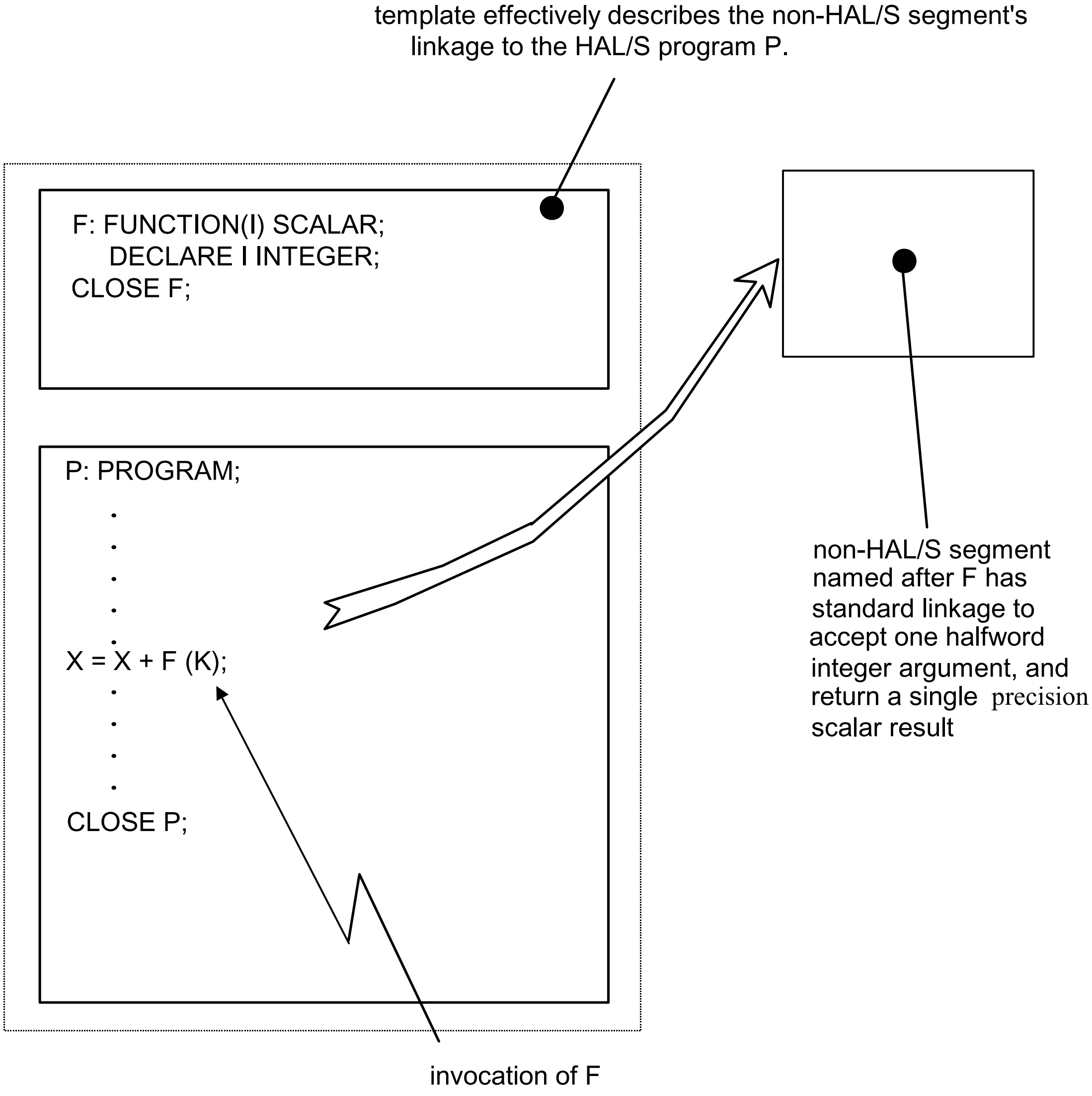

template effectively describes the non-HAL/S segment's linkage to the HAL/S program P.
F: FUNCTION(I) SCALAR;
DECLARE I INTEGER;
CLOSE F;
P: PROGRAM;
X = X + F (K);
CLOSE P;
non-HAL/S segment named after F has standard linkage to accept one halfword integer argument, and return a single precision scalar result
invocation of F

Figure 31-1

If a particular implementation of the HAL/S compiler system includes a software management scheme for insuring the consistency of templates and object modules, as alluded to in Section 15.1, it may not be permissible to use this method for invoking a non-HAL/S code segment (In such cases it would be natural to let one of the predefined alternate linkage forms be the standard linkage itself).

