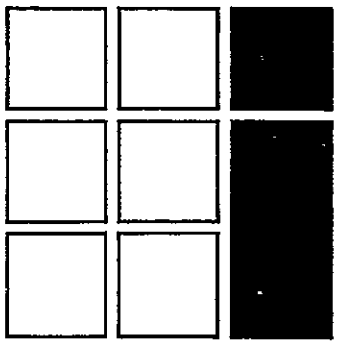


INTERMETRICS

CR 115121

FINAL REPORT
VOLUME II
A GUIDE TO THE
HAL
PROGRAMMING LANGUAGE
JUNE 1971

Submitted to:

National Aeronautics and Space Administration
Manned Spacecraft Center
Houston, Texas 77058

Prepared under Contract NAS 9-10542

INTERMETRICS, INC.
380 Green Street
Cambridge, Mass. 02139
(617) 868-1840

MANNED SPACECRAFT CENTER

NATIONAL AERONAUTICS AND SPACE ADMINISTRATION

HAL Guide

Errata Sheet

1. p. 1-5 First paragraph of Sec. 1.2 should be:

   The HAL Guide is divided into three parts. Part I presents an overall view of HAL and its essential elements; Part II defines a useful mathematical subset designated HAL_M; Part III completes the description of the full capabilities of the language in this implementation.

   The appendices .....

2. p. 3-1 The last sentence should be: Scalars and the elements of vectors and matrices are floating-point quantities.

3. p. 4-10 Example 4.2.1.1 a

   $\overline{X}$ = VECTOR.....

4. p. 4-14 $(x-p)^2 + (y-q)^2 = r^2$

5. p. 5-3 Example e)

   $[A]_{J \text{ TO } \#}$ instead of $[A_{J \text{ TO } \#}]$

6. p. 5-4 $(\overset{*}{E}_{1 \text{ TO } 3,\ 1 \text{ TO } 3})$

7. p. 5-5 (top of page)

   W = Y - 6;

   .... and W = Y - 6 will be executed.

8. p. 5-12 $A_0 = G[B_0 + \ldots$

(also)

READ(CARDS) G, N, ...

9. p. 5-13 AZERO = G(BZERO + ...

WRITE (LISTING) I, AZERO;

10. p. 6-8 ... polar form is me^{iQ}

should be

... polar form is $me^{i\phi}$

11. p. 6-8 • insert the following statement after the PHASOR PROCEDURE statement:

DECLARE PI CONSTANT(3.14159265);

• omit comment

PI IS A RESERVED HAL CONSTANT

12. p. 6-9 (near botton of page)

YINIT instead of YIMT

13. p. 7-4 (top of page)

...see Sec. 11.1.1.

(botton of page)

...see Sec. 11.1.2.

14. p. 7-5 (in PROGRAM A)

omit bar over Q

15. p. 7-10 reference should be to Sec. 11.2.1.2

16. p. 7-11 (within LIST)

insert SKIP(3), prior to each COLUMN( )

17. p. 9-5 (at botton of page)

omit (;) after $\{A\}_{35}$

18. p. 9-6 (at bottom of page)

At least one blank must separate structure level from identifier; i.e.,

2 B CHARACTER(10), etc.

19. p. 10-2 (in Example 2)

The first 7 bits of B ....

20. p. 10-7 (middle of page)

$\dot{D}\&\dot{E}$, $\dot{F}\&\dot{G}\&$ $\dot{H}$ ....

21. p. 10-10 (at top of page)

after the first END; insert

DO FOR I = 1 TO 12;

22. p. 10-13 Bars (-) should be placed over [P], VRESULT,[Q].

also,

BLOCK should be replaced by $\bar{V}$RESULT.

23. p. 10-21 (at bottom of page)

.... = $\dot{C}||\dot{E}$;

24. p. B-6 (within table)

$\bar{V}_m$ instead of $[\bar{V}]_m$

Foreword

This document is Volume II of the final report of a programming language development contract for advanced manned spacecraft. This effort was sponsored by the National Aeronautics and Space Administration's Manned Spacecraft Center, Houston, Texas under contract NAS 9-10542. It was performed by Intermetrics, Inc., Cambridge, Mass. under the technical direction of Mr. Daniel J. Lickly. The Technical Monitor for the Manned Spacecraft Center was initially Mr. John E. Williams and later was Mr. Jack R. Garman, FS/5.

PREFACE

This document is meant to serve as an introductory guide to the HAL programming language. The guide does not attempt to cover all the features of the language and is directed at the initial implementation of HAL on the IBM 360/75 at the Manned Spacecraft Center in Houston, Texas.

Complete specifications for HAL are given in "The Programming Language, HAL, - A Specification", Document #MSC-01846.

Table of Contents

Part III. General Capabilities

Part I.

AN INTRODUCTION TO HAL

1.0 Brief Description of HAL

HAL is a programming language developed by Intermetrics, Inc. for aerospace computer applications. It is intended to satisfy the requirements for both on-board and support software. The language contains features which provide for real-time control, vector-matrix and array data handling, and bit and character string manipulations.

1.1 The Basic Characteristics of HAL

1.1.1 Source Input/Source Listing

A singular feature of HAL is that it accepts source code in a multi-line format, corresponding to the natural notation of ordinary algebra. An equation which involves exponents and subscripts may be written, for example, as

$$C_I = (X\ A_J^2 + Y\ B_K^2)^{3/2}$$

instead of (as in FORTRAN or PL/1)

```
C(I) = (X*A(J)**2+Y*B(K)**2)**(3./2)
```

HAL also permits an optional single-line format; its construction is similar to the example above, with some minor changes; thus

```
C$I = (X A$J**2+Y B$K**2)**3/2
```

HAL source code may be input on cards or by data terminal. The input stream is free-form in that, for the most part, card or carriage column locations have no meaning; statements are separated simply by semi-colons.

In an effort to increase program reliability and promote HAL as a more direct communications medium between specifications and code, the HAL program listing is annotated with special marks. Vectors, matrices and arrays of data are instantly recognized by bars, stars and brackets. Thus, a vector becomes $\bar{V}$, a matrix $\overset{*}{M}$, and an array [A]. Further, bit strings appear with a dot, i.e., $\dot{B}$ and character strings with a comma, $\underset{,}{C}$. With these special marks as aids, the source listing is more easily understood and serves as an important step toward self-documentation. In addition to data marks, logical paragraphs, or blocks of code, are automatically indented so that dependence of one block on another may be seen clearly.

HAL is a higher-order language, designed to allow the programmer-analyst-engineer to communicate with the computer in a form which approximates 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.

1.1.2 Data Types and Computations

HAL provides facilities for manipulating a number of different data types. Arithmetic data may be declared as scalar, vector, matrix or integer (whole number). Individual bits may be treated as Boolean quantities or grouped together in strings. The language handles text by manipulating character strings via special instructions. Organizations of data-types may also be constructed; one-, two-, or three-dimensional arrays of any single type can be formulated, partitioned, and used in expressions. An hierarchical organization called a structure can be declared in which related data of different types may be stored and retrieved as a unit or by individual reference.

HAL requires that most data types be described explicitly; i.e., by declarations which assign a name and specify desired attributes. However, for scalars, 3-dimensional vectors, 3x3 matrices, and Booleans (1 bit bit-strings), the programmer can take advantage of HAL's implicit declarations and let the

compiler assign these variables appropriately.

The arithmetic data types together with the appropriate operators and built-in functions constitute a useful mathematical subset. HAL can be used directly as a "vector-matrix" language in implementing large portions of both on-board and support software. For example, a simplified equation of motion might appear as

$$\overline{A} = \overset{*}{B}\ \overline{ACC};$$

$$\overline{G} = -MU\ UNIT(\overline{R})/\overline{R}.\overline{R};$$

$$\overline{V}DOT = \overline{A} + \overline{G};$$

$$\overline{R}DOT = \overline{V};$$

where the matrix $\overset{*}{B}$ transforms acceleration from spacecraft to reference coordinates.

By combining data types within expressions and utilizing both implicit and explicit conversions from one type to another, HAL may be applied to a wide variety of problems with a powerful and versatile capability.

1.1.3 Real-Time Control

HAL is a real-time control language; that is, certain defined blocks of code called programs and tasks can be scheduled based on time or the occurrence of anticipated events. A limited subset of HAL's real-time capabilities will be included in the current implementation.

1.1.4 Program Reliability

Program reliability is enhanced when a software system can create effective isolation for various subsections of code as well as maintain and control commonly used data. HAL is a block-oriented language in that a block of code can be established with locally defined variables that cannot be altered by sections of program located outside the block. Independent programs can be compiled and run together with communication among the programs permitted through a centrally managed and highly visible data pool. For a real-time environment, HAL couples these precautions with a locking mechanism which can protect, by programmer directive, a block from being entered, a task from being initiated, and even an individual variable from being written into, until the lock is removed. (Locking is not included in this implementation.)

These measures cannot in themselves ensure total software reliability but HAL does offer the tools by which many anticipated problems, especially those prevalent in real-time control, can be isolated and solved.

1.2 Organization of The Guide

The HAL Guide is divided into four parts. Part I presents an overall view of HAL and its essential elements; Part II defines a useful mathematical subset designated HAL_M; Part III completes the description of the full capabilities of the language in this implementation; and Part IV discusses source code preparation and the HAL listing, aspects of 360/75 job control, and compile- and run-time diagnostics.

The appendices contain lists of keywords, built-in functions, features specifically dependent on the IBM 360/75 computer, and other information.

2. Language Elements

2.1 Program Structure

A HAL program consists of statements terminated by semicolons (;), groups of associated statements which are treated as a single statement (DO-groups), and blocks of statements organized as subroutines (procedures and functions). The statements and/or blocks must be compiled as a program unit, or as sets of independently compilable program units. Communication between programs is through a common data pool (COMPOOL) within a symbolic library.

2.2 Data

2.2.1 Data Declarations

In general all data types and organizations in HAL (i.e., scalar, integer, vector, matrix, bit and character string, array and structure) must be specifically declared by DECLARE statements. However, HAL does provide a subset of data which may be declared implicitly, i.e., simply by appearance in the program. Implicit data presumes certain default characteristics; e.g., vector and matrix dimensions.

2.2.2 Literals

A literal is a name which expresses its own value and is a constant during program execution. Literals can be numeric or string; e.g.,

```
12.6
 248          } - numeric literal
6.62E-2
OCT'7776'     - bit string literal
'HAL PROGRAM' - character string literal
```

2.3 Statements

In addition to the DECLARE statement, HAL statements provide for assigning expression results to variables, organizing statements in subroutines (procedures and functions), and controlling program logic flow. Control is accomplished through the IF-, GO TO -, and DO- statements. For a real-time control environment, HAL provides the commands to schedule programs and tasks through a real-time executive.

2.4 Input-Output

The HAL input-output statements READ, WRITE, and FILE facilitate the reading and writing of data and comments by identifying the external device (e.g., cards) and the quantities to be assigned or "displayed". Data may be in standard or non-standard formats. Statements are included to store and retrieve file data and to control printer page layout.

2.5 Source Code Preparation

The HAL program may be written in multi-line or single line format and loaded into the compiler on cards, data terminals

or other compatible devices. The multi-line format defines exponent, main, and subscript lines (E, M, S) and associates them into a single input stream. Comments may be included on any line by enclosing the comment within the symbol pairs; /* and */. Comments can also be introduced on comment lines (C-lines).

HAL is composed of five basic syntactical elements: identifiers, keywords, literals, special characters, and built-in functions. Complex syntactical units (i.e., statements) are constructed from these basic elements using a common set of input characters.

2.5.1 The Character Set

The characters which may be used in writing a HAL program are: the numerals 0 through 9, the letters A through Z, a blank character, and the following symbols.

=	(equals sign)	|	(OR symbol; also ¦)
+	(plus sign)	&	(ampersand)
-	(minus sign)	;	(semi-colon)
/	(slash)	:	(colon)
*	(asterisk)	.	(period)
<	(less than symbol)	,	(comma)
>	(greater than symbol)	'	(apostrophe)
¬	(not symbol; also ^)	(	(left parenthesis)

) (right parenthesis)

$ (dollar sign)

_ (break character)

(number sign)

@ (at sign)

[] (brackets)

{} (braces)

HAL will also accept other characters, restricting their use to within comments and character strings. Some examples are:

! (exclamation point)

% (percent sign)

? (question mark)

" (double quotation marks)

2.5.2 Identifiers

An identifier is a name which is assigned by the programmer to a data element, statement label, etc. Identifiers must satisfy the following rules:

a. The first character must be a letter.

b. It may contain 0 to 31 more characters, which may be any combination of letters, digits, or break characters, except that it must not end with a break character.

c. A qualified structure name (see Section 8.6.2.1) will contain imbedded periods and must not end in a period or break character.

d. An identifier may not be a compiler keyword.

Examples of valid identifiers:

```
A
R05
INTEGRATION_ROUTINE
SEXTANT_TO_NAVIGATION_BASE_MAT
STATE.COV_MATRIX
```

Examples of invalid identifiers:

1A	begins with a digit
SAMPLE_	ends in a break character
DECLARE	reserved word
POS VEC	contains a blank
STATEMENT_#200	contains a # character

2.5.3 Keywords

Keywords are words recognized by the compiler to have standard meanings within the language, and are usually unavailable for any other use; for example, operators, commands, attributes, and built-in function names. A list of HAL keywords is presented in Appendix A. Some examples are:

```
DECLARE
INTEGER
AND
VECTOR
SQRT
TRANSPOSE
```

2.5.4 Literals

See Section 2.2.2.

2.5.5 Special Characters

Special characters or combinations of characters are used in HAL as operators, separators, or other delimiters. These characters and some of their uses are described below.

2.5.5.1 Arithmetic Operators

Symbol	Definition
+	addition (or prefix plus)
-	subtraction (or prefix minus)
/	division (other uses also)
(see note below†)	multiplication
*	vector cross product (other uses also)
.	vector dot product (other uses also)
**	exponentiation (single-line)

† Note that HAL does not utilize a character as a multiplication operator. Instead a space (or spaces) between two distinct identifiers is interpreted as multiplication.

2.5.5.2 Relational Operators

Symbol	Definition
=	equal to
¬=	not equal to (or ^=)
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to
¬>	not greater than (or ^>)
¬<	not less than (or ^<)

2.5.5.3 String and Logical Operators

Symbol	Definition
AND (or &)	Boolean AND
OR (or |)	Boolean OR
NOT (or ¬ or ∧)	Boolean NOT
CAT (or ||)	Concatenation

2.5.5.4 Other Operators

Symbol	Definition
#	Indicates repetition within a list, or the last member of an array or string.
@	Scaling operator, or character to bit modifier
$	Subscript operator (single line format)

2.5.5.5 Separators

The following characters have meaning as separators in HAL:

Symbol		Definition
comma	,	(a) separates elements of a list; (b) separates indices in index expressions; (c) separates clauses in declare statements.
semicolon	;	(a) terminates statements; (b) separates structure indices from array element indices.
colon	:	(a) associates a statement label with the succeeding statement; (b) separates array element indices from sub element indices.
apostrophe	'	delimits string literal values (character or bit).
equals	=	indicates replace in assignment and DO FOR statements.
period	.	separates component names of qualified structures.
	/* */	encloses comments
	()	Parentheses have many uses in the language. They are used in expressions, for enclosing lists, function arguments, data dimension and initialization values, etc.

2.5.5.6 Built-in Function Names

Built-in function names are identified by the compiler as part of the language and are therefore keywords. A complete list of these functions appears in Appendix B. Some examples are:

ABS

TRUNCATE

COS

TAN

INVERSE

UNIT

2.5.5.7 Compiler-Generated Annotation

The following characters are used by the compiler to annotate the output of various data types in the language. Identical usage is also acceptable in the input stream.

Symbol	Definition
*	Over a name denotes a matrix
-	Over a name denotes a vector
.	Over a name denotes a bit string
'	Over a name denotes a character string
[ ]	Denotes an array organization
{ }	Denotes a structure organization

Part II

HAL_M

----A MATHEMATICAL SUBSET OF HAL

3. Introduction to HAL_M

HAL_M defines a useful mathematical subset of HAL. It is primarily directed at the programmer-analyst who wishes to gain rapid facility with HAL and use it in the solution of engineering problems as he would use FORTRAN or ALGOL. As such HAL_M concentrates on:

1) scalar, vector and matrix arithmetic.
2) simplified data declarations, arrays, and input-output commands.

and neglects:

1) bit- and character-string variables and manipulations (except for I/O where necessary).
2) complicated data arrays and structures.
3) real-time control and data-locking.

HAL_M is not a formal language subset; "full-HAL" statements can be freely mixed in HAL_M if so desired. However, the compiler implementation is such that programs written completely in HAL_M insure the greatest degree of machine independence and transferability.

3.1 Data Types and Data Declarations

3.1.1 Types

Three data-types are included: scalar, vector, and matrix. Scalars and the elements of vectors and matrices are floating-point single precision quantities.

3.1.2 Data Organizations

One-, two-, and three-dimensional arrays of scalars may be declared.

3.1.3 Literals

3.1.3.1 Arithmetic

Arithmetic literals are written as a series of decimal digits with an optional decimal point. The literal may contain powers of 10, 2 and/or 16, represented by E, B, and H respectively. The following are some acceptable forms of arithmetic literals (all are equivalent to the value, 6):

6, 6.0, +600E-2, 0.006E3, 12B-1, .12E+2B-1

3.1.3.2 Characters

Character literals are useful for messages, headings, etc. The simplified form is to enclose text; i.e., letters, digits, symbols, and blanks within quote marks. Thus,

'NAUTICAL MILES'

'ERROR_106'

'GO BACK, TRY AGAIN!'

are examples of character literals.

3.1.4 Declarations

HAL permits the implicit declaration of scalars, vectors, and matrices by their first appearance in the program listing. A "bar" (-); i.e., minus sign, on the E-line over an identifier denotes a vector; a "star" (*); i.e., asterisk, denotes a matrix; and the absence of any marks above an identifier means the quantity is a scalar. Once marked, the programmer need not continue to supply notation in the source code. The compiler will annotate

the output listing appropriately. The implicit declaration of vectors and matrices always results in default dimensions. The standard defaults are 3-dimensional vectors, and 3x3 matrices.

The following statements would suffice to declare the scalars A, B, the vectors V, W, Y and the matrix M.

$$\overline{W} = \overset{*}{M}\ \overline{V};$$

$$\overline{V} = A\ \overline{V} + B\ \overline{W} * \overline{Y};$$

3.1.4.1 DECLARE Statements

Three data DECLARE statements are necessary within HAL_M. These statements allow specification of vector and matrix dimensions (if defaults are not adequate) as well as the declaration of an array of scalars. Some examples are:

a. DECLARE V VECTOR (6);

   DECLARE VECTOR(8) T,U,V,W;

   The desired vector dimension must appear as an integer literal within parenthesis. The second declaration illustrates the factored DECLARE statement where T, U, V, W are declared in one statement.

b. DECLARE M MATRIX (4,4);

   DECLARE MATRIX(3,6) M,N,P,Q;

   The desired matrix dimensions must appear as integer literals within parentheses; i.e., (rows, columns). The second declaration is a factored DECLARE statement.

c. DECLARE A ARRAY(3);

   DECLARE ARRAY(2,3) A,B;

   DECLARE C ARRAY (2,3,4);

   The desired array shape must appear as integer literals within parentheses, Arrays may be one-, two-, or three-dimensional and consist of scalar elements.

3.1.4.2 Arrays, Vectors, Matrices

One-dimensional arrays are not vectors. Two-dimensional arrays are not matrices. That is, arrays obey sequential element-by-element operations and not vector-matrix arithmetic. Thus, for example,

```
DECLARE ARRAY(2,2) A,B,C;

[C] = [A] .[B];
```

is executed in the following order:

$$C_{11} = A_{11}\ B_{11};$$
$$C_{12} = A_{12}\ B_{12};$$
$$C_{21} = A_{21}\ B_{21};$$
$$C_{22} = A_{22}\ B_{22};$$

4. Arithmetic Expressions, Assignments and Control

4.1 Arithmetic Expressions

HAL_M contains three types of arithmetic operands:

Scalar	denoted S
Vector	denoted $\bar{V}$
Matrix	denoted $\overset{*}{M}$

An arithmetic expression is any meaningful arrangement of operators and operands, where parentheses may be freely used as in ordinary mathematical notation to specify the grouping or ordering of operations. An arithmetic expression is a string of arithmetic operations which, when evaluated, results in a a scalar, vector, or matrix.

4.1.1 The Order of Operations

In the evaluation of an expression, the order of operations is determined by parentheses and operator precedence. Operations within an expression are performed in the order of decreasing priority. For example, in the expression A+B**3, exponentiation is performed before addition. If an expression involves operations of the same priority, the general rule is that the operations are performed left to right.

If an expression is enclosed in parentheses, it is treated as a single operand. The parenthesized expression is evaluated before its associated operation is performed. For example, in the expression (A/B)C, A is divided by B and then the result is multiplied by C. Thus, parentheses modify the normal rules of priority.

The following chart illustrates all the possible arithmetic operations in HAL_M, as well as each one's priority, operand types, and value or result.

