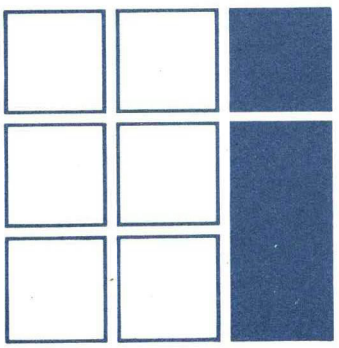
Bob Witty

DRAFT DOCUMENT

The Programming Language HAL - A specification

1 March 1971



INTERMETRICS

DRAFT DOCUMENT

The Programming Language
HAL
- A specification

1 March 1971

Submitted to:

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

FOREWORD TO THIS DRAFT

This document is a draft of The Programming Language, HAL - A Specification (Document # MSC-01848). The content is substantially complete. An index will be included in the final document.

This report was prepared by Intermetrics, Inc. under contract NAS-9-10542 from the Manned Spacecraft Center of the National Aeronautics and Space Administration. The Technical Monitor of this contract is Mr. Jack Garman/FS5.

The publication of this draft does not constitute a document release by Intermetrics and information contained herein shall not be disclosed outside the Government.

PREFACE

The HAL Programming Language has been developed by the staff of Intermetrics, Inc. based on many years of experience in producing software for aerospace applications.

HAL accomplishes three significant objectives: (1) increased readability, through the use of a natural two-dimensional mathematical format; (2) increased reliability, by providing for selective recognition of common data and subroutines, and by incorporating specific data-protect features; (3) real-time control facility, by including a comprehensive set of real-time control commands and signal conditions. Although HAL is designed primarily for programming on-board computers, it is general enough to meet nearly all the needs in the production, verification and support of aerospace, and other real-time applications.

The design of HAL exhibits a number of influences, the greatest being the syntax of PL/1 and ALGOL, and the two-dimensional format of MAC/360, a language developed at the M.I.T. Draper Laboratory. With respect to the latter, Intermetrics wishes to acknowledge the fundamental contribution, to the concept and implementation of MAC, made by Dr. J. Halcombe Laning of the M.I.T. Draper Laboratory.

March 1, 1971

TABLE OF CONTENTS

INTERMETRICS INCORPORATED • 380 GREEN STREET • CAMBRIDGE, MASSACHUSETTS 02139 • (617) 868-1840

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 $\overline{V}$, 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, $\acute{C}$. 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 programmers, analysts and engineers 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 permits the user to manipulate character strings, via special instructions. Organizations of data may also be constructed; multi-dimensional arrays of any single type can be formulated, partitioned, and used in expressions. A 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 data types with default attributes 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

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

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

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

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

where the matrix $\overset{*}{B}$ transforms acceleration from measurement 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. These events may include external interrupts, specific data conditions, and programmer-defined software signals. Undesirable or unexpected events, such as abnormal conditions, may be handled by instructions which enable the programmer to specify appropriate action.

HAL's real-time control features permit the initiation and scheduling of a number of active tasks. This is a necessity for any complex onboard space application.

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.

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.

2.0 HAL LANGUAGE ELEMENTS

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 (e.g. 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 (see Sec. 8).

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.

2.1 Syntax Notation

The following rules are used throughout this specification to describe the syntax of the various constructs in HAL.

2.1.1 Syntactical Elements

Syntactical elements represent the defined language elements which comprise HAL. Elements are denoted by lower case letters (allowing imbedded hyphens) enclosed by angle brackets. Some examples are:

<digit>

<identifier>

<expression>

<operand>

<label>

2.1.2 Keywords

A keyword is the literal occurrence in the language of the characters represented. They are made up of upper case letters and break characters. Some examples are:

DECLARE

INTEGER

AND

OR

NOT

CALL

PROCEDURE

PRIO_CHANGE

2.1.3 Vertical Strokes

The vertical stroke | indicates that a choice of syntactical units or other meaningful symbols is to be made; e.g.

<identifier>|<expression>

<name>|<label>

0|1|2|3|4

etc.

2.1.4 Braces

Braces { } are used to denote that a choice of one of the enclosures must be made. The choices may be stacked vertically, or horizontally using the vertical stroke. For example,

DECLARE<name>{MATRIX / VECTOR}

and

DECLARE<name>{MATRIX|VECTOR}

are identical.

2.1.5 Brackets

Brackets [ ] are used to denote that a choice of one or none is to be made. For example

[<label>:]END;

specifies that an END may but need not be, labeled; e.g.,

MARK: END;

or just

END;

2.1.6 Three Dots

Three dots ... denote that the immediately preceding syntactical unit may occur one or more times in succession; e.g.,

[<digit>]...

specifies a sequence of zero or more digits, while

{<digit>}...

specifies a sequence of one or more digits.

2.2 HAL Character Set

HAL's language syntax includes a total of 85 basic characters. These are:

52 English language alphabetic letters: upper case A through Z and lower case a through z. (Lower case is optional and may be used in identifiers when available.)

10 digits 0 through 9.

23 special characters. Each special character or combination of characters has a particular meaning within the language syntax. (Their uses are discussed in Section 2.3.4.) They are:

=	(equals sign)	.	(period)
+	(plus sign)	,	(comma)
-	(minus sign)	'	(apostrophe)
/	(slash)	(	(left parenthesis)
*	(asterisk)	)	(right parenthesis)
<	(less than symbol)	$	(dollar sign)
>	(greater than symbol)	_	(break character)
¬	(not symbol; also ^)	#	(number sign)
|	(OR symbol; also ¦)	@	(AT sign)
&	(ampersand)	[]	(brackets)
;	(semi-colon)	{}	(braces)
:	(colon)		

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)

2.3 Basic Syntax Elements

2.3.1 Identifiers

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

a. The first character must be a letter.

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

c. It must not be a compiler reserved word.

d. A qualified structure name will contain imbedded periods and must not end in a period or break character. A structure name must be 31 characters or less, including periods.

Examples of valid identifiers:

```
A

R05

INTEGRATION_ROUTINE

SEXTANT_TO_NAVIGATION_BASE_MAT

STATE.COV_MATRIX
```

Examples of invalid identifiers:

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

2.3.2 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 C. Some examples are:

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

2.3.3 Literals

A literal is a group of characters or digits which expresses its own value. For example, 248 and 12.6 are literals in that the compiler will assign these values to these "names". Literals are constants during program execution. There are two types of literals: arithmetic and string.

2.3.3.1 Arithmetic Literals. An arithmetic literal has the following general format:

<digits>[{E|B|H}<integer>]...

where

<digits> = one or more decimal digits with an optional decimal point.

<integer> = signed or unsigned whole number.

GENERAL RULES:

1. E, B, H represent powers of 10, 2, 16 respectively. (That is, 1.023E+2 ≡ 102.3, 32B-5 ≡ 1.)
2. No distinction is made by form between scalar and integer literals. (See Sec. 6.2.1.2 for the use of literals in expressions.)

EXAMPLES:

0.123E6B-3E10, 1E75, 1E-75, 456.789, 3 are all valid arithmetic literals.

2.3.3.2 Bit String Literals. Three forms of bit string literals are defined:

BIN[(<repetition>)] '<binary digit string>'

OCT[(<repetition>)] '<octal digit string>'

HEX[(<repetition>)] '<hexadecimal digit string>'

where <repetition> is an unsigned integer and the digit strings are of length 1 or more. When <repetition> is provided the resulting string length is equal to <repetition> times the number of digits in the particular <digit string>. Imbedded blanks are allowed between the apostrophes, but have no significance.

GENERAL RULES:

1. Binary digit strings may contain only zeros, ones, or blanks.
2. There are 4 special forms of bit string literals:

$$\{\begin{matrix}\text{TRUE}\\\text{ON}\end{matrix}\} \equiv \text{BIN'1'}$$

$$\{\begin{matrix}\text{FALSE}\\\text{OFF}\end{matrix}\} \equiv \text{BIN'0'}$$

EXAMPLES:

TRUE, BIN'10110', HEX 'ABCD', BIN(32)'1', OCT'3777' are all valid bit string literals.

2.3.3.3 Character String Literals. Two forms of character string literals are defined:

'<text>'

CHAR [(<repetition>)]'<text>'

where <text> may contain any character in the accepted character set. If it is desired to have an apostrophe in the resulting literal, it must be represented by an adjacent pair of apostrophes. The length of the resulting string is equal to the count of the characters plus the number of apostrophe pairs.

EXAMPLES:

'AB''''C', CHAR'57.3/C', CHAR(26)'POP', are all valid character-literals, having lengths of 5, 6, and 78 respectively.

NOTE: The character pair /* is always interpreted as an opening comment bracket by the compiler, even if it occurs within a character string literal.

2.3.4 Special Characters

Special characters or combinations of characters are used in HAL between or with identifiers as operators, separators, or other delimiters. These characters and their uses are defined below and described in more detail in Sec. 6.

2.3.4.1 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:

(1) a space (or spaces) between two distinct identifiers is interpreted as multiplication.

(2) one of the operands (identifer or expression) must be enclosed in parentheses.

(3) the leftmost operand must end with a parenthesis (function form; e.g., SIN(X)).

2.3.4.2 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 ^<)

The word NOT is equivalent to (¬|^) and may be applied to the combinations above.

2.3.4.3 String and Logical Operators.

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

Word operators (e.g., AND) may be substituted for symbols (e.g., &) except that they do not act as delimiters and must be appropriately delimited by blanks or otherwise. The use of these operators is described in more detail in Sec. 6.

2.3.4.4 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)

2.3.4.5 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.

2.3.4.6 Built-in Function Names. Built-in function names are identified by the compiler as names of functions which are part of the language. A complete list of these functions appears in Appendix A. Some examples are:

ABS

TRUNCATE

COS

TAN

INVERSE

UNIT

2.3.4.7 Compiler-Generated Annotation. The following characters are used by the compiler to annotate various data types as they appear in the listing. Identical usage is also acceptable in the input stream.

Symbol	Definition
*	Over a name denotes a matrix type.
-	Over a name denotes a vector type.
.	Over a name denotes a bit string type.
,	Over a name denotes a character string type.
[ ]	Denotes an array of a particular data type.
{ }	Denotes a structure organization.

3.0 SOURCE LANGUAGE INPUT

A source language program is presented to the compiler in the form of statements. Statements can be written in single line, one-dimensional format, as in FORTRAN, PL/I, and most languages, as (for example)

```
A = B**4 + 2(C+D)**2 ;
Z = R/(A-Z)**2;
C = A**B**2 + E**4;
```

However, one of the unique features incorporated into HAL, in order to improve readability and clarity, is that statements may also be written using a multi-line or two-dimensional format. That is:

$$A = B^{4} + 2(C+D)^{2};$$
$$Z = R/(A-Z)^{2};$$
$$C = A^{B^{2}} + E^{4};$$

The multi-line format introduces the added dimension of optional exponent and subscript lines. These lines are used for the exponentiating and subscripting of data on the main line of the statement. The exponent line is also used for annotation of variable names in order to indicate data types. Examples of the multi-line format are:

(1) an assignment statement involving scalar array elements :

$$A_{I,J} = B_J^2 + C_{2K+3}^2;$$

(2) a vector-matrix equation:

$$\overline{X} = (\overline{R}.\overline{V})\ \text{UNIT}\ (\overline{R}) + \overset{*T}{M}\ (\overline{R}*\overline{V});$$

(3) a complicated expression in multi-line format involving multiple exponents and multiple indices:

$$Y = 5\ \text{BAKER}^{5N}_{\text{INDEX\_TABLE}_{I,J}} + \text{COMBUF}^{2K}_{I,J,*};$$

The standard source language input is expected to be in two-dimensional format. The single-line format is provided as an alternate. If single-line input is used, the compiler will expand the single-line to multi-line in the output listing. The definitions and restrictions of the two-dimensional and single-line formats are described below.

3.1 Two-Dimensional Format

Source language input statements must always have a main or "M" line. An "M" line may optionally have associated with it zero or more "E lines" (exponent lines) and zero or more "S lines" (subscript lines). An input statement may be thought of as n continuous parallel streams of characters on the E-, M-, S lines that comprise the statement. A statement terminator (semi-colon) is used to terminate the n-line stream. The terminator must be on the main line and occur after (to the right of) all information on the main line and any associated E and S lines. Another statement may begin following the terminator.

The first character of each line of input must be the particular letter that identifies the line. The various identification letters recognized, are:

First character of line	Meaning
E	This line contains exponents for the main line, or another E line below it.
M	This line is a main line; a blank is assumed to be an M line.
S	This line contains subscripts for the main line or another S line above it.
C	This line contains comments.
D	This line contains compiler directives.

Statements and comments may occupy any part of the rest of the available lines (e.g. columns 2 through 80 for cards).

Continuation of a statement from one set of E, M, and S lines to another is permitted. For this purpose, column(s) 2 of the next set is considered equivalent to column(s) 81 of the current set. A statement may be continued in this manner until a terminator appears on an M line. The number of E and S lines in the succeeding set(s) need not be the same as the number of E and S lines used originally. An M line, however, must always be present in every set. For example,

```
E        5
E       K
M  A = B    + C
S       I


E  2
M    + D + E
```

which is equivalent to

```
E        5
E       K    2
M  A = B  + C  + D + E;
S       I
```

3.1.1 E and S Line Expressions. The E and S lines contain exponent and subscript expressions respectively, as well as certain data type annotations. Labels, terminators, statements, and expressions resulting in vectors, matrices, and character strings are not permitted on E or S lines.

S lines are evaluated from the lowest S line up to the main line; E lines are evaluated from the upper-most E line down to the main line. Subscripting is always evaluated prior to exponentiation. Exponent and subscript expressions follow the same arithmetic rules as for expressions on the main line (See Section 6).

Examples

```
M  Q  =  A   ;        J is an index for C, the result of
S         B           which is used to index B; the result is
S          C          then used to index A.
S           J

E            2
E           3              9
E          2              2            512
M  B  =  A   ; means B=A   ; or B = A    ;


E
E        2(D+E)
M  A = B                          +D ;
S        2(TABLE_1 +TABLE_2. )   K
S                 J          K
```

Expressions on an E or S line must appear following (to the right of) the associated identifier on the M line. Also, M line information cannot appear directly above S line or below E line expressions. Similar rules apply to E and S lines associated with other E or S lines.

The number of E and S lines allowed in a statement will be determined by the compiler implementation.

3.2 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; thus

5.	$X = A^{(B\$J)^P}$	X = A**(B$J)**P

3.2.1 Implicit Data Declarations.

Since data type annotation (-), (*), (.), (,) cannot be supplied by the programmer over a variable name using a single line, implicit data declarations (See Sec. 5.3) are not possible in this format.

3.3 Comments

3.3.1 Comments on Statement Lines.

Comments can be inserted on any E, M, or S line in a statement. A comment consists of any set of characters enclosed in the /* */ pair. These are the comment open and close brackets respectively. The */ combination cannot be used within a comment since it would be interpreted as the comment close bracket.

Comments on one M line, initiated by /*, can be continued to other M lines until terminating bracket */ appears on a succeeding M line. Comments initiated on an E or S line must be terminated before the end of the line (e.g., column 80 for cards). For example:

```
E            2    2    2      /*THIS IS A COMMENT*/
M  R_MAG = X  + Y  + Z        /*WHICH IS TO
S            I    I    I      /*SHOW HOW COMMENTS*/

M  CONTINUE */ + ALPHA;
```

Note that imbedding a comment within a statement is allowed. In general, comments are permitted wherever blanks are legal.

3.3.2 Comment Lines.

Comments may also be introduced by the use of comment lines. A comment line begins with a C in the first character position of the input line. The rest of the line contains the comment made up of characters recognized by the compiler implementation. Comment lines may only appear between statement line

groups; i.e., they are not permitted within the EMS combination that comprises a statement line.

EXAMPLE:

```
E     2
M  A=B ;
S     I

C  THIS IS AN EXAMPLE WHICH
C             2
C  SHOWS A = B  AND IS
C               I
C  COMPUTED ONLY WHEN FLAG 1 IS SET

M  X=Y;
```

3.4 Use of Blanks

Blanks are significant as separators between identifiers, keywords, and literals. The use of consecutive blanks is syntactically equivalent to the use of only one blank with the following exceptions:

(1) within EMS combinations when the horizontal position of items is important relative to the associated data above or below;

(2) within character strings.

4.0 DATA ELEMENTS

HAL classifies data elements by type and permits collections of types into data organizations. Types are further specified by data attributes. There are six data types in HAL; integer, scalar, vector, matrix, and character and bit strings. The type classification of an identifier determines the contexts in which it may be used.

The data types may also be combined into data organizations. There are two types of organizations in HAL: arrays and structures. Fig. 4-1 summarizes the relationship among the types and organizations.

HAL Data Types and Organizations

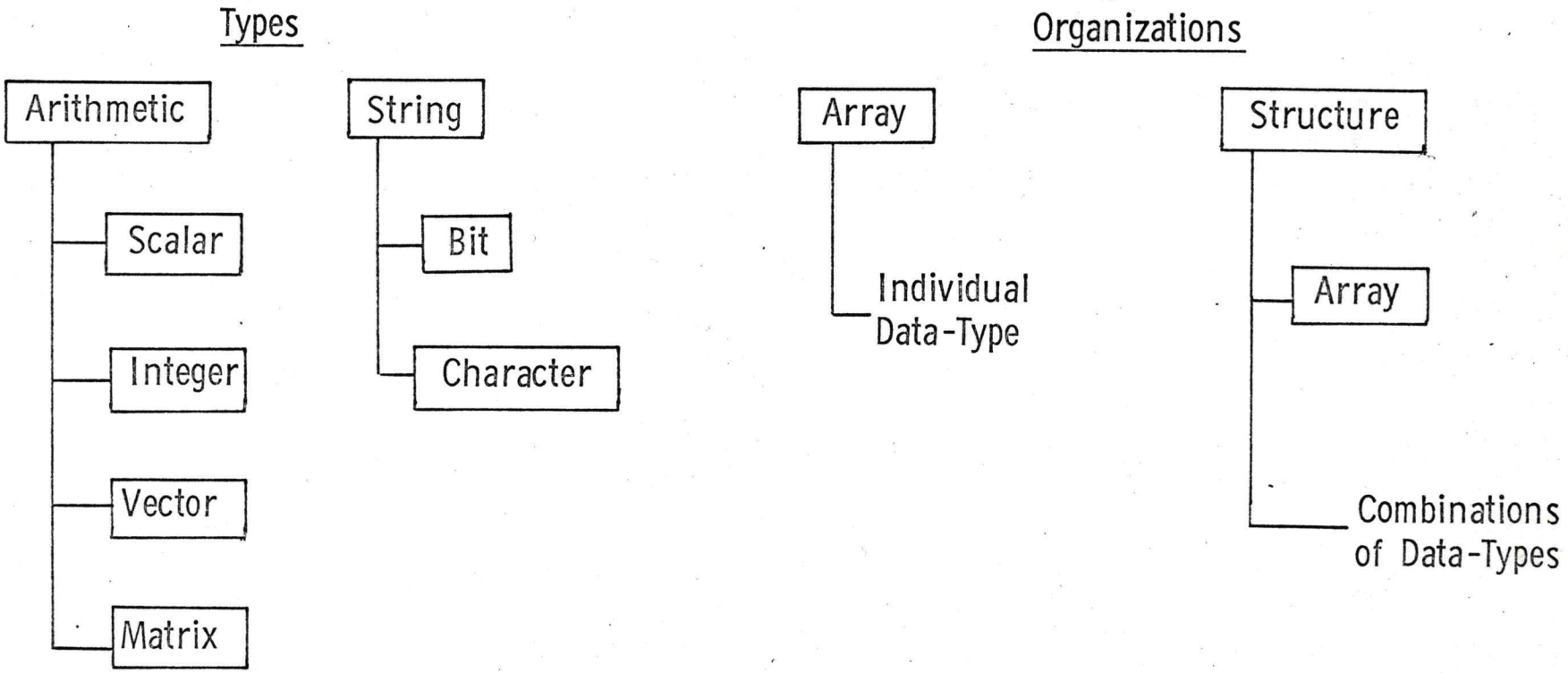


Figure 4-1

4.1 Data Types

4.1.1 Arithmetic Data

An arithmetic data item is one that has a numeric value and may be used in an arithmetic expression. There are four arithmetic types in HAL: scalar, integer, vector, and matrix.

4.1.1.1 Scalar. Scalar variables are numbers represented in a fixed or floating point form. The choice of form will depend on the target machine for a particular compiler implementation of the language (i.e., a compiler will implement either fixed or floating point, but not both). Fixed and floating point are alternate forms of scalars and are not mixed or used together.

4.1.1.2 Integer. An integer is a signed number containing only integral values - a whole number.

4.1.1.3 Vector. A vector corresponds to its normal mathematical definition, having magnitude and direction and represented by n-components within a coordinate system. The individual components of a vector item are scalars, by definition. Vectors obey the standard rules of vector arithmetic.

4.1.1.4 Matrix. A matrix corresponds to its normal mathematical definition, being a rectangular array of m rows and n columns of scalar elements. A matrix obeys the standard rules of matrix arithmetic.

4.1.2 String Data

There are two types of strings in HAL: character strings and bit strings. String data has a length property. A bit string of length one is a Boolean variable which may take on values of only 1 or 0. A bit string of length n can be considered as the concatenation (joining together) of n bit strings of length one. A character string may have fixed or varying length. A fixed length character string of size n always contains n characters. A varying character string is one whose length is dynamically controlled at execution time. A varying character string requires specification of its maximum size.

EXAMPLE

'ABCD? HELP!' is a character string of length 11, including the space between ? and H.

4.2 Data Organizations

A data organization is a collection of data items. There are two kinds of data organizations in HAL: arrays and structures.

4.2.1 Arrays

An array is an ordered collection of elements, known by one name, all of which have the same data type and attributes. For example, every vector in an array of vectors must have the same number of components; every character string in an array of varying character strings must have identical maximum length. The maximum number of dimensions of an array is implementation dependent.

4.2.2 Structures

A structure is a hierarchical organization of data which may contain other structures, arrays, or individual data types. A structure need not consist of identical data elements.

Briefly, when a structure name is declared it is immediately followed by a list of the names and attributes of the elements within it. Each name is preceded by a level number (non-zero integer literal) which identifies the level of organization. All elements having the same level number are at the same level of organization.

The outermost structure is called the major structure and is always at level one; all contained structures are minor structures. All elements of the structure must be at a level greater than one. If a minor structure is at the nth level,

its elements must be specified at the n+1 level. Each item in a structure is given a name. If the name of a structure is referenced, the entire structure, i.e., all elements, are addressed. If the name of an element which is a minor structure is referenced, all of the elements of that minor structure are addressed.

