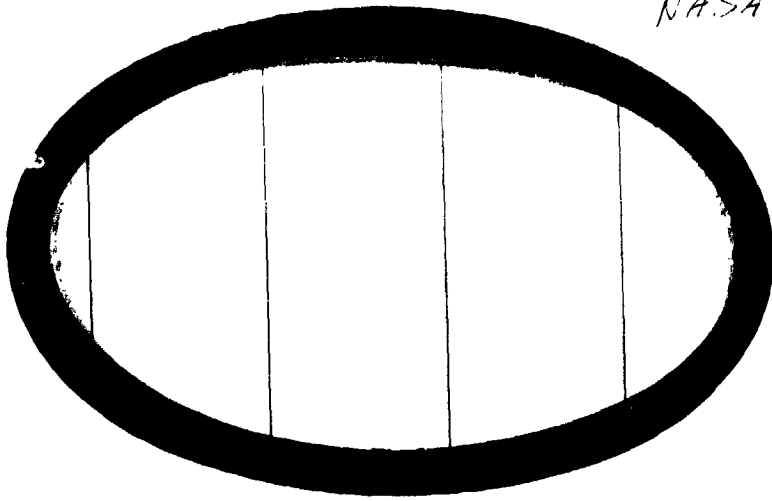


NASA CR-134371



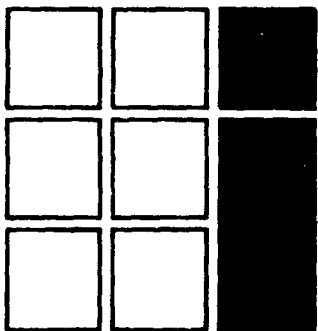
(NASA-CR-134371) HAL/S PROGRAMMER'S GUIDE  
(Intermetrics, Inc.) 190 p HC \$12.50

CSCL 09B

N74-29536

G3/08

Unclas  
45357



**INTERMETRICS**

HAL/S  
PROGRAMMER'S  
GUIDE

IR-63-3

3 July 1974

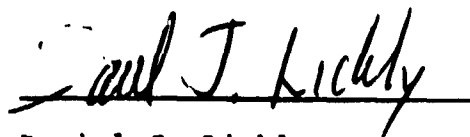
Prepared by:

P.M. Newbold  
R.L. Hotz

Typescript:

V.L. Cripps

Approved:



Daniel J. Lickly  
HAL Language/Compiler Dept.  
Head

Approved:



Dr. F. H. Martin  
Shuttle Program Manager

## FOREWORD

This document was prepared for the Johnson Space Center, Houston, Texas, under contract NAS 9-13864.

## CONTENTS OF PART I

	<u>Page</u>
1. STRUCTURE OF HAL/S	1-1
1.1 STRUCTURING AND HIGHER ORDER LANGUAGES	1-1
1.2 THE BLOCK STRUCTURE OF HAL/S	1-2
1.3 STATEMENT GROUPING IN HAL/S	1-8
1.4 SUMMARY	1-11
2. HAL/S SYMBOLOGY	2-1
2.1 THE CHARACTER SET	2-1
2.2 RESERVED WORDS, IDENTIFIERS, AND LITERALS	2-2
2.3 FORMAT OF SOURCE TEXT	2-8
2.4 STATEMENT DELIMITING	2-10
2.5 COMMENTS IN HAL/S	2-10
2.6 SUMMARY	2-11
3. A HAL/S COMPILATION - THE PROGRAM BLOCK	3-1
3.1 OPENING AND CLOSING THE BLOCK	3-1
3.2 POSITION OF DATA DECLARATIONS	3-2
3.3 FLOW OF EXECUTION IN THE PROGRAM	3-3
3.4 SUMMARY	3-4
4. DATA DECLARATION	4-1
4.1 HAL/S DATA TYPES	4-1
4.2 SIMPLE DECLARATION STATEMENTS	4-2
4.3 INITIALIZATION OF DATA	4-10
4.4 SUMMARY	4-13

	<u>Page</u>
5. REPLACE STATEMENTS	5-1
5.1 THE REPLACE STATEMENT	5-1
5.2 USING REPLACE STATEMENTS	5-2
5.3 SUMMARY	5-5
6. DATA REFERENCING AND SUBSCRIPTING	6-1
6.1 SUBSCRIPTS OF UNARRAYED DATA TYPES	6-1
6.2 SUBSCRIPTS OF ARRAYED DATA TYPES	6-8
6.3 SUMMARY	6-12
7. EXPRESSIONS	7-1
7.1 ARITHMETIC OPERATIONS	7-1
7.2 CHARACTER OPERATIONS	7-18
7.3 BOOLEAN OPERATIONS	7-20
7.4 COMBINING OPERATIONS & PRECEDENCE	7-23
7.5 SOME EXPLICIT CONVERSIONS	7-26
7.6 BUILT-IN FUNCTIONS	7-32
7.7 SUMMARY	7-36
8. ASSIGNMENTS	8-1
8.1 GENERAL FORM OF ASSIGNMENT	8-1
8.2 ARITHMETIC ASSIGNMENTS	8-2
8.3 CHARACTER ASSIGNMENTS	8-7
8.4 BOOLEAN ASSIGNMENTS	8-10
8.5 MULTIPLE ASSIGNMENTS	8-11
8.6 SUMMARY	8-13

	<u>Page</u>
9.    CONDITIONAL STATEMENTS AND BRANCHES	9-1
9.1 THE CONDITIONAL STATEMENT	9-1
9.2 RELATIONAL EXPRESSIONS	9-7
9.3 LABELS AND BRANCHES	9-15
9.4 SUMMARY	9-19
10.   STATEMENT GROUPS	10-1
10.1 DELIMITING STATEMENT GROUPS	10-1
10.2 REPETITIVE EXECUTION OF STATEMENT GROUPS	10-5
10.3 SELECTIVE EXECUTION OF STATEMENT GROUPS	10-13
10.4 BRANCHING IN STATEMENT GROUPS	10-15
10.5 SUMMARY	10-21
11.   FUNCTIONS AND PROCEDURES	11-1
11.1 BLOCK DEFINITIONS	11-1
11.2 PARAMETER LISTS	11-1
11.3 PROCEDURE CALLING	11-1
11.4 FUNCTION INVOCATION	11-1
11.5 SUMMARY	11-1
12.   INPUT/OUTPUT STATEMENTS	12-1
12.1 HAL/S INPUT/OUTPUT CONCEPTS	12-1
12.2 THE WRITE STATEMENT	12-4
12.3 THE READ STATEMENT	12-8
12.4 INPUT/OUTPUT FORMATTING	12-11

12.5	DEVICE ATTRIBUTES	12-18
12.6	SUMMARY	12-19
13.	REAL TIME FEATURES OF HAL/S	13-1
13.1	HAL/S REAL TIME CONCEPTS	13-1
13.2	DEFINING TASKS	13-1
13.3	SCHEDULING PROCESSES	13-1
13.4	OTHER FEATURES	13-1
13.5	SUMMARY	13-1
14.	SUMMARY OF PART 1	14-1

## INTRODUCTION

HAL/S is a programming language developed by Intermetrics, Inc. for the flight software of the NASA Space Shuttle program. HAL/S is intended to satisfy virtually all of the flight software requirements of the Space Shuttle. To achieve this, HAL/S incorporates a wide range of features, including applications-oriented data types and organizations, real time control mechanisms, and constructs for systems programming tasks.

As the name indicates, HAL/S is a dialect of the original HAL language previously developed by Intermetrics [1]. Changes have been incorporated to simplify syntax, curb excessive generality, or facilitate flight code emission.

## REVIEW OF THE LANGUAGE

HAL/S is a higher order language designed to allow programmers, analysts, and engineers to communicate with the computer in a form approximating natural mathematical expression. Parts of the English language are combined with standard notation to provide a tool that readily encourages programming without demanding computer hardware expertise.

HAL/S compilers accept two formats of the source text, the usual single line format, and also a multi-line format corresponding to the natural notation of ordinary algebra.

## 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 types are also incorporated.

HAL/S permits the formation of multi-dimensional arrays of homogeneous data types, and of tree-like structures which are organizations of non-homogeneous data types.



## REAL TIME CONTROL

HAL/S is a real time control language. Defined blocks of code called programs and tasks can be scheduled for execution in a variety of different ways. A wide range of commands for controlling their execution is also provided including mechanisms for interfacing with external interrupts and other environmental conditions.

## 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.

## SYSTEM LANGUAGE

HAL/S contains a number of features especially designed to facilitate its application to systems programming. Thus, it substantially eliminates the necessity of using an assembler language.

## PROGRAM RELIABILITY

Program reliability is enhanced when software can, by its design, create effective isolation between various sections of code, while maintaining ease of access to commonly used data. HAL/S is a block oriented language in that blocks of code may be established with locally defined variables that are not visible from outside the block. Separately compiled program blocks can be executed together and communicate through one or more centrally managed and highly visible data pools. In a real time environment, HAL/S couples these precautions with locking mechanisms preventing the uncontrolled usage of sensitive data or areas of code.

## ABOUT THE PROGRAMMER'S GUIDE

The Programmer's Guide presents an informal description of the HAL/S Language to the potential HAL/S programmer. It is in no way meant to be an exhaustive catalog of all the various rules of the language. That is the function of the HAL/S Language Specification Document. However, after the HAL/S programmer has absorbed the material presented here, he should have been able to gain enough insight into the workings of the language to enable him to use the Language Specification to clarify any ambiguities.

In order to execute a HAL/S program on any given machine, the programmer will need information contained in the HAL/S User's Manual appropriate for that machine.

The Programmer's Guide is divided into three parts:

- PART I is aimed at the new HAL/S user and contains enough information on the compiler language constructs to enable him to begin programming.
- PART II describes other, more complex, HAL/S constructs which will be used regularly in applications programming.
- PART III presents programming examples designed to illustrate and clarify important complex HAL/S Language constructs. Some of the examples are constructs too advanced to be described in PARTS I and II, but which are formally defined in the HAL/S Language Specification.

## PART I

Part I of the Programmer's Guide is oriented toward new users of HAL/S. It covers all the simpler constructs of the language and contains sufficient information for suprisingly complex programs to be written. Sections of text delimited by horizontal bars are comments referring to the existence of more complex HAL/S constructs to be explained in Part II.

## 1. 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 especial 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"\*. 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 on until each level of the problem has been relegated to the appropriate set of subroutines.

---

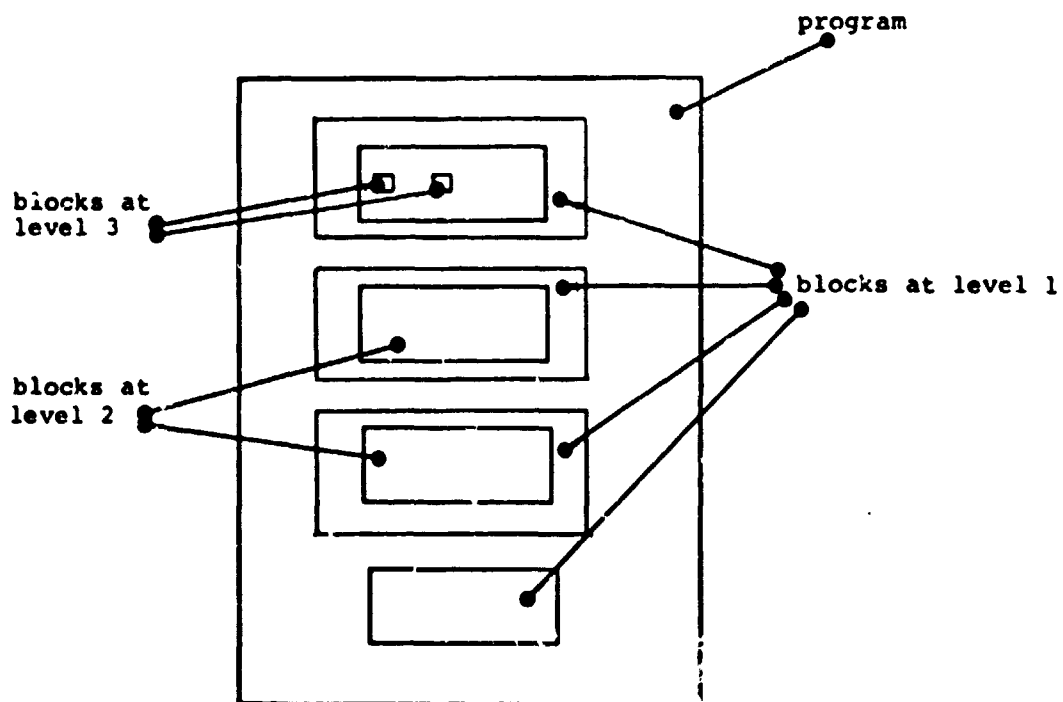
\* Here the term "subroutine" is loosely used in its generally recognized sense, conveying the idea of a subordinate block of code executable as a unit on demand. HAL/S uses different terminology, to be introduced later.

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.

## 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 procedure and function blocks nested within it.



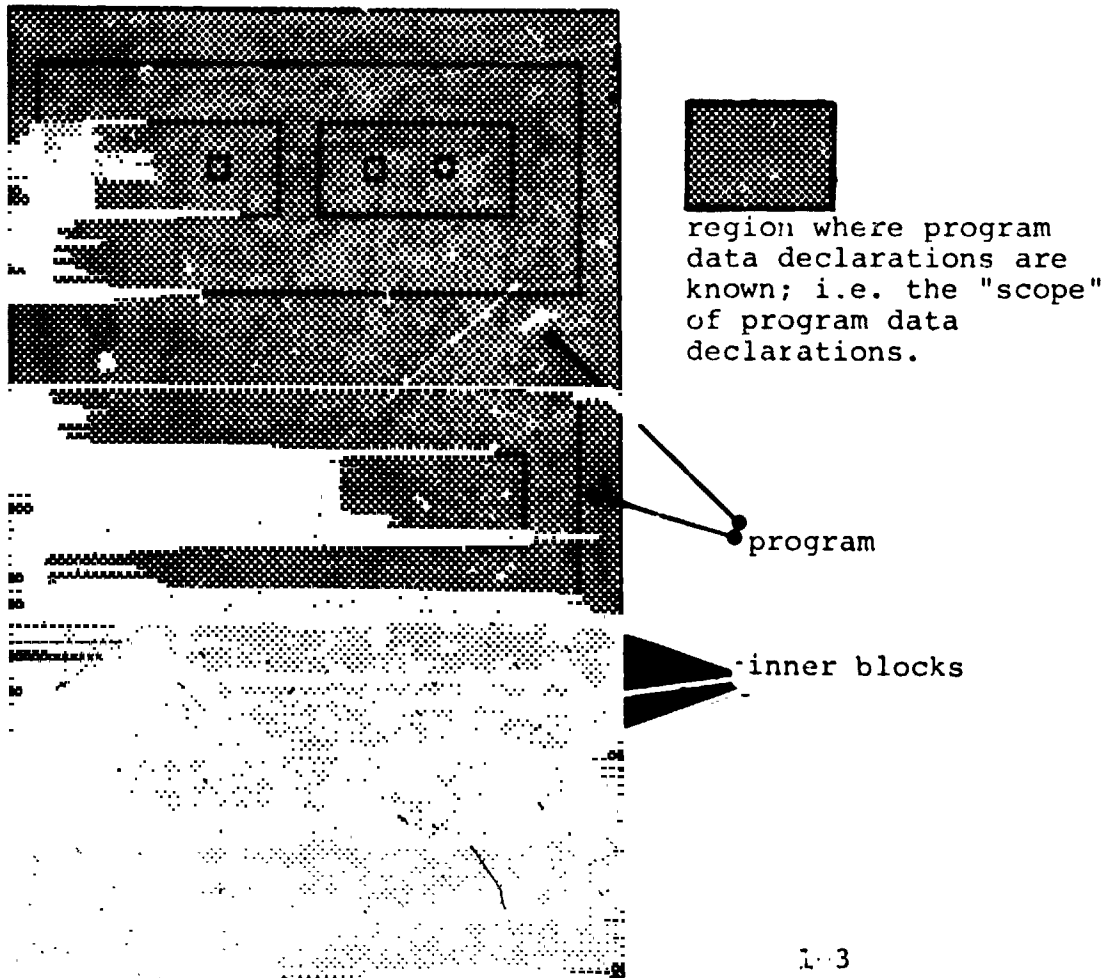
Function and procedure blocks comprise the "subroutines" of Section 1.1. The more deeply nested a block, the deeper the level of detail of the problem solution it is supposed to handle. The difference between function and procedure blocks lies in the manner in which they are invoked, and is clarified later in the Guide.

The HAL/S compilation, then, consists of blocks containing executable statements, some of which perform operations on defined data.