ALTERNATE LINKAGES

An alternate linkage to an external procedure or function is specified through the appearance of a declaration of the procedure or function in the compilation unit invoking the non-HAL/S code segment.

As before, the relationship of the declared block name and the actual name of the object module of the segment is implementation dependent.

- PROCEDURE FORM

The basic form of declaration for a procedure is:

```
DECLARE name PROCEDURE NONHAL(n);
```

1. *name* is the HAL/S identifier name by which the segment is known.
2. The unsigned integer n specifies which alternate linkage is to be assumed[†].

[†] The linkage corresponding to each value of n in a given implementation is given in the appropriate User's Manual.

Examples:

```
DECLARE P1 PROCEDURE NONHAL(3);
```

The declaration can be combined with any other kinds of declaration -

```
 DECLARE S SCALAR,
         P1 PROCEDURE NONHAL(2),
         C CHARACTER(80);
```

Its attributes can be factored thus:

```
DECLARE PROCEDURE NONHAL(1),P1,P2,P3;
```

Procedure invocations compatible with declarations of the above form may contain any number of input and assign arguments, including none. It is possible that in some linkage forms the number of arguments could vary from invocation to invocation. Implementation dependent restrictions upon the arguments may exist[49].

[49] See appropriate User's Manual.

- FUNCTION FORM

The basic form of declaration for a function is:

```
DECLARE name FUNCTION attributes NONHAL(n);
```

1. *name* is the HAL/S identifier name by which the segment is known.
2. *attributes* specify the type and precision of the function, as in the opening statement of a function definition (see Section 11.2).
3. The unsigned integer n specifies which alternate linkage is to be assumed.[†]

[†] The linkage corresponding to each value of n in a given implementation is given in the appropriate User's Manual.

Examples:

```
DECLARE F1 FUNCTION SCALAR NONHAL(1);
DECLARE F2 FUNCTION MATRIX(3,3) DOUBLE NONHAL(2);
```

As before several declarations can be combined with factoring:

```
DECLARE NONHAL(3), F3 FUNCTION INTEGER, P PROCEDURE;
```

Function invocations compatible with declarations of the above form are subject to the following:

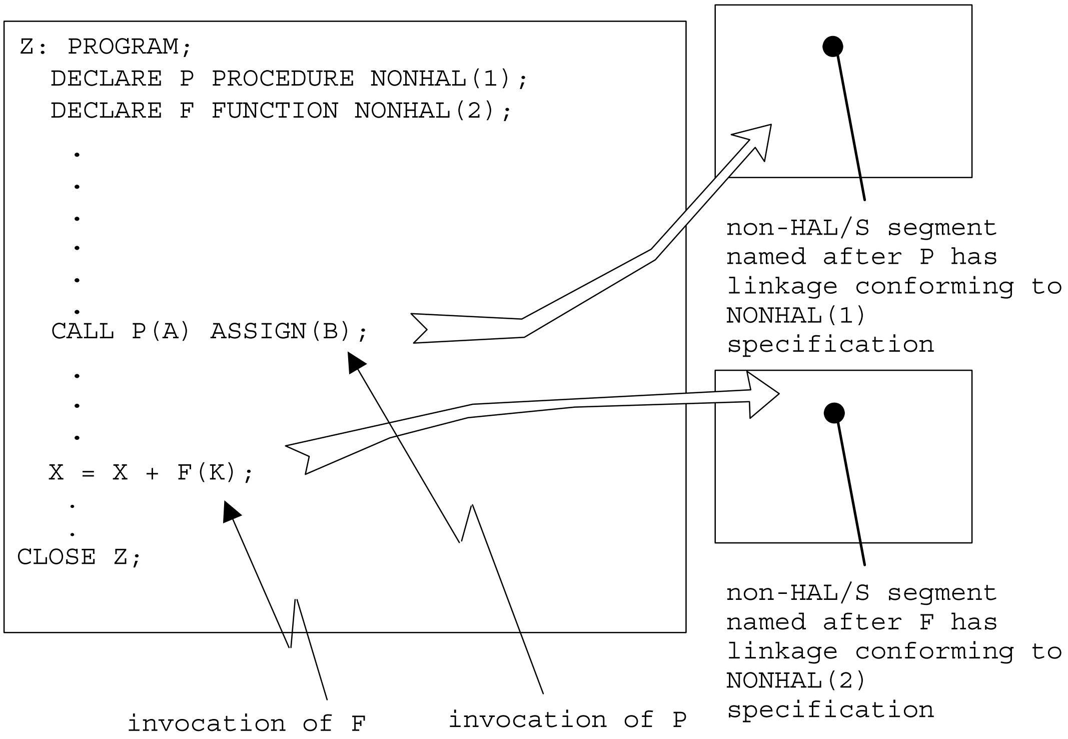
- they must possess at least one argument;
- arrayed arguments are passed in a single invocation rather than causing multiple elemental invocations as described in Section 20.6.