If any of the names assigned to items of a major structure are not unique within a name scope (See Sec. 8), the item must be referred to by the major structure name, the name of the minor structure in which the element is contained, and the name of the element. In referencing, all names of the hierarchy are separated by periods and the entire compound or qualified name becomes the element name. This type of structure, which requires all element names to be fully qualified, is called a qualified structure, and is specified with the attribute QUALIFIED in its declaration. Multiple copies of major or minor structures (i.e., arrays of structures) are permitted; these are limited to one-dimensional arrays.

4.2.2.1 A Not Qualified Example. One example of a hierarchical organization is the table of contents of a book. The name of a structure might be the name of the book and would contain as elements other structures which would be chapters in the book. Each chapter, as a minor structure could contain other elements which would be the sections of the chapter, and so forth. Thus,

```
DECLARE 1 BOOK_NOT_QUALIFIED,
          2 CHAPTER_ONE,
            3 INTRODUCTION,
            3 THEORY,
            3 SUMMARY,
          2 CHAPTER_TWO,
            3 BACKGROUND,
            3 DEVELOPMENT,
          2 CHAPTER_THREE,
            3 ORIENTATION,
            3 FUNCTIONAL_SPECIFICATION,
          2 CHAPTER_FOUR,
            3 CONCLUSIONS,
            3 FUTURE_PLANS;
```

4.2.2.2 A Qualified Example. An example of a structure which must be qualified is:

```
DECLARE 1 A QUALIFIED,
          2B,
          2C,
           3A,
           3B,
          2D,
           3B,
           3C;
```

Since the element names are not unique within the structure, each element must have a qualified name. The qualified names are:

A	is the major structure
A.B	is element B (at level 2)
A.C	is minor structure C
A.C.A	is element A of minor structure C
A.C.B	is element B of minor structure C
A.D	is minor structure D
A.D.B	is element B of minor structure D
A.D.C	is element C of minor structure D

4.2.2.3 An Aerospace Application. In a space application a structure can be used to collect and name sets of associated data elements of different types. Structure commands permit movement of data as well as other limited operations. For example, coasting flight navigation data can be grouped in a NAVIGATION_DATA_FILE structure; i.e.,

```
DECLARE,
    1 NAVIGATION_DATA_FILE,
      2 STATE_VECTOR,
        3 TIME,
        3 POSITION VECTOR,
        3 VELOCITY VECTOR,
      2 W_MATRIX MATRIX,
      2 STATE_CONTROL_FLAGS,
```

```
3 CENTRAL_BODY_FLAG BIT,
3 PERTURBATION_FLAG BIT,
3 MISSION_STATUS_FLAGS,
  4 RENDEZVOUS_FLAG BIT,
  4 ORBITAL_FLAG BIT,
  4 IN_TRANSIT_FLAG BIT;
```

4.3 Attributes

Attributes are used in conjunction with type and organization to specify to the compiler other characteristics associated with a type or organization name. There are five classes of attributes in HAL:

(1) Initialization

(2) Storage class

(3) Memory optimization

(4) Dynamic memory protection

(5) Special

4.3.1 Initialization Attributes

There are two forms of initialization attributes, INITIAL and CONSTANT. Both forms provide a technique which enables the programmer to preset values (numeric and string) into data elements. The use of the CONSTANT attribute will additionally make it illegal to assign new values to the identifier; i.e., to "write" into it. When either form is used as an attribute the other form may not be used. Both initialization attributes may be used with all data types (and arrays of data types). Neither can be used with major or minor structure names, but may be applied to the data elements of a structure.

4.3.2 Storage Class Attributes

Storage class attributes are used to specify storage allocation characteristics of data elements. There are two storage

class attributes: STATIC and AUTOMATIC. STATIC specifies that storage for the data element or organization is to be allocated when the program containing the data is loaded and initiated, and is not to be released until the program execution has been completed or terminated.

AUTOMATIC specifies that storage is to be allocated upon entry into the procedure, function, or task block containing the declaration. AUTOMATIC storage is released upon exit from the block. Since a program may contain procedures, functions, and tasks, data with AUTOMATIC attributes require storage only while the specific procedure, function, or task is active.*

4.3.3 Memory Optimization Attributes

These attributes are used to control the storage assignment and packing of data elements and organizations. There are two attributes: DENSE and ALIGNED. DENSE means that the amount of memory space occupied by the variable is more important than the time required to access it. Consequently, the compiler will attempt to conserve storage space by packing items. The result of packing by the compiler is dependent on the target computer characteristics and the compiler implementation.

ALIGNED means that the time required to access this data is more important than the space it occupies. This attribute

---

* See Secs. 7 and 9 for definitions of program, procedure, function, and task.

will cause the compiler to store the data for efficient access.

4.3.4 Dynamic Memory Protection Attributes

A real time system application may require the coexistence of many processes and the use of common data elements. The control techniques necessary to share these common data elements must include mechanisms for:

a. blocking other users from reading data elements, or organizations, while their current values are being changed (written).

b. preventing changes (writing) when data is being used (read).

For example, one job may be in the middle of using a matrix when it is interrupted by another job which updates the matrix. When the first job was interrupted it had used part of the 'old' matrix values, and when it continues it will be using the updated matrix. This problem could, of course, apply to any data element or organization which is shared among jobs in a real time system.

HAL provides the sharing control attribute LOCK_TYPE which specifies the type of sharing control that is to be used. The LOCK_TYPE attribute causes the compiler to perform checking on all programs which use the specified variable to help insure that the proper locking statements have been employed by the programmer.

The LOCK_TYPE attribute is only useful for STATIC storage and may be included in declarations at the program and COMPOOL* levels. If this attribute is not assigned to a variable, locking statements cannot be used (i.e., there will be no controlled sharing).

The defined locktypes are:

LOCK_TYPE (1)	This class of sharing allows the data to be read by any number of users. Read accesses will wait for writes. Write accesses will wait for any writes and for all previously initiated reads to be completed prior to writing.
LOCK_TYPE (2)	This type of sharing requires that write accesses wait for other writes. Read accesses can occur at any time.

4.3.5 Special Attributes

There are some attributes which can only be applied to certain data types or organizations. These are as follows:
(1) QUALIFIED and NOT_QUALIFIED are attributes which can only be applied to major structures. The attribute specifies whether the element names within that structure will always be qualified, or never qualified. If the NOT_QUALIFIED attribute is used, all

---

* See Section 8

the names within the structure must follow the rules that apply to unstructured identifiers. If the QUALIFIED attribute is used, then item names within the structure may be duplicated elsewhere, and all references to structure elements must be fully qualified.

(2) VARYING is an attribute which can only be applied to character strings. It signifies that the character string length may change at execution time. The maximum size of the string must be declared when VARYING is specified.

(3) The PRECISION attribute is applied to fixed and floating point scalars, vectors, and matrices, and arrays of these data types. It specifies the desired minimum precision of the numerical representation of data within the computer.

(4) The dimension (or length) attribute is applied to vectors, matrices, arrays, bit strings, fixed and varying character strings. It specifies the size and shape of vectors, matrices and arrays, the length of bit and fixed character strings, and the maximum length of varying character strings.

5.0 DATA DECLARATION

5.1 DECLARE Statement

The DECLARE statement is a non-executable statement used to specify explicitly the data organization, type, and attributes of identifiers. There are three forms of the DECLARE statements:

1. Simple DECLARE statement
2. Factored DECLARE statement
3. Structure DECLARE statement

5.1.1 Simple DECLARE Statement

The simple DECLARE statement is used to specify individually the organization, type and attributes of one or more identifiers.

GENERAL FORMAT:

DECLARE<name><specifications>[,<name><specifications>]...;

where <specifications> =

{[<array-spec>][<type-spec>][<attribute-list>]|
{PROGRAM|LABEL|FUNCTION[<type-spec>]}}

When no <specifications> are included, the compiler assigns default* type and attributes.

---

* HAL standard defaults are presented in Appendix B. (Also see Sec. 5.5, DEFAULT Statement.)

5.1.1.1 <array-spec>* An <array-spec> is written as follows:

ARRAY (<dimension-list>)

The array <dimension-list> can specify multiple dimensions in the form <m>[,<m>]... where <m> must be an unsigned integer literal greater than one; e.g., ARRAY (2,3) specifies a 2x3 array.

5.1.1.2 <type-spec>* A <type-spec> is written in one of the following forms:

INTEGER	
SCALAR	[PRECISION (<p>[,<q>])]
VECTOR	[(<length>)][PRECISION (<p>[,<q>])]
MATRIX	[(<rows><cols>)][PRECISION (<p>[,<q>])]
BIT	[(<length>)]
CHARACTER	[(<length>)]
CHARACTER	(<max-length>) VARYING

GENERAL RULES:

1. The <rows> and <cols> in the matrix declaration must be unsigned integer literals greater than one; they define the dimensions of the matrix.
2. For vectors, the <length> defines the vector dimension (i.e., the number of scalar components) and must be an unsigned integer literal greater than one. For bit and character strings, the <lengths> define the number of bits or characters in the object string and must be unsigned integer literals. For varying character

---

* See Sec. 5.4 for alternate form of specifying <array-spec> and <type-spec>.

strings, the <max-length> defines the maximum number of characters that may be assigned to that character variable.

3. The form PRECISION (<p> [,<q>]) defines the desired fixed or floating point precision of scalars, vectors and matrices.

   (a) For floating point, <p> must be an unsigned integer literal which specifies the desired minimum number of significant decimal digits.

   (b) For fixed point, (<p>,<q>) are integer literals such that

   $2^{<p>}$ > maximum absolute value to be represented

   (<p> being the number of integer bits)

   $2^{-<q>} \leq$ minimum absolute value to be represented

   (<q> being the number of fractional bits)

   and <p>+<q> = the minimum number of bits necessary to express the desired range of the scalar.

   (c) In general, the compiler will assign either a single word or a double word for scalars. For floating point, a double word will be assigned if <p> is greater than the number of decimal digits that can be represented in single precision in a particular machine.

   For fixed point, a double word will be assigned if <p>+<q> + the number of sign bits exceeds the number of bits for single precision representation in a particular machine.

   (d) Examples:

   (i) PRECISION(5,3) requires a minimum of 8 bits to accommodate a magnitude range of .125 $\leq$ magnitude

< 32. In this case, presuming a word length L>8, (not including sign bits) the compiler would assign 5 integer bits, and a number of fractional bits equal to L - 5 - the number of sign bits.

(ii) PRECISION(-5,39) requires a minimum of 34 bits to accommodate a magnitude range of $2^{-39} \leq$ magnitude $< 2^{-5}$. In this case, presuming a DP word is necessary, the compiler would assign -5 integer bits and a number of fractional bits equal to -5 + 2L - the number of sign bits.

4. If PRECISION and dimensions are not included in a <type-spec> the compiler will assign defaults. Defaults are presented in Appendix B.

5.1.1.3 <attribute list>. An <attribute list> may be specified by including zero or one attribute from each of the following classes, in any order:

1. Initialization attributes:

INITIAL (<value>)

CONSTANT (<value>)

where <value> must be a literal or a list of literals (see Sec. 5.1.1.4).

2. Storage class attributes:

STATIC

AUTOMATIC

3. Dynamic Sharing Control Attributes:

LOCK_TYPE (<n>)

INTERMETRICS INCORPORATED • 380 GREEN STREET • CAMBRIDGE, MASSACHUSETTS 02139 • (617) 868-1840

where <n> is an unsigned integer greater than zero literal defining the class of sharing control.

4. Storage optimization attributes:

DENSE

ALIGNED

GENERAL RULES:

1. If an attribute does not appear in a simple declaration, the compiler will assign the default* value for that attribute.
2. Restrictions on use of classes of attributes (also see Sec. 8):
   a. Initialization attributes may not be used at the COMPOOL level, nor in declaring <procedure-parameters> and <function-parameters> within procedures and functions (see Sec. 7.4).
   b. Storage class attributes may only be used at the task, procedure, and function levels.
   c. Sharing control attributes may only be used at the program and COMPOOL levels.
   d. Storage optimization attributes may not be used in declaring <procedure-parameters> and <function-parameters>.

5.1.1.4 Initialization. INITIAL and CONSTANT values of vectors, matrices, and arrays may be specified by lists of literals.

---

* See Appendix B.

GENERAL FORMAT:

{INITIAL|CONSTANT}({<list-of-literals>|
<list-of-literals>,*})

where

<list-of-literals> = [<n>#]{[<literal>]|<list-of-literals>}
[,[<n>#] {[<literal>]|<list-of-literals>}]

<n> is an unsigned integer literal.

GENERAL RULES:

1. <n>#<literal> specifies that there are <n> consecutive entries of this <literal> in the list.
2. <n># specifies <n> consecutive entries causing no initialization.
3. <n>#<list-of-literals> specifies that there are <n> consecutive entries of this "sub" <list-of-literals> within the list.
4. ,* indicates a partial initialization. That is, for a vector, matrix, array, and structure of data types not enough literals have been specified. After component-by-component assignment, all the rest are left uninitialized.
5. For vector and matrix declarations, if the number of <literals> in the <list-of-literals>:
   a. is equal to one, all the components are initialized to the <literal>.
   b. is equal exactly to the declared number of components, the vector or matrix is initialized, component-by-component, from the list.

c. In (b), if the number of <literals> is not exactly that required, the list must include a * as its last item and the rest of the vector or matrix will be uninitialized.

6. For array declarations of vectors and matrices, if the number of <literals> in the <list-of-literals>:

   a. is equal to one, all of the vector or matrix components in all the array elements are initialized to the <literal>.

   b. is exactly equal to the declared number of components in a vector or matrix element, each array element is initialized identically, component-by-component, from the list.

   c. is exactly equal to the total number of components, the entire array is initialized, component-by-component, from the list.

   d. In (b) and (c) above, if the number of <literals> is not exactly that required then the list must include a * as its last item and the rest of the array will be uninitialized.

7. For array declarations of scalars, integers, and bit and character strings, if the number of <literals> in the <list-of-literals>:

   a. is equal to one, all of the components are initialized to the <literal>.

   b. is equal exactly to the total number of components, the array is initialized, component-by-component, from the list.

c. In (b), if the number of <literals> is not exactly that required, see 6(d) above.

EXAMPLES:

1. DECLARE V VECTOR(9) CONSTANT (1,0,0,0,0,0,0,0,0,0);

may also be written as

```
DECLARE V VECTOR (9) CONSTANT (1, 8#0);
```

2. DECLARE A ARRAY (4,4) BIT (2)

```
INITIAL (BIN'10', BIN'10', 14#BIN'01');
```

3. ARRAY B ARRAY (3,3) VECTOR (5) INITIAL (0);

All the components of the 9 vectors in the array B, are initialized to 0.

4. DECLARE B ARRAY (3,3) VECTOR (5) INITIAL (25,0,5,0,1);

All 9 vectors in the array, B, are initialized to (25,0,5,0,1).

5. DECLARE B ARRAY (3,3) VECTOR (5)

```
INITIAL (15#0, 15#1, 15#2)
```

The number of literals in the initialization list is equal to the total number of components in the array. The components of the three vectors in the first row are initialized to 0, in the second row to 1, and in the third row to 2.

6. DECLARE B ARRAY (100)

```
INITIAL (5#(1,2,3,4,5),25#, 5#(6,7,8,9,10),*);
```

The first 25 items of the array B are initialized with the repeating pattern (1,2,3,4,5). The next 25 are left uninitialized. Items 51-75 are initialized to the

repeating pattern (6,7,8,9,10). The remaining items are not initialized.

7. DECLARE A ARRAY (10) INITIAL (2),

   B ARRAY (10) INITIAL (#,2,*)

   All the scalars of A are initialized to 2. Only the second scalar of B is initialized to 2, the rest being left uninitialized.

5.1.1.5 Declaration of Program, Function, and Statement Labels. The scopes of procedure, function and statement labels, i.e., the regions of the program in which they are recognized, are defined in Sec. 8.

GENERAL RULES:

1. Statement and procedure labels must be defined (by appearance or by DECLARE statement) before their use in the listing, or at least in the block (i.e. program, function or procedure) in which they are used.
2. Function labels must be defined (by appearance or by DECLARE statement) before their use, regardless of whether the FUNCTION statement and function reference appear in the same block.

3. <type-spec> specifies the data type returned by a function.

4. LABEL and FUNCTION may not be used at the COMPOOL level.

5.1.1.6 Examples of Simple Declaration Statements (Floating Point Implementation).

1. DECLARE I INTEGER INITIAL (65);

   I is an integer with an initial value = 65.

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

   X is a floating point scalar with at least 8 significant decimal digits.

3. DECLARE COMMAND_MODULE_STATE VECTOR (6) STATIC;

   COMMAND_MODULE_STATE is a 6-dimensional vector with single precision components (by default).

4. DECLARE SXT_TO_NB_MAT MATRIX CONSTANT

   (1, 0, 0, 0, 1, 0, 0, 0, 1);

   The matrix is a constant 3x3 identity matrix.

5. DECLARE A ARRAY (5, 3, 4) VECTOR (6) PRECISION (10);

   A is a 5x3x4 array of vectors. Each element is a 6-dimensional vector with components represented to at least 10 significant decimal digits.

6. DECLARE S BIT (100) INITIAL (BIN (100) '1');

   S is a bit string of length = 100. The initial value is all 1's.

7. DECLARE TRAKFLAG BIT AUTOMATIC;

   TRAKFLAG is a bit string of length = 1 (i.e. a Boolean).

INTERMETRICS INCORPORATED · 380 GREEN STREET · CAMBRIDGE, MASSACHUSETTS 02139 · (617) 868-1840

8. DECLARE MESSAGE CHARACTER (3) INITIAL (CHAR(3)'H');

   MESSAGE is a fixed character string of length = 3.

   The initial value is HHH.

9. DECLARE OUT ARRAY (132) CHAR (1) INITIAL (' ');

   OUT is a linear array of 132 character strings of length 1.

   Initially, all characters are blank.

5.1.2 Factored Declaration Statement.

A factored declaration statement eliminates the need for repeated specifications when an attribute or type is applicable to more than one identifier. All of the factors are placed prior to the first name in the declaration statement; other names, with or without specifications, are separated by commas.

GENERAL FORMAT:

DECLARE <factors>,<name>[<specifications>]
[,<name>[<specifications>]]...;

where both the <factors> and <specifications> are of the following form and order:

[<array-spec>][<type-spec>][<attribute-list>]

GENERAL RULES:

1. A <factor> applies to all names appearing in the factored declaration statement, where applicable (e.g., PRECISION will not be applied as a <factor> to a string type included in the statement).

2. If either INITIAL or CONSTANT is used as a <factor>, the other may not be used in the <specifications>.
3. If either STATIC or AUTOMATIC is used as a <factor>, the other may not be used in the <specifications>.
4. If either LOCK_TYPE(1) or LOCK_TYPE(2) is used as a <factor>, the other may not be used in the <specifications>.
5. If either DENSE or ALIGNED is used as a <factor>, the other may not be used in the <specifications>.

5.1.2.1 Examples of Factored Declarations.

1. DECLARE PRECISION (8), A VECTOR (6), B MATRIX (2,2) INITIAL (1,0,0,0);

   All elements of A and B are represented to at least 8 significant decimal digits.

2. DECLARE STATIC,

   A VECTOR (4) INITIAL (0,0,0,1),

   B MATRIX (5,5),

   C ARRAY (20);

   A, B, and C are allocated STATIC storage.

3. DECLARE MATRIX (3,4) INITIAL (0) AUTOMATIC,

   A, B, C PRECISION (10);

   A, B, and C are all (3,4) matrices with AUTOMATIC storage. Initially, all components are set to zero.

4. DECLARE INTEGER, A, B, C, D INITIAL (5);

5. DECLARE BIT DENSE INITIAL (OFF), TRACKING, RENDFLAG;

5.1.3 Structure Declaration Statement

The structure declaration statement is used to declare a structure organization.

GENERAL FORMAT:

DECLARE 1 <struct-name>[(<copies>)][<struct-attributes>],
{,2{<minor-struct-declaration> / <terminal-declaration>}}...;

GENERAL RULES:

1. <copies> must be an unsigned integer literal greater than 1; it defines the number of copies of the structure. For example, DECLARE 1 A (100), 2 B --- etc. declares that there are 100 copies of the structure A.

2. <struct-attributes> are attributes limited to

QUALIFIED|NOT_QUALIFIED

DENSE|ALIGNED

STATIC|AUTOMATIC

LOCK_TYPE(<n>)

   a. If any attributes are not provided in the declaration, the compiler will assign default* values.

   b. It should be noted that attributes apply to the entire structure and,with the exception of DENSE and ALIGNED, cannot be overridden in the minor structures or terminal declarations.

---

* See Appendix B

5.1.3.1 <terminal-declaration>. The <terminal-declaration> is similar to a simple declaration (Sec. 5.1.1); however, only a single name may be declared and the attribute list is limited to INITIAL or CONSTANT, and DENSE or ALIGNED.

GENERAL FORMAT:

[<next-level>]<name>[<array-spec>][<type-spec>]
[INITIAL|CONSTANT](<value>)[DENSE|ALIGNED]]{,|;}

GENERAL RULES:

1. If the <terminal-declaration> is contained in a <minor-struct-declaration> then

   <next-level> equals <this-level> + 1, where <this-level> is the level of the <minor-struct-declaration>, otherwise <next-level> equals 2.

2. The semi-colon (;) is used if the declaration is the last <terminal-declaration> of the structure declaration statement.

5.1.3.2 <minor-struct-declaration>.

GENERAL FORMAT:

<this-level><name>[(<copies>)][DENSE|ALIGNED],
{<minor-struct-declaration> / <terminal-declaration>}...

GENERAL RULES:

1. <this-level> is an unsigned integer literal $\geq$ 2 which identifies the level of hierarchy.

2. If a second <minor-struct-declaration> is contained within a first <minor-struct-declaration> then <this-level> of the

second declaration must be 1 greater than <this-level> of the first declaration.

5.1.3.3 Examples.

1.

```
DECLARE 1 A,                          (1) major structure A
          2 B,                        (2) minor structure B contains
                                          minor structure C and
            3 C,                          terminal element F
              4 D VECTOR(9),          (3) minor structure C contains
              4 E MATRIX (4,4),           terminal elements D and E
            3 F INTEGER;
```

2.

```
DECLARE 1 NAV_STATE(2)LOCK_TYPE(1)NOT_QUALIFIED,     (1)
          2 STATE,                                   (2)
            3 TIME PRECISION(8),                     (3)
            3 R VECTOR(3) PRECISION(10),             (4)
            3 V VECTOR(3) PRECISION(10),             (5)
          2 STATE_FLAGS DENSE,                       (6)
            3 BODY BIT INITIAL(TRUE),                (7)
            3 PHASE BIT,                             (7)
          2 W MATRIX(9,9) PRECISION(10);             (8)
```

Notes:

1. This is a structure whose name is NAV_STATE.
   The number of copies is 2 and it has a sharing class of 1.
2. This is a minor structure called STATE whose elements are defined at the next level.