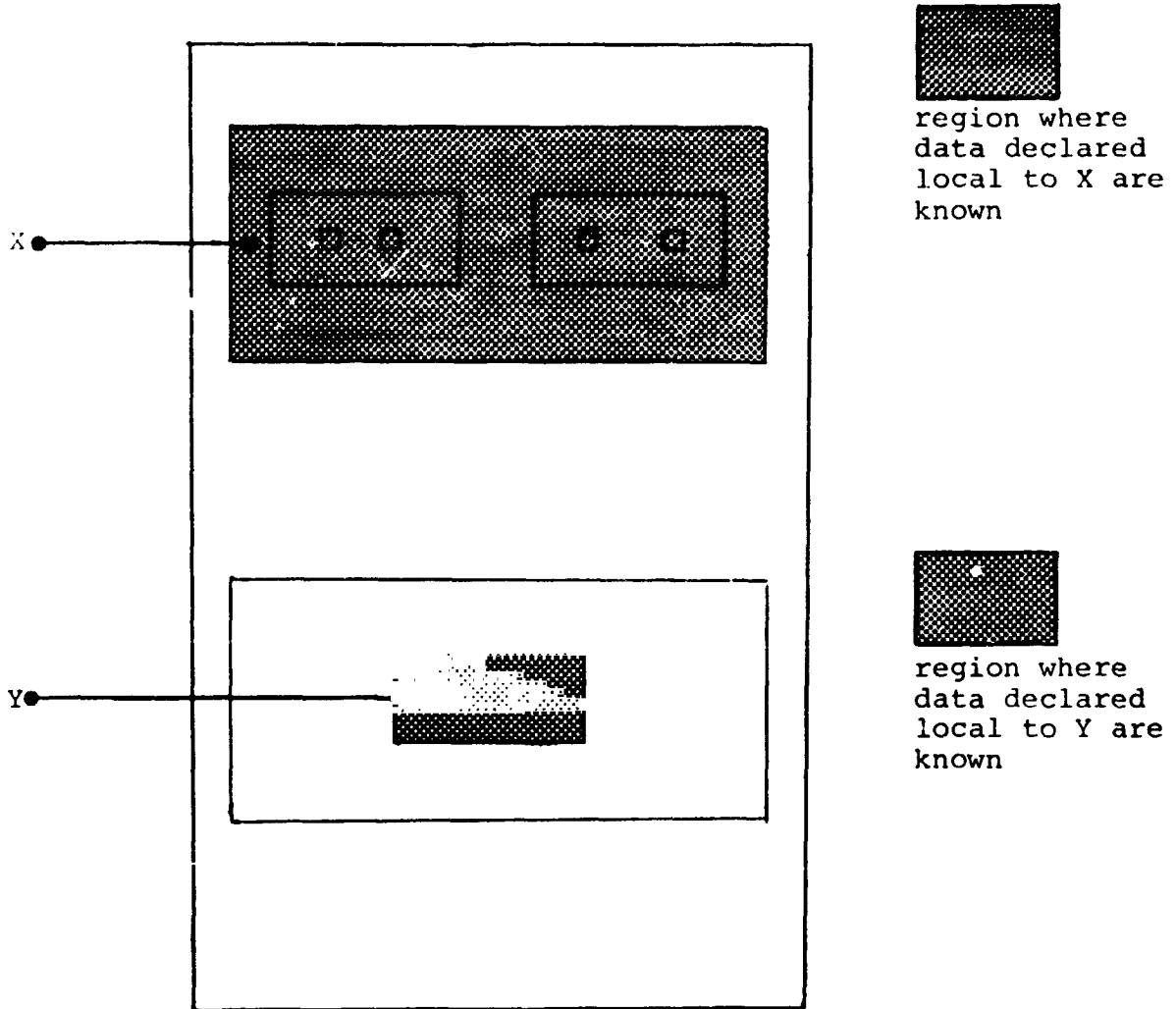
## SCOPING OF DATA

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:

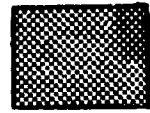
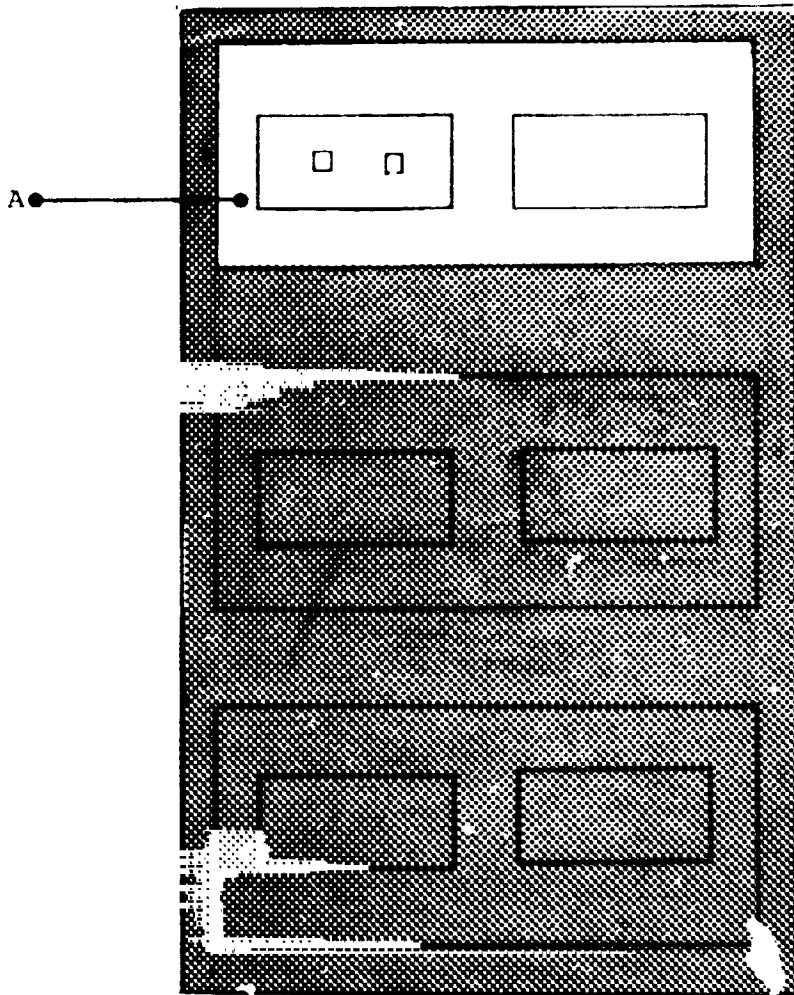


In addition, any procedure or function block nested within a program block may declare local data - data known only in that particular block and in blocks nested within it - as indicated below:



## SCOPING OF BLOCK NAMES

The program block, and every procedure or function within it are named: block names have scoping rules identical with the scoping rules for data already described. The name of any procedure or function block is deemed to have been "declared" in the outer block in which that procedure or function is nested. This bounds the region where the name is known, and therefore from where the procedure or function 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 except from inside itself, as indicated:

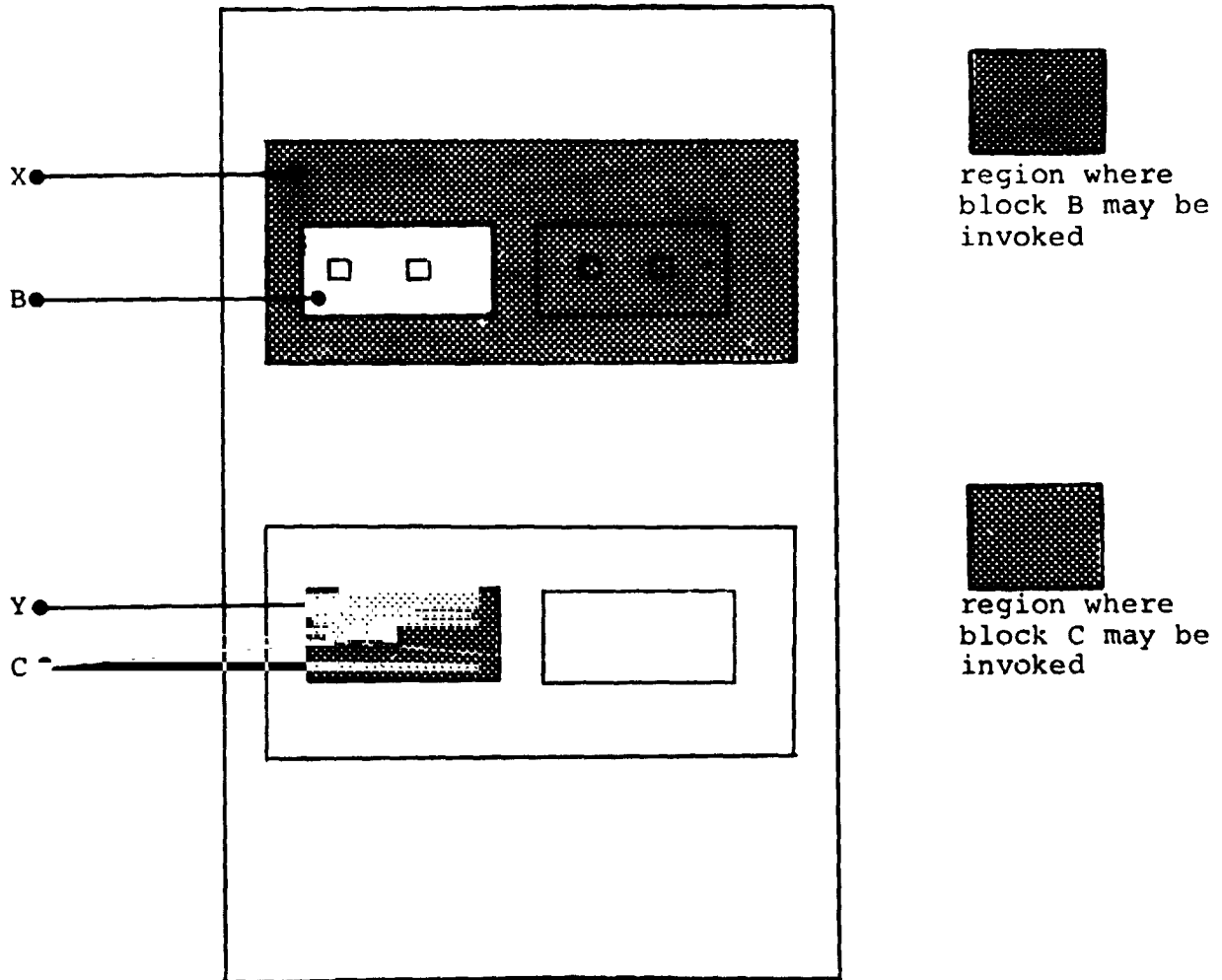


region where  
block A may be  
invoked



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:



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: (tbd)

---

### 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;
- one statement in the sequence may be selected as the only one to be executed.

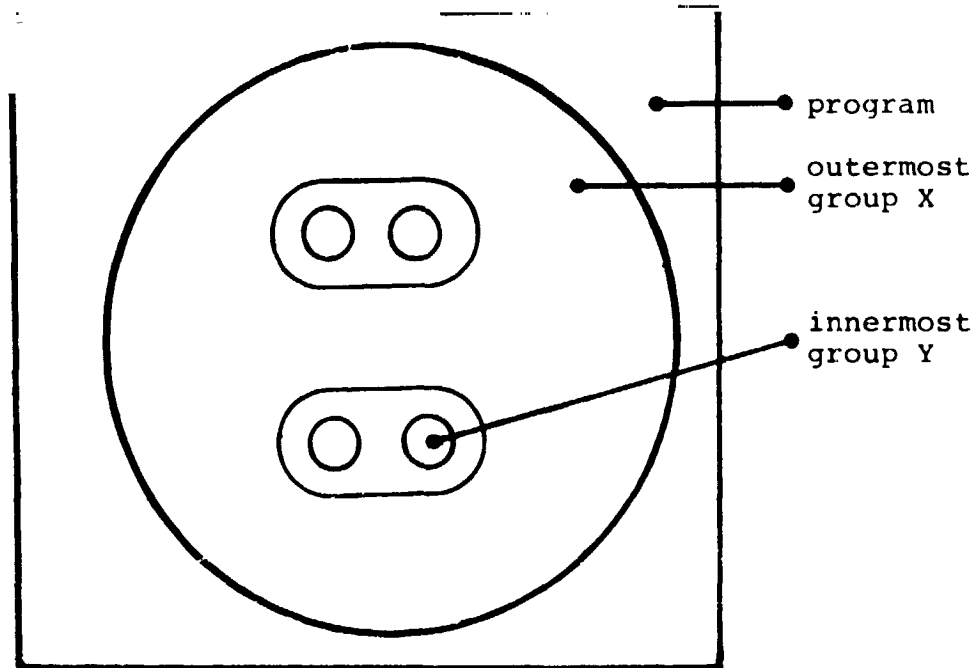
Sequences of compound statements may also be grouped together in the same way and, in turn, be treated as a more complex compound statement, and so on to an arbitrary degree of nesting.

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.

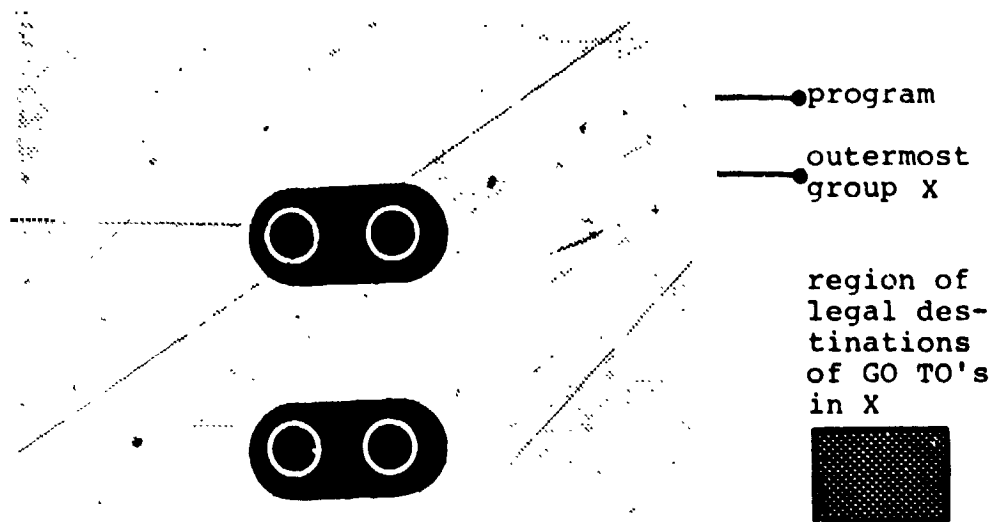
### STATEMENT GROUPS AND GO TO STATEMENTS

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 1.2.

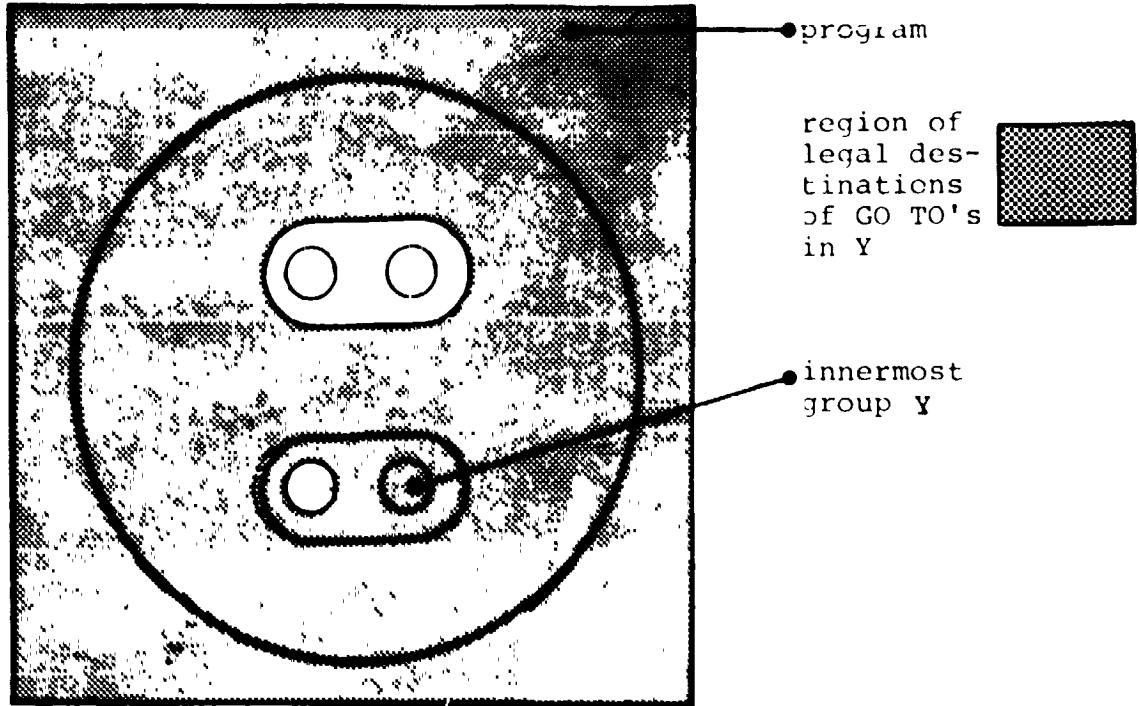
Consider a program containing nested groups of executable statements as shown below:



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



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



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 the only ways of entering and leaving procedures and functions, which is in accordance with structured programming principles.

## 1.4 SUMMARY

This section has been concerned with the structural properties of HAL/S compilations on an abstract level. It remains to be demonstrated in the ensuing sections of PART I how the properties are translated into sequences of actual HAL/S constructs. Section 2 begins this on the most basic level by describing the characteristics of HAL/S source text.

## 2. 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  
  
0123456789  
  
+-*./|-&=<>#@$,;:'")(_%  
  
(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 .
- **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 .



## 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\*.

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    }  
Al                       } legal  
P
```

```
lB                       }  
X_X_                     } illegal
```

---

\* Some implementations of HAL/S may place extra restrictions upon the names of identifiers.

## 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:

mantissa                      exponent  
+ddd.dddE±ddd

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 Appendix ).

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: (tbd).

---

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

'cccccc'

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\*.

---

\* This value is implementation dependent. See Appendix for exceptions.

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: (tbd).

---

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

TRUE ON	} expressing truth, or } binary "1"
FALSE OFF	} expressing falsehood } or binary "0"

---

Literal strings of binary values also exist.  
See (tbd).

---

## 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:

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

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

Example:

$a_{i+1}$  is coded as:

```
:M 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, 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: (tbd).