Other implementation dependent restrictions upon the arguments may exist.[50]

The following example shows the invocation of non-HAL/S code segments by the alternate linkage method:

[50] See appropriate User’s Manual.

Example:



```
Z: PROGRAM;
  DECLARE P PROCEDURE NONHAL(1);
  DECLARE F FUNCTION NONHAL(2);
    .
    .
    .
    .
    .
    .
  CALL P(A) ASSIGN(B);
    .
    .
    .
  X = X + F(K);
   .
   .
CLOSE Z;
```

non-HAL/S segment named after P has linkage conforming to NONHAL(1) specification
non-HAL/S segment named after F has linkage conforming to NONHAL(2) specification
invocation of F
invocation of P

Figure 31-2

This page intentionally left blank.

Appendix A STANDARD CONVERSION FORMATS

In relatively limited circumstances HAL/S allows conversion between scalar, integer, bit and character types. The following rules govern such conversions.

CONVERSIONS TO INTEGER TYPE:

- A bit type is converted to integer type by regarding it as the bit pattern of a signed integer of the desired precision (halfword or fullword). Left padding with binary zeros, or left truncation may occur.
- A scalar type is converted to integer type by rounding to the nearest whole number. Overflow errors may occur if the absolute value of the scalar type is too large to be represented as an integer of the desired precision.
- A character type is convertible to integer type only if its value represents a signed whole number (e.g., '-604') otherwise an error condition occurs. An error condition also occurs if the whole number is too large to be represented as an integer of the desired precision.

CONVERSIONS TO SCALAR TYPE:

- An integer type is converted directly to scalar form. Depending on the implementation, and the precisions, some decimal places of accuracy may be lost during conversion.
- A bit type is converted to scalar type by first converting it to double precision integer type according to the rule previously given, and then applying the integer to scalar conversion.
- A character type is convertible to scalar type only if its value represents a legal scalar- or integer-valued literal (e.g., '-1.5E-7'). See HAL/S Language Specification, Section 2.3.4 for details of arithmetic literals. Other values cause error conditions to arise.

CONVERSIONS TO BIT TYPE:

- An integer is converted to a bit string of maximum length (for double precision), or a halfword bit string (for single precision) in an unsubscripted BIT conversion function. The value is the bit pattern of the integer.
- A scalar type is first converted to double precision integer type according to the rule already given, and the integer to bit conversion rules are then applied.
- A character type is convertible to bit type only if its value is a string of '1's and '0's, and blanks, (but not all blanks), otherwise an error condition arises. The result of the conversion is always a maximum length bit string, irrespective of the argument type. If the argument has more then N bits, where N is the maximum allowable length of a bit operand, then only the N right-most are used. If the argument has fewer than N bits, the string is padded on the left with binary zeros.

CONVERSION TO CHARACTER TYPE:

- An integer type is converted to the representation:

  dddd (positive)

  -dddd (negative)

  where dddd represents an arbitrary number of decimal digits. Leading zeros are suppressed yielding a variable length result.