3. This is a terminal declaration of a scalar element, TIME.
4. This is a terminal declaration of the vector, R.
5. Same as (4) above except name is V.
6. This is a minor structure called STATE_FLAGS whose elements are defined below at the next level.
7. These are terminal declarations of the Boolean variables, BODY and PHASE.
8. This is a terminal declaration of the matrix, W.

5.1.3.4 Structure Initialization. A structure may be initialized by including the INITIAL or CONSTANT attribute in the <terminal-declarations>. If a <terminal-declaration> represents a single copy of the declared data item (i.e. the major structure and minor structures containing this item are single copies themselves) then initialization may be accomplished as described in Sec. 5.1.1.4.

If multiple copies are implied (i.e.,the major structure or minor structure(s) containing this item, or both, have more than 1 copy), two possibilities exist: (1) the data item may be initialized as if it were a single copy; or (2) the initialization <list-of-literals> may be designed to account generally for all copies.

GENERAL RULES:

1. If multiple copies exist and the data item is initialized as if it were a single copy, but not partially initialized, (see Rule 4 of Sec. 5.1.1.4), all copies will receive identical initialization for this data item.

2. If multiple copies exist and it is desired to initialize copies individually, or partially initialized the structure, the <list-of-literals> specifies consecutive entries for the data item, component-by-component, with copies running serially.

EXAMPLES:

1. DECLARE 1 A
   2 B INITIAL (6.061),
   2 C ARRAY(5) INITIAL(1,4#0);

   The structure A is initialized by initializing B and C.

2. DECLARE 1 A (20),
   2 B INITIAL (6.061),
   2 C ARRAY(5) INITIAL(1,4#0);

   The structure A has 20 copies; each is initialized identically.

3. DECLARE 1 A (20),
   2 B INITIAL (15#6.061,*),
   2 C ARRAY(5) INITIAL(15#(1,4#0),*);

   The structure A has 20 copies; the first 15 are initialized identically. The remaining copies are uninitialized.

4. DECLARE 1 A (20),
   2 B INITIAL (6.061,*),
   2 C ARRAY(5) INITIAL(19#(5#),(1,4#0));

   The structure A has 20 copies. The first copy of B is initialized to 6.061, the rest are uninitialized. The first 19 copies of C are uninitialized; the last copy is initialized to (1,0,0,0,0).

5.2 Notation of Data Types and Organizations

5.2.1 Data Type Notation.

The compiler will annotate certain names in order to enhance the readability of the output listing. The notation which signifies data type will be placed on the E line directly over the name on the M line. The notation characters are described below.

Data Type	Notational Character	Examples
VECTOR	-	POSITION = $\overline{R}$
MATRIX	*	REFMMAT = $\overset{*}{M}$
BIT	.	COM_BUFFER_{9}=TRACKFLAG
CHARACTER	,	MSG = $\overset{,}{B}$

There is no data type notation for INTEGER or SCALAR types. These types must be determined from context or from the declaration statements (or symbol table listing).

GENERAL RULES:

The annotation of an operand depends upon the resulting type of the operand itself and not upon the type associated with the identifier being referenced; for example:

1. When an element of a vector is referenced, it is not annotated; i.e., it is a scalar. For example, V_2 is the second scalar element of the vector $\overline{V}$.

2. When an element of a matrix is referenced, it is not annotated since it is a scalar. For example, $M_{1,2}$ is the scalar element in the 1st row, 2nd column of $\overset{*}{M}$.

3. When a row or column of a matrix is referenced, vector notation is used; for example,

   $\overline{M}_{*,2}$ is the 2nd column of the matrix $\overset{*}{M}$

4. When a partition of a matrix is referenced, matrix notation is used; for example,

   $\overset{*}{M}_{1\ TO\ 3,\ 1\ TO\ 2}$ is a partition of the matrix $\overset{*}{M}$; i.e. rows 1, 2, 3 and columns 1, 2.

5.2.2 Array Notation

The compiler will annotate arrays of data types with enclosing square brackets (i.e., [ ]).

If the array consists of vectors, matrices, bit or character strings, then the appropriate data type notation will also be presented. For example,

$[\overline{A}]$	A is an array of vectors,
$[\overset{*}{A}]$	A is an array of matrices,
$[\overset{.}{A}]$	A is an array of bit strings,
$[\overset{,}{A}]$	A is an array of character strings.

GENERAL RULES:

1. When a single array element is referenced, the compiler annotation will be consistent with the resulting data type. For example, suppose A is an array of matrices; then $\overline{A}_{2:*,1}$ has vector notation because the referenced item is a vector

(i.e., the first column vector from the second matrix element of A).

2. When a partition of an array is referenced, array notation is used; for example,

   $[A]_{2 \text{ TO } 4}$ is an array of elements A_2, A_3, A_4 from array A.

   The programmer may include the notation above as part of the input source code. This notation must be consistent with its use (e.g., a * must not be placed over a vector, etc.). If notation is not included then the compiler will annotate the output listing as described.

5.2.3 Structure Notation

The compiler will annotate major and minor structure names with enclosing braces (e.g. {A}).

GENERAL RULES:

1. When a single copy of a structure terminal is referenced, the compiler annotation will be consistent with the resulting data type or array. The notation will be the same as described in Secs. 5.2.1 and 5.2.2.
2. When multiple copies of a structure terminal are referenced, the compiler will annotate the terminal name with enclosing brackets and the appropriate data type. This reference remains a structure organization subject to the restrictions on structure manipulations imposed in Secs. 6 and 7.

EXAMPLES:

1. DECLARE 1 A (5),

```
        2 B BIT(10),
        2 C VECTOR,
        2 D MATRIX;
```

   a. $\{A\}_2$ is the second copy of A.

   b. $\dot{B}_4$; is the bit string in the 4th copy of A.

   c. $\{\bar{C}\}$ is a structure of all copies of the vector C.

   d. $\{\overset{*}{D}\}_{3 \text{ TO } 5}$ is a structure of the last three copies of the matrix D.

2. DECLARE 1 A (5),

```
        2 B CHARACTER(10);
```

   $\{\acute{B}\}$ is a structure of all copies of the string B.

3. DECLARE 1 A,

```
        2 B ARRAY(5) CHARACTER(10);
```

   $[\acute{B}]$ is the array terminal.

Note that while $\{\acute{B}\}$ in 2 and $[\acute{B}]$ in 3 contain the same data they are not identical in form and cannot be used interchangeably.

5.3 Implicit Declarations

In general, HAL requires that all data quantities be declared explicitly. The syntax of explicit data declarations has been presented in Sections 5.1 and 5.2. HAL also permits certain variables to be declared implicitly; namely,vector, matrix, bit and character string data types, by providing a (-), (*), (.), or (,) respectively, on the E line over the name of the data quantity. In the absence of an identifying symbol on the E line, the compiler will interpret the variable to be of a scalar type. The implicit declaration of integers, arrays, and structures is not allowed.

The compiler will assign characteristics, valid throughout the current scope (see Section 8 for further detail on scope of names), to implicitly declared names based on their first appearance in the listing. Thereafter, notation need not be supplied. For example, if $\overline{V}$ is used to declare a variable implicitly, then that variable may be referred to as V in any succeeding statement within the current scope. The compiler will supply the bar (-) on appropriate succeeding appearances of V when it has not been included by the programmer.

The implicit declaration of names as scalar, vector, matrix, bit or character string causes the assignment of default* values for all appropriate attributes.

---

* See Appendix B

5.4 Alternate DECLARE Form

All of the HAL data types, and arrays of these types, may be declared using an alternate form of the DECLARE statement.

GENERAL FORMAT:

```
                <note>
DECLARE <name><sizes> [INTEGER][PRECISION(<p>[,q])]
                                    [<attribute-list>];
```

where

<note> = [-|*|.|,]

and

<sizes> = [<array-shape>|<array-shape>:][<dimension>|
<string length>]

<array-shape> = <m>[,<m>]...

<dimension> = <m>[,n]

<string-length> = <r>

<m>, <n>, <p>, <q>, <r> must be integer literals. In addition, <m>, <n> must be greater than 1; <r> must be greater than 0.

GENERAL RULES:

1. (-), (*), (.), (,) appearing over the name specifies vector, matrix, bit string and character string data types respectively. If <note> and INTEGER are not provided, <name> is a scalar.
2. <dimension> specifies either vector length or the number of rows and columns.
3. <string-length> specifies bit or character length for fixed length strings or maximum length for varying strings.

4. Use of INTEGER, PRECISION, and <attribute-list> are described in Secs. 5.1.1.2, 5.1.1.3, and Sec. 5.1.2.

5. When declaring <procedure- or <function-parameters> (see Sec. 7.4), <note> may be omitted if the proper annotations are included on the parameters appearing in the CALL and function reference statements.

EXAMPLES:

1. DECLARE $\bar{V}_{5,3,4:6}$;
   - a 5x3x4 array of vectors. Each vector is of length 6.
2. DECLARE $\dot{S}_{100}$;
   - a bit string of length 100.
3. DECLARE $\dot{OUT}_{132:1}$;
   - a linear array of 132 character strings. Each string is of length 1.
4. DECLARE $\overset{*}{M}_{6,6}$;
   - a 6x6 matrix.
5. DECLARE A_{50};
   - a linear array of 50 scalars.

5.5 DEFAULT Statement

When variables are implicitly declared, or when variables or functions are explicitly declared with not all characteristics specified, the unspecified characteristics are supplied from a set of default characteristics. The standard set of these is described in Appendix B.

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.

GENERAL FORMAT:

DEFAULT {<type-spec>|[<type-spec>]<length-default-list>};

where

<type-spec> is defined in Sec. 5.1.1.2

<length-default-list> = {<length-default>}...

<length-default> may be one of the following forms:

BIT_LENGTH (<m>)

VECTOR_LENGTH (<m>)

MATRIX_DIM (<m>,<n>)

CHAR_LENGTH (<m>) [VARYING]

where <m> and <n> are literals of integral value.

<type-spec> is used to specify default type; e.g.

```
DEFAULT MATRIX(3,4);
DECLARE A, B, C SCALAR;
```

A and B are declared (3x4) matrices by default. The explicit form, SCALAR, becomes necessary because of this change in default type. <length-spec> is used to specify defaults for bit-string length, vector length, matrix row-column dimension, and character-string length (and VARYING-length). In the case of character strings, if VARYING is provided a maximum length (<m>) must be provided, whether in a DEFAULT or DECLARE statement. For example, the following two statements will cause an error message;

```
DEFAULT CHAR_LENGTH(20);
DECLARE C CHARACTER VARYING;
```

EXAMPLES:

1. ALPHA: PROGRAM;

```
    DEFAULT MATRIX(4,7) BIT_LENGTH(24);
    DECLARE A MATRIX, B, C BIT(10), D BIT;
      :
CLOSE ALPHA;
```

A and B are (4x7) matrices. D is a bit string of length 24.

2. BETA: PROCEDURE;

```
    DEFAULT BIT_LENGTH(16);
    DECLARE E, F BIT, G CHARACTER;
CLOSE BETA;
```

E is a scalar and G is a character string of standard default length. F is a bit string of length 16.

6.0 DATA MANIPULATION

6.1 Expressions

An expression is an algorithm used for computing a value. Variables, constants, literals, built-in functions, and programmer-defined functions combined with operators, form expressions. Expressions are of four types: arithmetic, string, array and relational. The type of an expression is the type of its result and is independent of the types of its operands. In the definitions that follow

<type-operand>={<type-name>|<type-function>|<type-expression>|(<type-expression>)}

where,

<type-name>={<type-variable>|<type-constant>|<type-literal>*}

and

<type- >={<integer- |<scalar- |<vector- |<matrix- |<bit- |<character- >}

6.1.1 Arithmetic Expressions

Arithmetic expressions yield arithmetic values; e.g.,a scalar expression is defined to be an expression yielding a scalar result. There are four types of arithmetic expressions: integer, scalar, vector and matrix.

---

* literals are only defined as being arithmetic, bit and character strings.

6.1.1.1 Integer Expressions. An <integer-expression> is composed of the following elementary operations:

GENERAL FORMAT:

{{[+]|-}<integer-operand>|
<integer-operand>{+|-|<mult>}<integer-operand>|
<integer-operand>**<positive-integer-literal>}

where

<positive-integer-literal> is a positive whole number literal or a bit string literal (interpreted by the compiler in this context as a positive whole number).

GENERAL RULES:

1. <mult> denotes multiplication by logical adjacency. The associated operands must be separated by at least one space (blank) unless one or both of the operands are parenthesized.

2. <integer-operands> and <positive-integer-literals> may be either integers or bit strings. Bit strings are converted implicitly to integers.
3. An integer result can only be derived from operations on <integer-operands>.
4. Division is not an integer operation; dividing one integer by another yields a scalar result.
5. In general, exponentiation will result in a scalar, except when the exponent is a <positive-integer-literal>.

EXAMPLES:

$$-P,\ Q+R,\ F^3,\ Q \dot{S},\ (-P)^4$$

are all integer expressions if P, Q, R, F are declared as integers.

6.1.1.2 Scalar Expressions. A <scalar-expression> is composed of the following elementary operations:

GENERAL FORMAT:

{{[+]|-}<scalar-operand>|
<scalar-operand>{+|-|/|<mult>}<scalar-operand>|
<scalar-operand>**<scalar-operand>|
<vector-operand>.<vector-operand>}

GENERAL RULES:

1. The <scalar-operand> may be a scalar, integer, or bit string except where the above format reduces to an <integer-expression>. Integers are converted implicitly to scalars. Bit strings are converted implicitly, first to integers and then to scalars.
2. Exponentiation is undefined when the <scalar-operand> is negative and the <scalar-operand> exponent has a non-integral value.
3. <vector-operand>.<vector-operand> denotes the vector inner product (dot-product). The dimensions of the two <vector-operands> must be equal.

EXAMPLES:

$$-P,\ P/R,\ P/\dot{S},\ R^{\dot{P}^2},\ \overline{V}.\overset{*}{M}\,\overline{V},\ \dot{S} + \dot{S}/R$$

are all valid scalar expressions if P is declared a scalar, and R is declared to be either an integer or a scalar.

INTERMETRICS INCORPORATED · 380 GREEN STREET · CAMBRIDGE, MASSACHUSETTS 02139 · (617) 868-1840

6.1.1.3 Vector Expressions. A <vector-expression> is composed of the following elementary operations:

GENERAL FORMAT:

{{[+]|-}<vector-operand>|
<vector-operand>{+|-|*}<vector-operand>|
<vector-operand>{/|<mult>}<scalar-operand>|
<scalar-operand><mult><vector-operand>|
<matrix-operand><mult><vector-operand>|
<vector-operand><mult><matrix-operand>}

GENERAL RULES:

1. The <scalar-operand> may be a scalar, an integer or a bit string. Integers and bit strings are converted implicitly to scalars.
2. Addition and subtraction must involve two vectors of identical dimensions.
3. <vector-operand>*<vector-operand> denotes vector cross-product, which is defined only for three-dimensional vectors.
4. Multiplication and division of a <vector-operand> by a <scalar-operand>, and negation of a vector, denote operations on each vector component.
5. <matrix-operand><mult><vector-operand> denotes formal mathematical matrix-vector multiplication; the vector dimension must equal the column dimension of the matrix.
6. <vector-operand><mult><matrix-operand> denotes formal mathematical vector-matrix multiplication; the vector

dimension must equal the row dimension of the matrix.

EXAMPLES:

$-\bar{P}*\bar{V}$, $\bar{P}+\bar{V}$, $5\bar{V}$, $\bar{V}\dot{S}$, $\bar{P}/(A+\dot{S})$, $(\bar{V}.\bar{P})^2\bar{F}$, $\overset{*}{M}\bar{V}$, $\bar{F}/(\bar{V}.\bar{V}\overset{*}{M})$ are all valid vector expressions.

6.1.1.4 Matrix Expressions. A <matrix-expression> is composed of the following elementary operations:

{{[+]|-}<matrix-operand>|
<matrix-operand>{+|-|<mult>}<matrix-operand>|
<matrix-operand>**{-1|T}|
<scalar-operand><mult><matrix-operand>|
<matrix-operand>{/|<mult>}<scalar-operand>|
<vector-operand><mult><vector-operand>}

GENERAL RULES:

1. The <scalar-operand> may be a scalar, an integer, or a bit string. Integers and bit strings are converted implicitly to scalars.
2. Matrix addition and subtraction must involve matrices of identical row and column dimensions.
3. <matrix-operand><mult><matrix-operand> denotes formal mathematical matrix multiplication; the column dimension of the left operand must equal the row dimension of the right.
4. Exponentiation of a matrix is restricted to the inverse (-1) and transpose (T) functions. These may also be written in functional form as INVERSE($\overset{*}{M}$), TRANSPOSE($\overset{*}{M}$).

5. Multiplication and division of a matrix by a scalar, and negation of a matrix, denote operations on each matrix element.
6. <vector-operand><mult><vector-operand> denotes the vector outer product; the result is a matrix whose row and column dimensions are the dimensions of the left and right operands, respectively.

EXAMPLES:

$-\overset{*}{M}\ \overset{*}{N}$, $\overset{*}{M}+\overset{*}{N}$, $\overset{*}{M}^{-1}$, $(\overset{*}{M}\ \overset{*}{N})^{T}$, $\overset{*}{M}/S$, $A\ \overset{*}{N}$, $\bar{V}\ \bar{V}$ are all valid matrix expressions.

6.1.2 String Expressions

String expressions yield string results; e.g., a bit string expression is defined as an expression yielding a bit string result. There are two types of string expressions: bit and character.

6.1.2.1 Bit String Expressions. Bit string expressions may contain bit string operands only. A <bit-expression> is composed of the following elementary operations:

GENERAL FORMAT:

{[NOT] bit-operand>|

<bit-operand>{AND|OR|CAT}<bit-operand>}

GENERAL RULES:

1. NOT complements each bit in the string.
2. AND, OR perform bit-by-bit logical AND and OR on the corresponding bits of the two operands.
3. When the string lengths are unequal, the shorter string is padded on the left with zeros until the strings are of equal length.
4. Concatenation, CAT or (||), links together two bit strings. The length of the result is the sum of the lengths of the two operands.

EXAMPLES:

NOT A, A OR (B AND C), A||NOT B||(B OR C) are all valid bit string expressions.

6.1.2.2 Character String Expressions. A character string expression must involve the concatenation of a character string and a bit string, integer, or scalar operand. A <character-expression> is composed of the following elementary operations:

GENERAL FORMAT:

{<character-operand>||<data-operand>|
<data-operand>||<character-operand>|
<character-operand>||<bit-operand>}

where <data-operand>={<integer-operand>|<scalar-operand>|
<character-operand>}

GENERAL RULES:

1. <integer- and <scalar-operands> are converted implicitly

to character numerical representation.

2. <bit-operands> are converted first to integers and then to characters.

EXAMPLES:

'TEXT||'HELP', A||'TEXT, 'TEXT||(A/S), 'TEXT||(B||C) are all valid character expressions.

6.1.3 Array Expressions

Array expressions yield array results. In general, most of the operations described in Sections 6.1.1 and 6.1.2 are valid for arrays if the operation is valid for elements of the arrays. There are two classes of array expressions: 1) where both operands are arrays; 2) where one operand is an array.

6.1.3.1 Two-array Expressons. For two-array expressions, all of the expressions detailed in Secs. 6.1.1 and 6.1.2 are valid by replacing the <type-operands> by <type-array-operands>. For example, in Sec. 6.1.1.2 the <scalar-operand> becomes a <scalar-array-operand> and the <vector-operand> becomes a <vector-array-operand>.

GENERAL RULES:

1. The two <array-operands> must be dimensionally identical.
2. The indicated operation is performed element-by-element, in sequence, on corresponding elements of the two arrays. For example, let [P] and [S] be two-dimensional arrays. Then [P] + [S] will be executed in the following sequence:

$$P_{1,1} + S_{1,1}$$

$$P_{1,2} + S_{1,2}$$

$$P_{1,3} + S_{1,3}$$

$$\vdots$$

$$P_{n,m} + S_{n,m}$$

3. The resulting array will be of the same dimensions as the <type-array-operands>.

EXAMPLES:

-[P], [P]/[$\dot{S}$], [$\bar{P}$]*[$\bar{V}$], [$\bar{V}$][$\dot{S}$], [R]$^{[P]}$, [$\dot{A}$]OR[$\dot{B}$], [A]||[$\dot{T}$EXT]

are all valid array expressions.

6.1.3.2 One-array Expressions. For one-array expressions, all of the expressions detailed in Secs. 6.1.1 and 6.1.2 are valid if one of the <type-operands> is replaced by a <type-array-operand>.

GENERAL RULES:

1. The indicated operation is performed, in sequence, using the single operand and each element of the array.
2. The resulting array will be of the same dimensions as the <type-array-operand>.

EXAMPLES:

[P]/$\dot{S}$, [P]*$\bar{V}$, $\bar{V}$[$\dot{S}$], R$^{[P]}$, [$\dot{A}$] OR $\dot{B}$, A||[$\dot{T}$EXT], $\bar{V}$/[A], [$\overset{*}{M}$]$\overset{*}{N}$, [A]+5 are all valid array expressions.

6.1.4 Structure Expressions

There are no structure expressions defined in HAL.

6.1.5 Relational Expressions

Relational expressions yield a single true (TRUE/ON) or false (FALSE/OFF) result of a comparison of operands. Relational operators are grouped as follows for use in different contexts:

$$\text{<P>} = \left\{ \begin{array}{ll} \neg= & \text{not equal} \\ = & \text{equal} \end{array} \right\}$$

$$\text{<Q>} = \left\{ \begin{array}{ll} \neg= & \text{not equal} \\ = & \text{equal} \\ < & \text{less than} \\ > & \text{greater than} \\ <= & \text{less than or equal} \\ >= & \text{greater than or equal} \\ \neg< & \text{not less than} \\ \neg> & \text{not greater than} \end{array} \right\}$$

6.1.5.1 Bit String Comparisons.

GENERAL FORMAT:

{<bit-operand><Q><bit-operand>|[¬]<single-bit-operand>}

where

<single-bit-operand>={<single-bit-name>|<single-bit-function>|
<single-bit-expression>|(<single-bit-expression>)}

A <single-bit-expression> is a string expression whose result is a bit string of length one.

GENERAL RULES:

1. When string lengths are unequal the shorter string is padded on the left with sufficient zeros to make the strings of equal length.
2. When comparing bits:

a. Proceeding from left to right, if the first comparison which is not equal is ">" then the total bit-string comparison is true for the relational operators >, >=, ¬<, ¬=, and false for the operators <, <=, ¬>, =.

b. Proceeding from left to right, if the first comparison which is not equal is "<" then the total bit-string comparison is true for the relational operators <, <=, ¬>, ¬=, and false for the operators >, >=, ¬<, =.

c. The total bit-string comparison is true for the relational operator = (and false for ¬=) if and only if all bit comparisons are =.