---

## 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. 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.

Example:

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

## 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 + 1;  
:C ADD ONE TO X  
:C THEN CARRY ON  
|
```

## 2.6 SUMMARY

In Section 2, the most basic elements of the HAL/S Language have been outlined: reserved words, identifiers, literals, the formatting of the source text, and alternate forms of comment insertion.

In Section 3, the overall form of a HAL/S program will be explained, with special references to how declarations of data and executable statements may be arranged within it.



### 3. 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 enclosed in statements delimiting its opening and closing, and identified with a name.

#### 3.1 OPENING AND CLOSING THE PROGRAM BLOCK

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

#### PROGRAM OPENING

The statement that opens the program block takes the form:

```
<label>: PROGRAM;
```

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


## PROGRAM CLOSING

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.

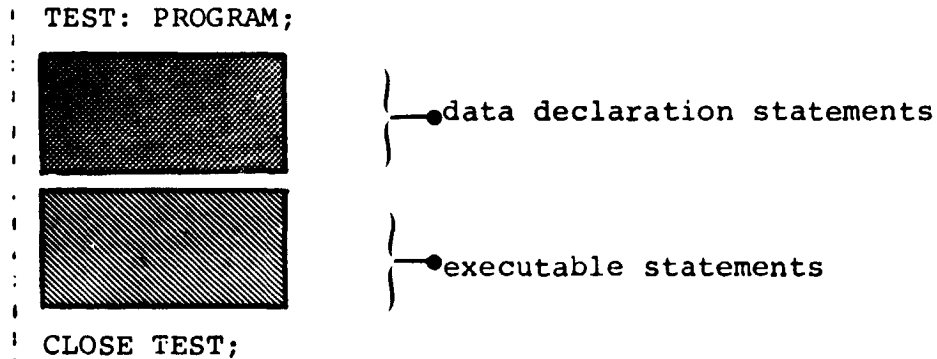
Example:

```
: TEST: PROGRAM;  
:  } body of program goes in here  
: 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:



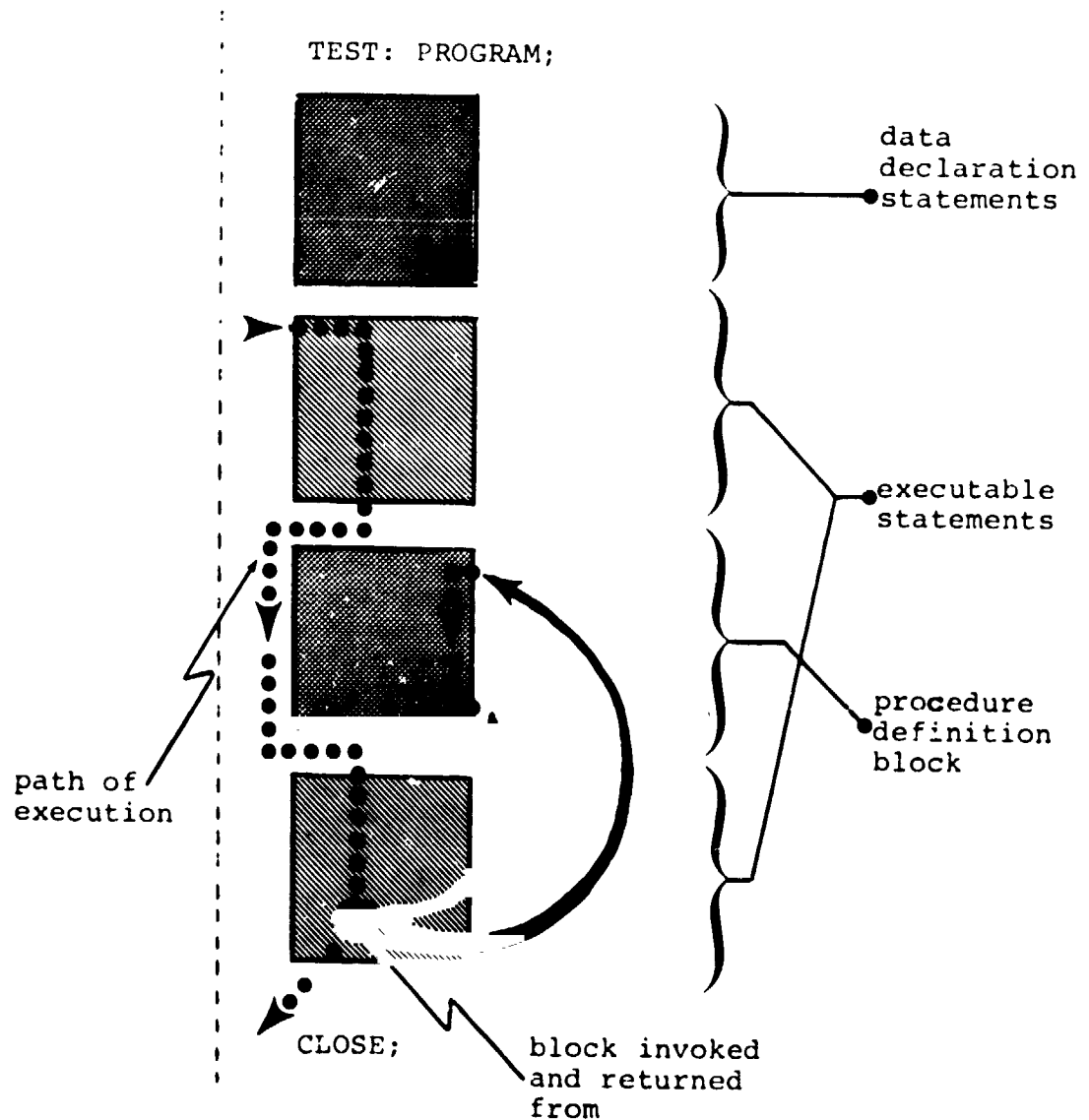
### 3.3 FLOW OF EXECUTION IN THE PROGRAM

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, or branching 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 non-existent.

\* This order is called the "natural order" of execution.

Example:



### 3.4 SUMMARY

Section 3 has described the opening and closing of a program block, has shown where data declarations are placed in it, and has explained the path of execution followed through a program block. The following chapters of Part I will begin to fill in the details of the possible contents of the block. Section 4 describes how data is declared and referenced. It begins to build on the fundamental information given in Section 2.

## 4. 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\* 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 3.

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.

---

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

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 may 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 in fact allows more data types than just those described here. It also allows hierarchical organizations of data-types called "structures".  
See: (tbd)

---

## 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\*.

### Examples:

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

---

\* See Appendix .

## SCALAR

```
DECLARE <name> SCALAR;  
DEC RE <name> SCALAR SINGLE;  
DECLARE <name> SCALAR DOUBLE;
```

1. In each of the forms, <name> is any legal 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 a 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\*.

### Examples:

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

---

\* See Appendix .

## MATRIX

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

1. In each form <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  $1 < m, n \leq 16^*$ .
4. If the size specification (m,n) is absent, a 3x3 matrix is assumed.

### Examples:

```
DECLARE M1 MATRIX(2,4);  
DECLARE M2 MATRIX(4,5) DOUBLE;  
DECLARE M3 MATRIX;
```

• a 3x3 matrix

---

\* This value may be implementation dependent. See Appendix for exceptions.

## 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  $1 < n \leq 16^*$ .
4. If the length specification (n) is omitted a length of 3 is assumed.

### Examples:

```
DECLARE V1 VECTOR(10);  
DECLARE V2 VECTOR(3) DOUBLE;  
DECLARE V3 VECTOR;
```

↙ a 3-vector

---

\* This value may be implementation dependent. See Appendix for exceptions.

## 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.

### Example:

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

## BOOLEAN

```
DECLARE <name> BOOLEAN;
```

1. <name> is any legal identifier.

### Example:

```
DECLARE B1 BOOLEAN;
```

---

\* This value may be implementation dependent. See Appendix

## ARRAYS

In any of the above declarations, regardless of data type, the part of the declaration between the <name> and the terminating semicolon which establishes the type (and possibly precision and size) constitutes the "attributes" of the declaration.

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

```
DECLARE <name> ARRAY(n) <attributes>;
```

1. <attributes> stands for any legal form of attributes for any data type described.
2. n denotes the number of elements in the array (i.e. entries in the table) and must lie in the range  $1 < n \leq 32768^*$ .

### Examples:

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

---

\* This value may be machine dependent. See Appendix for exceptions.

## COMPOUND 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);  
DECLARE B BOOLEAN;  
DECLARE C ARRAY(5) CHARACTER(20);  
DECLARE V ARRAY(3) VECTOR;
```

} separate declarations

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

} equivalent compound  
declaration

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 of declaration called a factored declaration can be used.  
See: (tbd)

---

### 4.3 INITIALIZATION OF DATA

A HAL/S data item of any type may be initialized by incorporating an INITIAL specification into its declaration statement. 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 INITIAL 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 form of INITIAL specification for 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>not</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	} arithmetic
SCALAR	

\* In many respects a data item initialized this way is akin to a literal.



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 specification for multi-valued data items:

```
INITIAL(<value>, <value>, .....)  
CONSTANT(<value>, <value>, .....)
```

1. The meaning of the keyword CONSTANT is the same as for uni-valued data items.
2. 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	arithmetic
SCALAR	
VECTOR	
MATRIX	

3. The number of <value>s in the list must equal the total number of elements implied by the data declaration.

Note that if all the elements of a multi-valued data item are to be initialized to the same value 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 of 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:

```
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:

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

---

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

---

#### 4.4 SUMMARY

Section 4 has dealt with how data is declared in HAL/S compilations, and how it initialized. The next logical step is to begin to discover how it may be used. However, this is put off until Section 6. Section 5 deals with a useful HAL/S construct which allows the user to replace frequently-repeated HAL/S expressions by defining and substituting a symbolic name.

Study of Section 5 can be omitted without detriment to the understanding of the remainder of Part I of the Guide.



## 5. REPLACE STATEMENTS

When it is necessary to repeat a particular HAL/S construct exactly many times during a program, the user can avoid the tedious process of laboriously writing it at length each time by defining a symbolic name to represent the construct, and then replacing the construct with the symbolic name.

This kind of substitution can be of advantage in several ways. For instance, the value of a literal recurring many times can be easily changed between successive compilations. The user need only define a symbolic name to represent the literal, then replace the one with the other. Only one line of the program needs to be recoded as opposed to the many lines that would need recoding if the user had to find and change the literal each time it occurred.

The definition and substitution of the symbolic name is accomplished by a REPLACE statement.

### 5.1 THE REPLACE STATEMENT

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

- ```
REPLACE <name> BY "XXXXXXXXXX";
```
1. XXXXXXXX represents the HAL/S source text which it is desired to substitute. The text is delimited by double quote marks, and must be written in single line format.
  2. <name> is the symbolic name chosen to represent the text. It may be any legal identifier name.
  3. XXXXXXXX may be any legal source text of arbitrary length. Imbedded double quote marks must be represented as a pair 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 comment.

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.

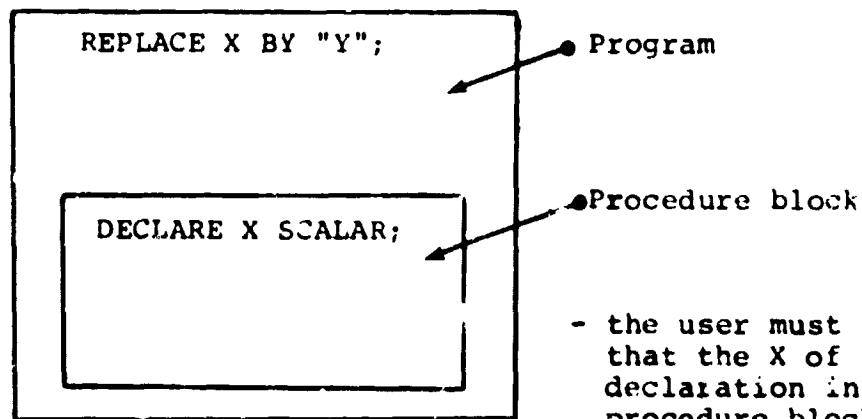
Example:

```
|  
| DECLARE V1 VECTOR(N);  
| REPLACE N BY "4";  
| DECLARE V2 VECTOR(N);  
| :  
| :
```

- the REPLACE statement will only be effective 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.

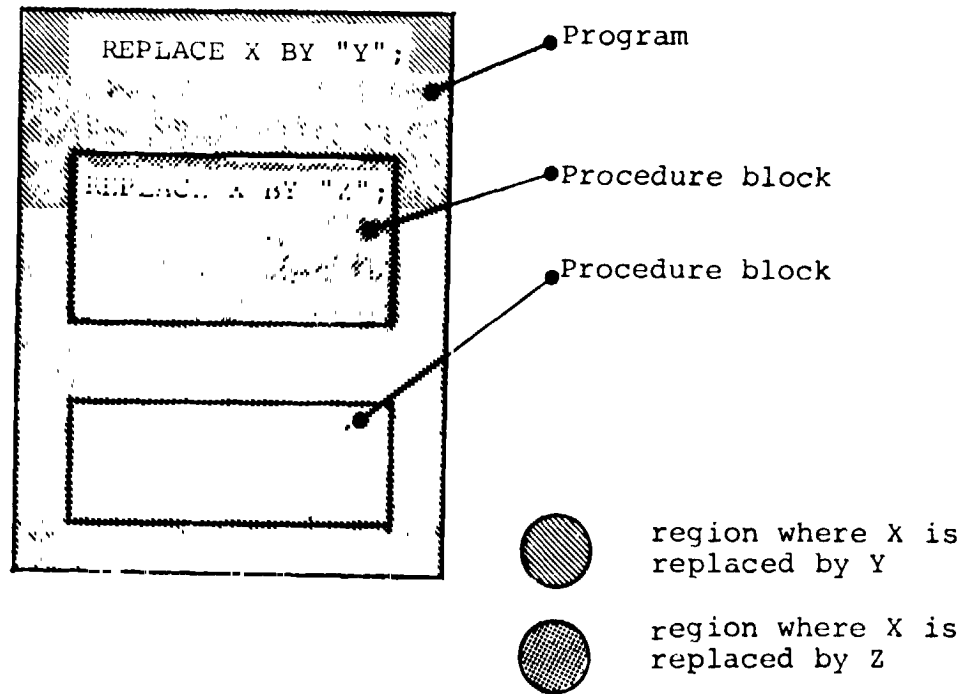
Example:



- 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.

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.

Example:



---

Replace statements may also possess parameters, turning them into a sophisticated macro expansion facility.  
See: (tbd).

---



### 5.3 SUMMARY

Section 5 has dealt with a mechanism for symbolic replacement of HAL/S source text. Section 6 begins to examine the way in which executable statements are constructed by describing how data is referenced.

1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13  
14  
15  
16  
17  
18  
19  
20  
21  
22  
23  
24  
25

## 6. 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)\*. 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.

---

\* 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.

## 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.\*

- To select the  $\alpha^{\text{th}}$  character from STRING:

|                                                                            |
|----------------------------------------------------------------------------|
| $\text{STRING}_{\alpha}$                                                   |
| 1. $\alpha$ is an integer expression in the range $1 \leq \alpha \leq L$ . |

- To select  $\alpha$  characters from STRING, starting from the  $\beta^{\text{th}}$ :

|                                                                  |
|------------------------------------------------------------------|
| $\text{STRING}_{\alpha} \text{ AT } \beta$                       |
| 1. $\alpha$ and $\beta$ are integer expressions.                 |
| 2. $\beta$ is in the range $1 \leq \beta \leq L$ .               |
| 3. $\alpha$ is in the range $0 \leq \alpha \leq L - \beta + 1$ . |

---

\* 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 forms hold true. An account of these is given in Section 8.3.

- To select a substring starting with the  $\alpha^{\text{th}}$  character of STRING, and ending with the  $\beta^{\text{th}}$ :

STRING <sub>$\alpha$</sub>  TO  $\beta$

1.  $\alpha$  and  $\beta$  are integer expressions in the range  $1 \leq \alpha, \beta \leq L$ .
2.  $\beta \leq \alpha$ .

Examples:

if the value of C is 'ABCDEF' then:

C<sub>5</sub> is 'E'

C<sub>2</sub> AT 2 is 'BC'

C<sub>4</sub> TO 6 is '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  $\alpha^{\text{th}}$  element from VEC:

VEC <sub>$\alpha$</sub>

1.  $\alpha$  is an integer expression in the range  $1 \leq \alpha \leq L$ .
2. The resulting data type is SCALAR.

- To select an  $\alpha$ -vector partition starting from the  $\beta$ th element of VEC:

VEC <sub>$\alpha$</sub>  AT  $\beta$

1.  $\alpha$  is an integer literal value in the range  $2 \leq \beta \leq L$ .
2.  $\beta$  is an integer expression in the range  $1 \leq \beta \leq L - \alpha + 1$ .

- To select a partition starting from the  $\alpha$ th element of VEC and ending with the  $\beta$ th.

VEC <sub>$\alpha$</sub>  TO  $\beta$

1.  $\alpha$  and  $\beta$  are integer literal values in the range  $1 \leq \alpha, \beta \leq L$ .
2.  $\beta > \alpha$ .