- A scalar type is converted to the representation:

  ƀd.ddddE±dd (positive)

  -d.ddddE±dd (negative)

  (except scalar 0 is converted to 0.0).

  The number of decimal digits d in the fractional part and exponent are implementation and precision dependent. The digit to the left of the decimal point is nonzero. There are no imbedded blanks. Leading zeros in the exponent are not suppressed. The representation includes a leading blank (ƀ) if the scalar is positive. In all cases, the result is fixed in length.

- A bit type is converted to a character string of '1's and '0's corresponding to the binary representation of the bit string argument.

Appendix B BUILT-IN FUNCTIONS

HAL/S typically supports the following set of built-in functions. Minor variations may arise between implementations.

ARITHMETIC FUNCTIONS

- arguments may be INTEGER or SCALAR types
- in functions with one argument, result type matches argument type (except as specifically noted)
- in functions with two arguments, unless specifically specified, result type is scalar if either or both arguments are scalar; otherwise the result type is integer
- arrayed arguments cause multiple invocations of the function, one for each array element - arrayness of arrayed arguments must match

Name, Arguments	Comments
ABS(α)	$\lvert \alpha \rvert$
CEILING(α)	smallest integer $\geq \alpha$
DIV(α,β)	integer division α/β (arguments rounded to integers)
FLOOR(α)	largest integer $\leq \alpha$
MIDVAL(α,β,γ)	the value of the argument which is algebraically between the other two. If two or more arguments are equal, the multiple value is returned. Result is always scalar.
MOD(α,β)	α MOD β
ODD(α)	$\left.\begin{array}{l}\text{TRUE 1 if } \alpha \text{ odd}\\ \text{FALSE 0 if } \alpha \text{ even}\end{array}\right\}$ result is BOOLEAN
REMAINDER(α,β)	signed remainder of integer division α/β (argument rounded to integer)
ROUND(α)	nearest integer to α
SIGN(α)	+1 $\alpha \geq 0$ -1 $\alpha < 0$
SIGNUM(α)	+1 $\alpha > 0$ 0 $\alpha = 0$ -1 $\alpha < 0$
TRUNCATE(α)	largest integer $\leq \lvert \alpha \rvert$ times SIGNUM (integer (α))

ALGEBRAIC FUNCTIONS

- arguments may be integer or scalar types - conversion to scalar occurs with integer arguments
- result type is scalar
- arrayed arguments cause multiple invocations of the function, one for each array element
- angular values are supplied or delivered in radians

Name, Arguments	Comments
ARCCOS(α)	$\cos^{-1}\alpha$, $\lvert\alpha\rvert \le 1$ This returns an angular value.
ARCCOSH(α)	$\cosh^{-1}\alpha \quad \ge 1$
ARCSIN(α)	$\sin^{-1}\alpha$, $\lvert\alpha\rvert \le 1$ This returns an angular value.
ARCSINH(α)	$\sinh^{-1}\alpha$
ARCTAN2(α,β)	$-\pi < \tan^{-1}(\alpha/\beta) \le \pi$ Proper Quadrant if: $\left.\begin{array}{l}\alpha = k\sin\theta \\ \beta = k\cos\theta\end{array}\right\\} \quad k > 0$ This returns an angular value.
ARCTAN(α)	$\tan^{-1}\alpha$
ARCTANH(α)	$\tanh^{-1}\alpha$, $\lvert\alpha\rvert < 1$
COS(α)	$\cos\alpha$ This takes an angular value
COSH(α)	$\cosh\alpha$
EXP(α)	e^{α}
LOG(α)	$\log_e\alpha$, $\alpha > 0$
SIN(α)	$\sin\alpha$ This takes an angular value.
SINH(α)	$\sinh\alpha$
SQRT(α)	$\sqrt{\alpha} \qquad \alpha \ge 0$
TAN(α)	$\tan\alpha$
TANH(α)	$\tanh\alpha$

VECTOR-MATRIX FUNCTIONS	
• arguments are vector or matrix types as indicated • result types are as implied by mathematical operation • arrayed arguments cause multiple invocation of the function, one for each array element	
Name, Arguments	**Comments**
ABVAL(α)	length of vector α.
DET(α)	determinant of square matrix α.
INVERSE(α)	inverse of a nonsingular square matrix α.
TRACE(α)	sum of diagonal elements of square matrix α
TRANSPOSE(α)	transpose of matrix α.
UNIT(α)	unit vector in same direction as vector α.