Summary of HAL Arithmetic Operations

Operation	Priority[1]	Form[2]	Results
Exponentiation	6	S^S	scalar
Matrix transpose (short form)	6	$\overset{*}{M}^T$	matrix
Matrix inverse (short form)	6	$\overset{*}{M}^{-1}$	matrix
Scalar-scalar product	5	S S	scalar
Scalar-vector or vector-scalar product	5	S $\overline{V}$ or $\overline{V}$ S	vector
Scalar-matrix or matrix-scalar product	5	S $\overset{*}{M}$ or $\overset{*}{M}$ S	matrix
Vector-matrix product	5	$\overline{V}$ $\overset{*}{M}$	vector
Matrix-vector product	5	$\overset{*}{M}$ $\overline{V}$	vector
Vector outer product	5	$\overline{V}$ $\overline{V}$	matrix
Matrix-matrix product	5	$\overset{*}{M}$ $\overset{*}{M}$	matrix
Vector cross product	4	$\overline{V}$ * $\overline{V}$	vector
Vector inner (dot) product	3	$\overline{V}$. $\overline{V}$	scalar
Scalar-scalar quotient	2	S / S	scalar
Vector-scalar quotient	2	$\overline{V}$ / S	vector

1. Higher priorities have larger numerical values.
2. S, $\overline{V}$, $\overset{*}{M}$ are general scalar, vector, matrix operands - the symbols represent operand type rather than value.

Operation (cont'd)	Priority	Form	Results
Matrix-scalar quotient	2	M / S	matrix
Scalar sum or difference	1	$S \pm S$	scalar
Vector sum or difference	1	$\overline{V} \pm \overline{V}$	vector
Matrix sum or difference	1	$\overset{*}{M} \pm \overset{*}{M}$	matrix

4.1.1.1 Some Exceptions

1. Exponentiation is right-to-left. Thus,

   a) $A**B**C \equiv A^{B^C} \equiv A**(B**C)$

   b) $SIN(X)^2 = (SIN(X))^2$, and not $SIN(X^2)$

2. Division is right-to-left. However, vector and matrix expressions may never appear as denominators in a quotient.

   a) $A/B/C \equiv A/(B/C) \equiv A\ C/B$

   b) $A/B\ X/C\ Y/D \equiv A/(B\ X/(C\ Y/D)) \equiv A\ C\ Y/B\ X\ D$

   c) $\overline{V}/A/B = \overline{V}/(A/B) \equiv B\ \overline{V}/A$

   d) $\overline{V}/A/\overline{R}$ is illegal

   e) $(\overline{V}/A)\overline{R}$ is OK

   f) $\overline{V}/\overline{R}.\overline{V}$ is OK

(See HAL specification document for more detail on exceptions.)

4.1.1.2 Product Operands

Note that in the product forms

$$S\ S \quad S\ \overline{V} \quad S\ \overset{*}{M} \quad \overline{V}\ \overset{*}{M} \quad \overline{V}\ \overline{V}$$

$$\overset{*}{M}\ \overset{*}{M} \quad \overline{V}\ S \quad \overset{*}{M}\ S \quad \overset{*}{M}\ \overline{V}$$

the multiply operator is implied from the "logical adjacency" of the operands. In HAL all such operands must be easily distinguishable.

4.1.1.3 Sum and Difference

The operands must agree in the number of scalar components as well as in type (S, $\overline{V}$ or $\overset{*}{M}$). In the form $\overline{V} \pm \overline{V}$, the vector operands must be of the same length. In the form $\overset{*}{M} \pm \overset{*}{M}$, the matrix operands must have the same row-column dimensionality.

4.1.1.4 Vector Cross Product

The operation, $\overline{V} * \overline{V}$, is defined in HAL only for vector operands of length three (3).

4.1.1.5 Vector-Matrix Products

a) $\overline{V}.\overline{V}$ Vector inner (dot) product is computed as

[1 x n] [n x 1] → [1 x 1],

a scalar.

Note: In arithmetic expressions, the character "." will be taken as a part of a literal if the context allows this interpretation. Thus, for example, $\overline{U}$ .2 $\overline{V}$ is interpreted as $\overline{U}$(0.2 $\overline{V}$) and not as $\overline{U}$. (2 $\overline{V}$).

b) $\overline{V}\ \overline{V}$ Vector outer product is computed as

[n x 1] [1 x m] → [n x m],

an n x m matrix.

c) $\overline{V}\ \overset{*}{M}$ Vector-matrix product is computed as

[1 x m] [m x n] → [1 x n]

an n-dimensional vector.

d) $\overset{*}{M}\ \overline{V}$ Matrix-vector product is computed as

[m x n]]n x 1] → [m x 1]

an m-dimensional vector.

e) $\overset{*}{M}\ \overset{*}{M}$ Matrix-matrix product is computed as

[m x n] [n x p] → [m x p],

an m x p matrix.

4.1.2 Some Examples of Arithmetic Expressions.

	MATHEMATICAL NOTATION	HAL EXPRESSION
1.	ab	A B
2.	$a(-b)$	A(-B) or -A B
3.	$-(a + b)$	-(A + B)
4.	a^{x+2}	A^{X+2}
5.	$a^{x+2}\ c$	A^{X+2} C
6.	ab/cd	A B/C D
7.	$(\frac{a+b}{c})^{2.5}$	((A+B)/C)**2.5
8.	$\cfrac{a}{1+\cfrac{b}{(2.7 + c)}}$	A/(1 + B/(2.7 + C))
9.	$(\underline{v}^T\underline{y})M^{-1}(\underline{v}+\underline{y})$	$(\overline{V}.\overline{Y})\overset{*}{M}^{-1}(\overline{V}+\overline{Y})$
10.	$a(\underline{y}\ \underline{v}^T)^T\ (\underline{v} \times \underline{w})$	$A(\overline{Y}\ \overline{V})^T(\overline{V} * \overline{W})$

4.1.3 Characters and Arithmetic Expressions. Character strings may be joined and/or combined with arithmetic expressions to produce messages or data printout, using the concatenation operator (CAT or ||). Thus,

'NAUTICAL'||' '||'MILES' becomes NAUTICAL MILES

and

X^2 + 5||'N.M.' becomes (value) N.M.

where (value) is the numerical value of X^2 + 5.

Note that, in effect, concatenation converts numerical data to characters. These "character expressions" can then be output using the WRITE I/O command.

4.1.4 Array Expressions. Within HAL_M only arrays of scalars are considered. Array expressions yield array results and may involve a combination of scalar and array operands. Scalar-scalar product, quotient and sum or difference are defined as array operations where at least one operand is an array. The indicated operation is performed on a sequential element-by-element basis.

4.1.4.1 Some Examples.

let $[A] = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$ $[B] = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$

then

a) $[A]/5 = \begin{bmatrix} 1/5 & 2/5 \\ 3/5 & 4/5 \end{bmatrix}$

b) $[A]/[B] = \begin{bmatrix} 1/5 & 2/6 \\ 3/7 & 4/8 \end{bmatrix}$

c) $[B] + 10 = \begin{bmatrix} 15 & 16 \\ 17 & 18 \end{bmatrix}$

d) $[A]^{[B]} = \begin{bmatrix} 1^5 & 2^6 \\ 3^7 & 4^8 \end{bmatrix}$

4.2 Assignment Statements

A HAL statement is an order to perform some action, and a HAL program is composed of a series of statements of various kinds. The fundamental statement is the assignment statement, which assigns a value to one or more variables. A simple assignment statement takes the form

```
Label:  Variable = Expression;
```

where Label: is optional. A single assignment statement can set several variables; e.g.

```
ABLE: A,B,C = 5;
```

or

```
* * * *
M, N, P = Q;
```

Note that, in general, the dimensions of the left side variables and the right side expressions must be the same. Exceptions are made for "zeroing" and assigning arrays. Thus,

```
A = 0;
B = 0;
*
C = 0;
[D] = 0;
```

are all valid "zeroing" statements in HAL. In the case of an array, a scalar expression assigns the scalar value to every element of the array; e.g.

```
[A] = 5;
```

or

```
[B] = X**2 + Y;
```

If there is more than one left side variable then the array dimensions of all must be identical.

4.2.1 Conversion Functions

It may be convenient to form a vector, matrix, or array of scalars, from its components. In HAL_M, three conversion functions are provided: VECTOR(list), MATRIX(list), SCALAR (list). These functions may be applied to mixed lists of scalars, vectors, matrices and arrays. The functions may be thought of as constructing a one-dimensional list of all the included elements. Vector, matrix, and array list-elements are equivalent to lists of their components. Matrices are unraveled by incrementing "right-most" index first (i.e., 1,1,1; 1,1,2; 1,1,3;...1,2,1; 1,2,2; etc.).

The resulting vector, matrix, or array is filled, respectively, element-by-element from the list.

a) If the list consists of only one scalar, all the elements will be assigned a value equal to this single list element. The desired dimensions (or shape) are indicated by subscripts. Thus, for example,

$$VECTOR_6(0)$$

$$MATRIX_{2,3}(5)$$

$$SCALAR_{3,3,3}(A)$$

The default dimensions will be applied to VECTOR and MATRIX if subscripts are not supplied.

b) If the list consists of one vector, one matrix or one array the resulting forms can be quite complex. See Sec. 9.2, Appendix F, and Sec. 6 of the HAL specification document.

c) If the list consists of several entries, VECTOR and SCALAR lengths will be equal to the number of elements in the list. MATRIX row and columns will be equal to the square root of the number of list elements (presuming an integral value) unless otherwise specified. The number of list entries must be compatible, thus

$$\text{VECTOR}(A^2, B^2, C^2, D, 6\#E)$$

$$\text{MATRIX}(\overline{X}, \overline{Y}, \overline{Z})$$

$$\text{SCALAR}(A, \overline{X}, \overset{*}{M})$$

Note: 6#E, indicates six entries of the quantity E (i.e., E,E,E,E,E,E).

4.2.1.1 Some Examples

a) $\dot{X}$ = VECTOR(1,A,0)* VECTOR(1)

b) DECLARE P ARRAY(15);

LOOK: [P] = SCALAR_{15}(5#A,5#B,5#C);

c) $\overline{RE}$ = $\text{MATRIX}_{3,3}(\overline{X},\overline{Y},\overline{Z})\overline{RS}$;

4.3 Control Statements and Relational Expressions

4.3.1 The GO TO Statement

The GO TO statement in HAL is a simple unconditional transfer to a labelled statement. The general form is:

GO TO L;

where L is the label of a statement elsewhere in the program. It specifies that the statement to be executed next is the one identified by the label and that control is to be transferred to that point in the program.

4.3.2 The IF Statement. The IF statement provides the capability to change the sequence of statement execution on the basis of what happens during execution of the program. The general forms are:

IF L_c THEN S

or

IF L_c THEN B ELSE S

where L_c denotes a logical condition or set of logical conditions, S may be any executable statement except END. B may be any executable statement except IF or END. For example:

```
IF X<5 THEN GO TO AGAIN;
```

or

```
IF X<5 THEN ABLE: GO TO AGAIN;
   ELSE IF X<10 THEN GO TO TRY;
```

The IF statement format requires that an ELSE be preceded by an IF and not by another ELSE. As a result, the execution of a statement following ELSE occurs only if the logical condition associated with the nearest preceding IF is false.

4.3.3 Logical Conditions. A logical condition may be expressed as an arithmetic comparison expression, for example

IF $\overset{*}{M} = \overset{*}{N}$ THEN . . .

or

IF X<2.064 10^8 Y THEN . . .

or

IF A+B = $\overline{V}.\overline{W}$ THEN . . .

4.3.3.1 Comparison Expressions. Within HAL_M, scalar quantities may be compared using the following relational operators.

Symbol	Operation
=	equal
¬=	not equal
<	less than
>	greater than
<=	less than or equal
>=	greater than or equal
¬<	not less than
¬>	not greater than

Vectors, matrices and arrays are restricted to

Symbol	Operation
=	equal
¬=	not equal

For the operator =, the comparison is true if and only if all the corresponding elements are equal. For the operator ¬=, the comparison is true if and only if any of the corresponding elements are not equal.

4.3.3.2 Sets of Logical Conditions. Logical conditions can be combined, using the logical operators NOT (¬,^), AND (&) and OR (|), into complex sets of logical conditions; the final result of any condition or set of conditions must be a single true or false answer. Thus, for example,

```
IF X>5 AND Y<A AND M*=N* THEN . . .
IF NOT (X<=C OR X>=C + DELTA) THEN . . .
```

4.3.3.3 The Order of Operations. In order to avoid ambiguity, the following rules are established when evaluating logical sets of relational expressions.

a. NOT (¬,^) must be followed by a relational expression (or set of expressions) within parentheses.

b. Relational expressions are evaluated before AND and OR.

c. AND is applied before OR.

Thus:

1. A>5 AND B>6 means (A>5) AND (B>6)

2. NOT (A>5 AND B>6 OR C<7)

   means NOT ((A>5 AND B>6) OR C<7)

4.4 Examples - I

In this section two examples illustrate the use of the HAL_M as presented thus far.

4.4.1 INTERSECTIONS

Find the intersections defined by the line,

$$ax + by + c = 0$$

and the circle,

$$(x - p)^2 + (y - q)^2 = p^2$$

where a, b, c, p, q, r are parameters. (This problem is adapted from An Introduction to ALGOL 60, C. Anderson, Addison-Wesley, 1964)

HAL$_M$ Program

```
INTERSECTIONS: PROGRAM;
    READ(CARDS) A,B,C,P,Q,R;
C   COMPUTE DISTANCE FROM LINE TO CENTER OF CIRCLE
C   CAN BE DERIVED BY TRANSLATING ORIGIN TO (P, Q)
    F = A^2 + B^2;
    D = (A P + B Q + C)/SQRT(F);
    IF D^2>R^2 THEN GO TO NO_SOLVE; /*NO SOLUTION*/
    Z = (A^2 Q - A B P - B C)/F;
    U = (B^2 P - A B Q - A C)/F;

C   WHERE (U, Z) IS THE POINT OF INTERSECTION BETWEEN
C   THE GIVEN LINE AND AN ORTHOGONAL LINE THROUGH (P, Q)
    DELTAX = -B SQRT (R^2 - D^2)/SQRT(F);
    DELTAY = (A/B) DELTAX;
    X1 = U - DELTAX; /*INTERSECTION #1*/
    Y1 = Z - DELTAY;
```

```
        X2 = U + DELTAX;  /*INTERSECTION #2*/
        Y2 = Z + DELTAY;
    WRITE (LISTING)  'X1 = '||X1, 'Y1 = '||Y1, 'X2 = '||X2,
                     'Y2 = '||Y2;
        GO TO FINISH;
     NO_SOLVE: WRITE(LISTING) 'NO SOLUTION';
     FINISH: CLOSE INTERSECTIONS;
```

4.4.2 TRANSFORM

Given the three-dimensional vectors $\underline{w}$, $\underline{x}$, $\underline{y}$, $\underline{z}$, form an orthonormal coordinate set from $\underline{w}$, $\underline{x}$, $\underline{y}$ and express $\underline{z}$ on this set.

HAL_M Program

```
                TRANSFORM: PROGRAM;
                READ (CARDS) W, X, Y, Z;
C   USE GRAM-SCHMIDT TO FIND ORTHONORMAL SET
        A1 = UNIT(W);
        A2 = UNIT(X - (X.A1)A1);
        A3 = UNIT(Y - (Y.A1)A1 - (Y.A2)A2);
        M* = MATRIX(A1, A2, A3);  /*TRANSFORMATION MATRIX*/
        ZNEW = M* Z;
C   FIRST ZOLD, THEN ZNEW
    WRITE(LISTING) Z, ZNEW;
                        CLOSE TRANSFORM;
```

5. Subscripted Variables and DO Statements

5.1 Subscripts

The elements of vectors, matrices and arrays within HAL_M may be referenced by appropriate subscripting.

The first component of a vector or a one-dimensional array, is given the subscript 1, the second 2, etc. up to the total number of elements. Thus for a 9 element vector, i.e.,

```
DECLARE V VECTOR(9);
```

the components may be written as,

$$V_1 \; V_2 \; V_3 \cdots V_9.$$

A matrix or two-dimensional array may be thought of as being composed of horizontal rows and vertical columns. The first of the two subscripts refers to the row number, running from 1 up to the number of rows, and the second to the column number, running from 1 up to the number of columns. For instance, a matrix of two rows and three columns would require the declaration

```
DECLARE B MATRIX(2,3);
```

and the elements could be referred to by writing:

$$B_{1,1} \; B_{1,2} \; B_{1,3} \; B_{2,1} \; B_{2,2} \; B_{2,3}$$

A three-dimensional array may be thought of as being composed of planes, each plane containing rows and columns. This interpretation depends somewhat on the purposes of the computation. The reference to an element would simply be, for example, $C_{3,2,1}$.

5.1.1 Subscript Expressions

A subscript expression can be any arithmetic expression resulting in a scalar† value. Before being used as a subscript the value is algebraically rounded to the nearest integer. Some examples are:

a) $B_{3X+Y,-3Y+X}$

b) C_{B_I}

c) $D_{(A+B)J**2}$

5.1.2 Subscript Range Expressions: Partitions

HAL provides two subscript range expressions which permit the partitioning of vectors, matrices and arrays; the forms are:

$A_{P \text{ TO } Q}$

$A_{R \text{ AT } S}$

where P, Q, R, S may be literals, variables or expressions.

5.1.2.1 The "TO" operator

The TO-operator is used to reference, or partition a set of elements by specifying the subscript (or index) limits. For example:

---

† For subscript variables which take on only integral values, some run-time efficiency may be gained by declaring these variables to be integers. The HAL integer data declaration is presented in Part III of this Guide.

a) $\overline{V}_{1 \text{ TO } 4}$ partitions a larger vector, $\overline{V}$, and selects the first four components to form a vector.

b) $\overset{*}{M}_{1 \text{ TO } P,\ 1 \text{ TO } Q}$ partitions a larger matrix and selects the first P rows and the first Q columns.

c) $\overset{*}{A}_{*,\ 3 \text{ TO } 5}$ partitions a larger matrix and selects all rows, columns 3,4,5. The asterisk used in this context means "all of the particular index".

d) $[A]_{P \text{ TO } (P+2),\ I,\ J}$ partitions a three-dimensional array of scalars. The result is a one-dimensional array of three elements.

e) $[A_{J \text{ TO } \#}]$ partitions a one-dimensional array from the J "to the end". The number sign, used in this context means "to the end of the subscript range".

5.1.2.2 The "AT" Operator

The AT-operator is used to reference, or partition, a set of elements by specifying the index size (or length) and the beginning value. For example:

a) $\overset{*}{M}_{4 \text{ AT } 5,\ 4 \text{ AT } 7}$ partitions a larger matrix and selects a 4x4 sub-matrix; i.e., rows 5, 6, 7, 8, and columns 7, 8, 9, 10.

b) $\overline{V}_{3 \text{ AT } 2}$ partitions a vector and selects three components starting with the second component.

5.1.2.3 An Application of Partitioning

Let E be a 9x9 covariance matrix involving errors in the estimation of aircraft position, velocity and ground beacon position. Find the current rms error in aircraft position.

The matrix E is declared by the statement

```
DECLARE E MATRIX(9,9);
```

and the rms error is directly

```
RMS_POS = SQRT(TRACE(E(1 TO 3, 1 TO 3)));
```

Of course, this presumes that the covariance terms in position occupy the upper left corner of the matrix.

5.2 DO Statements

The Do statements are used to define groups of HAL statements which are to be treated as a single unit. There are four statements:

a) DO...END

b) DO WHILE

c) DO FOR

d) DO CASE

5.2.1 DO...END

The DO...END statement simply serves to block out or group a set of statements. Its most frequent application is as an alternative within an IF statement. For example:

```
IF A>5 THEN DO;
        ABLE: X = Y + 6;
        BAKER: Z = X Y;
            IF Z>10 THEN Z = 10;
        END;
X = Y - 6;
```

All of the statements enclosed within the DO...END group will be executed if A>5. If A≤5 then control will pass over the entire DO...END group and X = Y - 6 will be executed.

5.2.2 DO WHILE

The DO WHILE statement provides a means of executing a DO...END group as long as a logical condition (or set of conditions) is satisfied. The general form is

```
DO WHILE Lc;
    .
    .
    .
  statements
    .
    .
    .
END;
```

where L_c denotes the logical condition(s) as defined in Section 4.3.3. As an example, consider the computation of a square root (based on an example in "A Guide to ALGOL Programming", McCracken, John Wiley, 1962). Using the Newton-Raphson iterative technique, to find the square root S, of a number A, the formula

$$S - 1/2(\frac{A}{S}, + S')$$

may be applied repeatedly. S' is the previous value of S. In this illustration the initial guess will be 1 and the number of iterations will not be a factor. Thus,

```
SLAST = 1;
DO WHILE ABS(S-SLAST)> 10^-6 S; /*CONVERGENCE*/
   S = (A/SLAST + SLAST)/2;    /*CRITERION*/
   SLAST = S;
END;
```

Note that, in effect, the logical condition is within the DO WHILE loop and is reevaluated each time before execution of the group of statements. When ABS(S-SLAST) $\leq 10^{-6}$ S, control will pass to the statement following END.