Examples:

$$\text{if } V = \begin{bmatrix} 4.5 \\ 9.3 \\ 7.1 \\ 2.7 \end{bmatrix} \quad \text{then:}$$

$$V_1 = 4.5 \quad (\text{scalar})$$

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

$$V_2 \text{ AT } 1 = \begin{bmatrix} 4.5 \\ 9.3 \end{bmatrix} \quad (2\text{-vector})$$

## 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  $\alpha^{\text{th}}$  row and  $\beta^{\text{th}}$  column:

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

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

- To select the  $\alpha^{\text{th}}$  row of MAT:

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

1.  $\alpha$  is an integer expression in the range  $1 \leq \alpha \leq M$ .
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  $\alpha^{\text{th}}$  row may be selected.

- To select the  $\beta$ th column of MAT:

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

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

- To select a  $\alpha \times \gamma$  matrix partition starting from the  $\beta$ th row and  $\delta$ th column of MAT:

$\text{MAT}_{\alpha \text{ AT } \beta, \gamma \text{ AT } \delta}$

1.  $\alpha, \gamma$  are integer literal values in ranges  $2 \leq \alpha \leq M, 2 \leq \gamma \leq N$  respectively.
2.  $\beta, \gamma$  are integer expression in ranges  $1 \leq \beta \leq M - \alpha + 1, 1 \leq \delta \leq N - \gamma +$  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:

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

$$M_{2,3} = 2.3 \quad (\text{scalar})$$

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

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

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

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

## 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  $\alpha$ th array element from  $TABLE$ :

| $TABLE_{\alpha}$ :                                                                               |
|--------------------------------------------------------------------------------------------------|
| 1. $\alpha$ is an integer expression in the range $1 \leq \alpha \leq L$ .                       |
| 2. The colon is <u>optional</u> if the data type of $TABLE$ is <u>INTEGER</u> or <u>SCALAR</u> . |

- To select a sub-array of length  $\alpha$  starting from the  $\beta$ th array element of  $TABLE$ :

| $TABLE_{\alpha}$ AT $\beta$ :                                                                    |
|--------------------------------------------------------------------------------------------------|
| 1. $\alpha$ is an integer <u>literal value</u> in the range $1 \leq \alpha \leq L$ .             |
| 2. $\beta$ 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 <u>INTEGER</u> or <u>SCALAR</u> . |

- To select a sub-array starting from the  $\alpha^{\text{th}}$  array element of TABLE and ending with the  $\beta^{\text{th}}$ .

TABLE <sub>$\alpha$</sub>  TO  $\beta$ :

1.  $\alpha, \beta$  are integer literal values in the range  $1 \leq \alpha, \beta \leq L$ .
2.  $\beta \leq \alpha$ .
3. The colon is optional if the data type of TABLE is INTEGER or SCALAR.

Examples:

if T is a 4-array of booleans with values (TRUE,FALSE,TRUE,TRUE) then:

T<sub>2</sub>: is FALSE (unarrayed)

T<sub>3</sub> TO 4: is (TRUE,TRUE) (still arrayed)

if T is a 4-array of integers with values (1,2,3,4) then:

|                              |                 |   |                        |
|------------------------------|-----------------|---|------------------------|
| T <sub>2</sub> is 2          | (unarrayed)     | } | optional colon omitted |
| T <sub>3</sub> TO 4 is (3,4) | (still arrayed) |   |                        |

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

C<sub>1</sub>: is 'YES' (selects first array element)

C<sub>2</sub> TO 3: is ('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.

|                                                                                                                                                                                                                                                                                                                                              |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <p>TABLE &lt;array ss&gt;:&lt;component ss&gt;</p> <ol style="list-style-type: none"><li>1. &lt;array ss&gt;: represents array subscripting of any of the forms previously described.</li><li>2. &lt;component ss&gt; represents any form of component subscripting legal for the data type of TABLE, as described in Section 6.1.</li></ol> |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

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 values ('YES', 'NO', 'MAYBE') then:

$C_{3:3}$  is 'Y' (selects 3<sup>rd</sup> character from third array element)

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

$\left( \begin{array}{|c|c|} \hline 1 & 2 \\ \hline 3 & 4 \\ \hline \end{array} , \begin{array}{|c|c|} \hline 5 & 6 \\ \hline 7 & 8 \\ \hline \end{array} \right)$  then:

$M_{2:2,2} = 8$  (element in 2<sup>nd</sup> row, 2<sup>nd</sup> 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: (tbd).

---

## 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:

TABLE \*: <component ss>

1. <component ss> 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 values ('YES', 'NO', 'MAYBE') then:

$C_{*:1}$  is ('Y', 'N', 'M') (makes 3-array from first character of each item)

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

$\left( \left( \begin{array}{cc} 1 & 2 \\ 3 & 4 \end{array} \right) , \left( \begin{array}{cc} 5 & 6 \\ 7 & 8 \end{array} \right) \right)$  then:

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

$M_{*:,2} = \left( \left( \begin{array}{c} 2 \\ 4 \end{array} \right) , \left( \begin{array}{c} 6 \\ 8 \end{array} \right) \right)$  (2-array of 2-vectors)

---

HAL/S allows more general forms of subscript expressions than just those stated in Section 6. In addition, a symbolic form of reference to the last array or other element of a data type is allowed. Even more complex forms of subscripts apply to parts of tree organizations of data ('structures').  
See: (tbd)

---

### 6.3 SUMMARY

This section has comprehensively described the forms of subscripting available in HAL/S. At this point in the Guide, sufficient information has been given to allow the user to be able to reference different kinds of data. Section 7 shows how operations may be performed on the data so referenced.

## 7. 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"\*. 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:

\* 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.

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

## NEGATION

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

|                                                                                                                                                                                                                                                                                                                 |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <p>Symbolic form: <math>- R</math></p> <p>1. The legal data types for <math>R</math> are given by the following table:</p> <p style="text-align: center;"><u>R-type</u><br/> MATRIX<br/> VECTOR<br/> SCALAR<br/> INTEGER</p> <p>2. Negation of vector and matrix types implies element-by-element negation.</p> |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

Examples:

if  $I$  is an integer and  $I = 5$

then  $-I = -5$

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

and  $-V = \begin{bmatrix} 1.5 \\ -4.2 \\ -5.1 \end{bmatrix}$



## 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:

| <u>L -type</u> | <u>R -type</u> |
|----------------|----------------|
| MATRIX         | MATRIX         |
| VECTOR         | VECTOR         |
| SCALAR } {     | SCALAR         |
| INTEGER } {    | INTEG' .       |

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  $I \equiv 5$

S is scalar with  $S = -4.2$

then

$I + 1 \equiv 6$  (integer result)

$I + 0.5 \equiv 5.5$  (scalar result)

$S + 1 \equiv -3.2$  (scalar result)

$I - S \equiv 9.2$  (scalar result)

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

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

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

but

$V1 - V2_{1 \text{ TO } 3} \equiv \begin{bmatrix} -1.5 \\ -2.5 \\ 1.0 \end{bmatrix}$

is legal because subscripting 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  $M1 = \begin{bmatrix} 1.0 & 0 \\ -0.5 & -1.0 \\ 0 & 0 \end{bmatrix}$

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

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

but,  $M1_{2 \text{ AT } 1,*} - M2 = \begin{bmatrix} 0.5 & 0.5 \\ -1.5 & -2.0 \end{bmatrix}$  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:

| <u>L-type</u> | <u>R-type</u>       |
|---------------|---------------------|
| MATRIX        | { SCALAR<br>INTEGER |
| VECTOR        |                     |
| 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)

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

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

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

S is a scalar with  $S = 0.5$

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

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

## 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 not 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:  $L \cdot R$

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

|        |        |
|--------|--------|
| L-type | R-type |
| VECTOR | VECTOR |

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

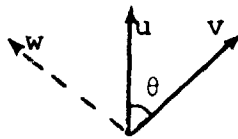
Example:

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  $|w| = |u||v|\sin \theta$   
then  $w = u \times v$

Symbolic form:  $L * R$

1. The type of the operands must be vector:

| <u>L-type</u> | <u>R.-type</u> |
|---------------|----------------|
| 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  $V1 \equiv \begin{bmatrix} 0.5 \\ 0 \\ 0 \end{bmatrix}$

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:

| <u>L-type</u> | <u>R-type</u> |
|---------------|---------------|
| INTEGER }     | { INTEGER     |
| SCALAR }      | { 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:

| <u>L-type</u> | <u>R-type</u> |
|---------------|---------------|
| INTEGER }     | { VECTOR      |
| SCALAR }      | { MATRIX      |
| VECTOR }      | { INTEGER     |
| MATRIX }      | { SCALAR      |

3. Element-by-element multiplication of the vector or matrix is implied.
4. If an operand is of integer type, it is first converted to single precision scalar.

Examples:

if  $S$  is scalar with  $S \equiv 1.5$

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

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

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

CASE (3)

2. The operand types are:

| <u>L-type</u> | <u>R-type</u> |
|---------------|---------------|
| VECTOR        | VECTOR        |

3. If the  $L$ -operand is of length  $m$ ,  
and the  $R$  operand is of length  $n$ ,  
the result is an  $m \times n$  matrix.

Examples:

If  $V_1$  is a 3-vector with  $V_1 \equiv \begin{bmatrix} 1.0 \\ -1.0 \\ 1.0 \end{bmatrix}$

$V_2$  is a 2-vector with  $V_2 \equiv \begin{bmatrix} 0.5 \\ 0.6 \end{bmatrix}$

then  $V_1 V_2 \equiv \begin{bmatrix} 0.5 & 0.6 \\ -0.5 & -0.6 \\ 0.5 & 0.6 \end{bmatrix}$  (3 x 2 matrix)

and  $V_2 V_1 \equiv \begin{bmatrix} 0.5 & -0.5 & 0.5 \\ 0.6 & -0.6 & 0.6 \end{bmatrix}$  (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)

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

Note that by using partitioning subscripts that

$M1_{*,2 \text{ TO } 3} M2$  is illegal because of dimension mismatch;

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

CASE (5)

2. The operand types are:

| <u>L-type</u> | <u>R-type</u> |
|---------------|---------------|
| VECTOR        | MATRIX        |
| MATRIX        | VECTOR        |

3. If the L operand is an  $m \times n$  matrix, the R operand must be an  $n$ -vector, and the result is an  $m$ -vector.

4. If the L operand is an  $m \times n$  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:

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

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

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

and  $M V$  is illegal because of dimension mismatch;

however,  $M V_1 \text{ TO } 2 \equiv \begin{bmatrix} -0.5 \\ -1.0 \\ -0.2 \end{bmatrix}$  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:

| <u>L -type</u> | <u>R -type</u> |
|----------------|----------------|
| 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 = 10 (integer result)

and I\*\*-1 = 0.2 (scalar result)

also 2\*\*0.5 =  $\sqrt{2}$  (scalar result)

CASE (2)

2. The operand types are:

| <u>L-type</u> | <u>R-type</u> |
|---------------|---------------|
| 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:

| <u>value</u> | <u>result</u>               |
|--------------|-----------------------------|
| $\leq -2$    | 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  $M \equiv \begin{bmatrix} 0.5 & 1 \\ -0.5 & 0 \end{bmatrix}$

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

$$M^{-1} \equiv \begin{bmatrix} 0 & -2 \\ 1 & 1 \end{bmatrix}$$

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

CASE (3)

2. The operand types are:

| <u>L-type</u> | <u>R-type</u> |
|---------------|---------------|
| 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 possible in unary operations: transposition is considered a unary operation.
2. Where an operation specifies type conversion from integer to single precision scalar, this conversion is carried out first.
3. If only one operand is integer and no type conversion is implied, no precision conversion takes place.
4. If both operands have the same precision, the result is of the same precision (even if not of the same type).
5. If the operands have mixed precision, the single precision operand is converted to double precision. Then rule 4 is applied.

## 7.2 CHARACTER OPERATIONS

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

| <u>Symbol</u>                                             | <u>Purpose</u> |
|-----------------------------------------------------------|----------------|
| $\begin{array}{l} \text{  } \\ \text{CAT} \end{array} \}$ | 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: $L \begin{array}{l} \text{  } \\ \text{CAT} \end{array} R$                                                                                 |                                                                                                               |
| 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. |                                                                                                               |
| $\begin{array}{l} \text{L-type} \\ \hline \text{INTEGER} \\ \text{SCALAR} \\ \text{CHARACTER} \end{array} \}$                                             | $\begin{array}{l} \text{R-type} \\ \hline \text{INTEGER} \\ \text{SCALAR} \\ \text{CHARACTER} \end{array} \}$ |
| 2. The rules for converting integer and scalar types to character type are to be found in Appendix .                                                      |                                                                                                               |



Examples:

If C is a character item with C  $\equiv$  ' UNITS'

I is integer with I  $\equiv$  10

then 'TEN' || C  $\equiv$  'TEN UNITS'

I || C  $\equiv$  '10 UNITS'

and I || I  $\equiv$  '1010'

## 7.3 BOOLEAN OPERATIONS

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

| Symbol   | Purpose              |
|----------|----------------------|
| &<br>AND | logical intersection |
| <br>OR   | logical conjunction  |
| ~<br>NOT | logical complement   |

### COMPLEMENT

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

|                                        |
|----------------------------------------|
| Symbolic form: $\sim R$<br>NOT         |
| 1. The $R$ operand is of Boolean type. |

Example:

If  $B$  is Boolean with  $B \equiv \text{TRUE}$

then  $\sim B \equiv \text{FALSE}$

## CONJUNCTION

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

Symbolic form:  $L \mid R$   
OR

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

| T=TRUE<br>F=FALSE |   | L |   |
|-------------------|---|---|---|
|                   |   | T | F |
| R                 | T | T | T |
|                   | F | T | F |

Examples:

If  $B$  is Boolean with  $B \equiv \text{FALSE}$

then  $B \mid B \equiv \text{FALSE}$

$B \mid \text{TRUE} \equiv \text{TRUE}$

## INTERSECTION

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

Symbolic form:  $L \ \& \ R$   
AND

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

| T=TRUE<br>F=FALSE |   | L |   |
|-------------------|---|---|---|
|                   |   | T | F |
| R                 | T | T | F |
|                   | F | F | F |

Examples:

If  $B$  is Boolean with  $B \equiv \text{FALSE}$

then  $B \& \text{TRUE} \equiv \text{FALSE}$

$B \& B \equiv \text{FALSE}$

## 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:

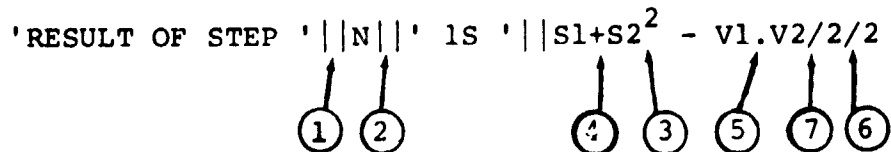
| Symbol  | Precedence | Purpose               |
|---------|------------|-----------------------|
|         | FIRST      |                       |
| **      | 1          | exponentiation, etc.  |
| (blank) | 2          | multiplication        |
| *       | 3          | cross product         |
| .       | 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.

Examples:

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



### BOOLEAN PRECEDENCE

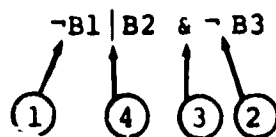
The precedence rules for Boolean operations are stated separately because there are no implicit conversions causing interaction with arithmetic and character operations.

| Symbol | Precedence | Purpose      |
|--------|------------|--------------|
| ¬, NOT | FIRST      | 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:

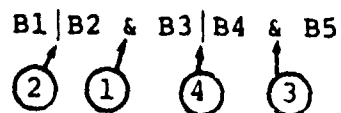


## OVERRIDING PRECEDENCE ORDER

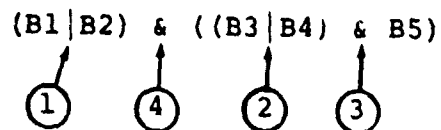
In HAL/S, the order of precedence can be overridden at will by the use of parentheses, nested to any arbitrary depth.

Examples:

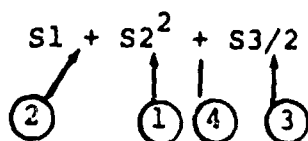
In the following Boolean expression,

$B1|B2 \& B3|B4 \& B5$   


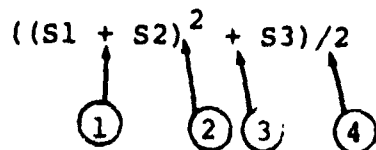
parentheses may change the precedence order as shown:

$(B1|B2) \& ((B3|B4) \& B5)$   


In the following arithmetic expression,

$S1 + S2^2 + S3/2$   


parentheses may change the precedence order as shown:

$((S1 + S2)^2 + S3)/2$   


---

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

---

## 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:

$(\langle \text{expression} \rangle) @ \text{DOUBLE}$

$(\langle \text{expression} \rangle) @ \text{SINGLE}$

1. In the first form, if  $\langle \text{expression} \rangle$  is a single precision arithmetic expression, it is converted to double precision. If it is already double precision, the conversion has no effect.
2. In the second form, if  $\langle \text{expression} \rangle$  is a double precision arithmetic expression it is rounded to single precision. If it is already single precision, the conversion has no effect.