3. The <single-bit-operand> implies the comparison:

<single-bit-operand> = TRUE

EXAMPLES:

A=B, A>B, A¬<B, etc. are all valid bit string comparisons.

6.1.5.2 Arithmetic Comparisons.

{<integer-operand><Q><integer-operand>|
<scalar-operand><Q><scalar-operand>|
<vector-operand><P><vector-operand>|
<matrix-operand><P><matrix-operand>}

GENERAL RULES:

1. The <integer-operand> may be either an integer or a bit string except where the <integer-operand> comparison reduces to a <bit-operand> comparison. Bit strings are converted implicitly to integers.

2. The <scalar-operand> may be a scalar, integer, or bit string except where the <scalar-operand> comparison reduces to either an <integer- or <bit-operand> comparison. Integers are converted implicitly to scalars. Bit strings are converted implicitly, first to integers and then to scalars

EXAMPLES:

I>J, I<A, $A^{\neg}=\dot{S}$, $\dot{S}<=(A+\overline{P}.\overline{V})$, $\overline{V}=\overline{B}$, $\overset{*}{M}^{\neg}=\overset{*}{N}$ are all valid arithmetic comparisons.

6.1.5.3 Character String Comparisons. Character comparisons have the following general format:

<character-operand><Q><character-operand>

GENERAL RULES:

1. When the string lengths are unequal, the shorter string is padded on the right with sufficient blanks to make the strings of equal length.
2. The character comparison involves left-to-right comparison of corresponding characters in each operand according to a collating sequence which may be implementation dependent.
3. Total character-string comparisons follow the same rules as described for bits in Sec. 6.1.5.1 (Rule 2).

6.1.5.4 Array Comparisons. Array comparisons are valid in comparing two <type-array-operands>, or one <type-array-operand> and one <type-operand>. The result must be a single true or false answer.

GENERAL FORMAT:

{<type-operand>|<type-array-operand>}<P>

{<type-operand>|<type-array-operand>}

GENERAL RULES:

1. Comparisons are on an element-by-element basis.
2. For the operator =, the comparison is true only if all the array elements are equal.
3. For the operator ¬=, the comparison is true if any of the array elements are not equal.

EXAMPLES:

[I] = [A], [A] ¬= S, [P]=[S] are valid array comparisons.

6.1.5.5 Structure Comparisons.

GENERAL FORMAT:

<structure-operand><P><structure-operand>

GENERAL RULES:

1. The two <structure-operands> must be identical in organization.

6.1.6 Precedence Order

In the evaluation of an expression, the order of operations is determined by parentheses and operator precedence. The precedence is divided into two groups, I and II; I is of higher priority. The groups are further ordered by relative priority number (the highest number being the highest priority).

6.1.6.1 Group I Arithmetic Operations

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

Matrix-scalar quotient	2	$\overset{*}{M} / S$
Scalar sum or difference	1	$S \pm S$
Vector sum or difference	1	$\overline{V} \pm \overline{V}$
Matrix sum or difference	1	$\overset{*}{M} \pm \overset{*}{M}$

S, $\overline{V}$, $\overset{*}{M}$ represent scalar (also integer and bit string), vector, and matrix operands.

6.1.6.2 Group II Relational and String Operations.

Operation	Priority	Form
NOT (¬, ^)	5	$\dot{B}$
CAT (||)	4	$\dot{A} \| \| \dot{B}$
(=, ¬=, >, ¬>, <, ¬<, >=, <=)	3	$S_1 >= S_2$
AND (&)	2	$\dot{A}$ & $\dot{B}$
OR (|)	1	$\dot{A} \| \dot{B}$

6.1.6.3 Further Comments on the Order of Operations.

1. 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. Exceptions are noted below.

2. Exponentiation is <u>right to left</u>. Thus,

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

3. Division is <u>right to left</u>. However, vector and matrix expressions may never appear as a denominator in a quotient.

   a. $A/B/C = A/(B/C) = A\ C/B$

   b. $A/BX/CY/D = A/(B\ X/(C\ Y/D)) = A\ C\ Y/B\ X\ D$

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

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

   e. $\overline{V}/\overline{R}.\overline{V} = \overline{V}/(\overline{R}.\overline{V})$

4. Within priority 5, in Group I, deviation from left-to-right order of scalar-vector-matrix products is permitted in order to simplify the computations. For example, in

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

the scalars are first multiplied together, then the vector is multiplied and finally the matrix. Strict left-to-right evaluation would cause 3 matrix-scalar and 1 matrix-vector product. However, since multiplications are associative, the forms are mathematically equivalent. 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.

6.2 Conversions

Conversions of data type and precision can be accomplished implicitly within expressions, and explicitly by special conversion functions. These conversions are detailed below.

6.2.1 Implicit Conversions

6.2.1.1 Data Type. Several implicit data-type conversions are described in Sec. 6.1 as occurring when operands of different types are combined by an operator. These conversions are also noted in the expressions summary of Sec. 6.4. 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.,

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

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

GENERAL RULES:

1. The prefix operations + and - applied to bit strings cause conversion of the strings to integers.
2. For arithmetic operations, other than exponentiation, involving two bit strings or a bit string and an integer, the strings are converted to integers, and the result is an integer.

3. Exponentiation involving integers and bit strings always causes conversion of integers to scalars, and conversion of bit strings first to integers, and then to scalars; the result is a scalar. There is an exception for the exponentiation of an integer by a <positive-integer-literal> (see Sec. 6.1.1.1). In this case the result is an integer.
4. For arithmetic operations involving a bit string and a scalar, the string is first converted to an integer and then to a scalar, and the result is a scalar.
5. For arithmetic operations involving an integer and a scalar, the integer is converted to a scalar, and the result is a scalar.
6. Division always causes the conversions of numerator and denominator to scalars, and produces a scalar result.
7. 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. Conversion of scalar to character produces a character string of specific length to be determined by the implementation. Conversion of integer to character produces a character string of minimum length sufficient to represent the integer as a signed decimal number.

6.2.1.2 Arithmetic Literals. If the representation of an arithmetic literal in the target machine is exactly an integral number (whole number) the literal will be treated as an <integer-operand> in operations and expressions, and with respect to the data-type conversions detailed in Sec. 6.2.1.1. Thus, 2, 2.00, 27.3E+3, 0.1024E+4B-5 are examples of "integer literals". If the literal has a fractional part, it will be treated as a single precision <scalar-operand>. Thus 1.5, 2.386E+2, etc. are examples of "scalar literals".

6.2.1.3 Precision. Implicit conversion of precision occurs when operands of different types or precisions are combined by an operator.

GENERAL RULES:

1. Conversion from bit to integer:

   Bit strings of length less than a machine word length are converted to integers by regarding the string as an unsigned integer. The result will be a full word positive integer. For string length exactly equal to word length, a sign bit is presumed, and the resulting integer will be a full word signed integer. For string lengths greater than a word length, conversion will not be performed; the compiler will issue an error statement.

2. Integer to Scalar:

   a. Floating Point

      When an integer can be represented exactly in single precision floating point then the conversion will be exact.

For larger integers, conversion will approximate the integer by the most significant portion that can be represented in a single precision floating point number.

b. Fixed Point

In conversion from integer to fixed point if the second operand is a scalar then the number of integer bits after conversion will be set equal to that of the scalar operand; otherwise the integer will be treated as a fixed point quantity of all integer bits and no fractional bits.

3. Conversions within Expressions:

a. Floating Point Operations

For operations involving two single precision operands, the result will always be single precision. For operations involving single and double precision, the single precision quantity will be converted to double precision and the operations will be performed in, and the result will be, double precision.

b. Fixed Point Operations

For operations involving two fixed point single precision operands (single word length) the result will be single precision.* For operations involving single

---

* For a particular target machine, the product of two single precision operands may be available to double precision.

and double precision operands, the conversion to double length will follow the same rules as for floating point.

The result of an operation also carries with it an implicit scaling based on the operation involved and the scaling of the operands.

GENERAL RULES:

1. Addition and subtraction

   The resultant number of integer bits equals the maximum of the integer bits of the two operands.

2. Multiplication

   The resultant number of integer bits equals the sum of the integer bits of the two operands.

3. Division

   The resultant number of integer bits equals the difference in the integer bits between numerator and denominator.

4. Exponentiation

   The resultant number of integer bits equals the product of the number of integer bits in the argument and the maximum absolute value of the exponent.

5. Fractional bits for all operations

   The resultant number of fractional bits equals that necessary to fill out a single or double word length, depending upon the context, and the sum of the integer and sign bits.

EXAMPLES:

Presuming a 32 bit word length including 1 sign bit:

1. DECLARE A PRECISION(5,12);

   DECLARE B PRECISION(15,13);

   X = A + B;

   The implicit scaling of the expressions A + B is (15,16). That is, the number of integer bits equals the maximum number of integer bits of A and B. The number of fractional bits equals 32 - (15+1).

2. DECLARE A PRECISION(5,12);

   DECLARE B PRECISION(20,20);

   X = A B;

   The implicit scaling of the expression A B is (25,38). That is, the number of integer bits equals the sum of the integer bits of A and B. The number of fractional bits equals 64 - (25+1). Conversion to double length is caused by the presence of B which requires a double word based on the declaration statement.

3. DECLARE PRECISION(3,12)A,B;

   $X = A^B$

   The implicit scaling of the expression A^B is (24,7). That is, the number of integer bits equals the product of the integer bits of A and the maximum value of B; i.e., $3x2^3 = 24$. The number of fractional bits equals 32 - (24+1).

c. Conversion of Literals

In converting a literal to a fixed point scalar, only the necessary number of integer bits will be used. First the literal is divided by 2^N so that its value is <1 but $\geq$ 0.5. 2^N is therefore the maximum range and N becomes the number of integer bits in the fixed point scaling. P-N-1 bits are assigned as fractional where P is the word length. (This presumes 1 sign bit.) For example, for the literal 250.87 the compiler would assign PRECISION(8, P-8-sign bits). For the literal .004875 the precision is PRECISION(-7, P+7-sign bits).

6.2.2 Explicit Conversions

Three classes of explicit conversions are specified: a data-type conversion to convert from one data-type to another, an array-type to convert a list of mixed data types to an array of a single type, and a set of special bit and character conversions.

6.2.2.1 Data Type. The explicit conversion of data types can be accomplished with the following set of conversion functions:

1. INTEGER (<single-operand>)
2. SCALAR (<single-operand>)
3. BIT[<index-expression>](<single-operand>)
4. CHARACTER[<index-expression>](<single-operand>)
5. VECTOR[<dimension>](<list>)
6. MATRIX[<dimension>](<list>)

where <single-operand> = {<type-operand>|<type-array-operand>}

<list> = <single-operand>[,<single-operand>]. . .

and <dimension> = <m>[,<n>]

<m> and <n> may be <bit-, <integer-, or <scalar-operands>; their values are converted to integers: <m> and <n> must be $\geq$ 2.

<index-expressions>, in the form of subscripts, are detailed in Sec. 6.3.1.

GENERAL RULES:

1. INTEGER converts bit strings, scalars and character strings to integers, and arrays of these types to arrays of integers. A bit string is converted according to the rules stated in Sec. 6.2.1.3. A scalar is converted to a signed full word integer by rounding to the nearest whole number. A character representation of a whole number is converted to a signed full word integer.
2. SCALAR converts bit strings, integers and character strings to scalars, and arrays of these types to arrays of scalars. A bit string is converted first to an integer (as in (1) above) and then to a scalar. An integer is converted to a scalar according to the rules stated in Sec. 6.2.1.3. A character representation of a decimal number is converted to a scalar.

A bit string may be converted directly to a floating (or fixed point) scalar, i.e. not converting to integer first, by use of the BIT pseudo-variable, described in Sec. 7.1.2.3.

3. BIT converts integers, scalars and character strings to bit strings. Integers and scalars are converted to full word bit strings; character strings are converted to the bit length representing the total character string. BIT may be subscripted by an <index-expression> to select a desired range of bits (see Sec. 6.3.1).

   BIT also converts arrays of integers, scalars, or characters to arrays of bit strings.

4. CHARACTER converts bit strings, integers, and scalars to character strings. Scalars are converted to specific length character strings; integers are converted to minimum length character representations (see Rule 7 of Sec. 6.2.1.1). Bit strings are converted to integers first and then to characters. CHARACTER may be subscripted by an <index-expression> to select a desired range of characters (see Sec. 6.3.1).

   CHARACTER also converts an array of bit strings, integers, or scalars, to an array of character strings.

5. VECTOR may be applied to a mixed list of all <type-operands> and <type-array-operands>. The VECTOR conversion-function may be thought of as constructing a scalar one-dimensional list of all the included elements. (Conversions follow SCALAR rules.) Vector, matrix, and array list-elements are equivalent to lists of their components. Matrices are unraveled by rows; arrays are unraveled by the "right-most" index first (i.e. 1,1,1; 1,1,2; 1,1,3; ... 1,2,1, 1,2,2,; etc.).

The rules for "filling" VECTOR are similar to those for initialization (see Sec. 5.1.1.5). The resulting vector is filled element-by-element from the <list>. The number of elements in the list may equal one, or the desired vector dimension. If equal to one (e.g., $VECTOR_6(0)$), the function may be subscripted by the desired dimension and all the components will be assigned values equal to this single element. The default dimension is used if a subscript is not provided.

If equal to the desired vector dimension (e.g., VECTOR $(3\#A^2,B^2,C^2,D^2)$ or $VECTOR_4((A,B,C,D))$, the function may or may not be subscripted by the correct dimension. In either case the vector dimension equals the number of elements in the list.

6. The MATRIX conversion-function constructs a one-dimensional list according to the same rules as for VECTOR. The resulting matrix is "filled" (element-by-element) by rows, and the shape (rows, columns) may be specified by subscripting the function. The number of elements in the list may equal one, or the desired total number of matrix elements. For example,

$$MATRIX_{4,4}(1)$$

$$MATRIX_{4,4}(A,4\#0,A,4\#0,A,4\#0,A)$$

are acceptable forms. If equal to one, the function may be subscripted by the desired dimension and all the components

will be assigned values equal to this single element. When subscripts are not provided, the default matrix dimensionality is used, and the number of elements in the list must equal one, or be consistent with the default.

EXAMPLES:

INTEGER($\dot{S}$), SCALAR($\dot{S}||\dot{B}$), BIT([A]), CHARACTER$_{1\ TO\ 5}$(A), VECTOR(A,$\dot{S}$,I,A^2), MATRIX$_{4,4}$(A,$\dot{S}$,I,A^2,P), SCALAR([$\dot{C}$]), MATRIX$_{8,8}$(1).

are all valid conversion-function applications.

6.2.2.2 Array-Type. The explicit formulation of arrays can be accomplished by adding "array shaping" subscripts to the functions presented in Sec. 6.2.2.1, thus:

1. INTEGER$_{[<array\text{-}shape>]}$(<list>)
2. SCALAR$_{[<array\text{-}shape>]}$(<list>)
3. BIT$_{[<array\text{-}shape>:][<index\text{-}expression>]}$(<list>)
4. CHARACTER$_{[<array\text{-}shape>:][<index\text{-}expression>]}$(<list>)
5. VECTOR$_{<array\text{-}shape>:[<dimension>]}$(<list>)
6. MATRIX$_{<array\text{-}shape>:[<dimension>]}$(<list>)

where

<array-shape> = <m>[,<m>]...

<m> may be <bit-, <integer-, or <scalar-operand>; the value of <m> is converted to an integer before use and must be $\geq$ 2.

GENERAL RULES:

1. INTEGER and SCALAR may be applied to mixed lists of all <type-operands> and <type-array-operands>. Conversions are the same as for the corresponding data-type functions. Except where these forms reduce to the data-type functions of Sec. 6.2.2.1, the number of elements in the lists may equal one, or the total number of array elements. Thus,
   a) if equal to one, the <array-shape> must be specified and all the elements in the array will be assigned values equal to this single list element.
   b) If equal to the total, the <array-shape> may be specified and the elements assigned on an element-by-element basis. If <array-shape> is not provided, a one-dimensional array of length n is presumed, where n is the number of elements in the list.
2. BIT and CHARACTER may be applied to mixed lists of all <type-operands> and <type-array-operands>. Conversions are the same as for the corresponding data-type functions. Except where these forms reduce to the data-type functions of Sec. 6.2.2.1, the number of elements in the list may equal one, or the total number of array elements.
   a) If equal to one, the <array-shape> must be specified and all the elements in the array will be assigned values equal to this single list element. If <index-expression> is not provided, default* string length will be used.

---

* See Appendix B.

b) If equal to the total, the <array-shape> may be specified and the elements assigned on an element-by-element basis. If <array-shape> is not provided a one-dimensional array of length n is presumed, where n is the number of elements in the list. If <index-expression> is not provided, a default* string length is used.

3. VECTOR and MATRIX may be applied to mixed list of all <type-operands> and <type-array-operands>. Conversions are the same as for the corresponding data-type functions. Except where these forms reduce to the data-type functions of Sec. 6.2.2.1, the number of elements in the list may equal one, or the number of components in a single vector or matrix array-element, or the total number of vector or matrix components in the array. Thus,

   a) if equal to one, the <array-shape> must be specified and all the vector or matrix components in the array will be assigned values equal to this single list element. If <dimensions> are not provided, default vector or matrix dimensions will be used.

   b) If equal to the number of components in a single vector or matrix array-element, the <array-shape> must be specified and all the vectors or matrices in the array will be assigned these list values. If MATRIX-<dimension> is not provided then the default matrix dimensions is used. (In this case the number of list elements must be consistent with

*See Appendix B

the default.) If VECTOR-<dimension> is not provided, the vector dimension equals the number of elements in the list.

c) If equal to the total number of components, the <array-shape> and the VECTOR-<dimension> must be specified. If MATRIX-<dimension> is not provided, the rule of (b) above applies. The components will be assigned, from the list, on an element-by-element basis.

EXAMPLES:

1. $\text{INTEGER}_{3,4}(\text{'ACE})$

   - A 3x4 array of integer-elements. Each element is equal to INTEGER('ACE'). 'ACE must be the character representation of an integer (e.g., '-604').

2. SCALAR(A,B,C,15#I)

   - A one-dimensional array of 18 scalar values.

3. $\text{BIT}(A,B^{2},C,D,E)$

   - A one-dimensional array of 5 default length bit strings.

4. $\text{BIT}_{3,2:1 \text{ TO } 8}(A)$

   - A 3x2 array of 8-bit bit strings. All array elements are equal to the eight "left most" bits of A.

5. $\text{VECTOR}_{9:4}(A,0,0,0)$

   - A one dimensional array of 9 four-component vectors. Each vector equals (A,0,0,0).

6. $VECTOR_{2,2:}(B)$

   - A 2x2 array of default length vectors. Every vector component is equal to B.

7. $VECTOR_{2,2:6}(6\#A,6\#B,1,0,0,1,0,0,6\#D)$

   - A 2x2 array of 6-component vectors.

8. $MATRIX_{10:}(A,B,C,--------)$

   - A one-dimensional array of 10 default size matrices. (Note that list length must be consistent with the default.)

9. $MATRIX_{2,3,4:5,5}(5\#A,5\#B,5\#C,5\#D,5\#E)$

   - A 2x3x4 array of 5x5 matrices.

6.2.2.3 Special Character-to-Bit, Bit-to-Character Functions

In addition to the BIT and CHARACTER functions presented in Sec. 6.2.2.1, special subscripting allows binary, octal and hexadecimal conversion from characters to bit string and vice-versa. The general forms are:

a) $BIT_{[<form>]}(<character\text{-}operand>)$

b) $CHARACTER_{[<form>]}(<bit\text{-}operand>)$

where

<form> = {@BIN|@OCT|@DEC|@HEX}

GENERAL RULES:

1. $BIT_{<form>}$ converts a character string (or array of character strings) of binary, octal, decimal or hexadecimal digits into a corresponding bit string (or array of bit strings). @BIN

requires the character string to be made up of only 1's and 0's, @OCT of only 0 to 7, etc.

2. $\text{CHARACTER}_{\text{<form>}}$ converts a bit string (or array of bit strings) into a character string of binary, octal, decimal or hexadecimal digits, depending on the subscript. If the bit string is too short for the required form, it will be padded on the left with zeros.

3. If <form> is not provided, these conversion functions revert to the unsubscripted functions of Sec. 6.2.2.1.

EXAMPLES:

$\text{BIT}_{\text{@OCT}}$('657'), $\text{CHAR}_{\text{@HEX}}$(B),

$\text{CHAR}_{\text{@BIN}}$('10101'), $\text{BIT}_{\text{@HEX}}$('FAD')

are all valid applications.

6.2.2.4 Precision. The precision of expression results can be specified or changed explicitly by the use of the <precision-expression>. That is:

{<type-operand>|<type-array-operand<}$_{\text{<precision-expression>}}$

where

<type-operand>={<integer- |<scalar- |<vector- |<matrix- |<bit- }operand>

and likewise for <type-array-operands>. If the <type-operand> is an expression or a subscripted name, the <operand> must appear within parentheses.

A. Floating point:

GENERAL FORMAT:

<precision-expression> = @p

GENERAL RULES:

1. p must be an unsigned integer literal and is equal to the minimum number of significant decimal places (same meaning as in the PRECISION attribute of the declaration statement).

EXAMPLES (presuming a 32-bit word):

1. DECLARE A PRECISION(10);

   DECLARE B ARRAY(5);

   $A = (B_I)_{@10}+C;$

   B_I is converted from single to double precision (i.e., at least 10 significant decimal places) and the sum is performed in double precision. Note that an indexed name requires parentheses.

2. REPLACE SP* BY '4';

   REPLACE DP BY '10';

   DECLARE X PREC(DP);

   $A = B +(X\ Y)_{@SP};$

   The double precision result of X Y will be converted to single precision. The final sum is computed in single precision.

B. Fixed point:

GENERAL FORMAT:

<precision-expression> =

{@p[,q]|{@<name>|@DP|@SP|@*}[+k -k]}

*See Sec. 7.3.6

where @DP and @SP are keywords; i.e., no spaces are allowed between characters.

GENERAL RULES:

1. @p, q specifies the number of integer and fractional bits (same meaning as in the PRECISION attribute of the declaration statement).
2. @<name>±k specifies the precision to be the same as that of <name> except with the binary point shifted relatively to the right (+) or left (-) by k places; i.e., increasing or decreasing, respectively, the number of integer bits.
3. @DP±k specifies conversion, first to double word length while maintaining the number of integer bits, and then a relative shift of the binary point right (+) or left (-) by k places.
4. @SP±k specifies a relative shift of the binary point to the right (+) or left (-) by k places first, and then conversion to single word length while maintaining the new number of integer bits.
5. @*±k specifies the current word length with the binary point shifted relatively to the right (+) or left (-) by k places.