5.2.3 DO FOR

The DO FOR statement provides a means of executing a DO...END group repetitively for a list of values of a control variable as well as for a logical condition. The list may contain a series of values and/or ranges of values. The general form is:

DO FOR VAR = A,B,...,C TO D BY E ...WHILE L_c;

where A, B, C, D, E may be scalar† expressions and VAR is a scalar† variable. "BY E" and "WHILE L_c" are optional.

The control variable, VAR, is initially set equal to the first element of the list, i.e., A, and then takes on successive values from the list on each pass through the group of statements. Between C and D, VAR is incremented by the value of E until VAR exceeds D (or is less than D, if the increment is negative). VAR is evaluated and compared to D prior to each pass. Note that if VAR = D then the statements will be executed for that value. The logical condition L_c, if present, must be true before any pass is initiated. It is processed after the control variable VAR is incremented and evaluated.

---

† or integer

If BY E is not provided, the increment is taken to be +1.

Note that the expressions B,C,D, and E are not within the loop structure of the DO FOR statement and are evaluated only once in the DO FOR statement at the beginning. If they are then modified within the loop, this will not affect their values in the DO statement. Two examples follow:

1. Evaluate $y = x^3 - \log |x|$ for the following values of x: -2, -1.2, 1 to 10 by 2's, 100.

   Thus,

```
DO FOR X = -2, -1.2, 1 TO 10 BY 2, 100;
   Y = X**3 - LOG(ABS(X));
WRITE(LISTING) X,Y;
END;
```

2. Evaluate e^x for x = .01, .1, 1, 10 using 20 terms of the infinite series.

   $$e^x = 1 + X + \frac{X^2}{2!} + \frac{X^3}{3!} + \ldots +$$

   Two DO FOR loops will be used; one to specify the values of x, and the other to sum the terms in the series:

```
DO FOR X = .01, .1, 1, 10;
        SERIES = 1;     /* INITIAL CONDITION */
        TERM = 1;
   EXP: DO FOR N = 1 TO 20;
        TERM = TERM X/N;
        SERIES = SERIES + TERM;
   END EXP;
 WRITE(LISTING) X, SERIES;
END;
```

5.2.4 DO CASE

The DO CASE statement provides a means of transferring control to any one of a number of statements, depending on the value of a scalar† expression.

Suppose it is necessary to transfer to one of five statements based on the value of N: the general form would

```
DO CASE N;
   S1;                    CASE 1††
   S2;                    CASE 2
   S3;                    CASE 3
   S4;                    CASE 4
   S5;                    CASE 5

END;
```

where S_1 to S_5 may be any executable statements, including other DO CASE statements.

If N is an expression, its value is rounded to the nearest integer. A value of 1 specifies the first statement (CASE 1), 2 the second, and so on. The compiler will issue an error message if the rounded value of N is negative, zero or greater than the number of statements provided.

---

† or integer

†† The compiler supplies these CASE indicators; they are not programmer-supplied comments.

The DO CASE statement can be used most effectively as a multi-decision point, allowing combinations of specific computations and transfers of control. For example;

```
CHOICE: DO CASE N;
          GO TO A1;                    CASE 1
          GO TO A2;                    CASE 2
          GO TO B1;                    CASE 3
          DO;                          CASE 4
               A = 3;
               GO TO C1;
          END;
          DO CASE P;                   CASE 5
               A = 4;                  CASE 1
               GO TO C2;               CASE 2
          END;
          IF A > 5 THEN GO TO D1;      CASE 6
             ELSE GO TO D2;
          GO TO F1;                    CASE 7
        END CHOICE;
```

5.3 Examples - II

Two examples are included in this section as further illustrations of HAL_M programming.

5.3.1 MEANS

Given n numbers $x_1, x_2, \ldots x_n$, compute

a) the arithmetic mean

$$a_m = (x_1 + x_2 + \ldots + x_n)/n,$$

b) the geometric mean

$$g_m = \sqrt[n]{x_1 \; x_2 \cdots x_n},$$

c) the harmonic mean

$$h_m = 1/x_1 + 1/x_2 + \ldots + 1/x_n,$$

(This problem is adapted from An Introduction to Algol 60, C. Anderson, Addison & Wesley, 1964.)

```
MEANS: PROGRAM;
  DECLARE X ARRAY(100); /* ALLOW UP TO 100 NUMBERS */
  READ (CARDS) N, [X]1 TO N;
    P = PRODUCT([X]1 TO N);
  ZERO_CHECK: IF P = 0 THEN WRITE (LISTING)
                    'HARMONIC MEAN DOES NOT EXIST';
  ARITH_MEAN: AM = SUM([X]1 TO N)/N;
      WRITE (LISTING) 'THE ARITHMETIC MEAN HAS THE
                        VALUE'||AM;
  GEO_MEAN: IF P<0 AND CEILING(N/2) - N/2 = 0
      THEN DO;
      WRITE(LISTING) 'GEOMETRIC MEAN IS UNDEFINED
      FOR NEGATIVE PRODUCT AND EVEN NUMBER OF TERMS';
          GO TO HAR_MEAN;
          END;
C   CEILING WILL ROUND UP TO NEAREST INTEGER
```

```
            GM = SIGN(P) (ABS(P))**(1/N);
            WRITE (LISTING) 'THE GEOMETRIC MEAN HAS THE VALUE'
                            ||GM;
        HAR_MEAN: IF P = 0 THEN GO TO FIN;
            HM = SUM(1/[X]1 TO N);
            WRITE (LISTING) 'THE HARMONIC MEAN HAS THE VALUE'
                            ||HM;
    FIN:  CLOSE MEANS;
```

5.3.2 FREQ_RESPONSE

Find the sinusoidal amplitude-frequency response, over a significant frequence spectrum, for the transfer function

$$G(s) = \frac{K}{(s\tau_1+1)\ (s\tau_2+1)}\ , \text{ where } \tau_1 > \tau_2$$

```
FREQ_RESPONSE: PROGRAM;
    READ (CARDS) K, TAU1, TAU2;
WRITE (LISTING) 'K ='||K, 'TAU1='||TAU1, 'TAU2='||TAU2;
WRITE (LISTING) 'RAD/SEC', 'AMPLITUDE';   /*HEADINGS*/
    T1SQ = TAU1**2;
    T2SQ = TAU2**2;
C   FIND SPECTRUM LIMITS IN POWERS OF 10
C   ASSUME ALL FREQUENCIES TO BE BETWEEN 10**-10 AND 10**10
        DO FOR I = 10 TO -10 BY -1 WHILE .1/TAU1 < 10**I;
        END;  /*THIS LOOP WILL FIND LOWER LIMIT OF SPECTRUM*/
```

```
        DO FOR J = -10 TO 10 WHILE 10/TAU2 > 10^J;
        END;  /*THIS LOOP WILL FIND UPPER LIMIT OF SPECTRUM*/
        SPECTRUM: DO FOR K = I TO J; /*STEP THROUGH SPECTRUM*/
            W = 10^K;
            TABLE: DO FOR OMEGA = W TO 9 W BY W; /*INCREMENT
                                                   FREQUENCY*/
            MAGN= K/SQRT(TAU1^2 OMEGA^2 + 1)SQRT(TAU2^2 OMEGA^2 + 1);
            WRITE (LISTING) OMEGA, MAGN;
            END TABLE;
        END SPECTRUM;
FINISH: CLOSE FREQ_RESPONSE;
```

5.3.3 FILTER

Find the step response for a digital filter represented by the difference equation:

$$A_0 = K[B_0 + \sum_{1}^{N} P_K B_K] + \sum_{1}^{M} Q_K A_K$$

where A_0, B_0 are current values of A and B, and A_K, B_K are K samples old.

```
    FILTER: PROGRAM;
        DECLARE VECTOR(20) A,B,P,Q;
            /*ALLOW FOR UP TO 20 PAST SAMPLES*/
            /*AND 20 COEFFICIENTS*/
        READ (CARDS) K, N, P_1 TO N, M, Q_1 TO M;
            /*P AND Q ARE COEFFICIENTS*/
        READ (CARDS) S; /*NUMBER OF DESIRED SAMPLE PERIODS IN
                          RESPONSE*/
```

```
HEADINGS: WRITE (LISTING) 'SAMPLE', 'OUTPUT RESPONSE';
      BZERO = 1; /*SET UNIT STEP*/
          Ā(1 TO M) = 0; /*ZERO MEMORY*/
          B̄(1 TO N) = 0;
DO FOR I = 0 TO S;
      AZERO = K(BZERO + P̄(1 TO N) • B̄(1 TO N) + Q̄(1 TO M) • Ā(1 TO M));
                          /*USE DOT PRODUCT*/
WRITE (LISTING) S, AZERO;
    Ā(1 TO N) = VECTOR (AZERO, Ā(1 TO (N-1))); /*INDEX OUTPUTS*/
    B̄(1 TO M) = VECTOR (BZERO, B̄(1 TO (M-1))); /*INDEX INPUTS*/
END;
FINISH: CLOSE FILTER;
```

6. Subroutines: Functions and Procedures

It often happens that some basic computation is required at a number of places in a program. It is possible, of course, to write out the necessary statements each time they are needed, but doing so wastes storage space and is conducive to errors. It is therefore desirable to be able to write the statements once and refer to them as required. Functions and procedures provide this capability.

6.1 Functions

HAL offers a number of built-in functions (see Appendix B) to compute such quantities as trigonometric functions, logarithms, vector absolute values, matrix determinants and inverses, etc. In order to use these functions, it is necessary only to write their names where they are needed, entering the desired expression(s) for the argument(s). For example,

```
X = A SINH(Y);
```

assigns the product of A and the hyperbolic sine of Y to the scalar X. Y may be a simple name or an expression. A more complicated example might be

```
A = ABVAL(X̄*Ȳ) TRACE(M̄*+N̄*) ABS(P LOG(S));
```

The HAL programmer need not be confined to the HAL built-in functions, but can develop and use programmer-defined functions. Suppose it is desired to make the computation of one root of the quadratic equation $ax^2+bx+c = 0$ into a function. The function's arguments are the coefficients;

its value is the root. Thus by simply writing the function name, as in

$A = Y^2$ ROOT1(E,F,G);

control is transferred to the function, its value computed and control is returned. The product of Y^2 and the value of the function ROOT1 is then assigned to A. Note that the coefficient arguments may be names and/or expressions; thus

$A = Y^2$ ROOT1(E^2, LOG(F), G/E);

would mean that the coefficient of x^2 is the value E^2, the coefficient of x is the value LOG(F), and the constant term is G/E.

The function name with its list of arguments may be considered the calling statement, or "function reference". The function itself must be defined elsewhere in the program by a FUNCTION statement and accompanying function body, or block of code. The FUNCTION statement names the function, names the parameters used within the function and specifies the data type of the function result; for example

ROOT1: FUNCTION(A,B,C)SCALAR;

Using HAL_M, only functions resulting in vectors or matrices need specify the function data type. If no specification is provided the function is presumed to be a scalar; e.g. ROOT1: FUNCTION(A,B,C); . Although a function may accept an array as input data, HAL does not permit the specification of an array data-type function.

The body of the function consists of the operations necessary to compute its value. For the example being considered, the complete function definition might appear as:

```
ROOT1:  FUNCTION(A,B,C);
    RETURN (-B+SQRT(B**2-4 A C))/2 A;
CLOSE ROOT1;
```

The RETURN statement terminates the execution of a function. The function body must have at least one RETURN statement. The "returned" expression must agree with the function data-type; in this case a scalar. The above function might be organized in other ways, too; for example,

```
ROOT1:  FUNCTION(A,B,C);
    T = B**2 - 4 A C;
    U = -B+SQRT(T);
    V = U/2.A;
    RETURN V;
CLOSE ROOT1;
```

In this example, T, U, V are introduced for programmer convenience. These are local variables, i.e., local to the defined function and unknown outside the function block. Local variables are discussed further under Scope of Names in Sec. 7.1 . The declaration of local variables follow the same general rules for variable declarations as described in Sec. 3.1.4.

The variables A,B,C are called formal parameters; that

is, they do not exist in of themselves and are no more than dummy variables that indicate what to do with the actual parameters in the function reference. The appearance of formal parameters in the function statement serves as their declaration; e.g., FUNCTION(A,$\overline{V}$,$\overset{*}{M}$). An explicit declaration is necessary if other than default characteristics are required; e.g.,

```
E: FUNCTION(A,V̄,M*);
DECLARE V VECTOR(6);
DECLARE M MATRIX(6,6);
    :
```

It is important to emphasize that the data types and dimensions provided in the function reference must match, correspondingly, the data types and dimensions of the formal parameters declared in the FUNCTION statement and function body.

The formal parameters in a FUNCTION statement cannot be assigned values; i.e., they may not appear on the left hand side of an assignment statement. The actual parameters are expressions involving actual variables that have been declared elsewhere in the program. In the ROOT1 example above, the formal parameter A would be replaced in the function body by E^2, the formal parameter B by LOG(F) and the formal parameter C by G/E.

A function accepting a particular data type will usually accept an array of that type also. For example:

```
[A] = Y**2 ROOT1([E],[F],[G]);
```

Presuming linear arrays, this assignment statement would be executed as follows:

```
A1 = Y**2 ROOT1(E1,F1,G1);
A2 = Y**2 ROOT1(E2,F2,G2);
    :
etc.
```

6.1.1 Some Examples

1) Compute $E = \dfrac{1}{x^5(e^{1.432/kx} - 1)}$

as a function.

```
E: FUNCTION(X);
    B = EXP(1.432/K X) -1;
    RETURN(1/X^5 B);
CLOSE E;
```

2) Define a function to compute

$$\underline{y}(\underline{a},\underline{x}) = \begin{cases} (1 + \sqrt{\underline{a}.\underline{a} + x^2})\,\dfrac{\underline{a}}{|\underline{a}|} & \text{if } x < 0 \\ 0 & \text{if } x = 0 \\ (1 - \sqrt{\underline{a}.\underline{a} + x^2})\,\dfrac{\underline{a}}{|\underline{a}|} & \text{if } x > 0 \end{cases}$$

where $\underline{y}$, $\underline{a}$ are six-dimensional vectors.

```
Y: FUNCTION(Ā,X)VECTOR(6);
    DECLARE A VECTOR(6);
    IF X = 0 THEN RETURN 0;
    B = SQRT(Ā.Ā + X^2);  /*B IS A LOCAL VARIABLE*/
    IF X < 0 THEN RETURN (1 + B) UNIT(Ā);
        ELSE RETURN (1 - B) UNIT(Ā);
CLOSE Y;
```

Note that the formal parameter A required explicit declaration because the desired vector dimension was not the default.

because the desired vector dimension was not the default.

6.2 Procedures

A procedure is like a function in that, once invoked, control is transferred to the procedure body, computations are performed and results are made available to the caller. Where a function is used simply by writing its name, and a particular data type is associated with the function result, a procedure must be called with a CALL statement and may provide many results of different types. The CALL statement has the form:

```
CALL Name (A,B,C)ASSIGN(T,U,V);
```

where Name is the name of the procedure. A, B, and C may be names and/or expressions; T, U, V must be names only. A, B, and C provide the "input" data to the procedure and the procedure results ("output") are assigned to T, U, and V.

As an example, suppose that instead of desiring one-root of a quadratic, as illustrated in the previous section, two roots are necessary. The CALL statement might be

```
CALL ROOT2(E**2,LOG(F),G/E)ASSIGN(X1,X2);
```

and the procedure definition would consist of a PROCEDURE statement and a procedure body, thus

```
ROOT2: PROCEDURE(A,B,C)ASSIGN(P,Q);
    T = SQRT(B**2 - 4 A C);
    P = (-B + T)/2 A;
    Q = (-B - T)/2 A;
CLOSE ROOT2;
```

As in the case of a function, A, B and C are formal parameters (dummy variables) representing the "input" data. These variables cannot be assigned values; they cannot appear on the lefthand side of =. The assign parameters, i.e., P, Q above, are also formal parameters in that they only stand for the actual assign parameters (X1, X2) in the CALL statement, but they can be assigned as shown in the example. Since P and Q are in fact X1 and X2, the assignment statements actually place new values into X1, X2 at these points in the procedure body. Interestingly enough, since P and Q may appear on either side of =, data can be input to a procedure via the assign parameters as well as the call parameters.

The declaration of formal parameters and local variables follow the same rules as for functions. Note that no data type is associated with a procedure name and therefore a procedure name must be called rather than simply used. A procedure may be terminated and control returned to the caller by reaching a RETURN or CLOSE statement.

6.3 Examples - III

Two examples are included in this section as further illustrations of HAL_M programming.

6.3.1 PHASOR

Write a procedure to transform a complex number from rectangular to polar form with the CALL statement

```
CALL PHASOR(A,B)ASSIGN(M,PHI);
```

where the rectangular form is a + ib, and the polar form is me^{iQ}.

```
PHASOR: PROCEDURE(XREAL,XIMAG)ASSIGN(MAGN,PHASE);
        MAGN = SQRT(XREAL**2 + XIMAG**2);
        IF MAGN = 0 THEN DO;
              WRITE(LISTING)'PHASOR UNDEFINED';
              MAGN = -1; /*-1 IS USED TO INDICATE*/
              PHASE = -1; /*UNDEFINED CASE*/
              RETURN;     /*RETURN FROM PROCEDURE*/
                    END;
C    KEEP ARCTAN COMPUTATION LESS THAN 45 DEG.
        IF ABS(XREAL) >= ABS(XIMAG) THEN
           DO;
              IF XREAL > 0 THEN PHASE = ARCTAN(XIMAG/XREAL);
                 ELSE PHASE = PI + ARCTAN(XIMAG/XREAL);
                 /*PI IS A RESERVED HAL CONSTANT*/
              RETURN;
           END;
        IF XIMAG > 0 THEN /*AT THIS POINT ABS(XIMAG) > ABS(XREAL*/
                 PHASE = PI/2 - ARCTAN(XREAL/XIMAG);
        ELSE PHASE = 3 PI/2 - ARCTAN(XREAL/XIMAG);
CLOSE PHASOR;
```

6.3.2 INTEGRATE

Integrate the differential equation

$$\frac{dp}{dt} = t\,p^2 + t^2\,p$$

from t_1 to t_2, where $p(t_1) = p_1$

Use the Runge Kutta technique, and an integration step of Δt.

```
INTEGRATE: PROGRAM;
     READ(CARDS) P1, T1, T2, DELT;
          T = T1;
          P = P1;  /*INITIAL CONDITIONS*/
C    INTEGRATION LIMITS
     LIMITS: DO FOR TLIM = T1 TO T2 BY DELT;
C    RUNGE_KUTTA REQUIRES FOUR PASSES FOR EACH STEP
          FOUR_PASSES: DO FOR I = 1 TO 4;
               DIFF_EQUAT: PDOT = T P**2 + T**2P;
               CALL RUNGE_KUTTA (PDOT,P1,TLIM,DELT,I)ASSIGN(P,T);
                    END FOUR_PASSES;
                        P1=P; /*INITIALIZE FOR NEXT TIME STEP*/
          WRITE(LISTING) T,P;
          END LIMITS;
     CLOSE INTEGRATE;
C    RUNGE_KUTTA PROCEDURE
     RUNGE_KUTTA: PROCEDURE(YDOT,YIMT,XINIT,DELX,J)ASSIGN(Y,X);
          OUTER;†
          DECLARE K ARRAY(4);
```

† See Section 7.3 for discussion of the OUTER statement.

```
            K_J = DELX YDOT;
PASSES:  DO CASE J; /*EACH CASE IS A PASS*/
            DO;                                    CASE 1
            X = XINIT + DELX/2;                 /*HALF-STEP*/
            Y = YINIT + K_J/2;
            END;
            Y = YINIT + K_J/2;                     CASE 2
                                                /*HALF-STEP AGAIN*/
            DO;                                    CASE 3
            X = XINIT + DELX;                   /*WHOLE-STEP*/
            Y = YINIT + K_J;
            END;
C    FINAL RESULT
            Y = YINIT + SUM([K])/6;                CASE 4
                                                /*WHOLE-STEP AGAIN*/
         END PASSES:
   CLOSE RUNGE_KUTTA;
```

Some comments on this example:

1) In the RUNGE_KUTTA procedure, YDOT, YINIT, XINIT, DELX, J, Y and X are formal parameters.
2) The array K is an actual local variable.
3) Note that data is "remembered" by the procedure from one call to the next; values of X and the elements of the array K are retained. X is computed in Cases 1 and 3 and held for Cases 2 and 4 respectively. The elements of K are assigned on successive calls and retained for the summation in Case 4.

7. Program Organization: Scope of Names, Input-Output

A HAL program, written using the features defined in HAL_M, may consist of statements (i.e., IF's, DO's, assignments, etc.), procedures, and functions within a PROGRAM-CLOSE block. The PROGRAM-CLOSE block constitutes the main program and is the smallest compilable unit in HAL; the procedures and functions are sub-programs and are not independently compilable. Any procedure or function may, in turn, contain statements and additional procedures and functions.