Example:

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

$(A + B) @ \text{DOUBLE}$

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:

|                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <p>VECTOR<sub>n</sub> (&lt;exp&gt;, &lt;exp&gt;.....)</p> <ol style="list-style-type: none"><li>1. The subscript number n specifies the length of the vector to be created, and lies in the range <math>1 &lt; n \leq 16^*</math>.</li><li>2. If n is omitted the resulting vector is assumed to be of length 3.</li><li>3. Each &lt;exp&gt; is a scalar or integer expression.</li><li>4. The number of expressions in the list <u>must</u> match the implicit or explicit result length.</li><li>5. The result of the above conversion is in single precision.</li><li>6. The matrix is assembled row by row from the list.</li></ol> |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

### Examples:

VECTOR(1, 2, 3)

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

\* This value may be implementation dependent. See Appendix for exceptions.

if S is a scalar with  $S \approx 0.5$  then

`VECTOR4 (S, S2, 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:

`VECTOR@ DOUBLE, n (<exp>, <exp> ....)`

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

Examples:

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

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

`VECTOR@ DOUBLE, 4 (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:

$\text{MATRIX}_{m, n}(\langle \text{exp} \rangle, \langle \text{exp} \rangle, \dots)$

1. The subscript numbers  $m, n$  specify the row and column dimensions of the matrix to be created, and must lie in the range  $1 < m, n \leq 16^*$ .
2. The subscript may be omitted, in which case the resulting matrix is assumed to be 3 by 3.
3. Each  $\langle \text{exp} \rangle$  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.

\* This value may be implementation dependent. See Appendix for exceptions.

Examples:

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

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

MATRIX<sub>2, 3</sub>(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:

MATRIX<sub>@</sub> DOUBLE, m, n (<exp>, <exp> .....

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:

MATRIX<sub>@</sub> DOUBLE(1, 2, 3, 4, 5, 6, 7, 8, 9)

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

MATRIX<sub>A</sub> 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: tbd.

---

## 7.6 BUILT-IN FUNCTIONS

HAL/S possesses a comprehensive range of 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 . 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( $\alpha$ )        | returns $ \alpha $ (the absolute value of $\alpha$ ). $\alpha$ may be integer or scalar.                                                                        |
| DIV( $\alpha, \beta$ ) | returns the result of integer division of $\alpha$ by $\beta$ . $\alpha$ and $\beta$ may be scalar or integer: scalar values are rounded to integer before use. |
| ROUND( $\alpha$ )      | rounds a scalar $\alpha$ to an integer.                                                                                                                         |
| ODD( $\alpha$ )        | returns a Boolean result, which is TRUE if $\alpha$ is odd, and FALSE if $\alpha$ is even.                                                                      |
| SIGN( $\alpha$ )       | 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( $\alpha$ )  | returns $\cos \alpha$   |
| EXP( $\alpha$ )  | returns $e^\alpha$      |
| LOG( $\alpha$ )  | returns $\log_e \alpha$ |
| SIN( $\alpha$ )  | returns $\sin \alpha$   |
| SQRT( $\alpha$ ) | returns $\sqrt{\alpha}$ |
| TAN( $\alpha$ )  | returns $\tan \alpha$   |

## VECTOR-MATRIX FUNCTIONS

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

| Function            | Comments                                                 |
|---------------------|----------------------------------------------------------|
| ABVAL( $\alpha$ )   | returns length of vector $\alpha$                        |
| INVERSE( $\alpha$ ) | returns inverse of square matrix $\alpha$                |
| UNIT( $\alpha$ )    | returns unit vector in same direction as vector $\alpha$ |

## CHARACTER FUNCTIONS

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

| Function           | Comments                                                |
|--------------------|---------------------------------------------------------|
| LENGTH( $\alpha$ ) | returns current length of character string $\alpha$     |
| TRIM( $\alpha$ )   | strips leading and trailing blanks from string $\alpha$ |

## MISCELLANEOUS FUNCTIONS

Some of the more important miscellaneous functions are:

| Function        | Comments                                                                     |
|-----------------|------------------------------------------------------------------------------|
| DATE            | returns date at time of execution.                                           |
| MAX( $\alpha$ ) | returns the maximum value in the integer or scalar array $\alpha$            |
| MIN( $\alpha$ ) | returns the minimum value in the integer or scalar array $\alpha$            |
| 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;
```

## 7.7 SUMMARY

Section 7 has described how HAL/S expressions are synthesized from operands and operators, and in what order such expressions are executed. Expressions, particularly of integer and scalar type, form parts of many HAL/S language constructs. Section 6 referred many times to the use of integer expressions in subscripting.

Section 8 describes the assignment statement, which causes the result of an expression to be stored in some data item or items.

## 8. 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:

| <u>L-type</u> | <u>R-type</u>                 |
|---------------|-------------------------------|
| MATRIX        | { MATRIX<br>INTEGRER (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  $M1 \equiv \begin{bmatrix} 1.0 & 1.0 & 2.0 \\ 0.5 & -0.5 & 1.0 \end{bmatrix}$

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

then

$M3 = -M1;$

results in  $M3 \equiv \begin{vmatrix} -1.0 & -1.0 & -2.0 \\ -0.5 & 0.5 & -1.0 \end{vmatrix}$

$M2 = M1;$  is illegal (column mismatch)

but

$M2 = M1_{*, 2 AT 2};$

results in  $M2 \equiv \begin{vmatrix} 1.0 & 2.0 \\ -0.5 & 1.0 \end{vmatrix}$

$M3 = 0;$  results in  $M3 = \begin{vmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \end{vmatrix}$

but

$M3 = 1;$  is illegal

## VECTOR

The receiving data item is a vector.

1. The operand types are:

| <u>L-type</u> | <u>R-type</u>                |
|---------------|------------------------------|
| VECTOR        | { VECTOR<br>INTEGER (rule 3) |

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 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 = V1; is illegal (type mismatch),  
|

but

| M2<sub>1</sub> = 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:

| <u>L-type</u> | <u>R-type</u> |
|---------------|---------------|
| INTEGER       | INTEGER       |
| SCALAR        | SCALAR        |

2. If the L- and R-types do not match, type conversion of the result of the R-expression takes place before assignment.
3. Scalar-to-integer conversion implies rounding of the value of the R-expression.

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<sub>2,2</sub> = I - S;

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

(? indicates values unchanged by assignment)

M<sub>2,\*</sub> = 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, it takes place before precision conversion.



### 8.3 CHARACTER ASSIGNMENTS

The receiving data item is character type.

1. The operand types are:

| <u>L-type</u> | <u>R-type</u>                    |
|---------------|----------------------------------|
| CHARACTER     | { CHARACTER<br>INTEGER<br>SCALAR |

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

Examples:

If C is a character with  $C \equiv \text{'ABCDE'}$  and

C2 is a character,

then

|  |                       |                      |
|--|-----------------------|----------------------|
|  | C2 = C <sub>3</sub> ; | 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 item 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

STRING <sub>$\alpha$</sub>

denote a receiving data item of character type:

N is declared maximum length  
n is working length before assignment

1. The range of the subscript expression  $\alpha$  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  $\alpha$ , either by truncation of the excess from the right, or by padding on the right with blanks.
3. If the range of  $\alpha$  lies inside the range 1-n, then simple substitution of the character positions implied takes place.
4. If the range of  $\alpha$  lies partly beyond the range 1 - n, then the working length of STRING is increased appropriately.
5. If the range of  $\alpha$  lies totally beyond the range 1 - n, the working length of STRING is increased appropriately, and the gap between the n<sup>th</sup> character and the first position implied by  $\alpha$  (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:

| C1<sub>2</sub> TO 3 = 'QQ';  
|  
results in C1 ≡ 'AQQD'  
| C1<sub>2</sub> TO 3 = '1234';  
|  
results in C1 ≡ 'A12D'  
| C1<sub>2</sub> TO 3 = 'X';  
|  
results in C1 ≡ 'AX D'

By Rules 2 and 4:

| C1<sub>4</sub> TO 5 = 'QQ';  
|  
results in C1 ≡ 'ABCQQ'  
| (working length increased by 1)  
| C1<sub>4</sub> TO 5 = 'X';  
|  
results in C1 ≡ 'ABCX '  
| (working length increased by 1)

By Rules 2 and 5:

| C1<sub>5</sub> TO 6 = 'QQ';  
|  
results in C1 ≡ 'ABCDQQ'  
| (working length increased by 2)  
| C1<sub>7</sub> TO 9 = 'FGH';  
|  
results in C1 ≡ 'ABCD FGH'  
| C1<sub>6</sub> = 'FCH';  
|  
results in C1 ≡ 'ABCD F'

## 8.4 BOOLEAN ASSIGNMENTS

The receiving data item is of a Boolean type.

1. The operand types are:

| <u>L-type</u> | <u>R-type</u> |
|---------------|---------------|
| 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:

$$L^1, L^2, \dots L^n = R;$$

1. The value of the R-expression is assigned to all  $L^1 \dots L^n$  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 order of assignment is guaranteed.

Examples:

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

| M1, V1 = 0;

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

If C is a character,  
I is an integer,

| C, I = 127.2;

results in  $C \equiv '1.2720000E+02'$ ,  $I \equiv 128$

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  $I \equiv 2$ , then

|  $V1_I, I = I + 1$ ;  
|

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

If I is assigned before  $V1_I$ ,

then  $V1_I \equiv \begin{bmatrix} ? \\ ? \\ 3 \end{bmatrix}$ , otherwise  $V1_I \equiv \begin{bmatrix} ? \\ 3 \\ ? \end{bmatrix}$

(? 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: tbd.

---

## 8.6 SUMMARY

Section 8 has described assignment statements by which the results of expressions can be assigned to one or more data items. Assignments often form the core of a program but are generally limited in effectiveness unless their execution can be controlled with a degree of flexibility.

Section 9 begins to describe how execution can be controlled by introducing the HAL/S conditional, or IF, statement.





## 9. 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 critically 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

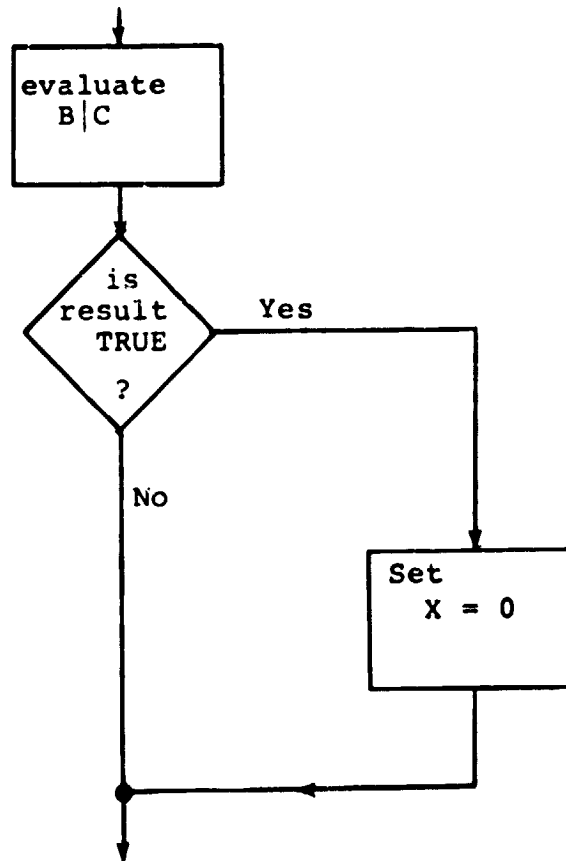
The form of the simple version is:

|                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>IF &lt;exp&gt; THEN &lt;statement&gt;;</pre> <ol style="list-style-type: none"><li>1. &lt;exp&gt; 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).</li><li>2. &lt;statement&gt; constitutes the true part of the conditional statement. Except as noted in Rule 3 it may be any executable statement, either <u>simple</u> or <u>compound</u>.</li><li>3. &lt;statement&gt; may <u>not</u> possess a label, and may <u>not</u> be another conditional statement.</li><li>4. If &lt;exp&gt; is FALSE, execution proceeds to the next statement. If TRUE, &lt;statement&gt; is executed first.</li></ol> |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

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:



```
| IF B|C THEN DO;  
|   X = X - 1;  
|   Y = Y + 1;  
| END;
```

The true part is a compound statement containing two assignments.

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

Illegal because true part is a conditional statement, in violation of Rule 3.

## AUGMENTED IF STATEMENT

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

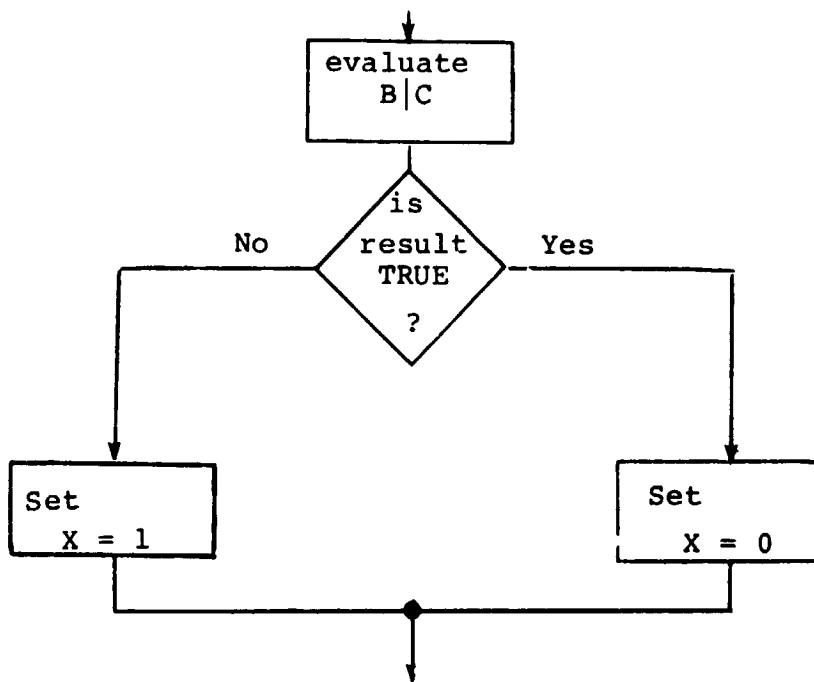
```
IF <exp> THEN <statement>;  
ELSE <else stmt>;
```

1. The form of the IF clause and true part are the same as in the simple conditional statement.
2. <else stmt> constitutes the false part of the conditional statement. It may be any unlabelled executable statement either simple or compound.
3. If <exp> is FALSE, execution proceeds to the next statement via <else stmt>. If TRUE, it proceeds to the next statement via <statement>.

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.

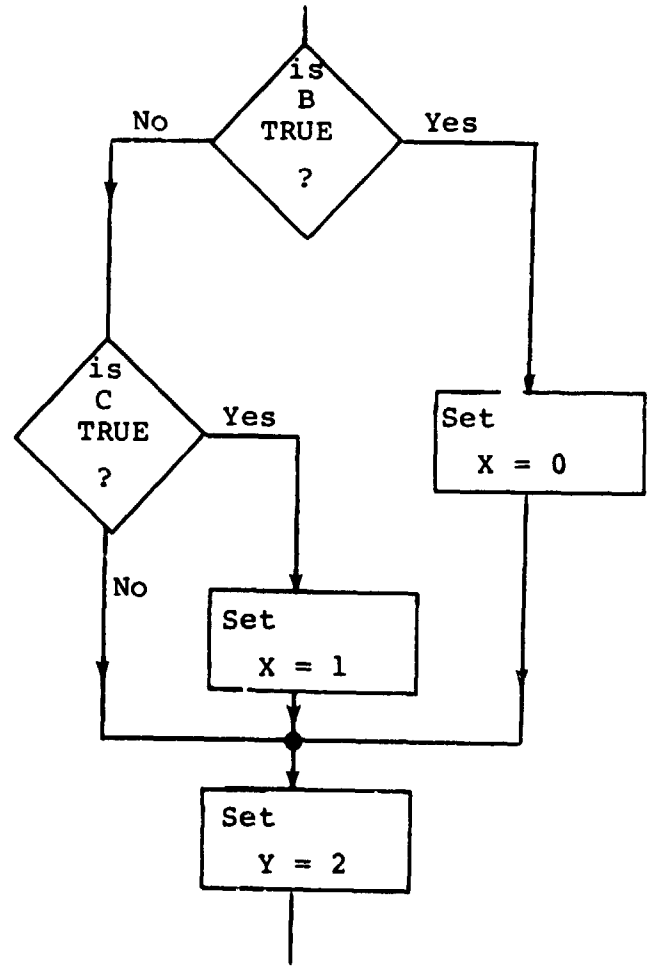