EXAMPLE:

Presuming a 32-bit word,

$$A = E_{@*-5} + (B + C_{@DP-8})D)_{@E-5}$$

C is converted from single to double precision and the

binary point shifted left eight places. B + C is performed in double precision and the result of the multiplication with D is rescaled to the same scaling as E except that the binary point is shifted left five places. This quantity is added to E after the binary point of E is also shifted left five places.

6.2.2.5 Summary* of Explicit Data-Type Conversions. The following table describes the resulting conversion for each function and operand type (I→S means integer to scalar, etc.):

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	✓

✓: Restores original argument (no operation).

Notes: (1) INTEGER and SCALAR only accept character string arguments which represent whole numbers and scalars, respectively. For example, INTEGER('30672') and SCALAR('362.06E+1') are valid applications.

* This section summarizes the conversions presented in Secs. 6.2.2.1 and 6.2.2.3.

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

(3) BIT and CHARACTER may be subscripted in order to select particular bits and characters, or to modify usage (see Section 6.2.2.3). A character string which represents binary, octal 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

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

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

(4) VECTOR and MATRIX cause the same conversions as SCALAR.

6.3 Subscripts

Subscript notation is used in HAL to specify single elements, or multiple-element partitions,of vectors, matrices, bit- and character-strings, arrays, and structures.

The first element of a vector, the first bit in a bit-string ("left-most" bit) and the first character in a character string ("left-most" character) are noted by the subscript 1, the second by 2, etc. up to the total number of components. Thus, for a 9-element vector the components may be written as

$$V_1\ V_2\ V_3 \cdots V_9$$

For a matrix, 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 example the elements of a 2x3 matrix could be referred to by writing:

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

The above data-types (including integers and scalars) may be arrayed in one, or multiple dimensions, and also organized into hierarchical data structures. In order to select and partition all quantities uniquely it is necessary to distinguish levels of subscripts. In the most general case, this is accomplished by separating structure subscripts from array subscripts with a semi-colon (;) and array subscripts from data-type subscripts with a colon (:). For example,

$$X.Y_{5;3:3,4}$$

refers to the scalar element in the 3rd row, 4th column of the 3rd component of the array of matrices Y which is in the 5th copy of the structure X.

6.3.1 Subscripting Data-Types and Arrays of Data-Types

Subscripting (i.e., selecting or partitioning) is accomplished by attaching a <subscript-expression> to a name, thus

GENERAL FORMAT:

{<type-name>|<type-array-name>}<subscript-expression>

where

<subscript-expression>
= [[<index-expression>[,<index-expression>]...]:]
[<index-expression>[,<index-expression>]]

and

<index-expression>
= <scalar-expression>[TO<scalar-expression>]|
[<scalar-expression>AT]<scalar-expression>}

<scalar-expressions> are evaluated and converted to the nearest integer before use. <scalar-expressions> must be $\geq$ 1.

6.3.2 Single-Element Reference

When referencing a single element the general format of Sec. 6.3.1 reduces to

[[<scalar-expression>[,<scalar-expression>]...]:]
[<scalar-expression>[,<scalar-expression>]]

GENERAL RULES:

1. The expressions to the left of the colon (:) reference the particular array element; the expressions to the right may be used to reference a matrix, vector, or string component.
2. For an array, if "left-expressions" are not provided, the colon being optional in this case, reference is made to an array of the particular matrix, vector, or string components.
3. For a vector or matrix, one or two <scalar-expressions> are used to reference a vector or matrix component.
4. For a bit- or character-string, one <scalar-expression> is used to reference a single bit or character in the string.
5. Use of a number sign (#) in place of a <scalar-expression> means "the last of a particular index".

EXAMPLES:

1. $M_{3,4}$ references the matrix-component in the third row, fourth column.
2. $A_{2,3,4}$ references a scalar or integer array element in the second plane, third row, fourth column.
3. $A_{2,3,4:3,4}$ references the component in the third row, fourth column of the matrix located in the second plane, third row, fourth column of the array, A.
4. $BIT_{16}(A)$ references the 16th bit in the bit representation of A.
5. $TEXT_{8}$ references the 8th character in the string.
6. $\overset{*}{M}_{3,7:}$ references the matrix in the third row, fourth column of the array of matrices, $[\overset{*}{M}]$.

7. $[V_5]$ references an array of the 5th components of all the vectors in the array of vectors $[\overline{V}]$. $[V_5]$ is an array of scalars.

6.3.3 Multiple-Element Partitions

6.3.3.1 The Use of *. An asterisk (*) may be used in place of <scalar-expression> to indicate "all of a particular index", thus establishing a cross-section of a matrix or an array.

EXAMPLES:

1. $\overline{M}_{*,4}$ references the fourth column of the matrix, which is a vector. (That is, all rows, fourth column.)
2. $[\overline{V}]_{2,*:}$ references the vectors in the second row of the array of vectors. Note that $[\overline{V}]_{2,*}$ is itself a one-dimensional array.

6.3.3.2 The "TO" Operator. The TO-operator may be used to reference (or partition) a set of elements by specifying the index limits.

GENERAL RULES:

1. The value of the expression to the left of TO refers to the element at which the partition begins.
2. The value of the expression to the right of TO refers to the element at which the partition ends.

EXAMPLES:

1. $\dot{B}_{5 \text{ TO } 10}$ selects bits 5, 6, 7, 8, 9, 10 from the bit string $\dot{B}$.

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

3. $\overset{\cdot}{[A]}_{P \text{ TO } (P+2),\ 1 \text{ TO } 3:4 \text{ TO } \#}$ partitions a two-dimensional array of bit strings. The result is an array of 3 rows and 3 columns. Each array element is a partition from bit 4 to the last bit of the corresponding original bit string.

6.3.3.3 The "AT" Operator. The AT-operator may be used to reference (or partition) a set of elements by specifying the size (or length) and the beginning index.

GENERAL RULES:

1. The value of the expression to the left of AT indicates the size of the partition.
2. The value of the expression to the right of AT refers to the element at which the partition begins.

EXAMPLES:

1. $\overset{\cdot}{B}_{6 \text{ AT } 5}$ selects 6 bits from the bit string B; i.e.,bits 5,6,7,8,9,10.
2. $\text{BIT}_{10 \text{ AT } P}(A)$ first converts the floating point (or fixed-point) scalar, A, to a bit string and then selects 10 bits starting at P.
3. $\overset{*}{M}_{4 \text{ AT } 5,\ 4 \text{ AT } 7}$ partitions a larger matrix by selecting a 4x4 sub-matrix.
4. $P_{Q \text{ TO } \#}$ partitions a character string by selecting the rest of the string starting at Q.

5. Note that

$$\dot{S}_{2 \text{ TO } 10} \equiv \dot{S}_{9 \text{ AT } 2}$$

6.3.4 Subscripting Structures

Subscripts may be used to specify terminal data elements and specific copies of the major structure, or contained minor structures.

GENERAL FORMAT:

<structure-name><structure-subscript-expression><subscript-expression>

where

<structure-subscript-expression>

= [[<index-expression>[,<index-expression>]...];]

<structure-name> = {<fully-qualified-name>|<not-qualified-name>}

and <index-expression> and <subscript-expression> are defined in Sec. 6.3.1.

GENERAL RULES:

1. When the <structure-subscript-expression> is included, all structure subscripts (major and minor) must be indicated.
2. The use of an asterisk * means "all of the particular index". Thus, $\{A.B.D\}_{26,*;}$ means D in all the copies of B which are in the 26th copy of A. If all indices are filled with * then the <index-expressions> may be omitted optionally; for example, $A.B.D_{*,*;} \equiv A.B.D$

EXAMPLES:

```
DECLARE 1 A(50)NOT QUALIFIED,
          2 B(25),
            3 C ARRAY(4,4) MATRIX(3,3),
            3 D INTEGER,
          2 E(15),
            3 G VECTOR(3),
          2 F BIT(1);
```

The following examples refer to the above structure declaration.

1. $C_{8,10;4,2:1,2}$

   This represents the scalar component in the first row, second column of the matrix which occupies the 4,2 position in the array C. This array is in the 10th copy of B which is in the 8th copy of A.

2. G_2

   This represents the second component of the vector G in all copies of E which are in all copies of A.

3. $\dot{F}_{25;}$

   This represents the single 1-bit bit-string in the 25th copy of A.

4. $\{\overset{*}{C}\}_{23,*;4,*:}$

   This represents all the matrices in the "4th row" of the array C, in all the copies of B which are in the 23rd copy of A.

6.4 Expression Summary

Tables 6.4-1 through 6.4-7 summarize the allowable operations between two operands. In most cases the valid result-type (or error) and any implied data conversions are indicated within the boxes. Array operations are generally valid wherever corresponding data-type operations are also valid.

Operation Prefix : $\{^{<P>}_{<Q>}\}$OPERAND

<P> = $\{^{[+]}_{-}\}$

<Q> = NOT

OPERAND	INTEGER	SCALAR	VECTOR	MATRIX	BIT STRING	CHARACTER STRING
	<P> INTEGER	<P> SCALAR	<P> VECTOR	<P> MATRIX	<P> Integer B→I* <Q> Bit String	ERROR

* B→I means conversion from bit to integer

Table 6.4-1

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

d: dimension check

Table 6.4-2

Operation Multiplication: $OPERAND_1$<mult>$OPERAND_2$

$OPERAND_1$ \ $OPERAND_2$	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	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

Special: 1) Vector DOT product $\overline{V}.\overline{V}$

2) Vector cross product $\overline{V}*\overline{V}$

d: dimension check

INTERMETRICS INCORPORATED • 380 GREEN STREET • CAMBRIDGE, MASSACHUSETTS 02139 • (617) 868-1840

. 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	ERROR
CHARACTER STRING	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR

Table 6.4-4

INTERMETRICS INCORPORATED • 380 GREEN STREET • CAMBRIDGE, MASSACHUSETTS 02139 • (617) 868-1840

. Operation Exponentiation : $OPERAND_1$**$OPERAND_2$

$OPERAND_1$ \ $OPERAND_2$	INTEGER	SCALAR	VECTOR	MATRIX	BIT STRING	CHARACTER STRING
INTEGER	SCALAR I→S, I→S (See Note 1)	SCALAR I→S (See Note 1)	ERROR	ERROR	SCALAR I→S, B→I→S (See Note 2)	ERROR
SCALAR	SCALAR I→S	SCALAR	ERROR	ERROR	SCALAR B→I→S	ERROR
VECTOR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR
MATRIX	All Errors except if "$OPERAND_2$" = {-1 | T};i.e. INVERSE or TRANSPOSE					
BIT STRING	SCALAR B→I→S, I→S (See Note 3)	SCALAR B→I→S (See Note 3)	ERROR	ERROR	SCALAR B→I→S, B→I→S (See Note 4)	ERROR
CHARACTER STRING	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR

Table 6.4-5

Notes:

1. Result is INTEGER if $OPERAND_2$ is a whole number literal ≥ 0.
2. Result is INTEGER if $OPERAND_2$ is a bit string literal which may be converted to an unsigned integer (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).

$$\text{OPERAND}_1 \{^{<P>}_{<Q>}\} \text{OPERAND}_2$$

<P> = {= | ¬ =}

<Q> = {= | ¬ = | > | < | <= | >= | ¬< | ¬>}

Operation Relational :

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> 1

Special: <structure><P><structure>
<array><P><array>

1: OPERAND padded to make lengths equal if necessary.

Table 6.4-6

Operation String :

$$OPERAND_1 \left\{ \begin{matrix} <P> \\ <Q> \end{matrix} \right\} OPERAND_2$$

<P> = ||

<Q> = { || / AND / OR }

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

Table 6.4-7

7.0 STATEMENTS

7.1 Assignment Statements

The assignment statement is used to evaluate an expression and to assign its value to one or more target variables. The target variables may be integer, scalar, vector, matrix, bit and character variables, array variables of these types, subscripted variables, or structures.

GENERAL FORMAT:

[<label>:]<variable-name>[,<variable-name>] = <data-expression>;

where,

<data-expression> = {<arithmetic|<string|<array}-expression

GENERAL RULES:

1. An assignment is performed in the following steps:
   a. subscripts of the target variables are evaluated;
   b. the expression on the right hand side of = is evaluated;
   c. the target variables are assigned.
2. If more than one <variable name> appears on the left hand side of = then all the names must be of identical data organization, (Several different data types may be included.)
3. The dimensionality of the right hand side expression must be identical to that of the left hand side variables with the following exceptions:
   a. string assignments (see Sec. 7.1.2);
   b. assigning zero (0) to arithmetic variables; e.g., $\overline{V}$, $\overline{R}$ = 0; $[\overset{*}{M}]$ = 0; A,B,C,D = 0; are acceptable forms;

c. array assignments (see Sec. 7.1.3)

EXAMPLES:

$\overline{A}, \overline{B}, \overline{C}, \overline{D}$ = VECTOR (1, 0, 0, P/C);

$\overline{D} = A\ \overline{P} + \overset{*}{M}\ \overline{F} + LOG(A)\overline{Z};$

BAKER: $\dot{B}_{1\ TO\ 8} = \dot{C}_{3\ TO\ 4} || \dot{A};$

$\overline{M}_{*,3} = \overline{A} * \overline{B};$

ABLE: $[\dot{A}], [\dot{I}] = BIT_{2,10:}([P]_{20\ AT\ J});$

are all valid assignment statements.

7.1.1 Implicit Conversions

7.1.1.1 Type Conversions. Implicit conversions are performed on the following assignments:

1. Scalar and bit expressions to integer target variables.
2. Integer and bit expressions to scalar target variables. The bit result is first converted to integer, and then to scalar.
3. Integer and scalar expressions to bit target variables. The scalar result is first converted to integer, and then to bit.
4. Integer, scalar and bit expressions to character target variables. The bit result is first converted to integer, and then to character.

EXAMPLES:

```
A = .S||.P;
I = A**2;
A = I - .S;
ABLE: TEXT = .S||.P;
```

7.1.1.2 Precision Conversion. The resultant precision of an expression is converted to the precision of the target variable:

EXAMPLES: (32 bit word length)

1. DECLARE PRECISION(10)VECTOR A;

   $\overline{X} = \overline{A} * \overline{B}$;

   All vectors are floating point; the components of $\overline{A}$ are

held in double precision. $\overline{B}$ is first converted to double precision, the cross-product is performed, and the result is converted to single precision on assignment to $\overline{X}$.

2. DECLARE PRECISION(5,12) A,B;

   DECLARE PRECISION(21,12) C;

   A = B + C;

   All quantities are fixed point; A and B are single length; C is double length. The number of fractional bits for each variable fills out the word length (less sign bit); thus, effectively,

   A and B become (5,26)

   C becomes (21,42)

   The precision conversions are as follows:

   a. B is converted to double precision and added to C.

   b. The result is converted back to single precision; i.e. (5,26) when assigned to A.

7.1.2 String Assignments

7.1.2.1 Bit Strings. When the length of a bit string expression and the target variable are unequal, the expression result is truncated on the left if it is too long, or padded with zeros on the left if it is too short. The resulting value is assigned to the target variables.

EXAMPLE:

$$\dot{S}_{1 \text{ TO } 6} = \dot{S}_{2 \text{ TO } 4};$$

$$\text{BAKER: } \dot{S}_{1 \text{ TO } 6} = \dot{B}_{20 \text{ AT } P};$$

are examples in which padding and truncation will occur.

7.1.2.2 "Boolean" Assignments. A one-bit string may be viewed as a Boolean variable and can be assigned as follows:

$$\dot{A} = \{\{\text{TRUE}|\text{ON}|\text{BIN'1'}\}|\{\text{FALSE}|\text{OFF}|\text{BIN'0'}\}\};$$

Note that TRUE and ON are literally the binary constant BIN'1'. A long bit string may be zeroed by an assignment; i.e.,

$$\dot{B}_{1 \text{ TO } 18} = \text{FALSE};$$

However, $\dot{B}_{1 \text{ TO } 18} = \text{TRUE};$ sets bit 18 equal to 1 and the rest equal to 0.

7.1.2.3 Pseudo-Variable Bit String Assignment. BIT strings may be assigned directly to the bit representation of other data types by using the pseudo-variable BIT.

GENERAL FORMAT:

$$\text{BIT}_{\langle\text{index-expression}\rangle}(\langle\text{variable-name}\rangle) = \langle\text{bit-string-expression}\rangle;$$

GENERAL RULES:

1. <variable-name> may be the name of an integer, scalar, bit, or character variable, or an array variable of these types.

EXAMPLES:

1. $BIT_{6\ TO\ 10}(A) = BIN(5)'1';$

   The scalar A is interpreted as a bit string and the bits 6 to 10 are assigned all 1's.

2. $BIT_{1\ TO\ 8}(C'_{16}) = H'||A';$

   The character C_{16} is interpreted as a bit string and bits 1 to 8 are assigned the result of the string concatenation.

3. $BIT(A) = S'||MANTISSA||EXPONENT;$

   The scalar A is interpreted as a bit string and is assigned a floating (or fixed) point format directly from a bit string expression.

7.1.2.4 Fixed Character Strings. Assignment is similar to that of bit strings except that extension or truncation is applied on the right. Thus, the expression value is truncated on the right if it is too long or padded on the right with blanks if it is too short. The resulting is assigned to the target variables.

EXAMPLES:

1. $C' = 'ABC';$ sets $C'_{1\ TO\ 3} = 'ABC'$ and blanks the rest of C'.
2. $C'_{1\ TO\ 3} = 'ABC';$ leaves the rest of C' alone.
3. $C'_{3\ TO\ 5} = 'ABC';$ leaves the rest of C' alone.
4. $C'_{3\ TO\ 80} = 'ABC';$ leaves characters 1 and 2 alone, and blanks characters 6 to 80.
5. $C'_{3\ TO\ 4} = 'ABC';$ sets $C'_{3\ TO\ 4} = 'AB'$ and leaves the rest of C' alone.

7.1.2.5 Varying Character Strings.

GENERAL RULES:

1. If the value of the expression is longer than the maximum length declared for the variable, the value is truncated on the right. The target string obtains a current length equal to its maximum length.
2. If the value of the expression is not greater than the maximum length, the value is assigned; the target string obtains a current length equal to the length of the value.
3. If the target string is subscripted, the string partition is considered a fixed length character string and the expression is assigned according to the rules of Sec. 7.1.2.3. If the target variable length is shorter than the upper index of the subscript expression, the target variable is padded on the right with blanks and the expression assigned. If the length is longer than the upper index, the expression is assigned, leaving the other characters alone. If the upper index exceeds the string maximum length, the assignment is truncated at the maximum length.

EXAMPLES: (let C be a varying string of maximum length 10)

1. C = 'ABC';sets the length of C to 3
2. C = 'ABC'||'BFG';sets the length of C to 6
3. $C_{3 \text{ TO } \#}$ = CHAR(3)'ABC';assigns;'ABCABCAB' from character 3 to the end; i.e. the length is set to the maximum length of 10.

4. $C_{7 \text{ TO } 9}$ = 'POP';assigns POP to characters 7, 8, 9. If the original length is <6, the string is extended with blanks and the length set to 9. For example, suppose C were equal to 'ABC', then the result of this assignment would be 'ABC⌀⌀⌀POP'.

7.1.3 Array Assignments

GENERAL FORMAT:

[<label>:]<array-variable-name>[,<array-variable-name>]...

= {<type-expression>|<type-array-expression>};

GENERAL RULES:

1. If the expression on the right hand side of = is a <type-expression>, the result of the expression is assigned to every target array element in sequence.

2. If the expression on the right hand side of = is a <type-array-expression>, the result of the expression is assigned to the target variables, in sequence, on an element-by-element basis.

EXAMPLES:

[A] = 5; [$\overline{V}$],[$\overline{W}$] = VECTOR(A,B,C,D);

[A] = [A] + 5; [A] = [B][C];are all valid array assignment statements.

7.2 Declaration Statements

See Section 5.0.

7.3 Control Statements

7.3.1 The GO TO Statement

The GO TO statement causes control to be transferred to the specified statement.

GENERAL FORMAT:

GO TO <label>;

EXAMPLE:

```
X = A;
GO TO BAKER;
ABLE: P = Z;
BAKER: V = M * Y;
```

7.3.2 DO Statements

The DO statements constitute a set of four executable statements. Each DO statement defines a group of statements which are treated as a single unit. The four DO statements are: the simple DO, the iterative DO WHILE and DO FOR, and the selective DO CASE.

A GO TO statement can transfer control from outside a group to a statement within a group. Special care must be taken to initialize necessary quantities in the cases of the iterative DO statements.

7.3.2.1 The Simple DO Statement.

GENERAL FORMAT:

[<label>:]DO;[[<label>:]<statement>]...[<label>:]END[<label>];

GENERAL RULES:

1. <statement> may be any executable statement including another DO statement.

EXAMPLES:

```
BAKER: DO;
         X = A;
         Y = B;
       DO;
         Z = C;
         W = M * V;
       END;
END BAKER;
```

Note that this example has been indented for clarity and does not imply an established input source-output listing format design.

7.3.2.2 DO WHILE Statement. The DO WHILE statement serves as a means of executing a group of statements repetitively as long as a condition is met.

GENERAL FORMAT:

[<label>:] DO WHILE <logical-condition>;

[[<label>:]<statement>]...[<label>:]END[<label>];

where

<logical-condition> = {<relational-expression>{AND|OR}
<relational-expression>|[NOT]<single-bit-expression>}

and the <single-bit-expression> is a bit string expression with only single bit operands (Booleans).

GENERAL RULES:

1. The <logical-condition> is within the loop structure of the DO WHILE group and is re-evaluated each time before execution of the group of statements.
2. When the <logical-condition> is not satisfied the DO WHILE loop is terminated and control is transferred to the first executable statement following the END statement.

EXAMPLE:

```
ABLE: DO WHILE (X>Y AND B6=TRUE)OR([A] = [B]);
          P = LOG(Z);
          M* = N* + Q*;
          Y = M3,2;
      END ABLE;
```

7.3.2.3 The DO FOR Statement. The DO FOR statement serves as a means of executing a group of statements repetitively for a list of values of a control variable and a logical condition.