Program, procedure, and function blocks define boundaries, or regions, within which names and labels are recognized and may be used for computation and control. Two blocks with mutually exclusive name regions may use the same name for different purposes without interference; e.g., X may be a vector in one procedure and a label in another. The region in which a name or label is potentially recognizable is called the scope of that name.

7.1 Scope of Names

The scope of a name or label, in HAL (or HAL_M), is defined from the outer-most block toward the inner. Thus, names declared at the main program level in a PROGRAM-CLOSE block are potentially recognizable within all nested procedures and functions. The names are only potentially known because any particular name can be declared again in an inner block and then its scope would become all the nested blocks within this block. In general, name and label scopes are based on the first appearance of the identifiers. An example may help to illustrate

these principles:

```
A: PROGRAM;
   DECLARE X VECTOR(6);
   :
   B: PROCEDURE;
      DECLARE M MATRIX(3,4);
      :
   CLOSE B;
C: FUNCTION;
   DECLARE X MATRIX(4,5);
   D: PROCEDURE;
      DECLARE VECTOR(6), A, M;
   CLOSE D;
CLOSE C;
CLOSE A;
```

Comments:

1. The scope of the program name (label), A, is all of A except D. Note that A is declared to be a vector in D.

2. The scope of the vector X is all of A except C and D. X is declared to be a matrix in C and its scope encompasses the nested procedure, D.

3. The scope of the matrix M is B.

4. The scope of the vector M is D.

For these examples of duplicate names within a single program, there are no ambiguities because of the different name scopes. HAL does not admit duplicate names within the same scope.

7.2 Scope of Labels

The scope of labels (statement labels, procedure and function names) generally follows the same rules as for names with some minor exceptions. The GO TO and PROCEDURE statements; i.e, GO TO X or CALL X (----), imply the existence of X as a label. If the label X does not appear in the block in which the statement is written, the GO TO or CALL must refer to a label in an outer block; if the label does appear in the same block, the statement refers to this label.

For example:

#1

```
A: PROGRAM;
   X: Y = Z + 3;
        :
   B: PROCEDURE;
     GO TO X;
        :
   CLOSE B;
        :
CLOSE A;
```

#2

```
A: PROGRAM;
   X: Y = Z + 3;
        :
   B: PROCEDURE:
     GO TO X;
        :
   X: F = G + H;
        :
   CLOSE B;
CLOSE A;
```

In #1, no label X appears in B, therefore control is transferred to the X appearing in A. In #2, control will be transferred to the X which appears in the same block as the GO TO X. With reference to #1, if the label X would have appeared in A after B, i.e., after its use in the GO TO statement, then X would have to be declared explicitly, prior to B, by a special

DECLARE statement (see Section 12.1.1).

A function name presents special problems because its appearance within a statement does not cue the fact that it is a label. For example

```
Y = 3 + SPECIAL(A**2 + 5);
```

does not convey whether SPECIAL is a function name or simply a data name. It is therefore necessary to locate the function definition statements at the beginning of a block so that the appearance of the function name causes no difficulties. For example:

```
A: PROGRAM;
   X: Y = Z + 3;
          :
   B: PROCEDURE;
      Z: FUNCTION:
           :
      CLOSE Z;
   P = Z + 3;
       :
   CLOSE B;
CLOSE A;
```

Note that even though Z is implicitly declared as a scalar at the program level, the reference to Z in B can only be to the function Z. (For an alternate technique, see Sec. 12.1.2.)

7.3 The OUTER Statement

Even though name scope allows for the duplication of names, it does not safely permit their implicit declaration (Sec. 3.1.4) within blocks in a program. For example, if a name were implicitly declared within a function and also declared at the program level, perhaps being unaware of the ambiguity, the program level scope would encompass the function, supercede the name's "function definition", and cause an error. In order to prevent such an occurrence HAL provides the means to isolate an inner block so that only intended names are recognized. The OUTER statement effects this isolation. For example:

```
A: PROGRAM;                          Z: PROGRAM;
   DECLARE VECTOR(4),X,Y,Z;             .
   P = Q + R;                           .
   X = -----                         B: PROCEDURE;
   .                                    OUTER Q, X, Y;
   .                                    .
   B: PROCEDURE;                        C: FUNCTION;
      OUTER Q, X, Y;                       .
      .                                    L = M + N;
      .                                    .
   CLOSE B;                             CLOSE C;
CLOSE A;                             CLOSE B;
                                     CLOSE Z;
```

The use of the OUTER statement here, means that of all the names (and labels) that might have been declared at the program level, only Q, X and Y are recognized inside B.

If OUTER is written without a list of identifiers, no "outer" names or labels will be recognized. It follows then that if it is desired to declare names implicitly an OUTER statement must be provided within the block, or the block must be within another block which contains an OUTER statement.

(See prgrams A and Z above. In A: P,Q and R are implicitly declared. In Z: L, M and N are implicitly declared.)

7.4 Explicit Declarations

In a program with nested procedures and/or functions, convenience may dictate the use of explicit declarations, even for scalars and standard default vectors and matrices. Instead of selecting outer names for each block with an OUTER statement and list, it may be easier to "accept all" outer names and declare explicitly the inner (or local) names. For example:

```
ABLE: PROGRAM;
      DECLARE VECTOR A, B, C, D, E,...,K;
      DECLARE MATRIX AM, BM, CM,...,KM;
      :
      BAKER: PROCEDURE;
             DECLARE A, B, L, M, N;
             DECLARE VECTOR X;
             DECLARE MATRIX D, W;
                :
      CLOSE BAKER;
```

In this example, all of the names declared at the program level (ABLE) and all of the names declared within BAKER are recognized in the procedure, BAKER. Note that within BAKER A and B are declared scalars, and D is a matrix. HAL permits the complete selection of inner- and outer-names by combining the use of DECLARE statements and the OUTER statement.

7.5 Communication Between Programs

The communication between independently compilable programs is provided by HAL through a common data pool (COMPOOL). This facility is discussed in detail in Sec.12.1.3. If the COMPOOL exists and is compiled with a set of programs the scope of the names in the COMPOOL comprises all of the programs. OUTER statements would then be required at the program levels if implicit declarations were to be made.

7.6 Input-Output

HAL provides three basic I/O statements: FILE, READ and WRITE. It is presumed that for the HAL_M programmer, a simplified usage will suffice.(A more complete discussion appears in Sec. 12.2).

7.6.1 FILE Statement

By "assigning" a name to a file, its value(s) is written into the file, thus;

```
FILE(Device,Record) = X;
```

Device is a three digit number specifying a tape or disc, etc., and Record is a program generated identification number (Record can be a scalar expression).

By "assigning" a file to a variable, the contents of the file are read and assigned, thus

```
X = FILE(Device, Record);
```

For filing and retrieval, X may be any data type or organization.

7.6.2 READ Statement

The READ statement causes input data to be read from an external device and assigned to a list of variables. The general format is:

```
READ(device†) A, B, C, ....;
```

where A, B, C are variable names. If the variable is a vector, matrix or array, the number of data fields to be read is the same as the number of elements; the order is the same as when a vector, matrix or array is "filled" from a list (see Sec. 4.2.1).

The following discussion assumes that the external device is a card reader.

Each READ statement presumes data begins in column 1 of a new card, and that each data field is separated by a comma and/or blanks. Control over the reading of cards is explained more fully in Sec. 12.2.1.1. If the READ statement requires more data than can be provided on a single card, subsequent cards will be read automatically as required. An example follows:

```
PROG:  PROGRAM;
       READ(CARDS) L, M, N, V;
       READ(CARDS) A, B, C, D, E;
         :
CLOSE PROG;
```

† "device" is a three digit number specifying a particular device (see Sec. 7.6). A programmer-defined name may be substituted by using the REPLACE statement. For example, suppose the I.D. number for the card reader were 696 then

```
REPLACE CARDS BY '696';
```

would permit the read control statement

```
READ(CARDS)A,B,C,....;
```

Cards:

```
col        1
CARD #1    64.06, -17.10, 45, -100.06, 67.17, 26.54
CARD #2    5, 7, 9, 12, 16
```

The input data may appear in a natural format. Any decimal number with or without a decimal point will be recognized. The letter E is used to express exponent powers of 10. Internal blanks may not appear in the number. The following are examples of acceptable input data:

```
 369.0
   8
-8.36E+2      (equivalent to -8.36 x 10^2)
+0.123E-06    (equivalent to 0.123 x 10^6)
 456.789
```

7.6.3 WRITE Statement

The WRITE statement transmits HAL internal data to an external device. The general format is:

```
WRITE(device) A, B, C, ....;
```

where the list A, B, C may be of variable names and/or expressions. If a member of the list is a vector, matrix or array the number of data fields to be written is equal to the number of elements; the order is the same as when a vector, matrix or array is "filled" from a list (see Sec. 4.2.1).

The following discussion assumes that the external device is a line printer.

Each WRITE statement presumes that data output will start in column 1 on a new line. The first executed WRITE statement presumes, in addition, that data output will start at the top of a new page of the listing. The programmer can control printing by including COLUMN(N) and SKIP(M) instructions in the WRITE list. For example:

```
WRITE(PRINTER)COLUMN(4),A,SKIP(2),B....;
```

will cause the printer to advance to column 4 before starting to print the value of A, and subsequently to skip 2 lines before starting to print B. If no print control is used, 5 blanks are inserted between each written field. If the WRITE statement delivers more data than can be written on one line, the printer automatically advances to the beginning of the next line and then continues. More about the control of printing is explained in Sec. 12.2.1.2.

Numerical output data appears in the following fixed format:

```
sx.xxxxxxxEtxx
└──────┘└┘
mantissa exponent
```

where

s is a blank or a minus sign;

x is a single digit, 0 to 9;

t is a plus or a minus sign.

An example follows:

```
    WRITE(PRINTER)COLUMN(20),'TRAJECTORY RESULTS',
                  SKIP(3),COLUMN(10),'RATE IN FT/SEC',
                  COLUMN(30),'TIME IN SEC',
                  COLUMN(50),'DISTANCE IN FT';
LIST:  DO FOR I=1 TO 50;
    WRITE(PRINTER)COLUMN(10) R_I,
                  COLUMN(30) T_I,
                  COLUMN(50) D_I;
   END LIST;
```

Printer:

```
col   10          20          30                  50

                  TRAJECTORY RESULTS

      RATE IN FT/SEC          TIME IN SEC         DISTANCE IN FT
      -6.3745228E+03          5.0000000E-01       5.7994673E+04
      -5.8812074E+03          1.0000000E+00       3.3210054E+04
      -5.2156354E+03          1.5000000E+00       2.1478935E+04
            .                       .                   .
            .                       .                   .
            .                       .                   .
       4.2573067E+02          2.5000000E+01       1.0057928E+04
```

Part III

GENERAL CAPABILITIES

Part III presents a description of some of the more general capabilities and complex aspects of HAL. A complete description and specification for HAL are given in the document "The Programming Language, HAL, - A Specification", Document # MSC-01846.

8. HAL Data

HAL classifies data into six types: integer, scalar, vector, matrix, character and bit string. Through use of DECLARE statements the programmer can specify, where applicable, attributes concerning the size, shape, precision, initialization, and storage class of any data. Figure 8.1 below, summarizes the allowable attributes for each type. The attribute effect appears within the chart. (See Sec. 8.4 and 8.5 for Initialization and Storage Class

Attribute / Type	Dimensions	Precision	Varying	Initialization	Storage Class
Integer	-	-	-	✓	✓
Scalar	-	decimal digits	-	✓	✓
Vector	length	decimal digits	-	✓	✓
Matrix	rows, columns	decimal digits	-	✓	✓
Bit	length	-	-	✓	✓
Character	length	-	max.length	✓	✓

Fig. 8.1 HAL Data Types and Attributes

8.1 Data Types

8.1.1 Scalar, Vector, Matrix

These data types are floating point quantities and correspond to normal mathematical definitions. A vector consists

of n-scalar components, a matrix of m rows, n columns of scalar components.

8.1.2 Integer

The integer data type is a full word signed number containing only integral values; i.e., a whole number.

8.1.3 Bit String

The bit string data type is simply a string of 1's and/or 0's of specified (fixed) length. A bit string of length equal to one may be used as a boolean variable.

8.1.4 Character String

The character string data type is a string of any of the HAL characters, and may be of fixed or varying length. The varying string is one whose length is dynamically controlled by the compiler at execution time, and requires specification of its maximum length.

8.2 Data Declarations

Each data type may be declared by a DECLARE statement. In addition, for convenience, several declarations may be made within a single statement. The general form is as follows:

DECLARE Name type dimensions precision other-attributes;

That is, the word DECLARE and then the name, followed by the type, including any dimensions and precision, followed thereafter by other attributes in any order. A few examples follow:

1. DECLARE J INTEGER INITIAL(65);

   J is an integer variable with an initial value = 65.

2. DECLARE X PRECISION(8)AUTOMATIC INITIAL(6.061);

X is a scalar variable with a precision of at least 8 decimal digits. The storage class is automatic and X has an initial value = 6.061. Note that when the type is not provided the HAL compiler presumes a scalar. The programmer can supply the word SCALAR at his option.

```
3.  DECLARE M MATRIX(3,3)STATIC
        INITIAL (1,0,0,0,1,0,0,0,1);
```

M is a 3x3 matrix variable with default precision supplied by the compiler. The storage class is static and M is initially set to an identity matrix.

Note that when the programmer does not supply an attribute, in most cases the compiler will presume a standard default. For example the default dimensions are VECTOR(3), MATRIX(3,3), BIT(1), CHARACTER(8). A list of all the HAL standard defaults may be found in the HAL specification document (Reference 1).

```
4.  DECLARE P BIT(12)INITIAL(OCT'4372');
```

P is a bit string variable of length 12 with an initial value of 100011111010. The default on storage class is STATIC.

8.2.1 Multiple Declarations

Several declarations may be made in a single statement by first separating individual declarations by commas, e.g.

```
DECLARE J INTEGER INITIAL(65),
        X PRECISION(8).....,
        P BIT(12).........,
        ...etc.............;
```

8.2.2 Factored Declarations

When a group of declarations have common factors, the declarations may be made in a single DECLARE statement with the common factors appearing first. For example,

1. DECLARE PRECISION(8) X INITIAL(6.061),
   M MATRIX(3,3), V VECTOR(6);

   All quantities have been declared to have a precision of at least 8 decimal digits.

2. DECLARE BIT(1)INITIAL(BIN'1'),A,B,C,D,E,F;

   A through F are 1 bit bit strings, all initially set equal to 1.

8.2.3 Implicit Declarations

As previously indicated in Sec. 3.1.4, scalars, vectors, and matrices may be declared implicitly (i.e., not by a DECLARE statement) by their first appearance in the program with an appropriate defining mark on the E-line over the variable name. Bit and character strings may also be declared in a like manner, with default characteristics, by marking the bit string with a period (.) and the character string with a comma (,). The standard default lengths for bit and character strings are one and eight, respectively. Thus the following statements would be sufficient to declare the strings $\dot{A}$, $\dot{B}$, $'C$, and $'D$.

```
Ȧ = BIN'1' OR Ḃ;
'C = 'ANSWER=' ||'D;
```

$\dot{A}$ and $\dot{B}$ are bit strings of length equal to one.
$'C$ and $'D$ are character strings of length equal to eight.

8.3 Precision

As indicated above, HAL allows the user to specify the precision of data in a DECLARE statement. The PRECISION attribute may only be applied to scalar, vector, and matrix and specifies the desired minimum number of decimal digits; the number must be a positive integer literal and appear within parenthesis as, for example:

```
DECLARE X PRECISION(4);
DECLARE V VECTOR(6)PRECISION(8);
```

For the IBM 360 implementation at MSC the compiler will provide either single or double precision floating point depending on the magnitude of the PRECISION specification. (The standard default is single precision.) For magnitudes greater than 7, double precision will be assigned.

8.4 Constants and Literals

HAL makes a distinction between quantities (names) which are declared as constant and those which literally express their own value (literals). Both remain constant during program execution.

8.4.1 Literals

There are two types of literals; arithmetic and string. An arithmetic literal appears as an ordinary decimal number and may exhibit exponent powers of 2, 10, 16. See Sec. 3.1.3 for examples of arithmetic and character string literals.

The bit string literal expresses its value as a series of binary, octal, decimal or hexadecimal digits. String literals must be enclosed in single quote marks. Some examples of bit string

literals are:

BIN.'101001'

OCT '77346'

DEC '943'

HEX '96FAB'

8.4.1.1 String Repetition

A convenient way to repeat a string pattern is to include a repetition factor indicating the number of "repeats". For example,

1) BIN(6)'10'

would produce 101010101010

2) OCT(4)'7'

would produce 7777

3) CHAR(26)'POP'

would produce POPPOPPOP.....POP.

Note that when repeating a character string,CHAR( ) must precede the string. The programmer may use CHAR for an unrepeated string at his option; i.e., 'ANSWER' and CHAR'ANSWER' are equivalent.

A repetition factor may not be included when expressing a string as DEC 'digits'.

8.4.1.2 Using Literals

Literals may be used in HAL wherever a constant number (or string) is required; for example, in the assignment statement

X = 3.064 Y;

8.4.1.3 The REPLACE Statement

The REPLACE statement provides a means of replacing a name literally by the string of characters enclosed within single quote marks. For example, the statement

```
REPLACE THRUST BY '10601.74';
```

would replace the name THRUST by the characters within the quote marks. The substitution is made whenever THRUST is encountered in the program. Substitution is accomplished within the compiler and does not appear in the listing. For example:

```
1.  A: PROGRAM;
       REPLACE BZERO BY '(-6.27)';
       DECLARE B INITIAL BZERO;
        :

2.  A: PROGRAM;
       REPLACE THRUST BY '10600';
          :
       ACCMAG= THRUST/MASS;
          :
```

The REPLACE statement may also be used to substitute short statements or expressions (or any character string); i.e.

```
1.  REPLACE FIRE_JETS BY 'GO TO F_J;';
       :
    X = B + C;
    FIRE_JETS
       :
      etc.
```

2. REPLACE FACTOR BY 'X**2 + Y';

P = M LOG(FACTOR);

In writing a REPLACE statement the character string must be in one-line format (see Appendix D) and the identifier to be replaced may not be a HAL keyword or symbol.

If a replace statement contains a string literal, double quotes must be used to distinguish them from the outer quotes; e.g.,

1) REPLACE A BY 'BIN"1010"';

2) REPLACE B BY '"THE ANSWER IS"'.

A would then be replaced by BIN'1010' and B by 'THE ANSWER IS'.

The scope of a REPLACE statement is the same as that for a name (Sec. 7.1) with the following exception: the name in a REPLACE statement is never "replaced" as a result of another REPLACE statement located in an outer block.

EXAMPLE:

```
ABLE: PROCEDURE;
  REPLACE X BY 'Y';
  DECLARE X INTEGER;
     :
  BAKER: PROCEDURE;
     REPLACE X BY 'Z';
        :
  CLOSE BAKER;
CLOSE ABLE;
```

The identifier X appearing in BAKER is replaced by Z. X outside of BAKER is replaced by Y.

8.4.2 Constants

The CONSTANT attribute when included in the DECLARE statement specifies that the named quantity is a constant during execution. The use of CONSTANT and INITIAL is mutually exclusive. Some examples are:

1) DECLARE J INTEGER CONSTANT(65);

2) DECLARE X CONSTANT(6.061);

3) DECLARE M MATRIX(3,3)CONSTANT(1,0,0,0,1,0,0,0,1);

4) DECLARE P BIT(12)CONSTANT(OCT'4372');

The declarations are similar to those at the beginning of Sec. 8.2 except J, X, M and P are constants.

8.4.2.1 Initialization Repetition

Initial and constant values of vectors and matrices may be specified by lists of literals and it may be convenient to repeat portions of the list. This is accomplished by use of the number (#) sign. As an illustration consider example (3) in Sec. 8.4.2 above. This could also be written:

```
DECLARE M MATRIX(3,3)CONSTANT(1,3#0,1,3#0,1);
```

or

```
DECLARE M MATRIX(3,3)CONSTANT(2#(1,3#0),1);
```

The term 3#0 means 0 repeated 3 times.

For vectors and matrices, the number of literals in the INITIAL or CONSTANT lists (including all repetitions) must either be equal to the total number of vector or matrix components, or be equal to one.

1) If equal to one, all the components are set equal to the literal (e.g., DECLARE M MATRIX INITIAL(0)).

2) If equal to the total number of components, the components are set equal to literals in the list

on an element-by-element basis.

The vector and/or matrix is 'filled' in the same manner as described in Sec. 4.2.1.

The uses and forms of INITIAL and CONSTANT are complex and it is suggested that the programmer consult the HAL specification document (reference 1) if more information is needed.

8.5 Storage Class

In HAL there are two ways in which data storage may be assigned: STATIC and AUTOMATIC. These attributes may only be applied to declarations made within procedures and functions.