MISCELLANEOUS FUNCTIONS		
• arguments are as indicated; if none are indicated the function has no arguments • result type is as indicated		
Name, Arguments	**Result Type**	**Comments**
CLOCKTIME	scalar	returns time of day
DATE	integer	returns date (implementation dependent format)
ERRGRP	integer	returns group number of last error detected, or zero
ERRNUM	integer	returns number of last error detected, or zero
PRIO	integer	returns priority of process calling function
RANDOM	scalar	returns random number from rectangular distribution over range 0-1
RANDOMG	scalar	returns random number from Gaussian distribution mean zero, variance one.
RUNTIME	scalar	returns Real Time Executive clock time (Section 8).
NEXTIME (<label>)	scalar	<label> is the name of a program or task. The value returned is determined as follows: a) If the specified process was scheduled with the IN or AT phrase, and has not yet begun execution, then the value is the time it will begin execution. b) Otherwise, the value is equal to the current time (RUNTIME function).

SHL(α,β)	Same as α	α may be integer or scalar type. β may be integer type or scalar type. If α is integer type, the result is an integer whose internal binary representation is that of α shifted left by β bit locations. The signed nature of the integer α is taken into account in an implementation dependent manner which depends upon the number system and word size of the target computer. Scalar types will be converted to integer types prior to shifting. If β is a literal or constant then it is illegal for it to be outside the range of (1,63). Otherwise, only the 6 right-most bits of the value are used, restricting the range to (0,63). Arrayed arguments produce multiple invocations of the function, one for each array element arrayness of arrayed arguments must match.
SHR(α,β)	Same as α	α may be integer or scalar type. β may be integer or scalar type. Results are as defined for the SHL function except that all shifting occurs to the right. The SHR is an arithmetic shift (sign bit is propogated). Arrayed arguments produce multiple invocations of the function, one for each array element - arrayness of arrayed arguments must match.

CHARACTER FUNCTIONS

- first argument is character type - second argument is as indicated (any argument indicated as character type may also be integer or scalar, whereupon conversion to character type is implicitly assumed)
- result type is as indicated
- arrayed arguments produce multiple invocations of the function, one for each array element - arraynesses of arrayed arguments must match

Name, Arguments	Result Type	Comments
INDEX (α,β)	integer	β is character type - if string β appears in string α, index pointing to the first character of β is returned; otherwise zero is returned
LENGTH(α)	integer	returns length of character string
LJUST(α,β)	character	β is integer type - string α is expanded to length β by padding on the right with blanks β ≥ length (α).
RJUST(α,β)	character	β is integer type - string α is expanded to length β by padding on the left with blanks β ≥ length (α)
TRIM(α)	character	leading and trailing blanks are stripped from α

BIT FUNCTIONS

- arguments are bit type
- result is bit type
- arrayed arguments produce multiple invocations of the function, one for each array element - arrayness of arrayed arguments must match

Name, Arguments	Result Type	Comments
XOR(α,β)	bit	Result is Exclusive OR of α and β. Length of result is length of longer argument. Shorter argument is left padded with binary zeros to length of longer argument.

<table>
<tr><th colspan="2">ARRAY FUNCTIONS</th></tr>
<tr><td colspan="2">• arguments are n-dimensional arrays where n is arbitrary.
• arguments are integer or scalar.
• result type matches argument type and is unarrayed</td></tr>
<tr><th>Name, Parameters</th><th>Comments</th></tr>
<tr><td>MAX(α)</td><td>maximum of all element of α.</td></tr>
<tr><td>MIN(α)</td><td>minimum of all element of α.</td></tr>
<tr><td>PROD(α)</td><td>product of all elements of α.</td></tr>
<tr><td>SUM(α)</td><td>sum of all elements of α.</td></tr>
<tr><th colspan="2">SIZE FUNCTION</th></tr>
<tr><th>Name, Argument</th><th>Comments</th></tr>
<tr><td>SIZE(α)</td><td>One of the following must hold:
• α is an unsubscripted arrayed variable with a one-dimensional array specification-function returns length of array.
• α is an unsubscripted major structure with a multiple copy specification-function returns number of copies.
• α is an unsubscripted structure terminal with a one-dimensional array specification-function returns length of array.
Result is of integer type</td></tr>
</table>