GENERAL FORMAT:

```
[<label>:]DO FOR<{scalar / integer} variable> =

  <for-list element>[,<for-list element>]...
[WHILE<logical-condition>];
[[<label>:]<statement>]...[<label>:]END[<label>];
```

where

<for-list element> = <{scalar / integer} expression>[<to-expression>

[<by-expression>]]

<to-expression> = TO<{scalar / integer}expression>

and

<by-expression> = BY<{scalar / integer}expression>

GENERAL RULES:

1. The scalar or integer assignment means that a single variable (control variable) will be assigned scalar or integer values.
2. The control variable takes on the successive values specified by the <for-list elements>. If the element is simply a scalar or integer expression, the control variable is set equal to this value prior to a pass through the loop. If the element involves <to- and <by-expressions>, the control variable is compared with the value of the <to-expression> prior to each pass, and is incremented by the <by-expression> at the conclusion of each pass.
3. If the <by-expression> is not provided, the group of statements will be evaluated repeatedly, incrementing the assigned control variable by 1 until the control variable is greater than the value of the <to-expression>.
4. If the <by-expression> is provided, the group will be evaluated repeatedly, incrementing the assigned control variable by the value of the <by-expression> until the control variable exceeds (if the <by-expression> is positive) or is less than (if the <by-expression> is negative) the value of the <to-expression>.

5. The effect of the <logical-condition>, if provided, is the same as for the DO WHILE statement.

6. The <to- and <by-expressions> are not within the loop structure of the DO FOR group and are evaluated only once. The <logical-condition> is within the loop and is evaluated before each pass.

EXAMPLES:

1. DO FOR I = 1 TO 10 BY 2;

```
        X = Y;
        Y = X^2 + Z;
    END;
```

This loop will be executed five times.

2. BAKER: L = Q/R;

```
ABLE: DO FOR I = P TO (N/S) BY L WHILE N > 0.046;
        X = Y^2 + A_I;
        N = N - .006 X;
END ABLE;
```

Note that the value of the <to-expression> (N/S) is only computed once. The condition N >0.046 is applied before each pass.

7.3.2.4 DO CASE Statement. The DO CASE statement provides a means of executing a selected statement from a group of statements.

GENERAL FORMAT:

```
[<label>:] DO CASE <case-expression>;
        [[<label>:<statement>]...[<label>:]END[<label>];
```

where the <case-expression> can be either an integer expression

or a scalar expression. The result of a <case-expression> is rounded to the nearest integer before use.

GENERAL RULES:

1. The <case-expression> results in an integer, used to designate which one of the included statements will be executed. A value of 1 specifies the first statement, 2 the second, and so on. An integer result outside the case range will be in error. The compiler will annotate the listing, indicating Case 1, Case 2, etc.
2. The <statements> may be any of the executable statements, including other DO statements.

EXAMPLES:

```
ABLE: DO CASE N;
        X = Y**2;           /*CASE 1*/
        DO CASE P;          /*CASE 2*/
            F = A + B;              /*CASE 1*/
            G = M * V;              /*CASE 2*/
        END;
        GO TO CHARLIE;      /*CASE 3*/
        Z = W + B;          /*CASE 4*/
    END ABLE;
```

7.3.3 END Statement

The END statement delimits the do-groups.

GENERAL FORMAT:

[<label>:]END[<label>];

GENERAL RULES:

1. The END statement terminates the group headed by the nearest preceding DO statement which has not already been terminated by an END statement.

2. If a label follows END, the corresponding DO statement must have that same label.

7.3.4 The IF Statement

The IF statement specifies the evaluation of a logical condition and a consequent flow of control dependent on whether the condition is true or false.

GENERAL FORMAT:

```
[<label>:]IF<logical-condition>THEN[<label>:]{<statement>|
                <basic-statement>ELSE[<label>:]<statement>}
```

where

a. the <logical-condition> has a true or false result; its format was described in Sec. 7.3.2.2.

b. the <basic-statement> is any executable statement except an IF or END statement.

c. the <statement> is any executable statement (including another IF statement) except an END statement.

GENERAL RULES:

1. If ELSE is not included, a true condition will cause execution of the statement following, and a false condition will cause control to pass to the statement following the IF statement.
2. If ELSE is present then a true condition will cause execution of the <basic-statement> following THEN and a false condition will cause transfer of control to the statement following ELSE.
3. 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.

EXAMPLES:

1.
```
ABLE; IF B THEN IF C THEN X = 5;
                    ELSE D: GO TO BAKER;
CHARLIE: Y = 6;
```

2.
```
IF X>100 AND Y<3 THEN P: GO TO ABLE;
ELSE IF B OR C THEN
          DO;
            Y = A + B;
            ABLE: Z = M* V;
          END;
        ELSE Y = A - B;
```

* IF statements within preceding do-groups do not apply.

7.3.5 The NULL Statement

The NULL statement is a no-operation.

GENERAL FORMAT:

```
[<label>:];
```

EXAMPLE:

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

7.3.6 REPLACE Statement

The REPLACE statement provides a means of specifying the substitution of a string of characters for an identifier. The character string must be contextually correct where substituted. This is a compile-time feature and not a run-time executable statement.

GENERAL FORMAT:

```
REPLACE<identifier>BY'<character-string>';
```

GENERAL RULES:

1. The <identifier> may not be a keyword or any word used by the language syntax (e.g.,TO or WHILE).
2. The <character-string> must be written in one-line format.
3. The <character-string> will be substituted, literally, whenever the identifier is encountered within the program. Substitution is accomplished within the compiler and does not appear in the listing.
4. The <identifier> may not be a <parameter> in the PROCEDURE or FUNCTION statements.

EXAMPLES:

```
1.  REPLACE P BY 'LOG(F) + Y**2';

            B = Z + P;

2.  REPLACE D BY 'GO TO ABLE;';

    IF B>6 THEN G ELSE D

3.  REPLACE A BY'(106.2B-32)';

    DECLARE B INITIAL A;

4.  REPLACE FIRE_JETS BY 'GO TO F_J;';
    :
    FIRE_JETS
        :
    F_J: DO; - - - - -

         - - - - -

         END;
```

7.4 Procedures and Functions

Procedures and functions are subroutines consisting of one or more statements which are intended to be written once but used at various points throughout a program. The primary distinction between procedure and function is that the procedure must be invoked by a CALL statement, and may accept and return lists of parameters of different data types, while a function is invoked by the appearance of its name as an operand and can return only a single data type or result.

7.4.1 Procedures

7.4.1.1 PROCEDURE Statement. The PROCEDURE statement identifies the beginning of a block of statements which forms a procedure; it defines the entry point and specifies the input and output parameters.

GENERAL FORMAT:

```
<procedure-label>:PROCEDURE[<procedure-parameters>]
            [ASSIGN<assign-parameters>];
            {[<label>:]<statement>|[<label>:]RETURN;}...
   [<label>:]CLOSE[<procedure-label>];
```

where

<procedure-parameters> = (<name>[,<name>]...)

and

<assign-parameters> = (<name>[,<name>]...)

GENERAL RULES:

1. The <procedure-parameters> are interpreted as input data to the procedure. They are formed parameters; that is, they do

not exist in of themselves and are no more than dummy variables that indicate what to do, within the procedure block, with the actual <call-arguments> in the CALL statement (see Sec. 7.4.1.2). If the <call-arguments> are names (not expressions), the <procedure-parameters> are in fact the same data locations as the <call-arguments>.

2. The <procedure-parameters> may not be assigned values within the procedure block; i.e., they may not appear on the left hand side of an assignment statement.

3. The <assign-parameters> are also dummy variables and represent the computed output data of the procedure. They are in fact the same data locations as the <assign-arguments> in the CALL statement.

4. The data-types and attributes of corresponding <call- and <assign-arguments>and <procedure- and <assign-parameters> must be identical (see Sec. 7.4.2.3).

5. Execution of a procedure may be terminated by a RETURN statement (see Sec. 7.6) or by logically reaching the CLOSE statement; control is returned to the caller.

6. Local variables may be defined within a procedure block by declaration statements and implicit declarations. See Sec. 8.1.1 for discussion of Scope of Names.

INTERMETRICS INCORPORATED · 380 GREEN STREET · CAMBRIDGE, MASSACHUSETTS 02139 · (617) 868-1840

EXAMPLE:

```
TIME: PROCEDURE(Ā,B)ASSIGN(C̄);
      C = Ā(F1+F2 B+F3 B^2);
      IF B>L THEN RETURN;
      C̄ = 100 C̄;
      CLOSE TIME;
```

7.4.1.2 CALL Statement. A procedure is invoked by a CALL statement which may define a set of input and output arguments and which transfers control to a specified entry point.

GENERAL FORMAT:

[<label>:]CALL<procedure-label>[<call-argument>]
[ASSIGN<assign-arguments>];

where <procedure-label> is the label associated with the PROCEDURE statement and

<call-arguments> = ({<name>|<expression>}[,{<name>|<expression>}]...

<assign-arguments> = (<name>[,<name>]...)

GENERAL RULES:

1. <call-arguments> will be used only as input information to the procedure.
2. <assign-arguments> may be assigned values computed within the procedure blocks and may also supply input information to the procedure.

EXAMPLE:

```
ABLE: PROCEDURE;
      V̄ = X̄ + Ȳ;
      CALL TIME (V̄, T) ASSIGN(W̄);
      S̄ = W̄*V̄;
      P = M* S̄;
CLOSE ABLE;
TIME: PROCEDURE(Ā,B) ASSIGN(C̄);
      C̄ = Ā(F1+F2 B+F3 B^2);
      IF B>L THEN RETURN;
      C̄ = 100 C̄;
CLOSE TIME;
```

7.4.2 Functions

7.4.2.1 FUNCTION Statement. The FUNCTION statement identifies the beginning of a block of statements which form a function; it defines the entry point and specifies the data-type of the result.

GENERAL FORMAT:

```
<function-label>:FUNCTION[<function-parameter>]
      [<type-spec>];
      {<statement>|RETURN(<expression>);}...
      [<label>:]CLOSE[<function-label>];
```

where

```
<function-parameters> = (<name>[,<name>]...)
```

GENERAL RULES:

1. If <type-spec> is not provided and is not specified in a declaration, default characteristics are used.
2. The <function-parameters> are interpreted as input data to the function. They are formal parameters; that is, they do not exist in of themselves and are no more than dummy variables that indicate what to do, within the function block, with the actual <function-arguments> in the function reference (See Sec. 7.4.2.2). If the <function-arguments> are names (not expressions), the <function-parameters> are in fact the same data locations as the <function-arguments>.
3. The <type-spec> specifies the characteristics of the function result. Arrays and structure organizations are not allowed.
4. The data-types and attributes of corresponding <function-arguments> and <function-parameters> in the reference and FUNCTION statements must be identical. (See Sec. 7.4.2.3)
5. A function must have at least one RETURN statement and execution may only be terminated by a RETURN statement; control is returned to the caller. An error message will be generated at run-time if the process logically reaches the CLOSE statement.
6. Local variables may be defined within a function block (see Sec. 8.1.1).

7.4.2.2 Function Reference. A function is invoked by a function reference which may define a set of input arguments and which transfers control to a specified entry point.

GENERAL FORMAT:

<function-label>[(<function-arguments>]...]

where

<function-arguments> = ({<name>|<expression>}[,{<name>|<expression>}]...

GENERAL RULES:

1. The <function-arguments> will be used only as input information to the function.
2. The <function-label> is treated as an operand whose value is computed within the function.

EXAMPLE:

```
ABLE: A* = M* TRACER(B*+C*);

TABLE: GO TO BAKER;

     TRACER: FUNCTION(Q*);

        R = TRACE(Q* Q*-1 + Q* + Q* Q* + Q* Q* Q*);

        IF R>100 THEN RETURN R;

        ELSE RETURN 0;

     CLOSE TRACER;
```

7.4.2.3 Parameter Declarations. Scalar, vector, matrix, bit and character string parameters may be declared implicitly, with default attributes, by their appearance in PROCEDURE and FUNCTION

statements with appropriate annotation. Thus, for example

ABLE: FUNCTION(A,$\overline{B}$,$\overset{*}{\overline{C}}$,D,E);

Array-parameters and parameters with other than default attributes require explicit DECLARE statements internal to the procedure or function blocks, in addition to appearing in the lists of parameters (annotation being optimal).

For certain applications it may be convenient not to specify the length or dimensions of parameters but instead, have the parameters take on these characteristics from the corresponding arguments in the CALL or function-reference statements. This may be accomplished by substituting an asterisk (*) for the length or dimensions in the DECLARE statements.

GENERAL RULES:

1. With reference to Sec. 5.1.1, vector length, bit length, character length and varying character maximum length may be specified by asterisks.
2. For arrays, shape may be specified by combinations of literals and/or asterisks.
3. For matrices, rows and columns may be specified combinations of literals and/or asterisks.

EXAMPLES:

1. TIME: PROCEDURE($\overline{A}$) ASSIGN($\overline{C}$);

```
    DECLARE VECTOR (*), A, C;
        .
        .
        .
    CLOSE TIME;
```

2. ABLE: PROCEDURE($\overline{V}$,$\overset{*}{M}$) ASSIGN($\overline{N}$);

```
        DECLARE V VECTOR (*);
        DECLARE M MATRIX (3,*);
        DECLARE Y VECTOR (4);
        Y = M V;
          .
          .
          .
    CLOSE ABLE;
```

Comment: $\overline{V}$, $\overset{*}{M}$, $\overline{N}$ are parameters. $\overline{Y}$ is a local variable. Note that $\overline{N}$ is declared by appearance as an <assign-parameter>. With no explicit DECLARE statement for $\overset{*}{N}$, default attributes are used.

7.4.2.4 Functions of an Array. When a <function-argument> is an array, the corresponding <function-parameter> may be either a single variable or an array-variable of the same data-type. If a single variable, the function has been designed to operate on each array element sequentially, element-by-element. If an array, the function accepts the input array as a unit.

EXAMPLES:

1.
```
    DECLARE B ARRAY (4);
    DECLARE C ARRAY (4);
    [C] = FUZZ([B]);
      FUZZ: FUNCTION(X);
            TEM = 1 + X/2 + X**2/6 + X**3/24;
            RETURN(TEM);
      CLOSE FUZZ;
```

FUZZ will be executed 4 times and return 4 scalar results which will be assigned to the component of array C, in sequence. If the <function-parameter> is an array-variable, then the function accepts the input array as a unit. The function will operate on the "inner-most" free indices of the array argument consistent with the expression.

2. DECLARE B ARRAY (4) VECTOR;

```
BUZZ: FUNCTION([X]);
      DECLARE X ARRAY (4) VECTOR;
      ADD = X1: + X2: + X3: + X4:;
      RETURN(ADD);
      CLOSE;
A = BUZZ ([B]);
```

BUZZ returns a single vector.

3. DECLARE A ARRAY(5), B ARRAY(5,4);

```
[A] = SUM([B]);
```

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

```
DO FOR I = 1 TO 5;
   AI = SUM([BI,*]);
END;
```

Note that SUM is a linear array function.

7.5 Programs

In HAL, a program is the smallest compilable unit. It may contain all of the program elements and statements defined, except PROGRAM statements; i.e. declarations, executable statements, procedures, etc.

7.5.1 PROGRAM Statement

GENERAL FORMAT:

```
<program-label>:PROGRAM;
        {<all-statements>}...
[<label>:]CLOSE[<program-label>];
```

GENERAL RULES:

1. <all-statements> may contain all valid syntax.
2. A program may be called using the CALL statement with the <program-label> (no parameters may be passed).
3. Execution of a program may be terminated by a RETURN statement (See Sec. 7.6) or by logically reaching the CLOSE statement; control is returned to the caller. (Also, see the real-time control statement TERMINATE in Sec. 9.)
4. A program can be scheduled in real-time through the system executive (see Sec. 9).

7.5.1.1 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-label>;

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. A program cannot call itself.

7.6 RETURN Statement

The RETURN statement terminates the execution of a procedure, function or program.

GENERAL FORMAT:

[<label>:]RETURN[<expression>];

GENERAL RULES:

1. In terminating a procedure or program, the RETURN statement must not include an expression.
2. In terminating a function the data type of the <expression> must agree with the type specified for the function.
3. The result of <expression> may not be an array.
4. The RETURN statement returns control to the caller.

7.7 CLOSE Statement

The CLOSE statement delimits the blocks of HAL statements which have name scope*; viz. procedures, functions, programs, tasks** and update** blocks.

GENERAL FORMAT:

[<label>:]CLOSE[<label>];

GENERAL RULES:

1. The CLOSE statement delimits the block headed by the nearest preceding PROCEDURE, FUNCTION, PROGRAM, TASK or UPDATE statement which has not already been delimited by a CLOSE statement.
2. If a label follows CLOSE, the corresponding "heading" statement must have that same label.
3. For a procedure, program or task, execution of the CLOSE statement returns control to the caller.
4. For an update block, execution of the CLOSE statement causes no operation.
5. For a function, execution of the CLOSE statement is an error.

---

* See Sec. 8.1.1

** See Sec. 9.

8.0 HAL PROGRAM ORGANIZATION

A HAL program organization consists of one or more independently compilable programs and a symbolic library. The library may contain a common data pool (COMPOOL) and all valid HAL syntax. Variables declared in the COMPOOL are available for use in any program. Library routines may be compiled into any program by directive. The organization is designed to provide programmer convenience and flexibility and yet maintain control and visibility of commonly used data.

8.1 Program Structure

A program (<program-block>) is the smallest compilable unit and is delimited by PROGRAM and CLOSE statements. The <program-block> may contain the following elements:

```
<program-block> = <program-statement>[<declare-group>]
 {<all-statements>|<task-block>|<sub-block>}...
                 <close-statement>
```

where,

```
<declare-group> = [<replace-statements>][<outer*-statements>]
                  [<default**-statements>][<declare-statements>]
<all-statements> =  all executable statements including
                    do-groups and update***-blocks
<task-block> = <task***-statement>[<declare-group>]
    {<all-statements>|<sub-blocks>}...<close-statement>
```

---

* See Sec. 8.1.2
** See Sec. 5.5
*** See Sec. 9.4.2

<sub-block> = {<procedure-statement>|<function-statement>}
[<declare-group>]
{<all-statements>|<sub-blocks>}...<close-statement>

<program-blocks> and contained <task-blocks> and <sub-block> (and further nested <sub-blocks>) define boundaries, or regions, within which names and labels are recognized and may be used for computation and control. The region in which a name or label is potentially recognizable is called its scope.

8.1.1 Scope of Names

The scope of a name is defined as the block in which it is declared and extends to all contained (and nested) blocks. For example, names defined in the COMPOOL are potentially recognized throughout every <program-block>; names defined in a <program-block> may be recognized in all enclosed <task- and <sub-blocks>; names defined in <task- and <sub-blocks> may be recognized in all nested <sub-blocks>, etc. Note that a name defined within an inner block is never recognized in an outer block. (To be more precise, the named variable or constant is never recognized in an outer block; the name itself, designating various data quantities, may appear in a number of blocks.)

Identical name declarations for two or more quantities cannot exist within the same name scope; however, duplicate names are allowed in different scopes. The following example illustrates this principle:

```
ABLE:  PROGRAM:
    DECLARE VECTOR(5) A, B;
        .
        .
        .
    BAKER: TASK;
        DECLARE A INTEGER;
            .
            .
            .
        CHARLIE: PROCEDURE;
            DECLARE A BIT;
            DECLARE X;
                .
                .
                .
        CLOSE CHARLIE:
    CLOSE BAKER;
    GRAB PROCEDURE;
        DECLARE X VECTOR(4);
            .
            .
            .
    CLOSE GRAB;
CLOSE ABLE;
```

Comments:

1. The vectors $\overline{A}$ and $\overline{B}$ have been declared at the program level and their scope is the entire program unless superseded by a declaration in an inner block (or obscured by omission from an OUTER statement, see Sec. 8.1.2).
2. In the task BAKER, A is an integer (the vector $\overline{A}$ will no longer be recognized); $\overline{B}$ is recognized.
3. In the inner procedure CHARLIE, A is re-defined again, being recognized within CHARLIE as a bit string. The scope of $\overline{B}$ remains the entire program.
4. In the procedure GRAB, $\overline{A}$ and $\overline{B}$ remain defined at the program level and $\overline{X}$ is declared at a local level. Note that although the names are the same, the variables represented by X in GRAB and X in CHARLIE are different.

8.1.2 Selective Inclusion of Outer Names

In the previous example names declared in an outer block were known to the inner block unless the inner block declared the same name. Another mechanism is provided to include (or reject) outer names selectively. The OUTER statement is an explicit means of specifying which "outer" names are to be known within the

block; outer names which would have been known but which are not listed are hidden. Thus, for example,

```
ABLE: PROGRAM;
      DECLARE A, B, C, D, E;
                .
                .
                .
      BAKER: TASK;
          OUTER B, D;
          DECLARE A;
                .
                .
                .
```

The program ABLE has declared names A, B, C, D, E which would be known in the task BAKER. However, the OUTER statement in BAKER only allows B and D to be known, and further BAKER redefines A locally. Note that the absence of an OUTER statement means that all outer names will be recognized within a particular inner block, while the inclusion of OUTER with no list of names completely isolates the inner block from any outer-declared names.

8.1.2.1 Inclusion of Structure Names. Structure names may also be included by listing the structure name(s) in the OUTER statement according to the following rules:

1. For a qualified structure, only the major structure name may be listed; the result being that all associated minor structure and terminal names are included implicitly.

2. For a not-qualified structure, the major structure name and all associated minor structure and terminal names may be listed. Only those names that are listed will be recognized within the block.

8.1.2.2 Implicit Declaration of Names. Implicit declaration of names will not be allowed unless the block contains an OUTER statement. Only those names appearing in an OUTER statement and those explicitly declared within a block will be unavailable for implicit declaration.

When no declarations precede the PROGRAM-statement, the compiler permits implicit declarations at the program level as though an OUTER-statement with no list had been included.

8.1.3 Scope of Labels

Labels are used for control purposes; to transfer control as in GO TO <label> or CALL <label>. The labels "name" the entry-points to programs, tasks, functions, procedures, updates, do-groups and statements. The scope of labels generally follows the same rules as for names with the following exceptions:

1. The GO TO and CALL statements imply the existence of a label. If the label 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 (before or after the statement), the statement refers to this label.

2. If a GO TO or CALL statement refers to a label in an outer block, the label must appear in the listing prior to the statement or be declared explicitly in a DECLARE statement.

3. Function names (i.e., <function-labels>) must always be defined in the listing prior to their use, even if the FUNCTION statement and the function reference appear within the same block. A function name may be defined by its appearance in a FUNCTION statement or by explicit declaration in a DECLARE statement.

EXAMPLES:

1.