STATIC storage is assigned when a program is activated and remains assigned until the end of a program. This is the kind of storage to which the FORTRAN programmer is accustomed. Consider the following example:

```
A: PROGRAM;
     .
     .
     .
   B: FUNCTION;
      DECLARE X INITIAL(5)STATIC;
         X = X + Y;
   CLOSE B;
CLOSE A;
```

In this example, X being a STATIC variable is assigned a storage location and initialized to five only when A is activated. Since its storage assignment does not depend upon B, the value of X, upon successive entries to B, will be the last computed; i.e., the value of X is held static ("remembered").

AUTOMATIC storage is assigned on entry to the block in which it is declared, and is released on exit from that

block. Suppose that in the example above an additional scalar Y is declared in B; thus

```
B: FUNCTION;
   DECLARE INITIAL(5) X STATIC, Y AUTOMATIC;
```

Y, being an AUTOMATIC variable is assigned storage only when control passes to the function B. Therefore, the last value of Y is not "remembered" and each invocation of B will cause Y to be initialized at a value of 5. AUTOMATIC storage is normally used for local data which must be reinitialized each time the block is entered.

8.6 Arrays and Structures

In HAL the programmer may associate the various data types into two organizations; arrays and structures. The array is an ordered collection of elements, known by one name, all of which have the same data type and attributes. The structure may be a collection of different data types, organized in a hierarchy.

8.6.1 Arrays

Any of the HAL data types may be organized into one-, two- or three-dimensional arrays. This is accomplished within the DECLARE statement; for example,

1) DECLARE J ARRAY(6)INTEGER INITIAL(65);

J is a one-dimensional array variable of 6 elements. Each element is an integer with an initial value = 65.

2) DECLARE M ARRAY(4,2)MATRIX(3,3);

M is a two-dimensional array (4x2) of 3x3 matrices.

In applying the INITIAL and CONSTANT attributes to arrays of data types the list of literals may specify the array value by the whole array, by a single array component (e.g., a matrix), or by an element of a component (e.g., a scalar element of a vector). The programmer should consult the HAL specification document (reference 1) for the allowable forms; some examples follow:

1) DECLARE V ARRAY(4) VECTOR(2) INITIAL(1,2,3,4,-4,-3,-2,-1);

The array V is initialized such that its first component has the value [1,2] and the second [3,4], etc.

2) DECLARE V ARRAY(4) VECTOR(2) INITIAL(1,2);

All four vectors in the array are initialized to the value [1,2].

3) DECLARE V ARRAY(4) VECTOR(2) INITIAL(1);

All of the vector elements in all of the vectors are initialized to 1.

8.6.2 Structures

Some programs are concerned with collections of data of different types. For example, in a spacecraft application the time, fuel, position and velocity vectors, navigation covariance matrix, cockpit switch positions and status monitoring flags might be collected periodically for storage or transmission to the ground. A programmer might wish to move (i.e., READ, WRITE, FILE, etc) all or only part of the collection. To do this he must be able to name and establish relationships among the data and to the whole. This is accomplished by the structure declaration, e.g.

```
DECLARE 1 SPACECRAFT_DATA,
          2 TIME INTEGER,
          2 FUEL,
          2 NAVIGATION,
            3 POSITION VECTOR,
            3 VELOCITY VECTOR,
            3 NAV_COV MATRIX(6,6),
          2 COCKPIT,
            3 POWER_SWITCHES BIT(20),
            3 LIFE_SWITCHES BIT(15),
          2 STATUS BIT(10);
```

The number preceding each name indicates the level of the name. The name SPACECRAFT_DATA has level 1, the highest level. This name refers to the major structure and includes all the names in the declaration. Thereafter, whenever a name at a higher level is followed by a name(s) at a lower level (higher number), the

name at the higher level is that of a minor structure and includes the other names within its structure. For example,

COCKPIT includes POWER_SWITCHES and LIFE_SWITCHES.

The data type declarations, i.e., not the major or minor structure names, follow the general rules for declarations stated earlier in this section. Note that the collection of items above could be represented pictorially as in Figure 8.6-1.

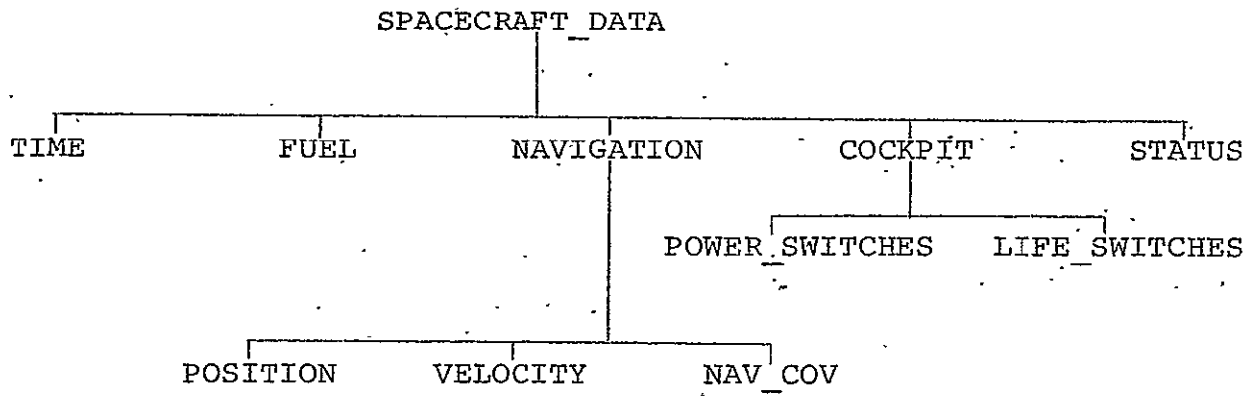


Figure 8.6-1 Hierarchy of Levels in Example Data Structure

8.6.2.1 Name Qualification

When all the names associated with a structure are unique, as in the example above, the data type names and the minor structure names may be referred to individually without ambiguity; i.e., FUEL, COCKPIT, POWER_SWITCHES, etc. Under these conditions the major structure may be given the attribute NONQUALIFIED, i.e., its names need no further qualification. Thus the declaration above would begin:

```
DECLARE 1 SPACECRAFT_DATA NONQUALIFIED,
          2 TIME INTEGER
            :
          etc.
```

However, the names within a structure need not be unique. It is permissible to use some or all of the lower-level names in several minor structures or in another major structure declared in the same part of the program. For example, consider the following structure where position and velocity are grouped into three intervals:

```
DECLARE 1 NAV_DATA QUALIFIED,
            2 FIRST,
              3 TIME INTEGER,
              3 POSITION VECTOR,
              3 VELOCITY VECTOR,
            2 SECOND,
              3 TIME INTEGER,
              3 POSITION VECTOR,
              3 VELOCITY VECTOR,
            2 THIRD,
              3 TIME INTEGER,
              3 POSITION VECTOR,
              3 VELOCITY VECTOR;
```

In order to distinguish among the variables with the same names, it is necessary to specify additional information. This is done by qualifying the names with higher-level names to make the identification unique. The rules for qualification

are that a name used in a structure must be qualified by prefixing it with the names of all the structures (major and minor) in which it is contained. The names are separated by a period and must be in order of level number, the most inclusive level appearing first. The major structure declaration must contain the attribute QUALIFIED. Thus in the example above, the three variables TIME would be referred to as:

```
NAV_DATA.FIRST.TIME
NAV_DATA.SECOND.TIME
NAV_DATA.THIRD.TIME
```

If the programmer does not provide a major structure attribute, the compiler presumes a NONQUALIFIED structure.

8.6.2.2 Multiple Copies of Structures

Multiple copies of major and/or minor structures may be declared by including a dimension in the DECLARE statement after the structure name; e.g.,

```
DECLARE 1 NAV_DATA(10)QUALIFIED,
          2 FIRST(5),
              :
          2 SECOND(5),
              :
          2 THIRD(5),
              .
              .
              .
```

In this case there are 10 copies of the major structure NAV_DATA. Each copy of NAV_DATA contains 5 copies of the minor-structures FIRST, SECOND, THIRD. To refer to a particular VELOCITY

the qualified name must be subscripted as follows:

NAV_DATA.FIRST.VELOCITY$_{8,3}$;

that is, the VELOCITY in the 3^{d} copy of FIRST which is in the 8th copy of NAV_DATA. Structure subscripting is presented in Section 9.

8.7 Formal Parameters and Functions

Functions, procedures and formal parameters were introduced and discussed in Section 6 in the context of HAL_M. In general, FUNCTION and PROCEDURE statements may contain lists of formal parameters of any data type, including arrays and structures. The FUNCTION statement may define the function result to be of any single data-type (arrays and structures are not permitted).

8.7.1 Formal Parameter Declarations

8.7.1.1 Specified Dimensions

Formal parameters with default attributes may be declared implicitly simply by their appearance in the list of parameters with appropriate annotation. Thus,

```
ABLE: FUNCTION(A,B,C,D,E);
```

declares A a scalar, $\overline{B}$ a three component vector, $\overset{*}{C}$ a 3x3 matrix, $\dot{D}$ a one bit bit string, and $\dot{E}$ a character string, eight characters long. Since the data type of the function result is not provided, a scalar result is presumed.

If other than default characteristics are desired, but with specified dimensions, the formal parameters must be declared within the function body (programmer-supplied annotation becomes optional). For example,

```
ABLE: FUNCTION(A,B,M,[D])VECTOR;
   DECLARE A PRECISION(10), B BIT(15);
   DECLARE M MATRIX(6,3), D ARRAY(10,5,3);
```

The DECLARE statements follow the forms presented in previous sections. Note that the function has a vector result of default dimension (i.e., 3) since the dimension has not been provided.

Implicit and explicit formal parameter declarations follow the same rules for functions and procedures.

8.7.1.2 Variable Dimensions

For certain applications it may be convenient not to specify the dimensions of parameters but instead, to have the parameters take on the dimensions of the corresponding arguments in the CALL or function-reference statements. This may be accomplished by substituting an asterisk (*) for the dimension literal. For example, suppose a function is written to accept any size matrix and returns some scalar result; i.e.,

```
ANY: FUNCTION(Q);
  DECLARE Q MATRIX(*,*);
```

The two asterisks mean that both the row and column dimensions will be determined at run time. A more complicated example might be

```
ABLE: PROCEDURE([C])ASSIGN(G);
   DECLARE C ARRAY(*,2)BIT(*);
```

This procedure expects to process an nx2 array of m-bit bit strings, where n and m will be determined at run time.

In general, the asterisk dimension may be applied to array, matrix and vector dimensions, as well as to bit and character string lengths.

8.7.2 Function Results

The FUNCTION statement defines the function result by indicating its data type and attributes. The type may be any of the six HAL data types but the attributes are limited to dimension and precision. The following are examples of valid FUNCTION statements:

```
A: FUNCTION(X,Y) PRECISION(10);

B: FUNCTION(X,Y) MATRIX(6,3) PRECISION(10);

C: FUNCTION(X,Y) CHARACTER(25);
```

8.8 Alternate DECLARE Statement Format

All of the HAL data types, and arrays of these types, may be declared using an alternate form of the DECLARE statement where the data type is indicated (except for scalar and integer) by an appropriate mark over the name and the size and shape designated by a subscript. (-), (*), (.), (,) appearing over a name specifies vector, matrix, bit string and character string data types respectively. Within the subscript, array shape must be separated from string or vector length, and matrix dimensions, by a colon (:).

The use of INTEGER, PRECISION and other attributes remain as described in Secs. 9.2 and 8.3.

EXAMPLES:

1) DECLARE A_{50};

   - a linear array of 50 scalars.

2) DECLARE $B_{2,3}$INTEGER;

   - a 2x3 array of integers.

3) DECLARE $\bar{V}_{10:6}$;

   - a linear array of 10 vectors of length 6.

4) DECLARE $\overset{*}{M}_{6,6}$;

   - a 6x6 matrix.

5) DECLARE $\dot{S}_{100}$;

   - a bit string of length 100.

8.9 The DEFAULT Statement

As detailed in Sec. 8.2, when names are implicitly declared, or explicitly declared with not all characteristics specified, the unspecified characteristics are supplied from a set of defaults. Standard defaults are included in Sec. 8.2 and a complete list appears in Appendix B of the HAL specification document.

In some cases it may be convenient to modify the standard default set to reduce the amount of source program coding required to achieve the given objective. For this purpose, the DEFAULT statement is provided, and the following "size" keywords defined:

BITLENGTH

VECTORLENGTH

MATRIXDIM

CHARLENGTH

The DEFAULT statement has the general format:

```
DEFAULT type(dimension) size;
```

EXAMPLES:

1)
```
DEFAULT MATRIX(4,7) BITLENGTH(24);
DECLARE A, B MATRIX, C BIT(10), D BIT;
```

The DEFAULT statement changes the type default from scalar to matrix, the matrix dimension from (3,3) to (4,7) and the bit length from 1 to 24. Therefore, the DECLARE statement declares A and B to be 4x7 matrices (note the MATRIX need not be supplied), and D to have length equal to 24 bits. It is to be emphasized that the defaults will "fill in" wherever the particular characteristics is not specified.

2) DEFAULT BITLENGTH(16);

DECLARE E, F, G;

The DEFAULT statement changes bit length to 16; all other defaults remain the same. Therefore, E is a scalar, F a bit string of length 16, and G a character string of length 8.

The scope of a DEFAULT statement, that is the region in which it is recognized, is the same as that for a DECLARE statement (see Sec. 7.1).

9. Subscripting

HAL makes use of subscripts for three purposes: 1) to select (i.e., index or partition) data items from complex data types, arrays and structures; 2) to formulate types and arrays from component parts; and 3) to modify the interpretation and usage of data quantities. All subscripting may be accomplished in a natural format by introducing the subscript expressions on the S-line.

9.1 Selection

9.1.1 Arrays of Vectors and Matrices

The referencing of individual components of vectors and matrices, and the partitioning of these data types, by subscripting, are presented in Sec. 5.1 of this guide. Since HAL also permits arrays of vectors and matrices it becomes necessary to introduce additional subscripting in order to select and partition all quantities. This is accomplished by separating the array subscripts from the array element subscripts with a colon (:), with the array subscripts always coming first. For example, consider the following array of matrices:

```
DECLARE M ARRAY(4,3)MATRIX(6,6);
-- a 4x3 array of 6x6 matrices.
```

A few subscript possibilities are:

1) $M_{1,2:3,4}$

This selects the scalar component in the 3rd row, 4th column of the matrix in the 1st row, 2nd column of the array.

2) $\overset{*}{M}_{1,2:}$

This selects the matrix in the 1st row, 2nd column of the array. The "trailing colon" means that the selection consists of the data types in the array, and not of elements within the data types. (Note that the compiler will supply the "over-star" indicating a matrix).

3) $[M]_{3,4}$

This selects the scalar components in the 3rd row, 4th column of all the matrices. The result is an <u>array of scalars</u>. If M were not an array of matrices, but a single matrix instead, $M_{3,4}$ would result in a single scalar. (Note that the compiler will supply the brackets indicating an array.)

4) $\overline{M}_{3,1:2,*}$

This selects a single 6-dimensional vector from the 2nd row (all columns) of the matrix in the 3rd row, 1st column of the array. (Note that the compiler will supply the "over-bar" indicating a vector.)

5) $[\overset{*}{M}]_{1\ \mathrm{TO}\ 3,2:1\ \mathrm{TO}\ 3,1\ \mathrm{TO}\ 3}$

This selects a sub-array of sub-matrices; i.e., the 1st three rows and 1st three columns of all the matrices in the 1st three rows, 2nd column of the original array. (Note that the compiler will supply the brackets and "over-star" indicating an array of matrices.)

It is evident that many complex forms can be developed

from this example. The important point is that by subscripting (indexing and partitioning) both the array and the array components, any selection can be made unambiguously.

9.1.1.1 The Use of * and #

The two symbols * and #† may be used in subscripting variables to indicate "all of a particular index" and "the last of a particular index" respectively. The * can only appear alone in a subscript position; i.e., $\overline{M}_{*,2}$ or $A_{*,*,1}$. The # may appear alone, as part of the expressions # ± K, or associated with "TO" or "AT" in the following forms:

± K TO

P AT # ± L

Examples:

1) $\overset{*}{M}_{1 \text{ TO } 6,*}$

   - a matrix partition: the first 6 rows, all columns.

2) $\overset{*}{M}_{\#-2 \text{ TO } \#,\#-2 \text{ TO } \#}$

   - a matrix partition: the last three rows and last three columns

3) $\overline{V}_{P \text{ AT } \#-Q}$

   - a vector partition: P elements starting at Q from the last element.

† Note that # is also used to indicate repetition within a list (see Sec. 8.4.2.1 and 9.2).

9.1.2 Bit and Character Strings

The individual bits and characters of strings, and the strings themselves within arrays, may be referenced by subscripting. The method is similar to that for vectors. Some examples follow based on the declaration:

```
DECLARE A BIT(15);
```

- a bit string of length 15.

1) $\dot{A}_P$

This selects the P^{th} bit in the string starting from the left. (Note that the compiler supplies the "over-dot" indicating a bit string.)

2) $\dot{A}_{1 \text{ TO } 8}$

This partitions the string and selects the 1st eight bits.

3) $\dot{A}_{P \text{ TO } \#}$

This partitions the string from the P^{th} bit to the end.

If the strings were arrayed, i.e.,

```
DECLARE A ARRAY(10)BIT(15);
```

then

4) $[\dot{A}]_P$

This selects the P^{th} bit from every string. The result is an array of 1 bit bit-strings.

5) A_P:

This selects the P^{th} bit string of the array. Note the "trailing colon".

6) $[A]_{1\ TO\ 6}$:

This selects the 1^{st} six strings of the array. The result is an array of 6 bit strings, each of 15 bits length.

9.1.3 Structures

Any data item within a structure may be referenced by appropriate subscripting of the item name. The structure may be QUALIFIED or NONQUALIFIED. The general method is "to reach" the item by first indicating the major structure copy, then the minor structure(s) copy(s), then the array position and finally the index within the data type. All structure subscripts must be separated from other subscripts by a semi-colon (;). The following example illustrates these points:

```
DECLARE 1 A(50) QUALIFIED,
        2 B(25),
            3 C ARRAY(4,4)MATRIX(3,3),
            3 D BIT(10),
        2 E VECTOR(6);
```

1. $\{A\}_{35}$;

This selects the 35th copy of the major structure, A. (Note that the compiler will supply the brackets indicating

a structure.)

2) $[A.B.C]_{35,10;}$

This selects the array of matrices, C, which are in the 10^{th} copy of B, which is in the 35^{th} copy of A. (Note that the compiler supplies the "over-star" and brackets indicating an array of matrices.)

3) $\{A.B.D\}_{*,1;5\ TO\ 8}$

This selects bits 5 to 8 of the bit string, D, in the 1^{st} copies of B which are in all copies of A. (Note that the compiler supplies the "over-dot" and brackets indicating a structure of bit strings.)

For a NONQUALIFIED structure the subscripting would be identical; thus, for example, (2) above would be written

$$[\overset{*}{C}]_{35,10;}$$

9.1.3.1 Structures of a Single Data Type

Consider the following two DECLARE statements:

```
1) DECLARE 1 A(5),
           2B CHARACTER(10);
2) DECLARE 1 A,
           2B ARRAY(5) CHARACTER(10);
```

From the first statement, $\{\dot{B}\}$ is a structure of all copies of string $\dot{B}$. From the second, $[\dot{B}]$ is the array of all strings.

Note that while {B'} in 1) and [B'] in 2) contain the same data they are not identical and cannot be used interchangeably.

Consider further,

3) DECLARE 1 A(5),

2B ARRAY(5) CHARACTER(10);

$\{[B']\}_{3 \text{ TO } 5;}$ is a structure of the last three copies of the array [B'].

It is suggested the reader consult the HAL specification document (Sec. 6) for more details on structure subscripting and manipulations.

9.2 Formulation

Vectors and matrices, and arrays of all data types may be formulated from their component parts by using special conversion functions and appropriate subscripting. In Sec. 4.2.1, the functions SCALAR, VECTOR, MATRIX were introduced. HAL also provides the following additional "formulating" functions:

INTEGER

BIT

CHARACTER

Each of these functions operates on lists of data and may be "filled" and "shaped" by subscripting.

9.2.1 VECTOR and MATRIX

These functions may be used both for formulating vector and matrix data types, as in Sec. 4.2.1, and for formulating arrays of these types. The distinction is made in the subscript format. For example

$$MATRIX_{2,3}(5)$$

formulates a 2x3 matrix, the elements of which all equal 5. On the other hand,

$$MATRIX_{6:3,3}(5)$$

formulates a one-dimensional array of 6, 2x3 matrices, the elements of which all equal 5. Several objectives may be accomplished using these functions depending upon the number of data items included in the list and the subscript format. For example,

1) $VECTOR_4(A,B,C,D)$

   formulates a 4 dimensional vector.

2) $VECTOR_{6:4}(A,B,C,D)$

   formulates a one dimension array of 6, 4 dimensional vectors.

3) $MATRIX_{10:4,2}(20\#A,20\#B,40\#C)$

   The arguments represent a linear list of 80 data items. This function formulates a one dimensional array of 10, 4x2 matrices in the following way: the first 8 items of the list "fill" the first 4x2 matrix

(by rows), the next 8 items "fill" the second matrix, etc.