Appendix C ORDERING OF DATA ELEMENTS

There are numerous kinds of operation in the HAL/S language which require operands with multiple components, array elements, and structure copies to be unraveled into a linear string of data elements. The reverse process of "reraveling" a linear string of data elements into components, array elements, and structure copies also occurs. Two instances of these processes are in I/O and in conversion functions. The former process is also involved in relating initial lists to the data items they initialize.

The standard order in which this unraveling and reraveling takes place is called the "natural sequence". By applying the following rules in the order they are stated, the natural sequence of unraveling is obtained. By applying the rules in reverse order, and replacing "unraveled" by "reraveled", the natural sequence for reraveling is obtained.

RULES FOR STRUCTURES:

1. If the operand is a structure with multiple copies, each copy is unraveled in turn, in order of increasing index. If the operand is a minor structure node in a multiple-copy structure, then the copy of the minor structure in each structure copy is unraveled in turn in order of increasing index.
2. The method of unraveling a copy is as follows. Each structure terminal which is part of the given structure operand is unraveled in turn. The order taken is the order of appearance of the terminals in the structure template.
3. Each structure terminal is unraveled according to the Rules given below.

Example:

```
STRUCTURE A:
   1 B,
     2 C SCALAR,
     2 D VECTOR(3),
   1 E INTEGER;
DECLARE A A-STRUCTURE(3);
```

order of unraveling of B is $B_{i'}$ i = 1, 2, 3

order of unraveling of each B_i is $C_{i'}$ D_i

RULES FOR OTHER OPERANDS:

1. An operand of any type (integer, scalar, vector, matrix, bit string, character, or event) may possess arrayness as described in Section 20.1. Each dimension of arrayness, starting from the leftmost is unraveled in turn, in order of increasing index.
2. Integer, scalar, bit string, character, and event types are considered for unraveling purposes as having only one data element.
3. Vector types are unraveled component by component, in order of increasing index.
4. Matrix types are unraveled row by row, in order of increasing index. The components of each row are unraveled in turn in order of increasing index.

Example:

```
DECLARE V ARRAY(2,2) VECTOR(3);
```

- order of unraveling of V is $V_{i,*:*}$, i = 1, 2
- order of unraveling of each $V_{i,*:*}$ is $V_{i,j:*}$,

  j = 1, 2
- order of unraveling of each $V_{i,j:*}$ is $V_{i,j:k}$,

  k = 1, 2, 3

Appendix D COMPILE-TIME COMPUTATIONS

References have been made in the Guide to the fact that in certain restricted cases, expressions which are computable at compile time may be substituted for literal values. Among the constructs allowing such substitutions are:

- initial lists in declarations;
- specification of dimensions or lengths in declarations;
- subscripting.

Only the restricted forms of integer, scalar, bit string and character expressions to be described can be used in such contexts. These forms are guaranteed to be computable at compile time in any implementation.

ARITHMETIC EXPRESSIONS

1. Expressions of integer and scalar type only can be computable at compile time.
2. The operators of such expressions are limited to the following:

```
+
-
(blank) - multiply
/
** - exponentiation
```

3. The operands of such expressions may either be literals or unarrayed unsubscripted data items of integer or scalar type. Such data items must previously have been declared, and initialized using the CONSTANT form.
4. The following built-in functions are also legal:

```
SIN   EXP   DATE
COS   LOG   CLOCKTIME
TAN   SQRT
```

DATE and CLOCKTIME are only computed at compile time if they appear in an initial list.

BIT STRING EXPRESSIONS

1. The operators which may appear in bit string expressions computable at compile time are limited to the following set:

```
¬
&
|
```

2. The operands of such expressions must be either literals or unarrayed unsubscripted data items of bit string type. Such data items must previously have been declared, and initialized using the CONSTANT form.