```
| If B|C THEN DO;  
|   X = 1;  
|   Y = 2;  
| END;  
| ELSE DO;  
|   X = 2;  
|   Y = 1;  
| END;
```

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:



## 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 to be 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  $\neg$ .

### COMPARATIVE OPERATIONS

HAL/S recognizes the following comparative operators:

| Symbol            | Purpose                | Class |
|-------------------|------------------------|-------|
| >                 | greater than           | I     |
| <                 | less than              |       |
| <=                | less than or equals    |       |
| NOT ><br>$\neg$ > | } not greater than     |       |
| > =               | greater than or equals |       |
| NOT <<br>$\neg$ < | } not less than        |       |
| =                 | equals                 | II    |
| NOT =<br>$\neg$ = | } not equals           |       |

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:  $L \overset{=}{\text{NOT}} = R$   
 $\overset{=}{\text{NOT}} =$

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

| L-type    | R-type    |
|-----------|-----------|
| VECTOR    | VECTOR    |
| MATRIX    | MATRIX    |
| BOOLEAN   | BOOLEAN   |
| CHARACTER | CHARACTER |

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 STRING is character type with  
STRING  $\equiv$  'ABC '

STRING = 'PQR'  
is FALSE.

STRING = 'ABC '  
is FALSE - character strings must be of the same  
length.

If V, W 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,

$V1 - V = 2 V$  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  $V1_{1 \text{ TO } 2} = V2$  is TRUE.

• CLASS I AND CLASS II

|                       |
|-----------------------|
| >                     |
| <                     |
| >=                    |
| <=                    |
| NOT >                 |
| Symbolic form: L ~> R |
| NOT <                 |
| ~ <                   |
| =                     |
| NOT =                 |
| ~ =                   |

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

| L-type  | R-type  |
|---------|---------|
| INTEGER | INTEGER |
| SCALAR  | SCALAR  |

2. In a mixed integer-scalar operation, the integer operand is converted to scalar before the comparison takes place.

Examples:

If I is an integer with I ≡ 5

then I = 5 is TRUE  
 I < 4 is FALSE  
 I >= 5 is TRUE

#### 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 takes place before the operation is executed. The rules under which it takes place are as follows:

1. Where an operation specifies type conversion from integer to single precision scalar, this conversion is carried out first.
2. If only one operand is integer and no type conversion is implied, no precision conversion takes place.
3. If both operands have the same precision, the result is of the same precision (even if not of the same type).
4. If the operands have mixed precision, the single precision operand is converted to double precision. Then rule 3 is applied.

## 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  $\equiv$  'ABC'

then

$$V1 = V2 | C_1 = 'A' \text{ is TRUE}$$

$$V1 = V2 \& C_1 = 'A' \text{ is FALSE}$$

If B is Boolean then

$$B | V1 = V2 \text{ is illegal}$$

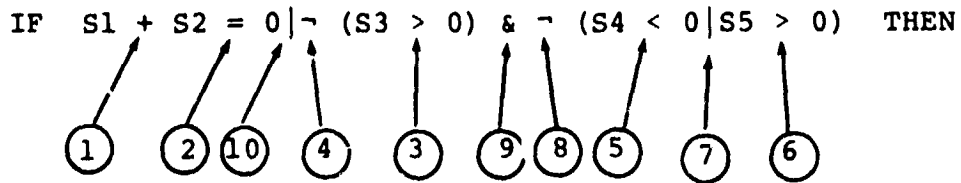
## PRECEDENCE

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

| Symbol                                                                                                                                    | Precedence | Purpose                                             |
|-------------------------------------------------------------------------------------------------------------------------------------------|------------|-----------------------------------------------------|
|                                                                                                                                           | FIRST      |                                                     |
|                                                                                                                                           | 1          | { operations involving<br>{ operands of comparisons |
| >                                                                                                                                         | }          | comparative<br>operations                           |
| <                                                                                                                                         |            |                                                     |
| <=                                                                                                                                        |            |                                                     |
| NOT >, ¬>                                                                                                                                 |            |                                                     |
| >=                                                                                                                                        |            |                                                     |
| NOT <, ¬<                                                                                                                                 |            |                                                     |
| =                                                                                                                                         | 2          |                                                     |
| NOT =, ¬=                                                                                                                                 |            |                                                     |
| &, AND                                                                                                                                    | 3          | logical operations<br>on comparisons                |
| , OR                                                                                                                                      | 4          |                                                     |
| ¬, NOT                                                                                                                                    | *          |                                                     |
| * Any operand of this operator <u>must always</u> be parenthesized, and is evaluated immediately after evaluation of the operator itself. |            |                                                     |

Example:

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



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

Examples:

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

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

The first statement will cause V<sub>3</sub> 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.

## 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), with two exceptions.
2. <statement> may not be the "true part" or "ELSE part" of a conditional statement.
3. <label> is a user-defined identifier name (see Section 2.2).

Examples:

```
| ONE:  X = X + 1;  
| TWO:  Y = 0;  
|
```

The following are illegal since they violate Rule 2:

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

However, the conditional statement itself may be labelled:

```
| THREE:  IF X = 0 THEN Y = 1;  
|
```

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

Example:

```
| 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 labelled:

```
| 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

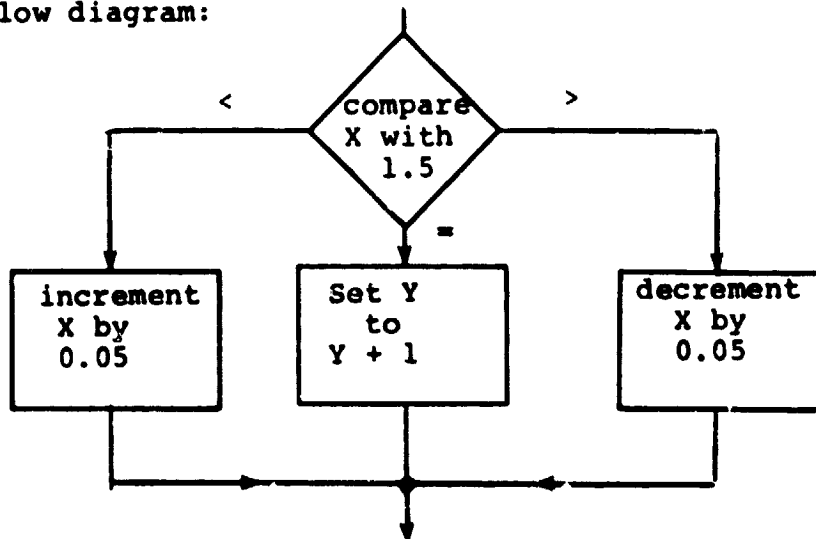
The Guide has stressed throughout that, according to structured programming principles, GO TO statements are inherently undesirable because they tend to disguise the program's flow of execution.

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.

Examples:

```
|          IF X > 1.5 THEN GO TO ALPHA;
|          IF X < 1.5 THEN TO TO BETA;
|          Y = Y + 1;
|          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:



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;
|
|   .
|   .
|   .
```

#### 9.4 SUMMARY

Section 9 has described conditional statements, labels, GO TO statements, and the ways in which they affect the flow of execution in a HAL/S program. Some attempt has been made to point out both the good and the bad ways of using these statements. Section 10 goes on to describe statement groups and how the usage of the constructs described in Sections 9 and 10 are very often interrelated in well-designed HAL/S programs.

## 10. 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>;                                                                                                                                                                                                                                                                                                                                                                                                                |
| ! |                                                                                                                                                                                                                                                                                                                                                                                                                              |
|   | <ol style="list-style-type: none"><li>1. &lt;control&gt; is a construct to be described. It specifies the manner in which the sequence of statements is to be executed.</li><li>2. &lt;control&gt; is optional. If it is absent, the sequence of statements is executed in its natural order* once only.</li><li>3. The DO statement is executable in that it may be labelled according to the Rules of Section 9.</li></ol> |

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

\* The "natural order" of execution was explained in Section 3.3.

## 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 labelled according to the Rules of Section 9.
2. <label> is optional: if present, the opening DO statement of the group must be labelled 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;
      |      I = I - 1;
      |      A = 0;
S     |      I
      |      END;
      |
      |
      |      FIX: DO FOR I = 1,25,16,2;
      |      A = -A ;
S     |      I      I
      |      END FIX;
      |
```

} opening DO statement  
} group of statements  
} closing END statement

} one statement in group  
} label specification in  
} 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 both assignments: one solution is -

```
IF S  $\neq$  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 <cond>;
```
1. <cond> 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 <cond> is TRUE.
  3. If the value of <cond> is FALSE, the stopping condition is satisfied. Execution proceeds to the statement following the END statement of the group.



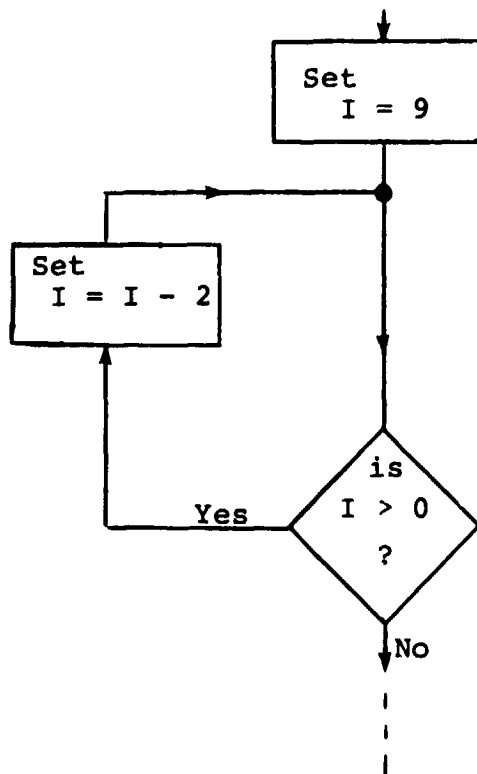
Examples:

```

: 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:



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 = 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;
| :
| :
```

---

There exists a variant of the DO WHILE statement called the DO UNTIL statement. Here execution of the group is assured at least once, whatever the value of the controlling expression. See: (tbd).

---

## THE DO FOR STATEMENT

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

```
DO FOR <var> = <init> TO <final> BY <inc>;
```

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. <init>, <final> and <inc> are integer or scalar expressions:
  - <init> is the initial value assigned to <var>.
  - <inc> 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 once prior to the first cycle of execution.

3. The stopping condition is met when the value of <var> lies outside the range bounded by <init> and <final>.
4. <inc> may be either positive or negative. The phrase

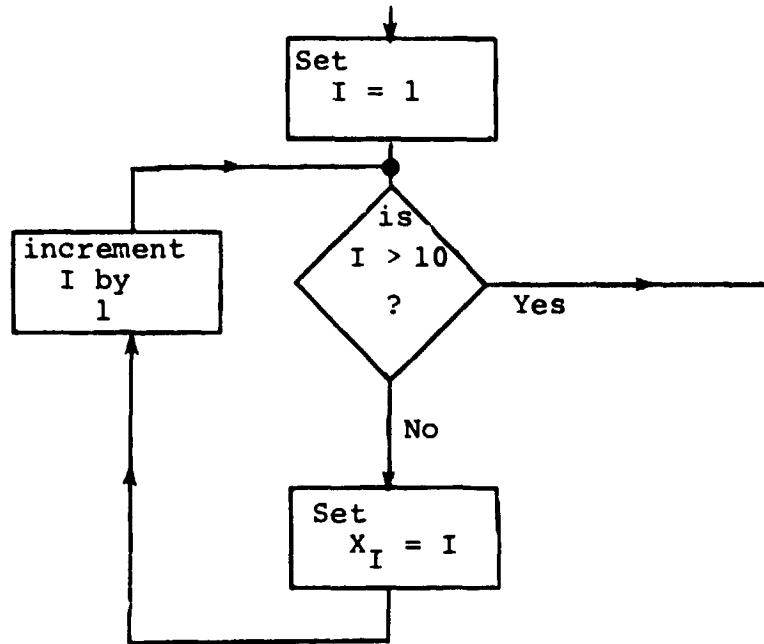
BY <inc>

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

Examples:

```
S DO FOR I = 1 TO 10;  
  X = I;  
  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:



```

: 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.

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

---

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>, <exp>, ... <exp>;
```

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.

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

---

### 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 <sup>th</sup> 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 is understood when it is realized that the selected statement may be a compound statement (i.e. it may itself be a statement group).

Example:

```
I = 3;
DO CASE I;
  X = 4;           case 1
  X = 3;           case 2
  DO;
    X = 7;         case 3
    Y = ;
  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 instead of an error being signalled, if the value of the case variable is outside the legal range for the statement group.  
See: (tbd).

---

## 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 respectively 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 only applies 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 <u>innermost</u> statement group in which it is enclosed.
2. Execution is directed to the first statement following the END of the group branched out of.

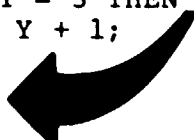
Examples:

```
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.

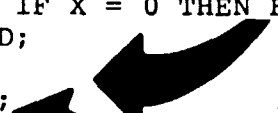
There exists a second form of the EXIT statement to allow branches out of other than the innermost statement group:

```
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;  
        I   I  
        IF X = I THEN EXIT ONE;  
        IF X = 0 THEN EXIT:  
      END;  
END;  
X = 0;
```

A diagram consisting of two thick black arrows. The first arrow starts at the 'EXIT ONE;' statement and points to the 'END;' statement that ends the 'DO WHILE' loop. The second arrow starts at the 'EXIT:' statement and points to the 'END;' statement that ends the 'DO FOR' loop. A dashed line indicates the path of the branch from the first EXIT statement to the outer loop's end.

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 <u>innermost</u> 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  $Y \equiv 1$  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  $X = 2$  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;
|       J   J
|       IF X = 25 THEN REPEAT;
|         J
|         IF X = 0 THEN REPEAT ONE;
|           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.

## 10.5 SUMMARY

Section 10 has explained how statements may be grouped together into compound statements, and how such groups may be executed repetitively or selectively.

At this point in the Guide, programs can be constructed using assignment statements, and controlling execution through conditional statements and statement groups.

The judicious use of procedures and user functions is essential to the well-ordered structure program . Section 11 thus goes on to describe how procedures and functions are defined and invoked.



10-22

INTERMETRICS INCORPORATED • 701 CONCORD AVENUE • CAMBRIDGE, MASSACHUSETTS 02138 • (617) 661-1840

## 11. FUNCTIONS AND PROCEDURES (TBD)

- 11.1 BLOCK DEFINITIONS (TBD)
- 11.2 PARAMETER LISTS (TBD)
- 11.3 PROCEDURE CALLING (TBD)
- 11.4 FUNCTION INVOCATION (TBD)
- 11.5 SUMMARY (TBD)

## 12. 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.

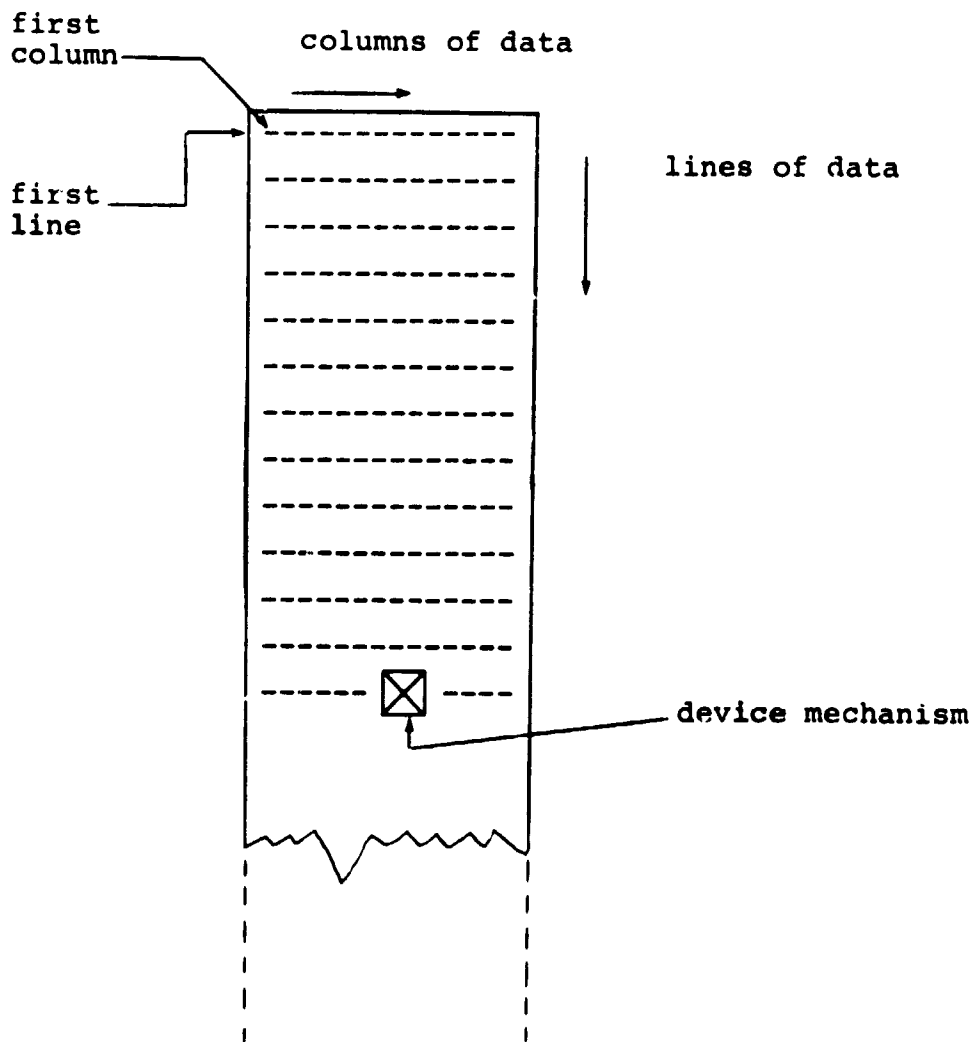
This section first introduces the HAL/S concept of sequential I/O and then goes on to describe the construction of I/O statements.

### 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:



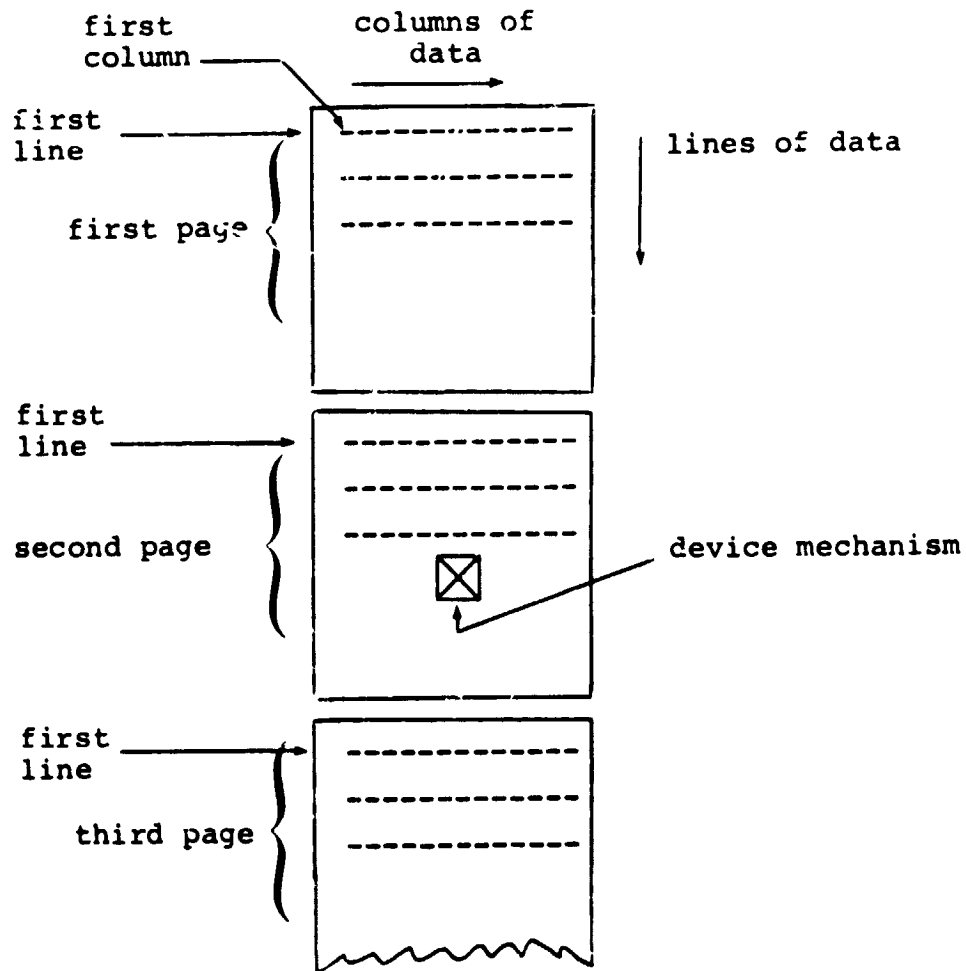
The device mechanism moves from column to column along each line, and from line to line as it performs I/O. Normally, the performance of I/O is accompanied by movement from left to right across each line, and downwards from one line to the next. However, special positioning commands can modify this behavior.

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 unpagged device is physically written so that it may, on some future occasion, be read in again via an unpagged 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 unpagged device, except that the lines of data are organized into "pages":



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 unpagged device. Such data cannot normally be read back again via an unpagged 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 unpagged I/O device. The form of the WRITE statement is as follows:

```
WRITE(n) <exp>, <exp>, ... <exp>;
```

1. n is the channel code number, and lies in the range  $0 \leq n \leq 9^*$ .
2. <exp> is any 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 Appendix I for exceptions.

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\*.
- 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.
- 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 of 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.

\* This value may be implementation dependent. Some implementations may allow the user to vary the value by a run-time option.

## 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 \_.

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:

- a  $l$ -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  $m \times n$  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  $3 \times 3$  matrix with  $M \equiv \begin{bmatrix} 0.5 & 1.5 & 0.0 \\ 2.5 & 1.0 & 1.0 \\ 0.5 & 0.1 & 10.1 \end{bmatrix}$

$I$  be a 3-array of integers  
with  $I \equiv (4 \ 6 \ -2)$

$C$  be a character with  $C \equiv \text{'VALUE'}$

$B$  be a Boolean with  $B \equiv \text{TRUE}$

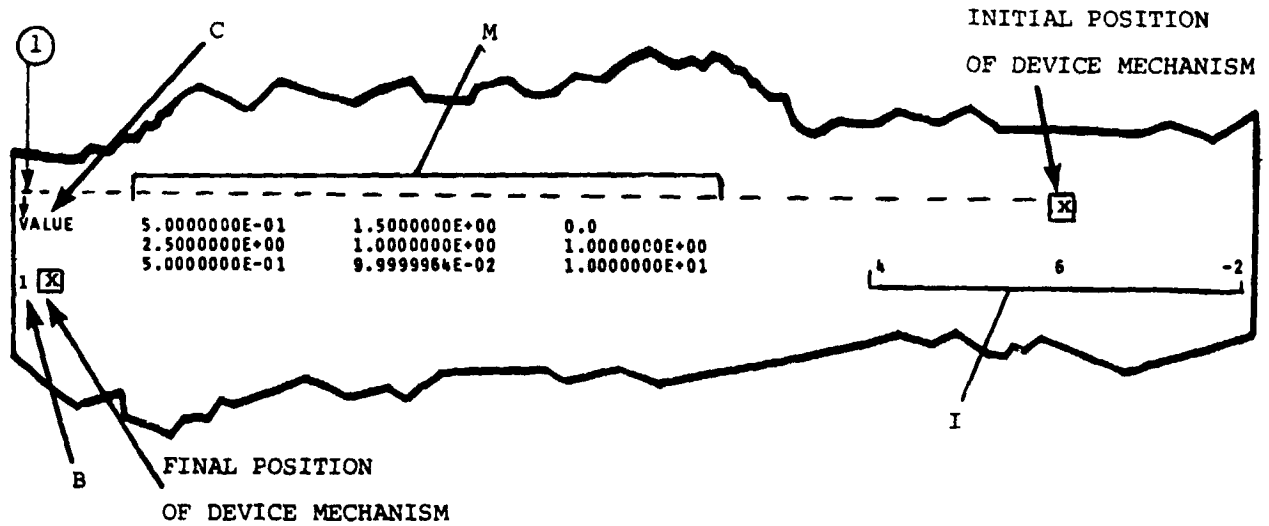
then

```
| WRITE(6) C,M,I;  
| WRITE(6) B;
```

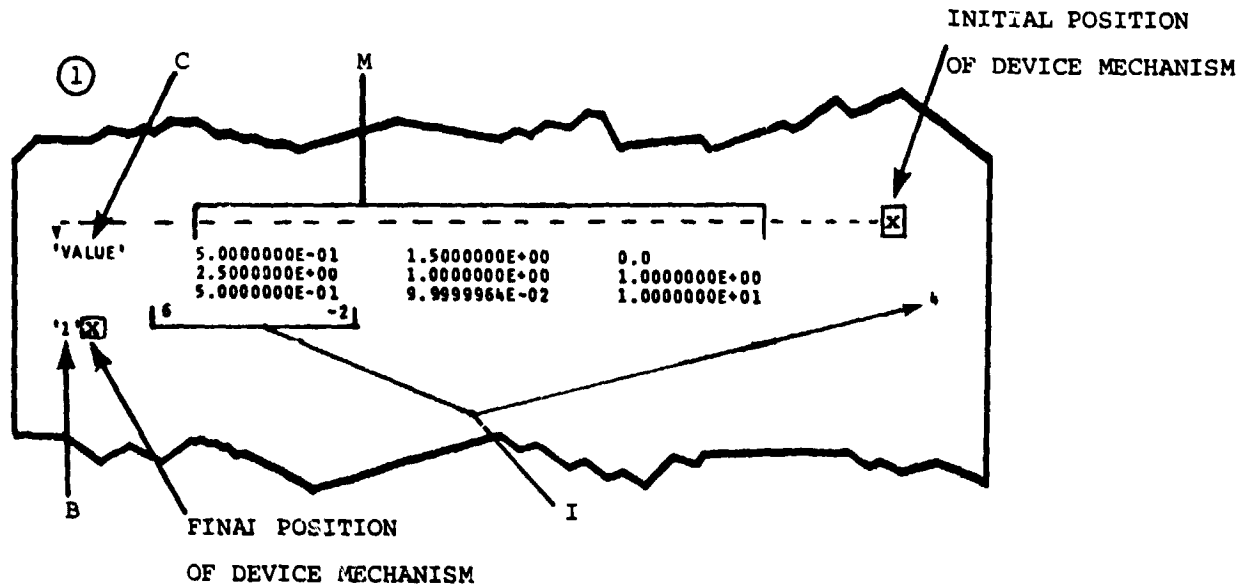
would result in output of the following form:



paged output: [132 columns/line]



unpaged output: [80 columns/line]



NOTES:

single precision scalar data fields are a fixed 14 columns wide.

single precision integer data fields are a fixed 11 columns wide.

## 12.3 THE READ STATEMENT

The READ statement is an executable statement for the input of data from an unpagged I/O device. The form of the READ statement is as follows:

```
READ(n) <var>, <var>, ... <var>;
```

1. n is the channel code number, and lies in the range  $0 \leq n \leq 9^*$ .
2. <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.

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.

\* This value may be implementation dependent. See Appendix \_ for exceptions.

- 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.

## 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 \_.

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,

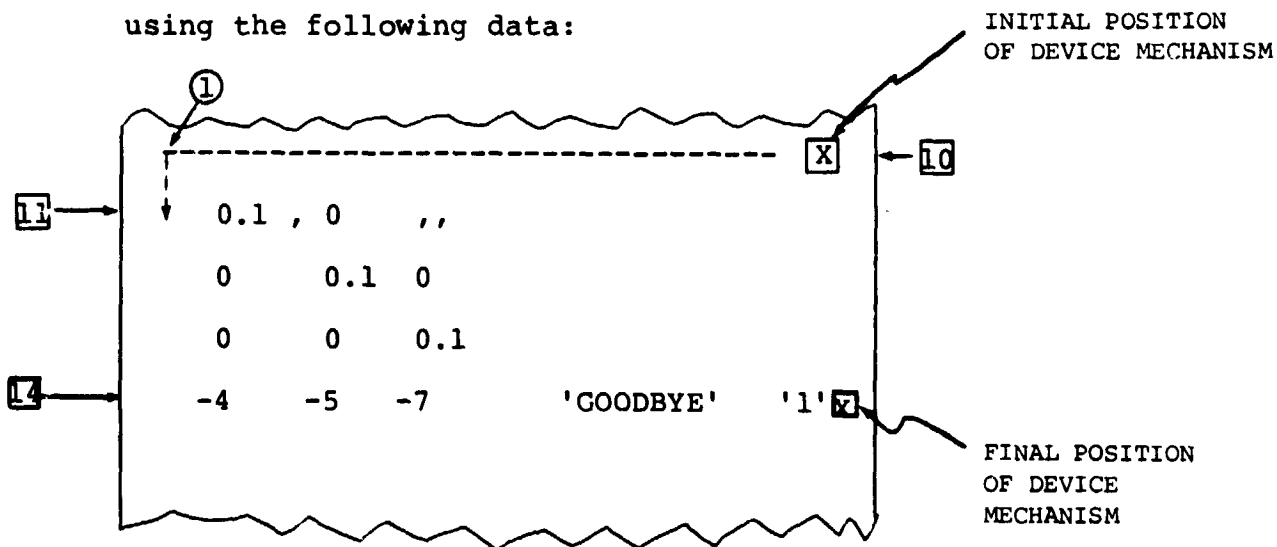
C be a character data item of maximum length 10,

B be a Boolean.

Then

```
READ(5) M,I,C;  
READ(5) B;
```

using the following data:



would result in:

$$M \equiv \begin{bmatrix} 0.1 & 0.0 & 0.0 \\ 0.0 & 0.1 & 0.0 \\ 0.0 & 0.0 & 0.1 \end{bmatrix} \leftarrow \text{this value not changed by READ statement.}$$

$$I \equiv (-4 \ -5 \ 1)$$

$$C \equiv \text{'GOODBYE'}$$

$$B \equiv \text{TRUE}$$

## 12.4 INPUT/OUTPUT FORMATTING

The formatting of I/O embraces two separate concepts:

- the shape of data fields;
- the position of data fields.

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.

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.

There is no direct capability in a READ or WRITE statement for defining different data field shapes. It should be noted however, that for output, the equivalent of arbitrary data field shaping can be achieved by using HAL/S's character string handling features.

---

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: (tbd)

---

## 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( $\alpha$ )  
COLUMN( $\beta$ )

1.  $\alpha$  and  $\beta$  are integer expressions.
2. TAB( $\alpha$ ) moves the device mechanism left or right by the number of columns specified by  $\alpha$ . Negative values of  $\alpha$  denote movement to the left; positive values, movement to the right.
3. COLUMN( $\beta$ ) moves the device mechanism left or right to the column indicated by  $\beta$ .
4. Values of  $\alpha$  or  $\beta$  must not be such as to try to move the device mechanism left past column 1, or right past the right-most column\*.

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.

\* The number of columns on any device (i.e. the logical record length) is assumed constant but implementation dependent. Its possible values may be found in the User's Manual for the implementation.

Example:

If C1, C2, C3 are character data items

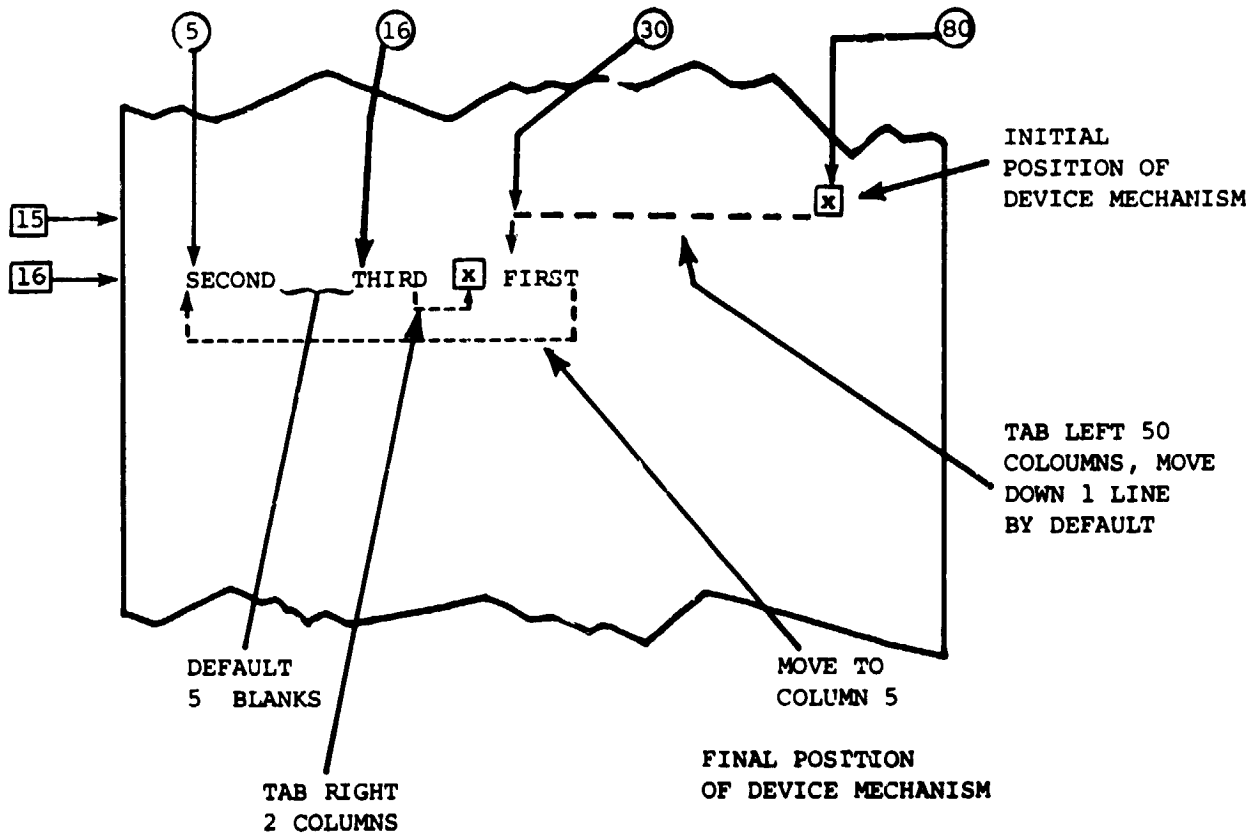
with C1 ≡ 'FIRST'  
C2 ≡ 'SECOND'  
C3 ≡ 'THIRD'

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:





## VERTICAL POSITIONING

The three pseudo-functions SKIP, PAGE, and LINE serve to position a device mechanism vertically. PAGE can only be used in I/O via a paged device; the behaviour of LINE is different depending on whether a device is paged or unpagged.

The form of the three pseudo-functions is as follows:

SKIP( $\alpha$ )  
PAGE( $\beta$ )  
LINE( $\gamma$ )

1.  $\alpha$ ,  $\beta$ , and  $\gamma$  are integer expressions.
2. SKIP( $\alpha$ ) moves the device mechanism downward by the number of lines specified by  $\alpha$ . The value of  $\alpha$  may be zero, in which case SKIP can suppress a default line advancement. However,  $\alpha$  may not be negative (indicating upwards movement). SKIPS over page boundaries are allowed.
3. PAGE( $\beta$ ) moves the device mechanism downward by the number of pages specified by  $\beta$ . As in SKIP,  $\beta$  may be negative in value. The relative line number remains unchanged.
4. For unpagged devices, LINE( $\gamma$ ) positions the device mechanism at line  $\gamma$ . The value of  $\gamma$  must not be such as to cause upwards movement of the device mechanism.
5. For paged devices, LINE( $\gamma$ ) has a different behaviour. Let the device mechanism be on line  $l$  prior to execution of LINE( $\gamma$ ). If  $\gamma < l$  then the device mechanism moves to line  $l$  on the next page. If  $\gamma \geq l$  then the device mechanism moves to line  $\gamma$  on the current page. The value of  $\gamma$  must lie in the range  $1 \leq \gamma \leq L$ , where  $L$  is the number of lines per page\*.

\* The number of lines per page is implementation dependent. Its value may be found in the User's Manual for a given implementation.

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

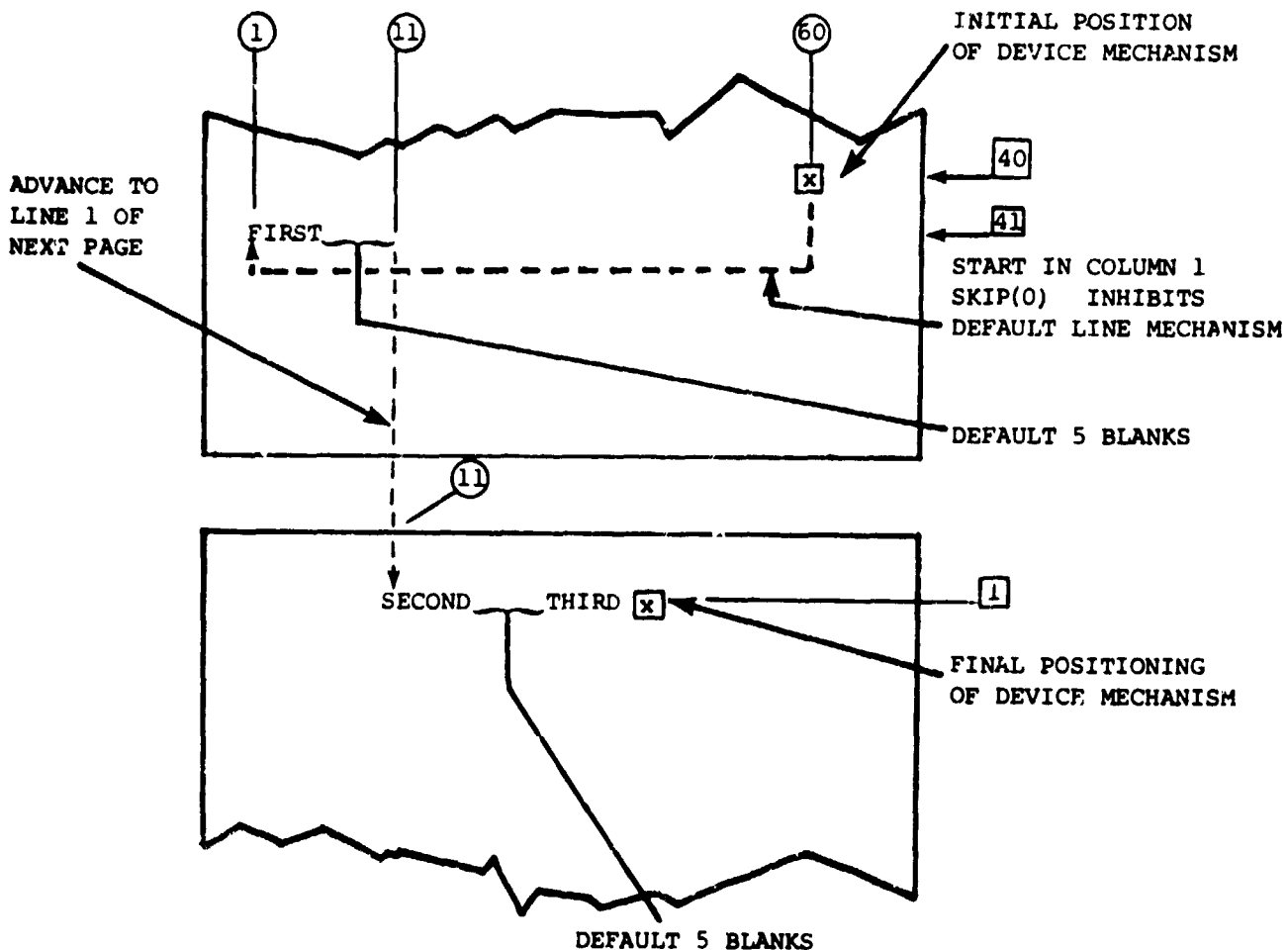
with C1 ≡ 'FIRST'  
 C2 ≡ 'SECOND'  
 C3 ≡ 'THIRD'

and if channel 6 is a paged device

then