The variations of VECTOR and MATRIX are numerous and the reader is advised to consult the HAL language specification (Ref. 1) if more information is needed. In general, though, three list sizes are acceptable: a single item which is "spread" over the data type or the data type array (also see Sec. 9.2.1.1 below); a number of items equal to those in the data type dimension (e.g., the total number of elements in a matrix) which is then repeated for all components of an array; and a number of items equal to the total number in the array which then simply "fills" the array on an element-by-element basis.

Vectors and matrices must consist of scalar elements, therefore other data types included within a list will be converted appropriately. (Conversions of types to types are discussed in Sec. 10.3.2.)

When the list contains more than one entry and the function is unsubscripted, the result is a vector of length equal to the number of elements in the list or a square matrix with rows and columns equal to the square root of the number of elements in the list. (The square root must be an integral number.) For example,

1) `VECTOR(A,B,C,D)`

   formulates a 4 dimensional vector

2) `MATRIX(20#A,5#B)`

   formulates a 5x5 matrix.

When the array shape is specified but dimension is not; e.g.,

$$VECTOR_{6:}(A,B,C;D)$$

the resultant vector(s) or matrices take on default dimensions and the number of elements in the list must be consistent with the default. In the example above, the function would evoke a compiler error message because the 4 elements in the list would not agree with the standard vector length default of 3.

9.2.1.1 VECTOR and MATRIX of a Single List Entry

If the number of entries in list is one; e.g., a single scalar, vector, matrix, etc., or a single array of any data type then two cases are of interest: subscripted and unsubscripted.

When the functions are subscripted and the list entry is a single data item (e.g., a scalar) its value is "spread" over the function as described above. If the single entry comprises a multiple data item (e.g., a matrix or array), the entry is first unraveled and the function "filled" according to the subscripted array shape and dimensions.

When the functions are unsubscripted, the final result depends upon the data type, array shape and dimension of the list entry. A summary of the resulting forms is presented in Appendix F.

9.2.2 INTEGER and SCALAR

The use of INTEGER and SCALAR are similar in that arrays of integers or scalars are formulated from lists of components with appropriate conversions (see Sec. 10.3.2) where necessary. Some examples are:

1) $\text{INTEGER}_{3,3,3}(J)$

   The result is a 3x3x3 array of integers. Every component of the array is set equal to J.

2) $\text{SCALAR}_{9}(\overset{*}{M})$

   The result is a one-dimensional array of scalars of length 9, where $\overset{*}{M}$ is a 3x3 matrix.

3) $\text{INTEGER}_{6,2}(3\#I,\overset{*}{D})$

   The result is a 6x2 array of integers (presuming $\overset{*}{D}$ is 3x3). The matrix is unraveled into a one-dimensional list (see Sec. 4.2.1). Note that the scalar elements of the matrix $\overset{*}{D}$ will be converted to integers.

When the list contains more than one entry and the function is unsubscripted, the result is a one-dimensional array of length equal to the number of elements in the list. For example,

$$\text{SCALAR}(\overline{V},\overset{*}{M})$$

The result is a one-dimensional array of scalars of as many components as in $\overline{V}$ plus $\overset{*}{M}$.

9.2.2.1 SCALAR and INTEGER of a Single List Entry

See Sec. 9.2.2.1 and Appendix F.

9.2.3 BIT and CHARACTER

BIT and CHARACTER may be used to formulate arrays of bit- and character-strings respectively. Appropriate conversions are made where necessary (see Sec. 10.3.2). Some examples are:

1) $BIT_{2,3:1\ TO\ 10}(A)$

   The result is a 2x3 array of bit strings. Each bit string equals the first 10 bits of the "bit-pattern" representation of the scalar, A.

2) $CHARACTER_{10}(X,Y,Z,\ 'COORDINATES')$

   The result is a one-dimensional array of 10 character strings. The first 9 strings are of the length necessary to represent the scalar (floating point) components of the vectors. Resulting character strings are implemented as varying.

3) $BIT_{4,3}(3\#A,3\#B,3\#C,3\#D)$

   The result is a 4x3 array of bit strings. All strings will be of the same length and equal to the maximum string length in the list of arguments.

When the list contains more than one entry and the function is unsubscripted, the result is a one-dimensional array of length equal to the number of elements in the list. Bit-string length corresponds to maximum string length in the list; character length is varying.

9.2.3.1 BIT and CHARACTER of a Single List Entry

See Sec. 9.2.2.1 and Appendix F. Note that subscript dimension for BIT and CHARACTER are different, in concept, than for VECTOR. VECTOR dimension specifies resultant vector length; BIT or CHARACTER dimension specifies the bits or characters to be selected from the string representations of the arguments.

Once again, the reader is advised to consult the HAL specification document (Reference 1) for more complete information on BIT and CHARACTER and the other functions presented in Sec. 9.2.

9.3 Modification

Two forms of subscripting allow the HAL programmer to modify the interpretation and/or usage of certain data types and expressions.

1) In converting from bits to characters and from characters to bits, the subscripts @BIN, @OCT, @DEC, @HEX provide binary, octal, decimal and hexadecimal interpretation, e.g., $BIT_{@OCT}('657')$ results in the bit string 110101111.

2) The precision of an expression can be specified explicitly by use of the subscript form @p, where p represents the minimum number of desired decimal digits. For example suppose the integer I has the value 311,648,726 and is to be added to the single precision floating point

scalar X. It is desired to maintain at least 10 digit precision in the floating point result. Thus the expression

$$I_{@10} + X$$

will

1) cause the integer to be converted to a scalar with precision of at least 10 decimal digits (i.e., a double precision mantissa on the IBM 360/75);
2) convert X to double precision because it is involved with a double precision operand;
3) perform the sum in double precision.

More examples of modification and HAL's automatic data conversions will be presented in Section 10.

10. Data Manipulation

In Part II of this guide the expressions and assignments associated with HAL_M were presented. These were largely confined to manipulation with the arithmetic types: scalar, vector, matrix. In this section, string and array operations are introduced as well as the conversions necessary for combining mixed data types.

A summary of all HAL data operations is presented in Appendix C.

10.1 String Operations

10.1.1 Bit Strings

The manipulation of bit strings, in HAL, is accomplished using the following four operators:

Operator	Definition
NOT (¬,^)	complement
CAT (||)	concatenation
AND (&)	logical AND
OR (| or ¦)	logical OR

and certain of the built-in functions listed in Appendix B. (Acceptable alternate forms for the above operators are shown in parentheses.) NOT complements each bit in the string; CAT forms one string by joining together the two operand strings; AND and OR perform bit-by-bit logical operations on the corresponding bits of two bit operands. If the strings are of unequal length for AND and OR, the shorter is padded on the left with zeros. When assigning a bit expression to a target variable, if the target and expression are of unequal length, then the following

steps are followed: if the expression result is too long, it is truncated on the left; if it is too short, it is padded with zeros on the left. As examples, consider

```
DECLARE BIT(12) A,B,C;
```

then,

1) NOT B

Each bit is complemented

2) C = B$_{1\ TO\ 7}$||A$_{\#-4\ TO\ \#}$

The first 8 bits of B and the last 5 bits of A are joined.

3) A = B$_{4\ TO\ 8}$ AND C$_{1\ TO\ 10}$

The two operands are of different lengths. B$_{4\ TO\ 8}$ is padded on the left with zeros until it matches the length of C$_{1\ TO\ 10}$. A logical AND is performed bit-by-bit; the result is a bit string of length 10. On assignment to A which is of length 12, the result is padded on the left with two zeros.

4. M = (D&E)|(F&G&¬H)|I

If all of these bit strings were declared implicitly then each represents a 1 bit string (i.e., a boolean) and this is an example of a complicated boolean expression and assignment. M is either TRUE or FALSE; i.e., either BIN '1' or BIN '0' depending upon the expression result. For example if D = BIN '1' and E = BIN '1' then M = BIN '1' (¬ is performed before &; see Sec. 10.1.3).

10.1.1.1 Bit Strings Within Logical Conditions

A logical condition or set of logical conditions, L_c, are conditions imposed upon IF and DO WHILE statements (see Sec. 4.3.2), i.e.,

```
IF Lc THEN....
```

or

```
DO WHILE Lc;
   :
```

As such the logical condition expresses a comparison (or comparisons) among data which is either true or false. For HAL_M, in Sec. 4.3.3.1, the relational operators were used to compare arithmetic data. These operators may also be extended to bit strings. Thus it becomes possible to test whether

$$\dot{A} < \dot{B}$$

$$\dot{A} >= \dot{B}$$

$$\dot{A} \neg= \dot{B}$$

etc.

The shorter string is padded on the left, as before. A bit comparison involves the left-to-right comparison of corresponding binary digits; BIN '1' is defined as greater than BIN '0'. The result of a bit string comparison is a single true or false answer. Thus BIN '101' >= BIN '1111' is false because the first bit comparison (starting on the left) fails. Note that, in this context $\dot{A} \neg= \dot{B}$, means that if any of the corresponding bits of $\dot{A}$ and $\dot{B}$ are not equal then the relation is true.

10.1.1.2 "Boolean" Conditions

If the logical condition in an IF or DO WHILE statement involves only single-bit bit strings (booleans) then the condition may be expressed as a boolean expression, similar to example (4) of Sec. 10.1.1.1. For example:

```
IF A AND (B OR C) THEN . . .
```

meaning if A is true (i.e., = BIN '1') and either B or C is true then...; or

```
DO WHILE ¬A|(B&C);
   :
```

meaning do the following statements while A is false (i.e., BIN '0') or, B and C are true.

10.1.1.3 Combining Comparisons and Boolean Expressions

Whenever it is desired to combine comparison expressions (arithmetic or string) with boolean expressions it becomes necessary to express all conditions as comparisons. That is,

```
IF X>5 AND B THEN . . .
```

is not an acceptable form using HAL. The statement must be written with the condition on B expressed as a comparison expression; thus,

```
IF X>5 AND B = TRUE THEN . . .
```

is correct. (Note TRUE ≡ BIN '1'.) A more complicated example might be:

```
IF(B||C = OCT '77') OR (X**2>5 AND FLAG1 = TRUE)THEN . . .
```

10.1.2 Character Strings.

When using HAL, input data from cards, terminals, files, etc. and output data to a printer or other device, are considered to be streams of characters. The acceptance and preparation of numerical data, message texts, headings, etc. requires the manipulation of character strings. The basic operations are presented here; I/O statements appear in Sections 7 and 11.

The manipulation of character strings, in HAL, is accomplished using the concatenation operator, CAT or (||), and certain of the built-in functions listed in Appendix B. Since character variables may be fixed or varying, a distinction must be made. When assigning a character expression to a fixed character string target variable, the result is similar to that for bit strings except that padding or truncation is applied on the right. Thus, the expression value is truncated on the right, if it is too long, or padded with blanks on the right, if it is too short. For example, consider

```
DECLARE CHARACTER(12) A,B,C;
```

then,

1. `C = 'ABC';`

   The first three characters of C are set to 'ABC', the rest are blanked.

2. `C(3 TO 4) = 'ABC';`

   Characters 3 and 4 are set to 'AB', the rest of C are left alone.

If the target variable is a varying character string, then, in general, the target string takes on a length equal to that of the right hand side expression. If the expression length is longer than the declared maximum length, the expression is truncated on the right.

The HAL language specification (Ref. 1) presents detailed rules and examples for the manipulation of character strings.

10.1.2.1 Character Strings Within Logical Conditions

Character string comparisons may be incorporated into logical conditions in the same manner as bit strings (Sec. 10.1.1.1). All of the relational operators of Sec. 4.3.3.1 may be applied in comparing two character strings. The shorter string is padded on the right with blanks. A character comparison involves left-to-right comparison of corresponding characters according to the collating sequence presented in Appendix E.

The result of a character string comparison is a single true or false answer. Thus

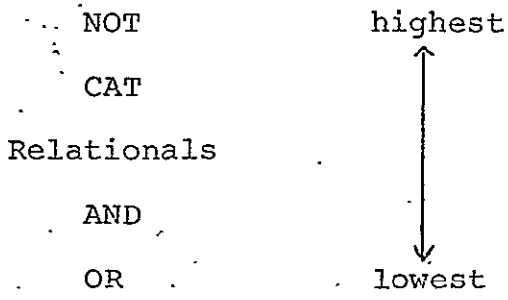
'ABCDE' = 'ABCEF' is false because the fourth character comparison (starting on the left) fails.†

---

† Note that in this context A ¬= B means that if any of the corresponding characters of A and B are not equal then the relation is true.

10.1.3 Order of Operations

In evaluating the expressions with a logical condition, an order of operations was established in Sec. 4.3.3.3. With the addition of the concatenation operator, this order may be generalized and applied to string expressions as well as logical conditions. The complete order is:



In illustrating the application of this precedence order, example (4) of Sec. 10.1.1 could have been written without parenthesis; i.e., M = D&E|F&G& ¬H|I. ¬H would be performed first, then the &'s from left-to-right: D&E, F&G& ¬H, and finally the two |'s. Other logical meanings would have required parenthesis; e.g.,

M = D&(E|F)&G& ¬(H|I);

As another example, consider the IF statement in Section 10.1.1.3. Again this could have been written without parentheses and no change in meaning:

IF B||C = OCT'77' OR X^2>5 AND FLAG1 = TRUE THEN...

B||C would be performed first, then the relationals from left to right: B||C = OCT'77', X^2>5, FLAG1 = TRUE, then AND, and

finally OR. A different logical intent would have required parentheses; e.g.

```
IF(B||C = OCT'77' OR X**2>5)AND FLAG1 = TRUE THEN...
```

Of course, the programmer can take advantage of HAL's built-in precedence rules but he is advised to use parentheses when in doubt in order to clarify the intent of the expression as it appears in the listing.

10.2 Array Operations

Most of the arithmetic and string operations in HAL can also be applied to arrays of appropriate data types, for example,

```
[C] = [B]||[A];
[E] = [F] AND ([G] OR [H]);
[V] = [M] * [W];
```

are valid array manipulations.

In general, operations with arrays are equivalent to operations with their components on a sequential component-by-component basis; i.e. by incrementing the "right-most" index first. Thus for two-dimensional arrays:

```
[A] = [B];
```

means $A_{1,1} = B_{1,1}$, $A_{1,2} = B_{1,2}$, ... $A_{n,m} = B_{n,m}$.
For array expressions and assignments, array dimensions must be compatible; i.e. if two arrays are involved in an operation, they must be of identical dimensions. If only one array is involved, the other operand may be a single data item; e.g., A[B] is a valid product in that A multiplies every

component of [B]. Note that an array may never be assigned to a single data item.

Some examples of array statements and their equivalents follow:

1) DECLARE ARRAY(10,10)A,B,C;

```
[A] = [B] + [C];
```

This array statement causes the addition of components of [A] and [B] on a component-by-component basis. Each sum is assigned to the corresponding component of [A]. The statement is equivalent to the following multiple "DO FOR - loops":

```
DO FOR I = 1 TO 10;
   DO FOR J = 1 TO 10;
   A(I,J) = B(I,J) + C(I,J);
   END;
END;
```

2) DECLARE ARRAY(12,6)D,E,F;

```
DECLARE G ARRAY(12,12);
[D] = [E] [F]/2 + X;
[G] = 0;
```

The components of [E] and [F] are multiplied on a component-by-component basis; each product is divided by 2, added to the scalar X, and assigned to the appropriate component of [D]. In addition, all components of [G] are set to zero. These statements are equivalent to the following "DO FOR -loops":

```
DO FOR I = 1 TO 12;
   DO FOR J = 1 TO 6;
      D(I,J) = E(I,J) F(I,J)/2 + X;
   END;
   DO FOR J = 1 TO 12;
      G(I,J) = 0;
   END;
END;
```

10.2.1 Partitioned Arrays

When array operations involve partitions of arrays the programmer is cautioned to remember the sequential nature of array computations. Consider the following two examples:

1)
```
DECLARE A ARRAY(25);
[A]2 TO 25 = [A]1 TO 24;
A1 = A_NEW;
```

The intention here is to shift the information in the array by one index position and incorporate new data into the first component of [A]. What is the actual result? This may be seen by writing the operations in sequence:

$$A_2 = A_1;$$
$$A_3 = A_2;$$
$$A_4 = A_3;$$
$$\vdots$$
$$A_1 = A\_NEW;$$

Unfortunately the "old" value of A_1 is propagated throughout

the entire array. The final result for [A] would be A_1 = A_NEW with the rest of the components set equal to the old value of A_1. The programmer could have accomplished the intended objective by writing

$[A]_{2 \text{ TO } 25}$ = SCALAR($[A]_{1 \text{ TO } 24}$);

A_1 = A_NEW;

2). DECLARE B ARRAY(10,10);

$[B]_{*,5} = [B]_{3,*}$;

The intention here is simply to replace the 5th "column" of the array by the contents of the 3rd "row". Note that both column and row represent one-dimensional arrays of 10 components each. The operations are performed as follows:

$B_{1,5} = B_{3,1}$;

$B_{2,5} = B_{3,2}$;

$B_{3,5} = B_{3,3}$;

$B_{4,5} = B_{3,4}$;

$B_{5,5} = B_{3,5}$;

$B_{6,5} = B_{3,6}$;

⋮

$B_{10,5} = B_{3,10}$;

and the result is wrong! That is, $B_{3,5}$ appears both on the right and left of the = sign and propagates $B_{3,3}$ into $B_{3,5}$ and $B_{5,5}$. The programmer could have accomplished his objective by writing:

$[B]_{*,5}$ = SCALAR($[B]_{3,*}$);

Thus, array manipulations do require some care and the programmer is urged to write out, in preliminary form, at least a partial sequence of operations in order to verify that the array statement will achieve the desired result.

10.2.2. Functions of Arrays

HAL built-in functions and programmer-defined functions may be given array expressions, of appropriate data-type, in the argument positions. Two classes of functions are of interest: 1) where the function's formal parameters or definition, calls for single data items, 2) where the function's formal parameters, or definition, calls for at least one array. (For built-in functions, the string-, arithmetic-, mathematical-, and matrix-vector-functions are of the first class; the linear array functions comprise the second. See Appendix B.)

10.2.2.1 Functions with Single Data Item Arguments

When arrays are processed by a function designed for single data item arguments, the result is a sequence of operations with the function being applied to the components of the arrays, component-by-component. Thus, for example, consider the sine function where the argument is an array of scalars; i.e.,

```
DECLARE ARRAY(10,5)A,B,;
    [B] = SIN([A]);
```

This statement is equivalent to the following "DO FOR -loop":

```
DO FOR I = 1 TO 10;
   DO FOR J = 1 TO 5;
      B(I,J) = SIN(A(I,J));
   END;
END;
```

For a function requiring more than one single data item, multiple array arguments must be of identical "shape". For example, let VRESULT be a programmer-defined function returning a vector, thus

```
VRESULT: FUNCTION(V,A,B)VECTOR(6)
                  :
```

and used in the statement

```
[P] = M* VRESULT([Q],[A],B);
```

where [P] and [Q] have been declared as 4x2 arrays of six component vectors and [A] is a 4x2 array of scalars. This statement is equivalent to the following sequence of operations:

```
DO FOR I = 1 TO 4;
   DO FOR J = 1 TO 2;
      P(I,J:) = M* BLOCK(Q(I,J:),A(I,J),B)
   END;
END;
```

Note that the same values $\overset{*}{M}$ and B are applied to the computation on every pass through the loop.

Both of the above examples illustrate that the manipulation of arrays with this class of functions is straight forward and is simply a sequence of component-by-component operations.

10.2.2.2 Functions With Array Arguments

HAL functions written, or designed, to accept array arguments must produce single data item results. For example

```
SUM([X])
```

accepts an array argument and returns a single result. The effect might be viewed as a "reduction in dimension". Consider the following examples:

1)
```
DECLARE A ARRAY(5), B ARRAY(5,4);
  [A] = SUM([B]);
```

This statement is equivalent to the following sequence of operations:

```
DO FOR I = 1 TO 5;
   A_I = SUM([B]_I,*);
END;
```

2)
```
DECLARE ARRAY(25,25,25)A,B;
[A]_3 TO 8,Q,* = MAX([B]_10 TO 15,*,*);
```

The left hand side represents a two-dimensional (6x25) sub-array; the argument of MAX is a three-dimensional (6x25x25) sub-array. The statement is equivalent to the following "DO FOR -loops":

```
DO FOR I = 3 TO 8;
   DO FOR J = 1 TO 25;
      A_I,Q,J = MAX([B]_I+7,J,*);
   END;
END;
```

Note that Q is specified at run-time and is outside the loop.

In "reducing the dimension", as illustrated in the examples above, the array functions operate on the "innermost" free index of the array argument (see Sec. F of Appendix B).

10.3 Manipulations With Mixed Data Types

HAL permits the mixing of most data types within expressions and the assignment of one data type result to another data type target variable. The mixing of data types is accomplished through prescribed sets of implicit and explicit conversions.