CHARACTER EXPRESSIONS

1. The catenation operator (||) only may appear in character expressions computable at compile time.
2. The operands of such expressions must be either literals, arithmetic expressions computable at compile time, or unarrayed unsubscripted data items of character type. Such data items must previously have been declared, and initialized using the CONSTANT form.

In some implementations, additional forms may also be computed at compile time. They will not, however, be regarded as legal in contexts where compile time computability is enforced by the rules of the HAL/S language.

Appendix E HAL/S KEYWORDS

The following table of keywords excludes built-in functions and %-macro names.

ACCESS
AFTER
ALIGNED
AND
ARRAY
ASSIGN
AT
AUTOMATIC

BIN
BIT
BOOLEAN
BY

CALL
CANCEL
CASE
CAT
CHAR
CHARACTER
CLOSE
COLUMN
COMPOOL
CONSTANT

DEC
DECLARE
DENSE
DEPENDENT
DO
DOUBLE

ELSE
END
EQUATE
ERROR
EVENT
EVERY
EXCLUSIVE
EXIT
EXTERNAL

FALSE
FILE
FOR
FUNCTION

GO

HEX

IF
IGNORE
IN
INITIAL
INTEGER

LATCHED
LINE
LOCK

MATRIX

NAME
NONHAL
NOT
NULL

OCT
OFF
ON
OR

PAGE
PRIORITY
PROCEDURE
PROGRAM

READ
READALL
REENTRANT
REMOTE
REPEAT
REPLACE
RESET
RETURN
RIGID

SCALAR
SCHEDULE
SEND
SET
SIGNAL
SINGLE
SKIP
STATIC
STRUCTURE
SUBBIT
SYSTEM

TAB
TASK
TEMPORARY
TERMINATE
THEN
TO
TRUE

UNTIL
UPDATE

VECTOR

WAIT
WHILE
WRITE

Appendix F STANDARD INPUT/OUTPUT FORMATS

Corresponding to each data type there exists a "standard external format" for the representation of its values on sequential I/O files. In any implementation the standard external format on output is fixed, on input the user has a certain flexibility in the format he can use.

OUTPUT FORMATS

1. Integer Type:

- The value of an integer is represented by a string of decimal digits, preceded if it is negative by a - sign. Leading zeros are suppressed.
- The string of digits is right justified in a field of fixed width. The width depends on the implementation, and on the precision of the integer.

1. Scalar Type:

- If the value of a scalar is positive it is represented by

  ± ƀd.dddddddEdd

  where d represents a decimal digit. One non-zero digits appears before the decimal point. The numbers of digits in the fractional part and exponent are fixed, and depend on the implementation and the precision of the scalar. Leading zeros in the exponent are not suppressed. The representation includes a leading blank (ƀ) .
- A negative value has the same form except that a - sign precedes the first decimal digit.
- If the value is exactly zero, it is represented as 0.0.
- The representation of a scalar is contained in a field of fixed width. The width is dependent on the implementation and the precision of the scalar. Justification is such that the decimal point occupies a fixed, precision dependent position in the field.

3. Bit Type (including BOOLEAN):

- There are two different representation of values of bit variables.
- The first representation consists of a string of binary digits corresponding to the bit variable. Leading binary zeros are not suppressed. The field width is equal to the number of binary digits in the string plus an inserted blank following every fourth digit (to enhance readability). This form is not compatible with the READ input (see Section 10.1.1).
- In the alternate representation, the string of binary digits plus inserted blanks is enclosed in the apostrophes. The field width is equal to the total of the number of digits, blanks, and two apostrophes.

3. Character Type

- There are two different representations of values of character variables.
- The first representation merely consists of the string of characters comprising the value. The field width is equal to the number of characters in the string. This representation is not compatible with READ (see Section 10.1.1).

- In the alternate representation, the string of characters is enclosed in apostrophes, and all internal apostrophes are converted to apostrophe pairs. The field width is equal to the total number of characters in the string, including added apostrophes.

NOTE: The two alternate representations for bit and character types occur on paged and unpaged output respectively.