#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;
```

If #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, in a DECLARE statement.

2.

#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.

8.1.4 Scope of the REPLACE Statement

With reference to the description presented in Sec. 7.3.6, the scope of a REPLACE statement is the same as that for a DECLARE statement with the following exception: the <identifier> 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.

8.1.5 Scope of the DEFAULT Statement

With reference to the description presented in Sec. 5.5, the scope of the DEFAULT statement is the same as that for a DECLARE statement.

EXAMPLE:

```
ALPHA: PROGRAM;
  DEFAULT MATRIX(4,7) BIT_LENGTH(24);
     .
     .
     .
  BETA: PROCEDURE;
     DEFAULT BIT_LENGTH(10);
     DECLARE E, F, BIT;
        .
        .
  CLOSE BETA;
CLOSE ALPHA;
```

In procedure BETA, which is nested within ALPHA, the default-type established in ALPHA remains valid so that E is a 4x7 matrix. F is a 16-bit string by virtue of the DEFAULT statement in BETA.

8.2 The COMPOOL

The COMPOOL is a centrally defined and centrally maintained group of statements. The statements are limited to REPLACE, OUTER and DECLARE (the <declare-group>), 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 COMPOOL A                INCLUDE COMPOOL A

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.

8.3 The Symbolic Library

The symbolic library is a centrally defined and centrally maintained pool of symbolic source code. The library is available to all programs and may be added to a program by use of the compiler directive*

INCLUDE<library-entry>

The appearance of this directive causes the symbolic code in the object file to be included in the compilation and inserted at that point. For example:

```
INCLUDE NAVDATA
A: PROGRAM;
INCLUDE AGLOBALS
INCLUDE ALOCALS
      .
      .
      .
   B: TASK;
      X = A;
      Y = B;
        .
        .
        .
   INCLUDE LOGIC
   CLOSE B;
   C:  PROCEDURE;
   IF L>100 GO TO ABLE;
            ELSE
   INCLUDE CHOICE
```

---

* Compiler directives require a D in column 1 of input source code line.

ABLE: _____

_____

_____

_____

CLOSE C;

CLOSE A;

GENERAL RULES:

The symbolic library may contain source code identical to that within a program except that INCLUDE directives are not allowed.

9.0 REAL TIME CONTROL

The real-time control of HAL programs consists of the interrelated scheduling of <program- and <task-blocks>, the reliable sharing of common data, and the recovery from abnormal error conditions.

The concepts and language features introduced in this section have been designed for general applicability to real-time control programming. It is recognized that depending upon specific hardware environments and operating system designs, certain features may not find utility.

9.1 TASK Statement

A task is a subroutine which is intended to be scheduled in real-time through an executive system. The TASK statement identifies the beginning of a block of statements which form a task and defines the entry point.

GENERAL FORMAT:

```
<task-label>: TASK;
 {[<label>:]<statement>|[<label>:]RETURN;}. . .
 [<label>:]CLOSE[<task-label>];
```

GENERAL RULES:

1. Unlike procedures, tasks do not provide for parameter passage and return. Rather, data exchange must be accomplished

through variables with common data scope (i.e., variables defined at the COMPOOL or program levels).

2. Local variables and constants may be declared as in procedures and functions.

3. Execution of a task may be terminated by a RETURN statement, a TERMINATE* statement or by logically reaching the CLOSE statement. If the task is activated by the executive, termination causes control to be returned to the executive. If the task is simply called, as a procedure, RETURN and CLOSE return control to the caller; TERMINATE always returns control to the executive.

9.1.1 Task Calls

The CALL statement may be used to call a task. The logical result is similar to calling a procedure; i.e., control is transferred to the task called and returned when the task is completed. The CALL statement is of the form:

CALL<task-label>;

In calling a task:

1) No arguments may be passed.

2) Control is returned to the caller at the statement following the CALL statement, when a RETURN or CLOSE statement is reached.

3) A task cannot call itself.

---

* See Section 9.2.4

9.2 Scheduling Statements

9.2.1 SCHEDULE Statement

The SCHEDULE statement is used to request initiation of a program or task based on three criteria:

a) at a specific time (<spec-time>)

b) in an incremental time (<inc-time>)

c) on events or combinations of events (<event-expression>)

where time is expressed in seconds or units specified by implementation, and an event is a programmer-defined (see Sec. 9.3.1) or system-defined occurrence. The general format of the SCHEDULE statement is:

```
[<label>:]SCHEDULE{<program-label>} [AT <spec-time>          ]
                  {<task-label>   } [IN <inc-time>           ]
                                    [ON <event-expression>   ]

  [PRIORITY{{<p>|PRIO + <q>})][,INDEPENDENT][,<task-id>];
```

<spec-time> and <inc-time> may be <scalar- or <integer-operands>. <event-expression> has the same form as the <single-bit-expression> (see Sec. 7.3.2.2); i.e., a logical combination (AND, OR, NOT) of event names.

GENERAL RULES:

1. A SCHEDULE statement within one <program-block> may be used to schedule the program itself, any task within the block, or another program. A task within one <program-block> may not be scheduled from another <program-block>.

2. Procedures, functions and labelled statements may not be scheduled.

3. <spec-time> and <inc-time> are rounded to the nearest integral number of time units before use.

4. PRIORITY(<p>) specifies the priority of initiation. If two programs (or a task and a program, etc.) are scheduled for the same time (or on the same event(s)), the one of higher priority will be initiated first. <p> may be a positive <scalar- or <integer-operand> and represents an absolute priority. Relative priorities may be established by using the function PRIO which returns the current program or task priority. Thus, PRIORITY(PRIO + <q>) requests a priority of <q> greater than current priority. <q> may be a positive or negative <scalar-or <integer-operand>.

5. If PRIORITY is not provided, scheduling will take place with current priority.

6. If INDEPENDENT is provided, the scheduled program or task is to be independent of the block in which it is scheduled. This means that an independent program or task can continue in an active state even after the scheduling block has been terminated. However, a task with STATIC variables or one which contains reference to identifiers declared at the program level cannot be scheduled as an independent task.

7. If INDEPENDENT is not provided, dependent scheduling will take place. All dependent programs and tasks are terminated when the block in which they were scheduled is terminated.

8. <task-id> is a name which will contain the unique identification data for the scheduled program or task.

9. If AT, IN, ON are not provided, initiation will take place as soon as possible (consistent with priority).

EXAMPLES:

1. SCHEDULE PROGRAM_20 PRIORITY(10), PROG_20;

   PROGRAM_20 is scheduled as a dependent block (program or task), priority 10, with identification stored in the variable PROG_20. Initiation will begin as soon as possible.

2. SCHEDULE ABLE PRIORITY(PRIO + 1);

   ABLE is scheduled as a depedent block at a priority 1 higher than the current priority.

3. SCHEDULE RADAR ON R_RUPT PRIORITY(HIGH);

   RADAR will be initiated on the occurrence of the event R_RUPT at priority HIGH.

4. SCHEDULE STEERING AT TIG-5 PRIORITY(6),INDEPENDENT;

   STEERING is scheduled, as an independent block, to begin at the time TIG-5 with priority 6.

5. SCHEDULE TRACK IN 5;

   TRACK is scheduled to begin in 5 units of time from the time the SCHEDULE statement is executed.

6. SCHEDULE ABLE ON (A AND B) OR C;

ABLE is scheduled to begin on the occurrence of either both events A and B, or event C.

7. IF X>10 AND TRACKFLAG = ON

THEN SCHEDULE AUTOMANEUVER PRIORITY(5);

ELSE GO TO BEGIN;

The SCHEDULE statement may be included as another executable statement. AUTOMANEUVER will be scheduled if X>10 and the TRACKFLAG is ON.

9.2.2 WAIT Statement

The WAIT statement is used to suspend an active program or task and reactivate it based on three criteria:

a) a specific time

b) an incremental time

c) a particular event or combination of events.

GENERAL FORMAT:

```
[<label>:]WAIT [UNTIL <spec-time>
               <inc-time>
               FOR <event-expression>];
```

where <spec-time>, <inc-time>, <event-expression> are the same as in Sec. 9.2 (SCHEDULE statement).

EXAMPLES:

1. WAIT 5;

The current block (program or task) is suspended for 5 units of time and then reactivated.

2. WAIT UNTIL TIG-5;

   The current block is suspended until the time TIG-5 and then reactivated.

3. WAIT FOR ABLE;

   The current block is suspended until the event ABLE occurs (i.e., ABLE is turned ON) and then reactivated.

4. WAIT FOR NOT(T1 AND T2) OR T3;

   The current block is suspended until the events T1 and T2 are OFF, or the event T3 is ON, and then reactivated.

9.2.3 PRIO_CHANGE Statement

This statement is used to change the priority of a task or program.

GENERAL FORMAT:

```
[<label>:]PRIO_CHANGE({<p>|PRIO + <q>})[<task-id>]
                    [({<p>|PRIO + <q>}),<task-id>]. . .;
```

where <p>, <q> are defined in Sec. 9.2.1.

GENERAL RULES:

1. <p> or <q> are new absolute and relative priorities, respectively, for the corresponding <task-id's>.
2. The current program or task priority may be changed by the statement

```
PRIO_CHANGE({<p>| PRIO + <q>});
```

EXAMPLES:

1. IF AFLAG THEN PRIO_CHANGE (PRIO + 5);

   If AFLAG is on then current priority is increased by 5.

2. PRIO_CHANGE (8), (10) TASK_1, (13) TASK_2, (PRIO + A) TASK_3;

   The current priority is changed to 8, TASK_3's priority is changed to the current priority plus A (i.e., 8 + A). Note that a <task-id> can be omitted only before the first comma, meaning the current task or program.

9.2.4 TERMINATE Statement

This statement is used to terminate a program or task and return control to the executive.

GENERAL FORMAT:

[<label>:]TERMINATE[<task-id>[,<task-id>]. . .];

GENERAL RULES:

1. Execution of this statement terminates all identified tasks and programs and all their dependent tasks and programs.
2. If <task-id> is not provided, the current program or task and all dependent programs and tasks are terminated.

EXAMPLES:

1. TERMINATE PROG_20, T2;

   The blocks (task or program) identified by PROG_20 and T2 are terminated.

2. TERMINATE;

   The current program or task is terminated.

9.3 Events and Signals

Programs and tasks may be scheduled by the occurrence of events or combinations of events. An event is a programmer-named condition and can be stimulated only by the execution of the SIGNAL statement.

9.3.1 Events

<event-variables> must be declared using DECLARE statements. The format is similar to that described for data declarations, thus:

GENERAL FORMAT:

DECLARE<event-variable>EVENT[LATCHED[INITIAL{ON/OFF}]];

GENERAL RULES:

1. <event-variables> may only be declared at the COMPOOL and program levels. Scope rules are the same as for data.
2. If the attribute LATCHED is provided, the <event-variable> will hold its signalled value; i.e., if signalled on, it will remain on.
3. If LATCHED is not provided, the <event-variable> when signalled on, will remain on only for a short interval of time. The time interval is implementation dependent.
4. The declaration of an <event-variable> can be incorporated in the same DECLARE statement with other identifiers; e.g.

```
DECLARE V VECTOR, M MATRIX, B EVENT;
```

5. EVENT, LATCHED, INITIAL may be factors in a DECLARE statement; e.g.

DECLARE EVENT, A, B, C INITIAL(ON);

6. If INITIAL is not provided for <event-variables> with the LATCHED attribute, a default value of OFF is presumed.

EXAMPLE:

DECLARE EVENT, A, B LATCHED;

A and B are declared "unlatched" and "latched" events. Both are set off initially. It should be noted that an unlatched event cannot be set on initially.

9.3.2 SIGNAL Statement

This statement is used to cause the occurrence of an event. The specific effect depends upon whether the <event-variable> has the attribute LATCHED.

GENERAL FORMAT:

[<label>:]SIGNAL<event-variable>[ON/OFF][,<event-variable>[ON/OFF]]. . . ;

GENERAL RULES for LATCHED <event-variables>:

1. <event-variables> may be on or off initially.
2. If an <event-variable> is off:
   a) it may be turned on by SIGNAL<event-variable>ON;
   b) it may be left off by SIGNAL<event-variable>OFF;
   c) if ON or OFF is not provided, SIGNAL<event-variable>; turns the <event-variable> on for a short time interval, and then off. The time interval is implementation dependent.

3. If an <event-variable> is on:

   a) it may be turned off by SIGNAL<event-variable>OFF;

   b) it may be left on by SIGNAL<event-variable>ON;

   c) if ON or OFF is not provided, SIGNAL<event-variable>; turns the <event-variable> off after a short interval.

GENERAL RULES for "unlatched" <event-variables>:

1. <event-variables> are always off intially.
2. SIGNAL<event-variable>[ON]; turns the <event-variable> on for a short interval, and then off. The time interval is implementation dependent.
3. SIGNAL<event-variable>OFF; causes no action.

EXAMPLE:

```
SYNCHRO: PROGRAM;
        DECLARE EVENT LATCHED, A, B;
        SCHEDULE ABLE INDEPENDENT;
        SCHEDULE BAKER INDEPENDENT;
        SCHEDULE CHARLIE ON A AND B;
        Z̄ = W̄ + V̄;
        M̄ = Z̄ N̄*;
          .
          .
          .
        TERMINATE;
    ABLE: TASK;     /*INDEPENDENT TASK*/
        .
        .
        .
    SIGNAL A ON;
    CLOSE ABLE;
```

```
        BAKER: TASK;   /*INDEPENDENT TASK*/
                .
                .
                .
          SIGNAL B ON;
        CLOSE BAKER;
        CHARLIE: TASK;
                .
                .
                .
        CLOSE CHARLIE;
CLOSE SYNCHRO;
```

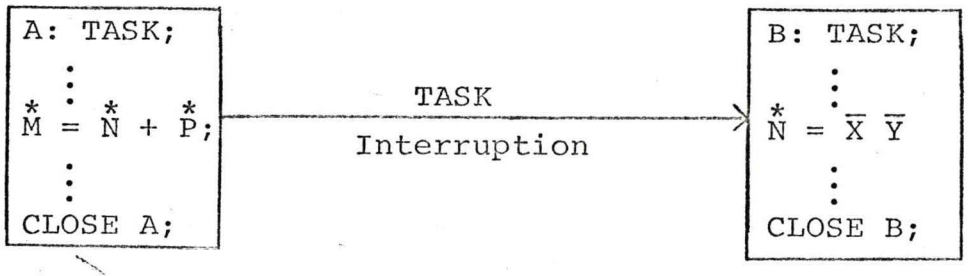
9.4 Dynamic Control of Shared Data

HAL provides features to control the sharing of variables in order to prevent conflicts in their utilization. These features include the attribute LOCK_TYPE to designate shared variables and an update block of statements in which shared variables may be changed in a controlled environment. Although the approach taken is basically implemented in software, it does depend on the ability to perform an "uninterruptable" instruction similar to the Test and Set instruction available on IBM 360 computers.

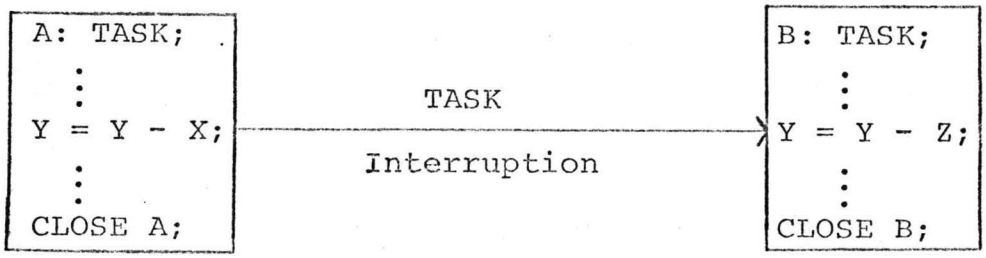
9.4.1 Conflicts in Sharing Data

In order to illustrate the problems that can arise in sharing data consider the following two examples:

Example 1: Read/Write Conflicts



Example 2: Serial Updating Conflicts



In both examples TASK B interrupts TASK A during the execution of a statement. The interruption may be caused by a hardware or software (SIGNAL) interrupt or by a "job swap" based on priority. In Example 1, presume that the interruption occurred while the matrix $\overset{*}{N}$ was being read. When TASK A resumes, the computation of $\overset{*}{M}$ will continue using some "old" $\overset{*}{N}$ data and the "new" $\overset{*}{N}$ data assigned in TASK B. In order to prevent this conflict, initiation of TASK B would have to be stalled until the reading of $\overset{*}{N}$ in TASK A is completed.

In Example 2, presume that the interruption occurs first after the current value of Y is loaded into the accumulator. When TASK A resumes, the "old" value of Y (i.e., not reflecting the update of Y in TASK B) is restored into the accumulator, X is subtracted and the result assigned to Y. In order to prevent this conflict, the initiation of TASK B would have to be stalled until the value of Y is updated in TASK A (i.e., each variable declared with the LOCK_TYPE attribute, see Secs. 4.3.4, 5.1.1.3).

The approach taken in solving the problems represented above, using HAL, is to confine the read and write accesses of shared variables to identified update blocks and for the compiler to assign a locking control variable to each shared variable (i.e., to each variable declared with the LOCK_TYPE attribute). The value of the "lock" is examined at run-time and only consistent (i.e., safe) accesses are permitted.

9.4.2 The Update Block

The <update-block> of statements provides a controlled environment for the reading and writing of shared data variables. All LOCK_TYPE(1) variables, and LOCK_TYPE(2) variables to be assigned new values (i.e., updated) must appear within <update-blocks>. LOCK_TYPE(2) variables which are to be read only need not be confined to these blocks. The <update-block> may contain the following elements:

<update-block> = <update-statement>[<declare-group>]
{<all-statements>|<sub-blocks>}. . .<close-statement>

subject to the restrictions below.

GENERAL FORMAT:

[<update-label>:]UPDATE;
{[<label>:]<statement>}. . .
[<label>:]CLOSE[<update-label>];

GENERAL RULES:

1. <statements> within an <update-block> (and enclosed <sub-blocks>) may not include I/O statements (see Sec. 10), or additional UPDATE statements.
2. Name scope rules are the same as described in Sec. 8.1.1 except <statements> may not contain <procedure-labels> or <function-labels> defined outside the block. (HAL built-in function names are permitted.)

3. Execution of the UPDATE statement attempts "to lock" all shared variables within the block. A variable to be assigned will be write-locked, variables to read only will be read-locked. Once locks are established they are not opened until execution of the CLOSE statement at the end of the block.

4. If all desired locks cannot be established at the UPDATE statement because one or all of the shared variables are not available (i.e., they are already locked elsewhere), the current program or task will be stalled (placed in "wait" by the executive) until all variables become available.

5. After all locks are established, copies are made of the shared variables to be assigned (if any), and the <statements> within the <update-block> are executed using this copy-data.

6. Execution of the CLOSE statement first opens all read-locks and then attempts to transfer the updated copy-data into the actual shared variables (to be assigned). If read-locks are in effect on these variables (i.e., they are still locked within other <update-blocks>), the current program or task will be stalled until these locks are opened. After the copy-data has been transferred all write-locks are opened and execution continues at the statement following CLOSE.

7. In conjunction with (4) above, a stall will occur at the UPDATE statement if any of the shared variables to be assigned in the block already are write-locked. In other words, a write-lock cannot be established on a variable that is already write-locked. A stall will also occur if any shared variables to be read are currently being written in other <update-blocks> , i.e., a read-lock cannot be established while the variable is being assigned a new value.

8. Transferring control outside the update block by a GO TO statement or in response to an error condition (see Sec. 9.5) is considered an "error exit". As a result, all read- and write-locks are opened and no copy-cycle is performed.

9. LOCK_TYPE(2) variables which are to be read only need not be confined to update blocks. This attribute should only be applied to those data types which can be accessed in a single uninterruptable instruction.

9.4.2.1 Summary on Entering an Update Block (LOCK_TYPE(1) Variables.)

Present State / Variables	Free	Read-Locked	Write-Locked	Writing
To be assigned in block	Write-Lock	Write-Lock	Stall	Stall
To be read in block	Read-Lock	✓	Read-Lock	Stall

Table 9-1

Table 9-1 indicates that on entering an <update-block>, if variables to be assigned are free or read-locked, write-locks will be established; otherwise execution will stall until variables are available. If variables to be read are free, read-locked or write-locked, read-locks will be established; otherwise execution will stall until variables are available. (✓ means read-lock already established, new lock is unnecessary.)

9.4.2.2 Summary on Leaving an Update Block (LOCK_TYPE(1) Variables.)

Present State / Actual Variables	Free	Read-Locked	Write-Locked	Writing
To be written	N.A.	Stall	Copy	N.A.

Table 9-2

Table 9-2 indicates that on leaving an <update-block>, if variables to be written are write-locked the copy-cycle will proceed; otherwise execution will stall until variables are available. (N.A. means not applicable. Once in an <update-block>, variables cannot be free nor in the process of being written within another <update-block>.)

9.4.2.3 Examples. Consider the two examples at the beginning of Sec. 9.4.1 and suppose that the statements in question were enclosed within <update-blocks>, e.g.,