```
WRITE(6) SKIP(0),C1,LINE(1),C2,C3;
```

produces output of the following form:

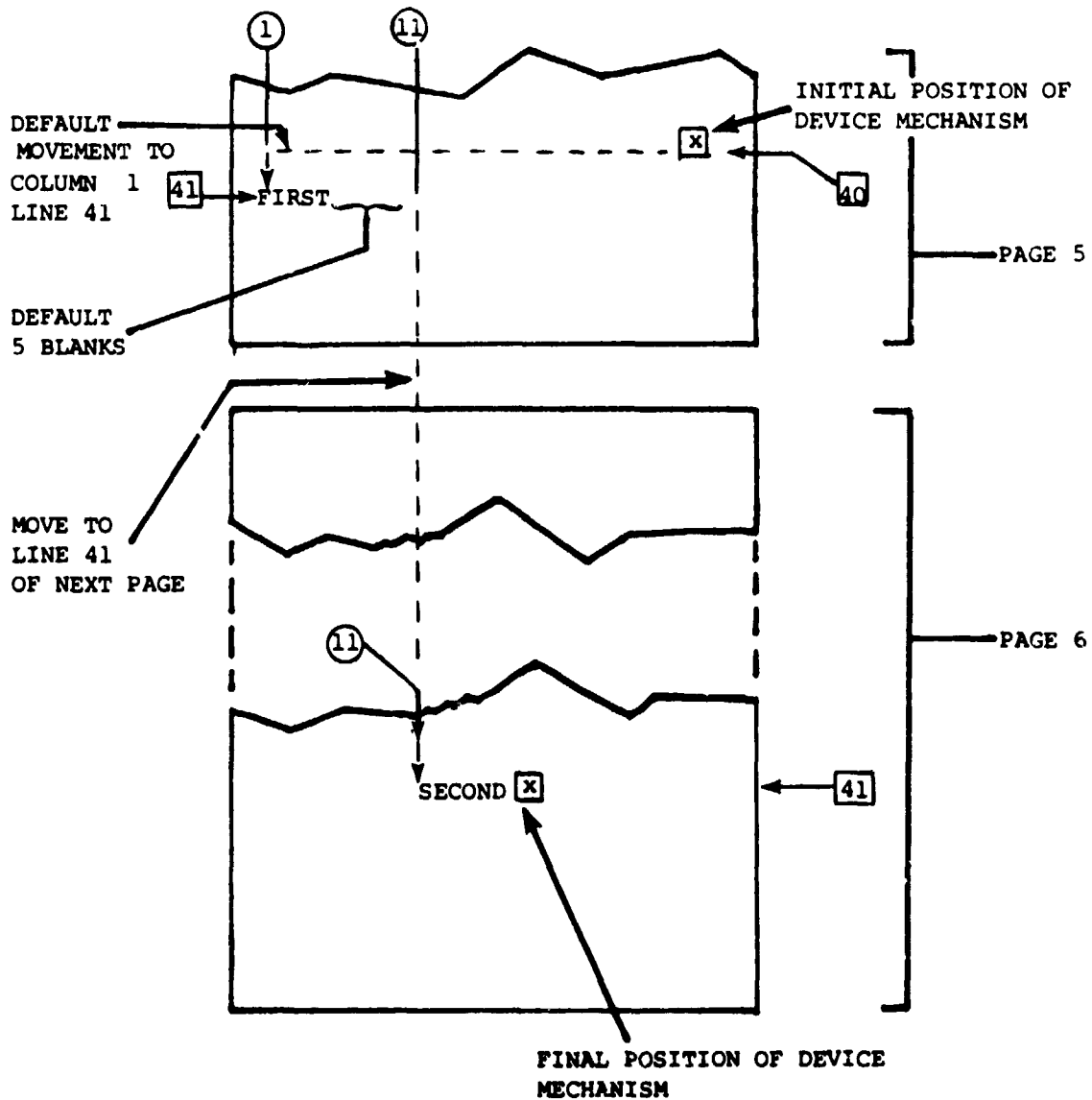


Note: If channel 6 were unpagged, the WRITE statement would be illegal since it would be calling for an upwards movement from line 40 to line 1.

Further,

```
WRITE(6) C1,PAGE(1),C2;
```

produces the output of the form:



## 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\*.

\* See the User's Manual for a given implementation.

## 12.6 SUMMARY

Section 12 of the Guide has described in detail the HAL/S constructs concerning sequential I/O, and has discussed the results of using different kinds of READ and WRITE statements. Section 13 introduces the user to the basic concepts involved in real time programming in HAL/S.

---

HAL/S contains a FILE statement by which random-access I/O may be effected.  
See: (tbd)

---

13. REAL TIME FEATURES OF HAL/S(TBD)

13.1 HAL/S REAL TIME CONCEPTS (TBD)

13.2 DEFINING TASKS (TBD)

13.3 SCHEDULING PROCESSES (TBD)

13.4 OTHER FEATURES (TBD)

13.5 SUMMARY (TBD)

14. SUMMARY OF PART 1

## INDEX

ABS	7-32
ABVAL	7-33
addition and subtraction	7-3, 7-4, 7-5
algebraic functions	7-33
arithmetic functions	7-32
arithmetic operations	7-1, 7-2
arithmetic precedence	7-23
array	4-2, 4-8, 12-6
array subscr.	6-8, 6-1
arrayed data types	6-8
array and component subscripting	6-10
ARRAY	4-8, 4-12
asterisk	6-6, 6-11
asterisk, in subs.	6-5
AT-	6-5
attributes	4-8
AT	6-6
assignment statement	7-1
augmented IF statement	9-4
blocks	1-2
block structure	1-2
built-in function names	2-2
Boolean	2-4, 2-7, 9-2, 4-7, 4-10, 4-11, 9-7, 9-12 10-5



Boolean	
d.type	4-2
operations	7-20
precedence	7-24
subs.	6-1
binary literal strings	2-7
branches	9-15
branching	10-15
built-in functions	7-32
catenation	7-18
character	4-7, 4-10, 4-11
data type	4-2
functions	7-34
operations	7-18
precedence	7-23
character string literals	2-4, 2-6
character set	2-1
character subscripts	6-1, 6-2, 6-3
channels	12-1, 12-4
class I operators	9-7, 9-10
class II operators	9-7
class II	9-8, 9-10
colon	
use of	6-10
use in Boolean data types	6-11
use of in arrayed subscript	6-8, 6-9
columns	12-1, 12-13
combining operations and precedence	7-23
combining comparative operations	9-12

comma	
in declarations	4-9
use in double precision	7-28
use of in Matrix double precision	7-30
use of	12-9
comments, HAL/S	2-10
comment lines	2-11
comparative operations	9-7
complement	7-20
compound statements	1-8, 10-13
compound declarations	4-9
component subscripting	6-1
component subscript	6-11
conditional statement	9-1, 10-1
<cond>	10-5
conjunction	7-21
control variables	10-5, 10-10
<control>	10-2
constant	4-10, 4-11
COS	7-33
crossproduct	7-7
data declarations	1-3, 4-1, 5-1
position of	3-2
data fields	12-4, 12-5, 12-6, 12-8, 12-11
data formats	12-6, 12-9
data referencing	6-1
data storage	12-4
data subscripting	6-1
data types	4-1

DATE	7-34
decimal notation	2-4
DECLARE	4-9
declaration statements	4-3, 4-10
delimiters	2-2
delimiting statement groups	10-1
device attributes	12-18
device mechanism	12-1, 12-2, 12-5, 12-8
device mechanism positioning	12-12
division	7-5
DIV	7-32
DO statement	10-2
DO CASE statement	10-13
DO CASE...ELSE	10-14
DO...END	10-4
DO FOR statement	10-5, 10-8
DO FOR	10-10, 10-18, 10-20
DO UNTIL statement	10-7
DO WHILE statement	10-5
DO WHILE	10-16, 10-18, 10-20
Dot Product	7-6
DOUBLE	4-3, 4-4, 4-5, 4-7, 7-26, 7-28, 7-30
double precision	4-2, 4-4
ELSE	9-1, 9-15
<else stmt>	9-4
END statement	10-3

error recovery	1-2
execution, path of	3-4
EXIT	10-17, 10-16
EXIT stmt	10-15
exponents	2-8, 2-9
exponentiation	7-13
expressions	7-1
EXPRESSION	7-1
<expression>	7-26
<exp>	7-27, 7-28, 7-29, 12-4
EXP	7-33
factored declaration	4-9
FALSE	9-1, 9-2, 9-4, 9-7
<final>	10-8
floating point	4-1
flow of execution	3-3
format	2-8
single line	I-1
multi line	I-1
fractional-valued literal	2-5
full word	4-3
function block	1-2, 1-3, 5-1
function name	1-4
GO TO statement	1-8, 9-17
GO TO	9-1, 10 15
GO TO statements	
elimination of	9-18
and statement groups	1-8
legal destinations	1-9, 1-10
block structure	1-10

halfword	4-3
horizontal positioning	12-12
identifiers	2-2, 2-3, 4-1, 4-2, 5-1
IF clause	9-1
IF statement	9-2
imbedded comment	5-1
implicitly-declared data items	4-1
<INC>	10-8
INITIAL	4-10, 4-11
<init>	10-8
initialization of data	4-10
I/O device	12-1
input/output formatting	12-11
input/output statements	12-1
integer data type	4-10, 4-11, 6-1, 7-1 4-1, 4-3
intersection	7-22
integral-valued literals	2-5
inversion	7-13
INVERSE	7-33
keywords	2-2, 4-3
<label>	3-1, 3-2, 9-15, 9-17, 10-3
label (<statement>)	9-2
labels	9-15
LENGTH	7-34
lines	12-1
LINE	12-15, 12-16
literals	2-1, 2-2, 2-4, 5-1

local data	1-4
LOG	7-33
matrix	4-5, 4-11, 4-12, 5-2, 6-1, 7-1
matrix conversion	7-29
matrix, data type	4-1
matrix subscripting	6-5, 6-6, 6-7
MAX	7-34
MIN	7-34
miscellaneous functions	7-34
multi-line format	2-9
multiplication	7-8, 7-9, 7-10, 7-11, 7-12
multiple exponents	2-5
multi-valued data items	4-10, 12-9
multi-valued data	4-11, 4-12
multi-valued expressions	12-6
<NAME>	4-3, 4-4, 4-5, 4-6, 4-7 4-8, 5-1
negation	7-2
nesting	1-2, 1-8
nested substitution	5-2
ODP	7-32
operators	2-2
order of initialization	4-12
overriding Precedence Order	7-25
output listings	2-1
PAGE	12-15, 12-16

paged I/O device	12-3, 12-4, 12-18
parenthesis	
use of in expressions	7-1
Boolean	7-25
partial initialization	4-13
precedence (relational)	9-13
precision conversion	7-17, 7-26, 9-11
procedure block	1-2, 1-3, 5-1
procedure name	1-4
program block	1-2, 3-1, 5-1
program block name	1-4
program closing	3-2
program opening	3-1
pseudo-functions	12-12
quotation marks	5-1
RANDOM	7-34
READ statement	12-8, 12-9, 12-10, 12-11 12-12, 12-13, 12-16, 12-18
REPEAT statement	10-15, 10-18, 10-19
real time control	1-2
recursion	1-4, 1-7
relational expressions	9-7, 9-12, 10-15
repetition (literal)	2-7
replace statements	5-1, 5-2
REPLACE	5-1, 5-2, 5-3
and block structure	5-3
replace parameters	5-4
reserved words	2-1, 2-2, 5-1
round	7-32

rounding	7-26, 10-10, 10-12
SCALAR	4-10, 4-11, 6-3
scalar, data type	4-1
scalar	4-4, 7-1
scalar subscripts	6-1
scoping	1-3
scoping of block names	1-4
sequence (Boolean)	7-24
sequence (precedence)	7-23
sequential I/O	12-1
semicolon, use of	2-10, 4-8
separators	2-2 (see special characters)
SIGN	7-32
SIN	7-33
SINGLE	4-3, 4-4, 4-5, 4-6, 7-26,
single line format	
single precision	4-2
SKIP	12-15, 12-16
source text	2-1, 2-8, 5-1
special characters	2-1, 2-2
SQRT	7-33
<statement>	9-2, 9-15
statement delimiting	2-10
statement grouping	1-3
statement groups	10-1
statement labels	1-10



stream-oriented I/O	12-4
structures	4-2, 6-12
structured programming	1-2
structuring	1-1
subroutines	1-1
subscripts	2-8, 2-9
subscripts of unarrayed data items	6-1
symbolic name	5-1
TAB	12-13
TAN	7-33
TO-	6-5
transpose	7-6, 7-13, 7-16
TRUE	9-1, 9-2, 9-4, 9-7
UNIT	7-37
unpaged I/O device	12-1, 12-2, 12-3, 12-8, 12-18
uni-valued data	4-10
uni-valued data items	12-9
uni-valued expressions	12-6
<value>	4-10, 4-11
<var>	10-8
vector, data type	
vector	4-6, 5-2, 6-1, 7-1
VECTOR	4-11, 4-12
vector conversion	7-2
vector - matrix functions	7-33
vector subscripts	6-3, 6-4

vertical positioning

12-15

well-bracketed

1-8, 10-1

WRITE statement

12-4, 12-5, 12-11, 12-12  
12-13, 12-16, 12-18