INPUT FORMATS

1. Scalar and Integer Types:

- There are three basic representations; whole-number, floating-point, and fraction.
- The whole number representation consists of a string of decimal digits preceded by an optional '-' sign. The maximum number of digits allowed is implementation dependent. Conversion to mantissa-exponent form takes place for scalar types.
- The floating-point representation is either

$$\text{ddd.dddd}$$

or

$$\text{dddd.dddd} \begin{Bmatrix} \text{E} \\ \text{B} \\ \text{H} \end{Bmatrix} \pm \text{dd}$$

- where d is a decimal digit. Any number of digits is allowed in the mantissa to an implementation dependent maximum. The decimal point may appear in any position. E, B, and H represent the exponent digits to be powers of 10, 2, and 16 respectively. A choice of one is indicated. The maximum number of digits in the exponent is implementation dependent. For bit and integer types, the representation is rounded to the nearest integral value. For bit types the binary representation of the result is taken.
- The floating-point representation may be prefixed by + or - signs to indicate the sign of the value. Without such prefix the value is positive.

2. Character Type:

- The representation of character type is a string of characters from the HAL/S extended set enclosed in apostrophes. The number of characters may vary between zero (a "null string") and an implementation dependent maximum. Within the string apostrophes must be represented by an apostrophe pair.

3. Bit Type:

- The representation of bit type is a string of '1's and '0's enclosed in apostrophes. Imbedded blanks are ignored. The number of digits may vary between one and an implementation maximum.

Appendix G Change History

Revision	Release	Date	Change Authority	Sections Changed
03	11.0	12/16/75		
04	11.1	06/11/76		
05	19.9	12/81		
06	23.1	02/04/91		Title page, p. 12-8
07	24.0	03/30/92		Title page, p. 12-8
08	27.0/11.0	06/21/96		Total reprint.
09	27.1/11.1	07/01/96		- p. 28-20
10	28.0/12.0	08/22/97		Total Reprint to Bring to HAL Documentation Standards and Reformat for HTML compatibility.
			CR12712	26.5 - Deleted 32.0 - Deleted App. A - pp. A-1, A-2 App. B - pp. B-1, B-2, B-3, B-4, B-5 App. E - p. E-1 App. F - pp. F-1, F-2 Index - pp. Index-3, Index-5
			DR109035	29.3 - p. 29-6 29.4 - pp. 29-6, 29-7
			DR109048	4.3 - p. 4-7
				Table of Contents - pp. vii, x, xv 12.4 - p. 12-12 14.0 - Deleted Index - p. Index-5

This page intentionally left blank.

Index

This is the Last Page of this Document

DRD NUMBER: 1.4.3.8-b DOCUMENT NUMBER: USA003087

TITLE: HAL/S Programmer's Guide

NASA-JSC

*BV	N. Moses
MS4	D. Stamper
EV111	EV Library (D. Wall)

USA-Houston

*USH-121G	SFOC Technical Library
USH-634G	Abel Puente
USH-64A6X	L.W. Wingo
USH-633L	Anita Senviel
USH-633L	Benjamin L. Peterson
USH-633L	Cory L. Driskill
USH-633L	Judy M. Hardin
USH-633L	Mark E. Lading
USH-633L	Quinn L. Larson
USH-633L	James T. Tidwell
USH-633L	Vicente Aguilar
USH-633L	Betty A. Pages
USH-633L	Jeremy C. Battan
USH-633L	George H. Ashworth
USH-634L	Mark Caronna
USH-634L	Burk J. Royer
*USH-635L	Joy C. King
USH-635L	Ling J. Kuo
USH-635L	Trang K. Nguyen
USH-635L	Billy L. Pate
USH-635L	Karen H. Pham
*USH-635L	Dan A. Strauss
USH-635L	Pete Koester
USH-632L	Renne Siewers
*USH-635L	Barbara Whitfield (2)

Boeing

HS1-40	B. Frere

blake.a.frere@boeing.com

* Denotes hard copy

Submit NASA distribution changes, including initiator's name and phone number, to JSC Data Management/BV or call 281-244-8506. Submit USA distribution changes to USA Data Management/USH-121E or via e-mail to *usadm@usa-spaceops.com*. Most documents are available electronically via USA Intranet Web (*usa1.unitedspacealliance.com*), Space Flight Operations Contract (SFOC), SFOC Data and Deliverables.

Indicates hardcopy

11/23/2005 7:08 AM