10.3.1 Implicit Conversions

Some representative examples of implicit conversions follow:

1) DECLARE INTEGER I,J;

```
J = A + I;
```

The addition (subtraction or multiplication) of an integer and a scalar causes conversion of the integer to the scalar type. The assignment of a scalar result to an integer target causes conversion of the scalar to integer before assignment.

2) DECLARE B BIT(10), I INTEGER;

X = B + I;

The addition (subtraction or multiplication) of a bit string and an integer causes conversion of the string to an integer. The assignment of an integer result to a scalar target causes conversion of the integer to scalar before assignment.

3) DECLARE BIT(10)A,B,C;

A = B/C;

Division is defined as a scalar operation. Bit string operands are converted to scalars by first converting the strings to integers and then to scalars. The quotient is always a scalar quantity. The assignment of a scalar to a bit target variable causes conversion of the scalar first to integer and then to bit string before assignment.

4) DECLARE C CHARACTER(25)VARYING;

C = 'THE ANSWER IS'||X;

The concatenation of a character string and a scalar, integer or bit string causes conversion of the scalar or integer to a character string, and the conversion of a bit string first to an integer and then to a character string.

In general, but with certain restrictions, implicit conversions within expressions follow a progression:

$$\text{from bit-to-integer-} \begin{cases} \text{to-scalar-to-character} \\ \text{to-character} \end{cases}$$

i.e.,

$$B \rightarrow I \rightarrow \begin{cases} S \rightarrow C \\ C \end{cases}$$

and from single precision (SP) to double precision (DP). Vector and matrix operands cause the same effects as scalars.

In assigning expressions, conversions go both ways depending upon the data type of the target variable; i.e., either

$$B \rightarrow I \rightarrow \begin{cases} S \rightarrow C \\ C \end{cases}$$

or

$$S \rightarrow I \rightarrow B$$

For example, if a bit string (B) is assigned to a scalar target (S), the string is first converted to an integer and then to a scalar ($B \rightarrow I \rightarrow S$); if a scalar is assigned to a bit string, the scalar is first converted to an integer and then to a bit string ($S \rightarrow I \rightarrow B$).

The following tables summarize the implicit conversions when two operands of different types are involved in expressions or assignments:

A. Expressions

Operand 1 \ Operand 2	I	S	B	C
I	-	I→S	B→I	I→C
S	I→S	-	B→I→S	S→C
B	B→I	B→I→S	-(2)	-(1)
C	I→C	S→C	B→I→C	-

Notes: (1) The concatenation of a character string and a bit string is only valid if the character string is the left hand operand (i.e., C||X).

(2) When bit strings are used in arithmetic operations the strings are converted to integers.

B. Assignments

Target \ Expression (2)	I	S	B	C
I	-	S→I	B→I	(1)
S	I→S	-	B→I→S	(1)
B	I→B	S→I→B	-	(1)
C	I→C	S→C	B→I→C	-

Notes: (1) Character expressions may not be assigned to arithmetic or bit string target variables.

(2) Vector, matrix, and array expressions may only be assigned to vector, matrix and array target variables respectively. Structures may only be assigned to structures of identical component declarations.

10.3.1.1 Conversion of Arithmetic Literals

Arithmetic lietrals exhibit the following default data types and precision:

a) if the literal's value has no fractional part (e.g., 6, 6.0, .12E+2B-1, etc.), it is considered to be an integer data type.

b) if the literal's value has a fractional part (e.g., 6.1, .123, 6.024E-5, etc.), it is considered to be a scalar data type, and its precision will be determined by context, if possible; otherwise default precision will be used.

Subsequent conversion within expressions of mixed data types, follows the rules described above, in Sec. 10.3.1. Note that double precision representation of literals will be utilized when required within an expression, i.e., when the literal is involved with a double precision operand.

EXAMPLES

1. `X = I + 3.1;`

   Presuming that I is an integer, it is converted to a scalar (because 3.1 is a scalar) and added to 3.1. The result is assigned to the scalar X. Since the precision of 3.1 cannot be determined by context (i.e., being added to an integer) default precision will be used. The standard default is double precision.

2. `DECLARE PRECISION(10) X, Y;`

   `X = 4.06372 Y;`

   Since Y is a double precision scalar, the literal 4.06372 is utilized as a DP quantity and multiplied by Y.

3. `REPLACE K BY '1060';`

   `:`

   `A = (K + 3.064) B;`

   K is treated as an integer and therefore converted to a scalar before being added to the "scalar literal", 3.064. Both literals will be expressed in default precision.

10.3.2 Explicit Conversions

Four functions are provided for the explicit conversion of one data type to another. The four are:

INTEGER

SCALAR

BIT

CHARACTER

These functions* may be applied to integer, scalar, bit- and character-string arguments, and result in the named data types. Thus,

```
I = INTEGER(X/Y) + J;
A = BIT8 TO 12(X)||B;
C = CHARACTER(X)||CHARACTER(J);
```

are examples of the use of conversion functions. The following table describes the resulting conversion for each function and type:

Function \ Type	I	S	B	C
INTEGER	-	S→I	B→I	C→I (1)
SCALAR	I→S	-	B→I→S	C→S (1)
BIT (3)	I→B	S→B (2)	-	C→B (2)
CHARACTER	I→C	S→C	B→I→C	-

Notes: (1) INTEGER and SCALAR only accept character string arguments which represent whole numbers and scalars, respectively. For example, INTEGER('30672') and

* Also see Sec. 9.2.

SCALAR ('362.06E+1') are valid applications.

(2) BIT converts scalars and character strings directly to bit strings. That is, a floating point scalar argument would result in a 32-bit bit string, the string representing the 360/75 "bit-pattern" of the floating point quantity. A character byte is converted to its 8 bit pattern.

(3) BIT and CHARACTER may be subscripted in order to select particular bits and characters, or to modify usage (see Section 9.3). A character string which represents binary, octal, decimal or hexadecimal digits can be converted to a corresponding bit string; i.e.,

$BIT_{@BIN}$('1011') becomes 1011

$BIT_{@OCT}$('657') becomes 110 101 111

$BIT_{@HEX}$('FAD') becomes 1111 1010 1101

$BIT_{@DEC}$('78') becomes 1001110

Likewise bit strings can be converted to binary, octal, decimal or hexadecimal character digits; e.g.,

$CHARACTER_{@HEX}$(BIN'11111010')

In addition to using conversion functions within expressions, the "pseudo-variable" SUBBIT is defined, and may appear on the left hand side of an assignment statement. That is, a bit string expression may be assigned directly to the bit representation of other data types. For example,

$SUBBIT_{6\ TO\ 20}(A) = C\ E;$

or

$SUBBIT(C_6') = HEX'9F';$

Through the use of the SUBBIT pseudo-variable, the basic bit pattern (machine representation) of any data type may be manipulated by the programmer.

10.4 HAL Operations - A Summary

HAL provides the programmer with full facilities for:

1) scalar and integer arithmetic
2) vector and matrix arithmetic
3) bit- and character-string manipulations
4) array operations
5) structure handling

Most of the common operators are valid with most of the data types as operands and yield results that might be expected intuitively. However, some operations with particular data types are not allowed, and others imply specific conversions. A summary of all HAL operations, involving one or two operands, is included in Appendix C. For most operations the valid result data type (or error) and the implicit data conversion(s) are indicated. The tables in Appendix C have been taken from the HAL language specification document (Reference 1) and are presented here for programmer convenience.

11. Additional Program Organization and I/O

11.1 Organization

In Sec. 7 the basic features of program organization and name scope were presented in the context of HAL_M. For the most part, these features also apply to the full HAL language. The additional data types: integer, bit- and character-strings, arrays of these types, and structures of all data types, follow the same rules with respect to scope and DECLARE and OUTER statements, as scalars, vectors and matrices.

It is the intention of this section to describe, to the programmer, additional details of program organization bearing principally upon the logical arrangement of blocks of code within a program and the relationship of one program to another, and to the Symbolic Library.

11.1.1 Declaration of Labels

It was pointed out in Sec. 7.2 that the scope of labels, in a HAL program, generally follows the same rules as the scope of names. The statement or procedure label must be defined before its use, or, at least, in the block in which it is used.

When a label appears after its use in a GO TO or CALL statement and outside the block in which it is used, then the label must be declared explicitly. For example:

```
A: PROGRAM;
   DECLARE X LABEL;
      .
      .
      .
   B: PROCEDURE;
      .
      .
      .
   GO TO X;
      .
      .
      .
   CLOSE B;
      .
      .
      .
   X: Y = LOG(P);
      .
      .
      .
CLOSE A;
```

The label X appears in the listing after GO TO X and outside B, and therefore requires the DECLARE statement. The LABEL attribute may not be factored in a DECLARE statement; i.e.,

```
DECLARE LABEL A,B,C;
```

is not permitted.

11.1.2 Declaration of Function Names

Function names must always be defined before their use, even if the FUNCTION statement and function reference appear within the same block.

On occasion it may prove awkward to locate in the listing, all function blocks prior to the statements in which the function names are actually used. This requirement may be avoided by declaring the function name in a DECLARE statement. For example:

#1

```
A: PROGRAM;
ZAP: FUNCTION VECTOR;
       :
 CLOSE ZAP;
   B: PROCEDURE;
      Y = X + ZAP;
          :
   CLOSE B;
       :
CLOSE A;
```

#2

```
A: PROGRAM;
   DECLARE ZAP FUNCTION VECTOR;
       :
   B: PROCEDURE;
      Y = X + ZAP;
          :
  CLOSE B;
 ZAP: FUNCTION VECTOR;
          :
    CLOSE ZAP;
CLOSE A;
```

In #1, the function ZAP is recognized in B because its definition precedes its use. In #2 the definition has been relocated after its use, therefore ZAP must be declared, first, using a DECLARE statement.

The DECLARE statements for a function have the following form:

```
DECLARE A FUNCTION type dimensions precision;
```

The type and attributes may be written in factored form; thus

```
DECLARE FUNCTION MATRIX(4,4)
                 PRECISION(10) A,B,C;
```

11.1.3 Communication Between Programs

The program (i.e., PROGRAM-CLOSE block) is the only independently compilable HAL program-unit. A program can call another program and communicate data through a common data pool (COMPOOL). Data may not be transferred between programs by lists of arguments and formal parameters as with procedures and functions.

11.1.3.1 The COMPOOL

The COMPOOL is a centrally defined and centrally maintained group of statements. The statements are limited to REPLACE, OUTER and DECLARE, and the attributes in the DECLARE statements are further restricted to LABEL, FUNCTION, dimensions and PRECISION (also VARYING for character strings). The names and labels declared in the COMPOOL are potentially known to all programs and, in fact, provide the only means of communication between programs.

In order to take advantage of the COMPOOL as a data sharing mechanism, the programmer must include the COMPOOL statements before the PROGRAM statement during compilation. In a sense, the COMPOOL is placed "outside" the program block and its scope encompasses the program. If another program is compiled in a similar manner, using the same COMPOOL, the variables declared in the COMPOOL will be recognized in both programs. Thus, for example,

```
   INCLUDE COMPOOLA                    INCLUDE COMPOOLA
A: PROGRAM;                         B: PROGRAM;
   :                                   :
CLOSE A;                            CLOSE B;
```

It should be noted that if the COMPOOL is included after the PROGRAM statement; i.e., within the program block then its scope can encompass only the program itself, and declared variables cannot be shared by another program.

11.1.3.2 The Symbolic Library and the INCLUDE Directive

The COMPOOL statements reside in a symbolic library and are entered into the library using specific 360/75 utility commands (to be specified at a later date). Once in the library, the COMPOOL may be retrieved and compiled with any HAL program by using the compiler directive*

```
INCLUDE
```

along with certain other utility commands. The name associated with INCLUDE may be up to 8 characters in length with the first being an alphabetic character. Thus

```
INCLUDE COMPL106
```

or

```
INCLUDE NAVDATA
```

are valid directives.

The symbolic library may also be used to store any symbolic source code; e.g., complete programs, procedures, single statements. The library entries are available to all programs and may be

---

* Compiler directives require a D in column 1 of the input source code.

included in the compilation of a program, at any point, by utilizing the INCLUDE directive with the proper library name. Statements from the Symbolic Library will then be compiled as if they were supplied by the programmer in his source code.

11.1.4 Program Calls

The CALL statement may be used to call one program from another program. The logical result is similar to calling a procedure; i.e., control is transferred to the program called and returned when the program is completed. The CALL statement is of the form:

```
CALL program-name;
```

In calling a program:

1) no arguments may be passed; all communications must be through a COMPOOL.
2) All static variables are allocated on program initiation, and released when the program ends; i.e., variables with the INITIAL attribute are initialized, others take on unspecified values.
3) Control is returned to the caller at the statement following the CALL statement, when a RETURN or CLOSE statement is reached.
4) Control may be returned to the executive by executing a TERMINATE statement; i.e.,

```
TERMINATE;
```

5) A program cannot call itself.

11.1.4.1 Program Declaration

In order to call a program, its name must be known within the calling program. This is accomplished by the DECLARE statement

```
DECLARE A PROGRAM;
```

This statement may be placed in the COMPOOL, elsewhere in the symbolic library, or in the program body. In any case the declaration must appear before the name of the program is used in a CALL statement.

11.1.4.2 Example

```
A: PROGRAM;
    DECLARE XX PROGRAM;
      :
    CALL XX;
      :
      B: PROCEDURE;
         DECLARE YY PROGRAM;
            :
         CALL XX;
         CALL YY;
            :
      CLOSE B;
CLOSE A;
```

11.2 Input-Output

In this section the I/O control functions and the standard data formats complementing Sec. 7.6 will be presented. The material is specialized to card reader and line printer types of devices. The programmer is urged to consult the HAL language specification document (Reference 1) for a more general treatment.

11.2.1 Read and Write Control Functions

External data media, either providing input information to a HAL program or accepting output data, are treated as two-dimensional devices. Data occupies a grid consisting of horizontal lines with each line being made up of column positions; for example, a deck of punched cards where each card is a line, or a 132-column high speed printer. The "read mechanism" or "write mechanism" is located at some point on this two-dimensional grid, and moves in a conventional way along each line and from line to line as reading or writing takes place. Read- and write-control functions are used to move the "read mechanism" or "write mechanism" to any reachable location desired in readiness for reading or writing. The definition of "reachable"varies depending on the physical device involved.

11.2.1.1 Read

In this section discussion is restricted to the card reader as a read device. The "read mechanism" is located on the

two-dimensional grid by the read-control functions SKIP, TAB and COLUMN. A READ statement without these functions will always begin on column 1 of the next card, and will then read consecutive data fields, card after card until all variables have been assigned values, unless interrupted by a semicolon terminating a data field. In this latter case, variables not having yet been assigned values retain their previous ones. Following is an example of a simple READ statement:

READ(device†)A, B, C, D, E, F, ... etc.;

The SKIP read-control function controls the vertical position of the "read mechanism"; that is, it controls which card is next to be read. The form is

SKIP(N), where N $\geq$ 0

A SKIP(0) in the middle or at the end of a list of variable names has no effect. A SKIP(0) before the first variable name in a READ statement causes reading to continue on the same card as that last read by the previous READ statement. Multiple SKIP(N) specifications are cumulative in effect. Any SKIP function appearing before the first variable name overrides the implicit SKIP(1) which normally causes reading to start in the next card.

In the example

READ(CARDS)A, B, SKIP(3), C, SKIP(5), D;

values for A and B are on the first card to be read, the value for C is on the 4th card, and the value for D on the 9th card.

---

† See footnote of Sec. 7.6.2.

There is no relocation of the horizontal position of the "read-mechanism" during the skips.

The TAB and COLUMN read-control functions control the horizontal position of the "read mechanism", at which reading is to start or resume. The TAB function moves the "read mechanism" left or right by the specified number of columns. Its form is

TAB(N), where

N<0: move to left;

N=0: no effect;

N>0: move to right.

N must be of such a value that the column arrived at is in the range 1 through 80. The COLUMN function moves the "read mechanism" to the specified colume. Its form is

COLUMN(N), $1 \leq N \leq 80$

Multiple TAB functions are cumulative. A TAB or COLUMN function appearing before the list variable name in a READ statement overrides the implicit COLUMN(1) normally causing reading of a card to start at column 1.

In the example

```
READ(CARDS) A, B, TAB(6), C';
READ(CARDS) SKIP(0), COLUMN(7), D,E,F';
```

with the data fields

```
column     (1)   (7)
          -5.6, 7.2E+5, 113, 'SECONDS'
```

the first READ statement causes A, B and C' to take the values

-5.6, 7.2E+5, and 'SECONDS' respectively. The second READ statement rereads the same card starting at column 7, causing D, E and F to take the values 7.2E+5, 113, and 'SECONDS' respectively.

11.2.1.2 Write

In this section discussion is limited to the line printer as a write device. The "write mechanism" is located in the two-dimensional grid by the write-control functions LINE, SKIP, PAGE, TAB and COLUMN. A WRITE statement without these functions will always begin at column 1 of the printer, and print out the values of the variables and/or expressions in turn, each data field separated from the next by five blanks. When the end of the line is nearly reached, and the next data field is too long to be printed, one of two things happens. If the data is numerical, printing is deferred to the beginning of the next line. If the data is character, then printing continues until the end of the line is reached and then the remainder of the field is printed at the beginning of the next line. Following is an example of a simple WRITE statement:

WRITE(PRINTER)A,$\overset{*}{B}$,C+D^2,$\bar{E}$,....;

The TAB and COLUMN write-control functions have the same effect as when used as read control functions. The valid range of columns is 1 through 132, however. Note that if use of the functions to move the "write mechanism" to the left is made before printing, whatever was in those column positions

of the same line beforehand,is overwritten. Thus for example

```
WRITE(PRINTER)5132,COLUMN(1),-66;
```

causes the following line to be printed (see Sec. 11.2.2.5).

```
Column        (1)          (11)
                          -66
```

because -66 overwrites 5132. A way of causing multiple overwriting of characters is indicated later. Use of the TAB or COLUMN functions between two entries in the WRITE statement inhibits the 5-blank interfield spacing normally occurring at that point.

The LINE, PAGE, and SKIP write-control functions control the vertical position of the "write mechanism". The PAGE function is of the form

```
PAGE(N), N≥0
```

and causes the printer to advance N pages, remaining on the same line relative to the head of the page. (Each page has 58 lines.) If N=0 the function is ignored. For example if

```
WRITE(PRINTER)X,PAGE(2),Y,PAGE(0),Z;
```

causes printing of the value of X on line 7 or the current page, then the value of Y will be printed on line 7 of the next page but one. The value of 2 is printed on line 7 of this same page immediately following Y.

The SKIP write-control function operates in a similar way to the read-control function. The only difference in behavior results from the use of SKIP(0) in the middle of a list of variables and/or expression. This behavior is best demonstrated by an example:

```
WRITE(PRINTER)'RESULT',SKIP(0),COLUMN(1),'_______';
```

causes the following to be printed:

RESULT

Note the overprinting, which would not have resulted from the statement

```
WRITE(PRINTER)'RESULT',COLUMN(1),'_______';
```

which just results in the following being printed:

_____

In SKIP(N), N may be any positive number: it is immaterial whether or not page boundaries are crossed.

The LINE write-control function forces printing to continue on the line specified. Its form is

LINE(N), $1 \leq N \leq 58$.

If N is equal in value to the current line number, the effect is the same as a SKIP(0). If N is greater than the current line number then the "write mechanism" moves to that line on the current page. If N is less than the current line number then the "write-mechanism" moves to that line on the next page. For example if

```
WRITE(PRINTER)X,LINE(20),Y,LINE(1),2;
```

causes the value of X to be printed on line 15 of the current page, the value of Y will be printed on line 20 of the same page, and 2 on the first line of the next page.

11.2.2 Standard Data Formats

11.2.2.1 Numerical Input Data

Numerical data may be input to a HAL program as a signed (+ is optional) decimal number (with or without a decimal point) and raised, optionally, to the powers 10, 2 or 16. The format is as follows:

$$\pm\text{decimal number}\left\{\begin{matrix}E\\B\\H\end{matrix}\right\}\pm\text{integer}$$

where E, B and H represent 10, 2 and 16 respectively. Internal blanks are not allowed. Data may contain repeated powers. Some examples follow:

```
369.0
8
-8.36E+2B-1
+0.123E6B-3H4
1E-75
.337
```

Numerical data may be assigned to integer, scalar, vector, matrix and bit string data types. For integers and bit strings the data form must represent integral values. For a bit string assignment, data is first converted to a full word bit string and then assigned to the corresponding bit variable named in the read statement. The following statement could accept the

examples shown above:

```
READ(CARDS)I,B,V,A;
```

I is an integer, A is a scalar.

11.2.2.2 Character Input Data

Character data may be input to a HAL program as a string of characters enclosed by single quote marks. Thus, for example,

```
'A B'
'C'
'57.3/C'
'NUMBER_ONE'
'ON,OFF,OFF,ON'
```

Bit string data may be input directly in binary, octal, decimal and hexadecimal forms by representing the data as a character string and then interpreting the string within the HAL program. For example, suppose it is desired to assign a bit string with the octal number 37776. The data may be input as

```
'37776'
```

and the HAL statements might be:

```
DECLARE B BIT(15);
DECLARE C CHARACTER(5);
READ(CARDS)C;
B = BIT@OCT(C);
```

This last statement interprets the character string as an octal bit pattern and converts the quantity to a bit string.

11.2.2.3 Non-standard Data Formats

It is possible for a HAL program to accept data in forms other than those described above. The READALL statement is defined for this purpose. It is suggested that the programmer consult the HAL language specification (Ref. 1) if he desires to use non-standard input data.

11.2.2.4 Scalar Output Data

The standard single precision scalar output data from a HAL program is presented in Sec. 7.6.3. For double precision the number of decimal digits is increased from 8 to 17. The total field size is 14 character positions for single precision numbers, and 23 character positions for double precision numbers.

11.2.2.5 Integer and Bit String Output Data

Integer and bit string data are output from a HAL program as signed integral values (a positive number is indicated by a blank). The total field size is 11 character positions. Leading zeros are suppressed and appear as blanks, except for a single zero value. For example, the following WRITE statements:

```
WRITE(PRINTER) B;
WRITE(PRINTER) J;
WRITE(PRINTER) K;
WRITE(PRINTER) C;
```

might result in the print-out,

```
column        (1)              (11)
                                 5
                              -4673
                                 0
                           2684736
```

Note the conversion of bit strings to integer form.

11.2.2.6 Character Output Data

Character data output from a HAL program appears as a variable size field equal to the string length of the character variable, or expression, in the WRITE statement. For example, the statement

```
WRITE(PRINTER)'DISTANCE= '||A||' MILES';
```

might produce the printed line

```
column        (1)                          (30)
              DISTANCE= 8.6034768E+06 MILES
```

Note the blank characters after the = sign and before MILES.

Bit string data may be output in binary, octal, decimal or hexadecimal form by first converting the string to characters; for example, the statement

```
WRITE(PRINTER)CHAR@OCT(B);
```

would result in writing a bit string of value = 101110100 in the form:

```
564
```

The HAL language specification document (Ref. 1) contains other examples of character output data.

References

1. MSC-01846, The Programming Language HAL - A Specification, prepared under NAS 9-10542, NASA/MSC, Houston, Texas (to be published).

Appendices

Appendix A

HAL Keywords

(not including built-in functions)

The following words are HAL keywords and are usually unavailable for any other use.

ACCESS
AND
ARRAY
ASSIGN
AT
AUTOMATIC
BIN
BIT
BITLENGTH
BY
CALL
CASE
CAT
CHAR
CHARACTER
CHARLENGTH
CLOSE
COLUMN
CONSTANT
DEC
DECLARE
DO
ELSE
END
ERROR
EVENT
EXCLUSIVE
FALSE
FILE
FOR
FUNCTION
GO
HEX
IDCODE
IF
IN
INCLUDE
INDEPENDENT
INITIAL
INTEGER
LABEL
LATCHED
LINE
MATRIX
MATRIXDIM
NOT
NONQUALIFIED
OCT
OFF
ON
OR
OUTER
PAGE
PRECISION
PRIO
PRIOCHANGE
PRIORITY
PROCEDURE
PROGRAM
QUALIFIED
READ
READALL
REPLACE
RETURN
SCALAR
SCHEDULE
SEND
SIGNAL
SKIP
STATIC
SYSTEM
TAB
TASK
THEN
TERMINATE
TO
TRUE
UNTIL
UPDATE
VARYING
VECTOR
VECTORLENGTH
WAIT
WHILE
WRITE

Appendix B

HAL Built-In Functions and Pseudo-Variables

The built-in functions and pseudo-variables available in HAL are given in this appendix, and are presented in alphabetical order under their respective headings. The allowable data-types fpr the arguments are indicated using the following abbreviations:

I: integer

S: scalar

V: vector

M: matrix

B: bit

C: character

A. Conversion Functions (See Secs. 9.2.1, 9.2.2, 10.3.2)

Arguments: I,S,V,M,B,C

1. INTEGER
2. SCALAR
3. BIT
4. CHARACTER
5. VECTOR
6. MATRIX

B. String Functions

1. INDEX (string, config)

Arguments: B,C. Searches a string for a specified bit or character configuration. The starting location of that configuration

within the string is returned as an integer data type.

2. LENGTH (string)

   Arguments: B,C. Finds the string length and returns it as an integer data type.

3. LJUST (character-string)

   Result: LJUST removes all the leading blanks of a character string operand and returns the resultant character string.

4. RJUST (character-string, p)

   Result: RJUST creates a new character string of length, p. The character string argument is truncated on the left, or padded with blanks on the left, depending on whether its length is greater or less than p. p is a scalar expression which is rounded to the nearest integer before use.

C. Arithmetic Functions (B,I,S)

These functions return the same data type as the argument (bit arguments are first converted to integers; the function returns an integer). Array arguments yield array results.

1. ABS

   Finds the absolute value of the argument.

2. CEILING

   Determines the smallest integral value that is greater than or equal to the argument.

3. FLOOR

   Determines the largest integral value that does not exceed the argument.

4. ROUND

   Rounds the argument to nearest integral value.

5. SIGNUM

   Returns 0, +1, -1 as argument is zero, positive, and negative, respectively.

6. SIGN

   Returns +1, -1 as argument is positive or zero, and negative, respectively.

7. TRUNCATE

   Returns 0 if argument is less than +1 but greater than -1; otherwise returns equivalent of SIGN (argument) times the largest positive integral value that does not exceed ABS (argument).

8. MOD(a,b)

   MOD extracts the remainder c such that (a-c)/b=N, where N is an integral number. c is the smallest positive number that must be subtracted from a in order to make N an integral number.

D. Mathematical Functions

These functions return a scalar data type. Arguments may be B,I,S. (Bits and integers are converted to scalars.) Array arguments yield array results.

1. ARCCOS

   Trigonometric cosine; argument in closed interval [-1,1]; results in closed interval [0, π].

2. ARCCOSH

   Inverse hyperbolic cosine; arg not less than 1.

3. ARCSIN

   Inverse trigonometric sine; arg in closed interval [-1,1]; result in closed interval [$-\pi/2$, $\pi/2$].

4. ARCSINH

   Inverse hyperbolic arc sine; arg any value.

5. ARCTAN

   Inverse trigonometric tangent; arg any value; result in open interval ($-\pi/2$, $\pi/2$).

6. ARCTANH

   Inverse hyperbolic tangent; $|arg|<1$.

7. COS

   Trigonometric cosine; arg in radians; $|arg|<K1$.

8. COSH

   Hyperbolic cosine; $|arg|<K3$.

9. EXP

   Exponential, (e^{arg}); $|arg|<K3$.

10. LOG

    Natural logarithm; arg positive and non-zero.

11. SIN

    Trigonometric sine; arg in radians; $|arg|<K1$.

12. SINH

    Hyperbolic sine; $|arg|<K3$.

13. TAN

Trigonometric tangent; arg in radians; arg may not be an odd multiple of $\pi/2$; $|arg|<K2$.

14. TANH

Hyperbolic tangent; arg any value.

15. SQRT

Square root; arg positive.

Note: K1, K2 and K3 are upper limits which depend upon 360/75 machine characteristics (to be supplied at a later date).

E. Matrix-Vector Functions

Arguments may be vectors or matrices (as applicable). Array arguments yield array results.

1. ABVAL

Absolute value of magnitude of vector; argument may be a vector of any length.

2. ADJ

Adjoint; argument is invertible square matrix of any dimensions; result is equal to DETERMINANT (argument) times INVERSE (argument).

3. DET

Determinant; argument is a square matrix.

4. INVERSE

Inverse; argument is square matrix; result is inverse if argument is invertible.

5. TRACE

Trace; argument is square matrix; result is sum of diagonal matrix elements.

6. TRANSPOSE

Transpose; argument is matrix of any dimensions; result is the interchange of the rows and columns of the argument.

7. UNIT

Unit vector; argument is vector of any length; result is a vector of magnitude 1 and in line with argument.

F. Linear Array Functions

These functions have the following general format:

function-label(single-operand)

where the function will operate on the "linear array" representing the "inner-most" free index of the argument. The single-operand may be of (B,I,S,V,M) data types or arrays of these types. The following table indicates the array shape and dimension of the function result.

Argument	$[X]_{a}^{(1)}$	$[X]_{a,b}^{(1)}$	$\overline{V}_{\ell}$	$[\overline{V}]_{a,b:\ell}$	$\overset{*}{M}_{m,n}$	$\overset{*}{M}_{a,b:m,n}$
Function Label	$A^{(2)}$	$[A]_{a}^{(2)}$	$S^{(3)}$	$[S]_{a,b}^{(3)}$	$[\overline{V}]_{m}$	$[\overline{V}]_{a,b:m}$

Subscripts indicate shape and dimension (i.e., array-shape:dimension)

$\ell \equiv$ vector length; $m,n \equiv$ matrix rows, columns; $a,b \equiv$ array shape.

(In general, the argument array shape may be a,b,c,... etc.)

NOTES:

(1) X may be bit string, integer or scalar

(2) A is an integer if X is a bit string or integer

(3) S indicates scalar

The linear array functions are:

1. SUM

   Sums over inner-most free index.

2. PROD

   Forms product over inner-most free index.

3. MAX

   Finds maximum element value over inner-most free index.

4. MIN

   Finds minimum element value over inner-most free index.

EXAMPLES:

1. DECLARE A ARRAY(2,4,6);

   $SUM([A]_{2,*,6})$ results in a 2x6 array of scalars. Sum is performed over second index because it is free.

2. DECLARE ARRAY(25,25,10)A,B;

   $[A]_{3\ TO\ 8,4,*} = MAX([B]_{10\ TO\ 15,*,*})$;

   The result is a 6x10 array of scalars. Each scalar is equal to the maximum value encountered along the inner most index of [B]. The statement is equivalent to the following "DO FOR-loops":

```
DO FOR I = 3 TO 8;
   DO FOR J = 1 TO 10;
      A(I,4,J) = MAX([B]I+7,J,*);
   END;
END;
```

3. DECLARE D ARRAY(10)VECTOR(6);

SUM([$\overline{D}$]) results in an array of scalars of length 10. Each scalar is the sum of the 6 components of each of the 10 vectors.

G. Miscellaneous Functions

1. RANDOM

Result is the current base random number in the pseudo-random number generator. This function enables the programmer to make successive runs of a program without repeating sequences of pseudo-random numbers.

2. RANDOMG

Selects a random number from a Gaussian distribution.

3. TIME

Returns current time as an integer.

4. DATE

Returns current date as an integer.

H. Pseudo-Variables

A pseudo-variable, in HAL, is a function that can only appear on the left of an equal sign (=) in an assignment or DO statement. The only defined pseudo-variable is SUBBIT. See Sec. 10.3.2.

Appendix C

Summary of HAL Operations

The following tables summarize the allowable operations between two operands. In most cases the valid result-type (or an error) and any implied data conversions are indicated within the boxes.

Operation Prefix : $\begin{Bmatrix} P \\ Q \end{Bmatrix}$ OPERAND

P: $\begin{Bmatrix} [+] \\ - \end{Bmatrix}$

Q: NOT

OPERAND	INTEGER	SCALAR	VECTOR	MATRIX	BIT STRING	CHARACTER STRING
	P INTEGER	P SCALAR	P VECTOR	P MATRIX	P Q INTEGER BIT B→I* STRING	ERROR
		P refers to P-group of operators shown above. Q refers to Q-group of operators shown above.				

* B→I means conversion from bit to integer

Table C-1

Operation Addition & Subtract : $OPERAND_1 \pm OPERAND_2$

$OPERAND_1$ \ $OPERAND_2$	INTEGER	SCALAR	VECTOR	MATRIX	BIT STRING	CHARACTER STRING
INTEGER	INTEGER	SCALAR I→S*	ERROR	ERROR	INTEGER B→I	ERROR
SCALAR	SCALAR I→S	SCALAR	ERROR	ERROR	SCALAR B→I→S	ERROR
VECTOR	ERROR	ERROR	VECTOR d	ERROR	ERROR	ERROR
MATRIX	ERROR	ERROR	ERROR	MATRIX d	ERROR	ERROR
BIT STRING	INTEGER B→I	SCALAR B→I→S	ERROR	ERROR	INTEGER B→I	ERROR
CHARACTER STRING	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR

*I→S means conversion of integer to scalar d: dimension check

Table C-2

Operation Multiplication: $OPERAND_1$ $OPERAND_2$

$OPERAND_1$ \ $OPERAND_2$	INTEGER	SCALAR	VECTOR	MATRIX	BIT STRING	CHARACTER STRING
INTEGER	INTEGER	SCALAR I→S	VECTOR I→S	MATRIX I→S	INTEGER B→I	ERROR
SCALAR	SCALAR I→S	SCALAR	VECTOR	MATRIX	SCALAR B→I→S	ERROR
VECTOR	VECTOR I→S	VECTOR	MATRIX(1) SCALAR(2) VECTOR(3)	VECTOR d	VECTOR B→I→S	ERROR
MATRIX	MATRIX I→S	MATRIX	VECTOR d	MATRIX d	MATRIX B→I→S	ERROR
BIT STRING	INTEGER B→I	SCALAR B→I→S	VECTOR B→I→S	MATRIX B→I→S	INTEGER B→I, B→I	ERROR
CHARACTER STRING	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR

Notes: (1) Vector outer product $\overline{V}\ \overline{V}$
(2) Vector DOT product $\overline{V}.\overline{V}(d)$
(3) Vector cross product $\overline{V}*\overline{V}(d$, restricted to 3-element vectors)

d: dimension check

Table C-3

Operation Division : $OPERAND_1/OPERAND_2$

$OPERAND_1$ \ $OPERAND_2$	INTEGER	SCALAR	VECTOR	MATRIX	BIT STRING	CHARACTER STRING
INTEGER	SCALAR I→S, I→S	SCALAR I→S	ERROR	ERROR	SCALAR I→S, B→I→S	ERROR
SCALAR	SCALAR I→S	SCALAR	ERROR	ERROR	SCALAR B→I→S	ERROR
VECTOR	VECTOR I→S	VECTOR	ERROR	ERROR	VECTOR B→I→S	ERROR
MATRIX	MATRIX I→S	MATRIX	ERROR	ERROR	MATRIX B→I→S	ERROR
BIT STRING	SCALAR B→I→S, I→S	SCALAR B→I→S	ERROR	ERROR	SCALAR B→I→S, B→I→S.	ERROR
CHARACTER STRING	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR

Table C-4

Operation Exponentiation : $OPERAND_1$**$OPERAND_2$

$OPERAND_1$ \ $OPERAND_2$	INTEGER	SCALAR	VECTOR	MATRIX	BIT STRING	CHARACTER STRING
INTEGER	SCALAR (1) I→S	SCALAR (1) I→S	ERROR	ERROR	SCALAR (2) I→S, B→I→S	ERROR
SCALAR	SCALAR	SCALAR	ERROR	ERROR	SCALAR B→I→S	ERROR
VECTOR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR
MATRIX (5)	MATRIX	MATRIX S→I	ERROR	ERROR	MATRIX B→I	ERROR
BIT STRING	SCALAR (3) B→I→S, I→S	SCALAR (3) B→I→S	ERROR	ERROR	SCALAR (4) B→I→S, B→I→S	ERROR
CHARACTER STRING	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR

Notes:
(1) Result is INTEGER if $OPERAND_2$ is a whole number literal ≥ 0 (no I→S).
(2) Result is INTEGER if $OPERAND_2$ is a bit string literal which may be converted to an unsigned integer (no I→S, B→I).
(3) Result is INTEGER if $OPERAND_2$ is a whole number literal ≥ 0 (B→I).
(4) Same as (2) except (B→I, B→I).
(5) See Sec. 6.1.1.4

Table C-5

$OPERAND_1 \{^{P}_{Q}\} OPERAND_2$

P: =, ¬ =

Q: =, ¬ =, <, >, <=, >=, ¬<, ¬>

Operation Comparison :

Table shows valid relational operators; the result is always true or false.

OPERAND$_1$ \ OPERAND$_2$	INTEGER	SCALAR	VECTOR	MATRIX	BIT STRING	CHARACTER STRING
INTEGER	Q	Q I→S	ERROR	ERROR	Q B→I	ERROR
SCALAR	Q I→S	Q	ERROR	ERROR	Q B→I→S	ERROR
VECTOR	ERROR	ERROR	P	ERROR	ERROR	ERROR
MATRIX	ERROR	ERROR	ERROR	P	ERROR	ERROR
BIT STRING	Q B→I	Q B→I→S	ERROR	ERROR	Q(1)	ERROR
CHARACTER STRING	ERROR	ERROR	ERROR	ERROR	ERROR	Q(2)

Speical: structure-operand$_1$ P structure-operand$_2$

Notes: (1) OPERAND padded on the left to make lengths equal if necessary.

(2) OPERAND padded on the right to make lengths equal if necessary.

Table C-6

OPERAND$_1$ { P / Q } OPERAND$_2$

P: ||

Q: ||, AND, OR

Operation String :

OPERAND$_1$ \ OPERAND$_2$	INTEGER	SCALAR	VECTOR	MATRIX	BIT STRING	CHARACTER STRING
INTEGER	← ERROR →				→	P CHARACTER I→C*
SCALAR	← ERROR →				→	P CHARACTER S→C **
VECTOR	← ERROR →					
MATRIX	← ERROR →					
BIT STRING	← ERROR →				Q BIT STRING	ERROR
CHARACTER STRING	P CHARACTER I→C	P CHARACTER S→C	ERROR	ERROR	P CHARACTER B→I→C	P CHARACTER

* I→C means conversion from integer to character

** S→C means conversion from scalar to character

Table C-7

Appendix D

HAL Single-Line Format

Most HAL statements can be written in a single line, similar to FORTRAN or PL/1. The single line format requires the use of the following operators:

** for exponentiation

$ for subscripting

Examples

	Multi-Line	Single-Line
1.	$X = A^2 + B^2;$	X = A**2 + B**2;
2.	$X = A_I + B_I;$	X = A$I + B$I;

If the exponent or subscript is an expression (or a multiple subscript) rather than a simple name or literal, the expression, in single-line format, must be enclosed in parentheses:

3.	$X = A^{2P}_{J,K}$	X = A$(J,K)**(2P)
4.	$X = B^{2}_{A_{J,K+3}}$	X = B$(A$(J,K+3))**2

When subscripting an exponent or exponentiating a subscript, it becomes necessary to introduce the single-line format into the multi-line statement as well.

D.1 Implicit Data Declarations

Since data type annotation (-), (*), (.), (,) cannot be supplied by the programmer over a variable name, using a single-line, implicit data declarations are not possible in this format.

Appendix E

Character Collating Sequence

(To be supplied at a later date)

Appendix F

Formulating ("shaping") Functions

(unsubscripted with single entry lists. Also see Sec. 9.2)

The tables below indicate the resulting array shape and type dimensions for the functions

SCALAR

INTEGER

BIT

CHARACTER

VECTOR

MATRIX

where the functions themselves are unsubscripted and the arguments consist of a single entry (e.g. a scalar, a vector, etc. or an array of any data type).

Table F.1 SCALAR, INTEGER, BIT, CHARACTER.

Argument	$\overline{V}_{\ell}$	$[\overline{V}]_{a,b:\ell}$	$\overset{*}{M}_{m,n}$	$[\overset{*}{M}]_{a:m,n}$
Resulting Array Shape	$[X]_{\ell}$	$[X]_{a,b,\ell}$	$[X]_{m,n}$	$[X]_{a,m,n}$

Subscripts above indicate shape and dimension (i.e., array-shape:dimension) ℓ ≡ vector length; m,n ≡ matrix rows, columns; a,b ≡ array shape (in general, the argument array shape may be a,b,c,...etc.). X represents bit string, integer, scalar, or character string.

Table F.2 VECTOR, MATRIX

Argument ⟶

Resulting Array Shape & Dimensions	$X^{(1)}$	$[X]_{a,b}$	$\overline{V}_{\ell}$ or $[X]$	$[\overline{V}]_{a,b:\ell}$	$\overset{*}{M}_{m,n}$	$[\overset{*}{M}]_{a,b:m,n}$
VECTOR	$\overline{V}^{(2)}_{default}$	$\overline{V}_{a:b}$	$\overline{V}_{\ell}$	$[\overline{V}]_{a,b:\ell}$	$[\overline{V}]_{m:n}$	$[\overline{V}]_{a,b,m:n}$
MATRIX	$\overset{*}{M}^{(2)}_{default}$	$\overset{*}{M}_{a,b}$	$\overset{*}{M}^{(3)}_{default}$	$[\overset{*}{M}]_{a:b,\ell}$	$\overset{*}{M}_{m,n}$	$[\overset{*}{M}]_{a,b:m,n}$

Subscripts are defined in Table F.1

Notes:

(1) X refers to bit string, integer, scalar or character operands. Appropriate conversion to scalar is accomplished.

(2) All components are set equal to X.

(3) The length ℓ must equal the product of the product of the matrix default dimensions. (In general, the argument array shape may be a,b,c,...etc.)