```
A: TASK;
     .
     .
     .
   UPDATE;
     .
     .
     .
    M* = N* + P*;
     .
     .
     .
   CLOSE;
     .
     .
     .
CLOSE A;
```

Example 1

In TASK A a read-lock is established for $\overset{*}{N}$. After the interruption, a write-lock is established for $\overset{*}{N}$ and TASK B proceeds toward completion using copy-data for $\overset{*}{N}$. At the end of the <update-block> in TASK B the process stalls because of the read-lock imposed in TASK A. As a result, TASK A is allowed to continue with consistent "old" $\overset{*}{N}$ data. After completion of TASK A, the copy-cycle in TASK B is effected and $\overset{*}{N}$ is updated. All conflicts are eliminated.

Example 2

In TASK A read- and write-locks as well as copy-data are established for Y. As before, the value of Y (now copy-data) is placed in the accumulator. After the interruption, execution

of the UPDATE statement in TASK B attempts to establish read- and write-locks for Y. The process stalls because a write-lock already exists for Y. Therefore, control is transferred back to TASK A and execution allowed to continue. Y is updated in TASK A by X and a copy-cycle completed. TASK B now begins again. This time Y is free and read- and write-locks are established. TASK B runs through in a straightforward manner. Y is updated properly by both X and Z with no conflicts.

9.4.3 Exclusive Subroutines

The attribute EXCLUSIVE may be applied to programs, procedures, functions and tasks which are intended to be executed serially. The object is to avoid reentrant use of a subroutine either because the variables are not protected by locks (i.e., have not been declared with LOCK_TYPE attributes) or because dynamic design dictates serial use.

GENERAL FORMAT:

{<program-|<procedure-|<function-|<task-statement>}EXCLUSIVE;

GENERAL RULES:

1. The compiler will insert code at the beginning of the subroutine to cause the current program or task to stall if the subroutine is in use. At the end of the subroutine, the stalled programs or tasks of highest priority will be reactivated.

EXAMPLES:

1. ABLE: PROCEDURE(A,B) ASSIGN(C) EXCLUSIVE;

2. BAKER: TASK EXCLUSIVE;

The above are valid statements using the EXCLUSIVE attribute.

9.4.4 Access Rights

The general use of COMPOOL data within programs may be restricted by attaching access rights to the DECLARE statements within the COMPOOL. Programs are identified by number and permitted to access only those variables which have been declared with corresponding identification numbers. An illegal reference to a COMPOOL variable will prevent successful compilation of the problem.

GENERAL FORMATS:

<program statement>ID_CODE<p>;

<declare-statement>ACCESS(<p>[<p>]...);

where <p> is an unsigned integer literal.

GENERAL RULES:

1. If ACCESS is provided, declared variables will only be recognized in programs whose identification numbers are listed.
2. If ACCESS is not provided, declared variables will be recognized in all programs.
3. Compilation will abort if proper access rights have not been established for a reference to a COMPOOL variable.

9.5 Error Recovery

During execution of HAL programs an error condition may be detected by the system. Examples of errors might be:

overflow/underflow
divide by zero
negative square root argument
sine argument greater than 1
subscript out of range

Depending upon implementation such errors may be hardware or software detected. In any case, execution cannot continue and the system must offer generally applicable alternatives (e.g. aborting the current task, etc.).

In order to provide the programmer with some control after the occurrence of an error, perhaps to reset flags or previously initiated I/O commands, HAL permits programmer-defined error conditions and alternatives.

9.5.1 ON Statement

The ON statement may be used to direct the transfer of control on the occurrence of one or more specific error conditions.

GENERAL FORMAT:

[<label>:] ON ERROR$_{\text{<p>[TO<q>]}}$ [{GO TO <label>|SYSTEM}];

where <p> and <q> are integer literals.

GENERAL RULES:

1. For any implementation, unique <literals> are assigned to every system error condition; e.g.

$ERROR_5$	floating point overflow
$ERROR_6$	floating point underflow

and to programmer-defined error conditions.

2. A group of error conditions may be specified using the subscript range expression (e.g., $ERROR_{1\ TO\ 10}$).
3. Upon execution of the ON statement the alternatives GO TO <label> or SYSTEM are made available for the scope of the statement. The scope of an ON statement follows the same rules as the name scope of a variable (i.e. from the "outermost" block toward the inner, see Sec. 8.1.1).
4. If the specified error condition occurs within the defined scope the desired alternative is activated (i.e. control is either transferred to the statement <label> indicated or to the system).
5. If GO TO <label> or SYSTEM is not provided the default is SYSTEM.

9.5.2 ER_RUPT Statement

The ER_RUPT statement is used to announce the occurrence of programmer-defined error conditions.

GENERAL FORMAT:

ER_RUPT $ERROR_{<p>}$ [TO <q>];

where <p> and <q> are integer literals.

EXAMPLE:

```
D = B^2 - 4A C;
IF D<0 THEN ER_RUPT ERROR50;
X = (-B - SQRT(D),/2A;
```

9.5.3 EXAMPLES

1. ON ERROR$_{1\ TO\ 5}$ GO TO ABLE;

   If any of error conditions 1 through 5 occurs within the scope of this statement, control is transferred to ABLE.

2. ON ERROR$_{1\ TO\ 5}$ SYSTEM;

   If any of error conditions 1 through 5 occurs within the scope of this statement, system action is taken.

3.
```
A: PROCEDURE;
   ON ERROR1 GO TO BETA;
        :
   CALL B;
      :
   B: PROCEDURE;
      ON ERROR1 GO TO ALPHA
           :
      CLOSE B;
           :
ALPHA: FLAG1 = OFF;
       TERMINATE;
BETA: FLAG2 = OFF;
      TERMINATE;
CLOSE B;
```

If $ERROR_1$ occurs during procedure B control is transferred to ALPHA, otherwise it is transferred to BETA.

4.
```
A: TASK;
   DECLARE X - - -;
   ON ERROR1 TO 10 GO TO RECOVERY1;
   RETRY: R̄ = M̄* X;
   CALL B (R̄, V̄, TD...);
   ON ERROR1 TO 10 GO TO RECOVERY2;
   CALL JETS;
   ON ERROR1 TO 10 SYSTEM;
        ⋮
   B: PROCEDURE (----);
        ⋮
   CLOSE B;
   RECOVERY1: X̄ = X̄ + D̄ECTAX;
      GO TO RETRY;
   RECOVERY2: CALL JETS_OFF;
      GO TO ABORT;
CLOSE A;
```

RECOVERY1 and RECOVERY2 are established as different recovery points for TASK A. Control is transferred to either one depending on where the error conditions occur. The system action is established after control is returned from the procedure JETS.

This example illustrates that the programmer can develop arbitrary restarting points within a HAL program.

10.0 INPUT-OUTPUT

The HAL input-output statements provide for the filing, retrieval, reading and writing of data to and from external storage media. Filing is record-oriented in that a file statement causes a single record to be transmitted to or from a storage device; transmission is direct without any conversions. Reading and writing are stream-oriented in that data is considered to be a continuous stream of characters; conversions may occur during transmission.

The HAL I/O syntax consists of four statements and a small set of control functions.

10.1 FILE Statement

The FILE statement has the appearance of an assignment statement and may be used for both filing and retrieving data depending upon which side of the = sign FILE appears.

GENERAL FORMAT:

1. for filing data

```
[<label>:]FILE(<device>,<record-i.d.>)={<data-expression>|
                                        <structure>};
```

2. for retrieving data

```
[<label>:]<variable-name>=FILE(<device>,<record-i.d.>);
```

GENERAL RULES:

1. <device> is an integer literal identifying the external device. The maximum number of digits is implementation dependent.
2. <record-i.d.> is the record identification number and may be an integer or scalar expression. The result of <record-i.d.> is rounded to the nearest integer before use.
3. In retrieving data, the size of the record, i.e., the number of words (or perhaps bytes, etc.), must match the size (dimension) attributes of the <variable-name> on the left hand side of =. Because the filed information does not carry data-type or attributes, conversion errors can occur even if the sizes match properly.

EXAMPLES:

```
FILE(TAPE,I) = [A];
{B} = FILE(DISC,A_I);
```

are valid FILE statements, where TAPE and DISC represent integer literals.

10.2 READ Statements

Two READ statements are defined in HAL: READ and READ_ALL. READ is used to process data presented in standard formats; READ_ALL admits all characters and provides the flexibility to accept data in non-standard (arbitrary) formats.

10.2.1 READ Statement

The READ statement causes data, in standard formats from an external source, to be assigned to a list of variables.

GENERAL FORMAT:

READ(<device>)[<read-control>|<variable-name>]
[,[<read-control>|<variable-name>]]...;

where

<read-control> = {SKIP(<p>)|TAB(<p>)|COLUMN(<p>)}

and

<p> is an integer or scalar expression, rounded to the nearest integer before use.

GENERAL RULES:

1. The READ statement implies the input transmission of a stream of data fields, each field being separated by a comma or a semi-colon. (A blank or blanks may be used optionally instead of a comma, between data fields.)
2. The <variable-names> in the list may be of single elements, arrays of elements and/or structures. The number of fields transmitted, for each <variable-name>, corresponds to the size, or dimension, attribute of the <variable-name>. For example, READ $\overset{*}{M}$; (where $\overset{*}{M}$ is a 4x4 matrix) will cause 16 fields of data to be transmitted. It is presumed that vectors, matrices and arrays will be filled according to the rules and conversions for processing <lists>, as described in Sec. 6.2.2.1. The arrangement of structure data is described

in (5) below. The absence of a <variable name> in the list of names; i.e., nothing between commas, or leading or trailing commas causes the "read-mechanism" to skip over one data field (for example: READ(CARDS), A, , B, ;).

3. The external device is visualized as being two-dimensional in that data occupies horizontal lines, each line being made up of column positions. A data field is defined as a segment of contiguous columns, delimited by commas (blanks) or semi-colons. (The first column of line n+1 follows the last column of line n.) The <read-control> functions locate the "read-mechanism" on this "grid". If a <read-control> function is not provided immediately following READ(<device>), blanks being ignored, a default SKIP(1), COLUMN(1) is presumed; i.e. READ (<device>) causes the next line to be selected and reading to begin at column 1.

4. The appearance of SKIP(<p>) and/or COLUMN(<p>) within the list of <variable-names> sets up the "read-mechanism" to skip <p> lines and/or begin reading at column <p> when the next data field is encountered. The TAB(<p>) function causes a relative column location; i.e. TAB(8) would cause the "read-mechanism" to "move" eight columns. The presence of a semi-colon, separating fields of data causes termination of the current READ statement. Unassigned <variable-names> in the statement are left with their previous values. If additional data fields follow the semi-colon, on the same

line, they may be processed by the next read statement if a SKIP(0) is provided; e.g. the data card,

```
5,6,7;8,9;10;
```

could be processed by the following READ statements:

```
READ(CARDS) A,B,C,D,E;
READ(CARDS) SKIP(0), F,G,H,I,J;
READ(CARDS) SKIP(0), K,L,M,N,P;
```

The first semi-colon on the data card causes termination of the first READ statement after A,B and C are assigned. The second READ statement begins "reading" immediately after C, on the same line, because of the SKIP(0), and assigns F and G only. The last READ statement assigns K. Note that after the three READ statements D,E,H,I,J,L,M,N,P will retain their previous values.

5. If the <variable-name> is a structure, the elements of the structure are transmitted in the order specified in the structure declaration. Multiple-copy structures are transmitted one copy at a time. For the structure

```
DECLARE 1 A(5), 2B ARRAY(4,5), 2C VECTOR(4);
```

the statement

```
READ{A};
```

would result in an input transmission order of

$A.B_{1;1,1}$ $A.B_{1;1,2}\cdots$ $A.B_{1;4,5}$ $A.C_{1;1}\cdots$ $A.C_{1;4}$

$A.B_{2;1,1}$ ..... $A.B_{2;4,5}$ $A.C_{2;1}\cdots$ $A.C_{2;4}$

$\vdots$

$A.B_{5;1,1}$ $A.B_{5;1,2}\cdots$ $A.B_{5;4,5}$ $A.C_{5;1}\cdots$ $A.C_{5;4}$

EXAMPLES:

1. READ(CARDS) A,B,C,D,[E],{F};

   This statement causes transmission of enough data fields to assign the variables listed. Note that CARDS represents an integer literal.

2. READ(CARDS) COLUMN(20),A,B,
   SKIP(1), COLUMN(20),C,D,
   SKIP(1), COLUMN(20),E,F,
   :
   etc.

   This statement causes two fields of data to be read on each successive card. The data will be read starting in column 20.

3. READ(CARDS) A, TAB(40), C;

   This statement is designed to skip over some data fields (40 columns) known to be on the input cards.

10.2.2 Standard Input Data Formats

The list of variables in a READ statement may be of any data type. Each type requires the input data to be presented in a standard format.

10.2.2.1 Standard Arithmetic Data Formats. Integer, scalar, vector, matrix and bit string data may be presented in the following format:

{[+]|-}⌀<digits>[{⌀E|⌀B|⌀H}⌀{[+]|-}<integer>]...

where ⌀ represents optional blanks. Note that this is almost the same form as an arithmetic literal except for the optional blanks. See Sec. 2.3.3.1 for definition of terms.

GENERAL RULES:

1. For integers and bit strings the data form must represent integral values. Bit string data is first converted to a full word bit string and then assigned to the corresponding bit variable according to the rules stated in Sec. 7.1.2.1.
2. The data forms for scalars, vectors and matrices are identical.

EXAMPLES:

1. 369.0, 8, -8.36E+2 B-1 are valid forms of integer and bit string input data.
2. +0.123E6 B-3 H4, 1E-75, 3, 456.789 are valid forms of scalar, vector and matrix input data.

10.2.2.2 Standard Character Data Format. Character data may be presented as any character or string of characters (in the HAL set) enclosed in apostrophes. If it is desired to place an apostrophe in the string, it must be represented by an adjacent pair of apostrophes.

EXAMPLES:

1. 'AB''''C', '57.3/C', 'NUMBER_ONE', 'ON,OFF,OFF,ON' are valid forms of character data.
2. The following input data field and statements will assign a bit string variable using an octal input data form.

```
DECLARE B BIT(15);
DECLARE C CHARACTER(10) VARYING;
READ C;
B = BIT@OCT(C);
```

column (1)

input data: '37776'

10.2.2.3 Arrays and Structures. Arrays and structures consist of the above data types,and the forms presented are acceptable as required.

10.2.3 READ_ALL Statement

The READ_ALL statement allows data in non-standard form to be assigned to HAL character-string variables. This is accomplished by not defining fields of data but accepting all characters encountered in the input stream, including blanks, commas, semi-colons and apostrophes.

GENERAL FORMAT:

Same as for the READ statement except READ_ALL replaces READ and the <variable-names> may pertain to character strings only.

GENERAL RULES:

1. The READ_ALL statement implies the input transmission of a continuous stream of characters.

2. The <variable-names> in the list may be of character strings, arrays of character strings and/or structures containing only character strings.

EXAMPLE:

Suppose the following data card has been generated at a computer facility. It is desired to process this data in a HAL program.

```
column (1)                (30)              (60)
        DATE: 25/12/70      8,632 06          101101
```

where the scalar starting in column(20) is equivalent to 8.632E06 and the data starting in column(40) is a set of six boolean variables.

```
        DECLARE B BIT(6);
        DECLARE CHARACTER(20), C,D,E;
        READ_ALL(CARD)C', COLUMN(30), D', COLUMN(60), E';
C PUT SCALAR IN PROPER FORM
        D'2 = '.'; /*CHANGE COMMA TO PERIOD*/
        I = 3;
    LOOP: DO WHILE D'I ¬= ' '; /*LOOK FOR BLANK*/
        I = I + 1;
        END LOOP;
        D'I = 'E';  /*CHANGE BLANK TO E* /
        A = SCALAR(D'); /*ASSIGN SCALAR TO A* /
```

```
C    PUT BOOLEAN VALUES IN PROPER FORM
          B = BIT@BIN(E);
    /*FINISH*/
```

10.3 WRITE Statement

The WRITE statement causes the transmission of data to an external device. Data items transmitted are the character string representations, in standard formats, of values of HAL expressions.

GENERAL FORMAT:

```
WRITE(<device>)[<write-control>|{<variable-name>|<data-expression>}]
     [,[<write-control>|{<variable-name>|<data-expression>}]]...;
```

where

```
<data-expression>={<arithmetic-|<string-|<array}-<expression>
```

and

```
<write-control>={SKIP(<p>)|TAB(<p>)|COLUMN(<p>)|
                                PAGE(<p>)|LINE(<p>)}
```

<p> is an integer or scalar expression, rounded to the nearest integer before use.

GENERAL RULES:

1. The WRITE statement implies the output transmission of a continuous stream of characters.
2. The <variable-names> in the list may be the same as defined

for the READ statement. The <data-expressions> may be any valid arithmetic, string and/or array expressions.

3. The external device is visualized as being two-dimensional in that output data will occupy horizontal lines, each line being made up of column positions. A page is defined as a default number of lines. The <write-control> functions locate the "write-mechanism" on this "grid". If a <write-control> function is not provided immediately following WRITE(<device>), blanks being ignored, a default SKIP(1), COLUMN(1) is presumed; i.e. WRITE(<device>) causes the next line to be selected and writing to begin at column 1.

4. The appearance of <write-control> functions within the list of <variable-names> and/or <data-expressions> sets up the "write-mechanism" for execution when the next name or expression is encountered. SKIP, COLUMN and TAB perform the same functions as in the READ statement.

   LINE(<p>) redefines the value of the current line. If <p> is greater than the current line, blank lines are inserted so that the next line will be the p^{th} line of the current page. If <p> is less than the current line, the next line will be the p^{th} line on the next page.

   PAGE(<p>) causes <p> pages to be skipped upon execution.

5. If COLUMN and/or TAB functions are not provided the presence of a comma will cause a tab of a default number of columns. For example,

```
WRITE A, TAB(10), B, COLUMN(50), C;
```

causes A to begin in column 1, B to begin 10 columns after A, and C to begin in column 50.

```
WRITE A, B, C;
```

causes A to begin in column 1, B to begin a default number of columns after A, and C to begin a default number of columns after C.

6. If the <variable-name> is a vector, matrix, or array, the effect is to unravel these types by rows (Sec. 6.2.2), separating each element by the tab default.

   If the <variable-name> is a structure the effect is to unravel the structure into the order in which it was declared, copy-by-copy, (see READ statement), separating each element by the tab default.

EXAMPLES:

1. WRITE(LISTING) $A, \dot{B}, \bar{C}, \overset{*}{D}$, [E], {F};

   This statement causes transmission of all the named data to the output device. The data is converted to a continuous stream of characters with the elements separated by the tab default. Note that LISTING represents an integer literal.

2. DO FOR I = 1 TO 3;

   WRITE(LISTING) COLUMN(20), $\bar{M}_{I,*}$;

   END;

   These statements will cause the matrix $\overset{*}{M}$ to be printed in rectangular form, each row starting in column (20).

10.3.1 Standard Output Data Formats

The list of variables and expressions in a WRITE statement may be of any data type. Each type produces a standard output character format.

10.3.1.1 Scalars, Vectors, and Matrices. The standard output format for scalar, and components of vectors and matrices is:

sx.<digits>ØEsyy

where s is a blank or a minus sign,

x and y are single digits, 0 to 9,

<digits> is a string of digits, 0 to 9,

Ø is one blank.

sx.<digits> represents the mantissa, syy represents the exponent power of 10. The number of digits in <digits> is fixed and set by machine implementation. The total field of characters in this standard form is 8 plus the number of <digits>.

EXAMPLES:

8.0603478E 06, -7.5436210E-11, 0.0000000E 00, are standard scalar output data.

10.3.1.2 Integers and Bit Strings. The standard output format for integers and bit strings is:

<blanks>s<digits>

where <blanks> is a string of blanks

s is a blank or a minus sign

<digits> is a string of digits, 0 to 9.

The total field of characters in this standard form is fixed in size to be the same as that for scalars; leading zeros are suppressed and appear as blanks, except for a single zero. For example, suppose the character field has been fixed by implementation at 15, then integers might appear as:

```
(1)                    (15)
                         5
                     -4673
                         0
                   2684736
```

Note that when bit strings appear in the WRITE statement they are converted to integers according to the rules stated in Sec. 6.2.1.3.

10.3.1.3 Characters. The standard output format for characters is simply a variable field size equal to the string length of the character variable or expression in the WRITE statement.

EXAMPLES:

1. `WRITE(LISTING)COLUMN(20),'DIST.='||A||'MILES';`

   This statement might result in the following printed line:

```
(20)
DIST.=3.0654767E 06 MILES
```

2. Suppose it is desired to print the same data as above in the non-standard format sxxx.xxx, where s is a blank or

minus, and the x's represent digits. Then,

```
WRITE(LISTING)COLUMN(20)'DIST.='||
        PICTURE('sxxx.xxx',A)||'MILES';
```

The function PICTURE could be a programmer-defined function which accepts the character literal 'sxxx.xxx' and a scalar, A, and returns a character variable representing the scalar quantity in the desired form.

3. Print an array of bit strings in octal format.

```
WRITE(LISTING) CHAR@OCT([B]);
```

Note that the character strings representing the octal values will be separated, on each line, by the tab default. The result might be

```
03664     04662     37774     03725
06437     77172     46162     12346
  :
  :
etc.
```

10.4 Input/Output Manipulations

In addition to the <read- and <write-control> functions SKIP, TAB, COLUMN, PAGE and LINE, several others are defined for programmer convenience.

10.4.1 I/O Functions

```
PAGE_OF(<device>)
LINE_OF(<device>)
COLUMN_OF(<device>)
```

are functions which result in the current page, line and column numbers.

10.4.2 Character String Functions

LEFT(<character-expression>)

RIGHT(<character-expression>,<p>)

are functions for the left and right justification of character strings.

LEFT removes all leading blanks of the <character-expression>.

RIGHT creates a string of length <p> and truncates on the left or pads with blanks on the left depending on whether the <character-expression> length is greater or less than <p>.

APPENDIX A
Built-In Functions on 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 for the arguments are indicated using the following abbreviations:

I: integer

S: scalar

V: vector

M: matrix

B: bit

C: character

A. Conversion Functions (see Sec. 6.2.2).

1. INTEGER

   Arguments: B,I,S,C. Converts an argument to an integer, or a list of arguments to an array of integers.

2. SCALAR

   Arguments: B,I,S,C. Converts an argument to an integer, (or fixed-) point scalar, or a list of arguments to an array of scalars.

3. BIT

   Arguments: B,I,S,V,M,C. Converts an argument to a bit string or a list of arguments to an array of bit strings (V,M are interpreted as lists of scalars).

4. CHARACTER:

   Arguments: B,I,S,V,M,C. Converts an argument to a character string or a list of arguments to an array of character strings (V,M are interpreted as lists of scalars).

5. VECTOR

   Arguments: B,I,S,V,M,C. Converts a list of arguments to a vector, or an array of vectors.

6. MATRIX

   Arguments: B,I,S,V,M,C. Converts a list of arguments to a matrix or an array of matrices.

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. LEFT (character-string)

   Result: LEFT removes all the leading blanks of a character string operand and returns the resultant character string.

4. RIGHT (character-string, p)

   Result: RIGHT 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

These functions return the same data type as the argument (bit arguments are first converted to integers; the function returns an integer).

1. ABS

   Finds the absolute value of the argument.

2. CEIL, 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. SGN, SIGNUM

   Returns 0, +1, -1 as argument is positive or zero, and

negative, respectively.

6. SIGN

Returns +1, -1 as argument is positive or zero, and negative, respectively.

7. TRUNC, TRUNCATE

Returns 0 if argument is less than +1 but greater than -1; otherwise equivalent of SIGN (argument) times the largest positive integral value that does not exceed ABS (argument).

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. ACOS, 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 trignonometrictangent; 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. REMAINDER, REM

    Result is the remainder from the division of two arguments.

12. SIN

    Trigonometric sine; arg in radians; |arg|<K1

13. SINH

    Hyperbolic sine; |arg|<K3

14. TAN

    Trigonometric tangent; arg in radians; arg not odd multiple of $\pi/2$; |arg|<K2

15. TANH

    Hyperbolic tangent; arg any value.

16. SQRT

    Square root; arg positive.

Note: K1, K2 and K3 are upper limits which depend upon target machine characteristics.

E. Matrix-Vector Functions

   Arguments may be vectors or matrices (as applicable).

1. ABVAL

   Absolute value of magnitude of vector; argument may be a vector of any length.

2. ADJOINT, ADJ

   Adjoint; argument is invertible square matrix of any dimension; result is equal to DETERMINANT (argument) INVERSE (argument)

3. DETERMINANT, DET

   Determinant; argument is a square matrix

4. INVERSE

   Inverse; argument is square matrix; result is inverse if argument is invertible.

5. TRACE, TR

   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>(<linear-array>)

where the function will operate on the <linear-array> representing the "inner-most" free index of the array argument.

The <linear-array> may be of (B,I,S,V,M) data types. The functions return a scalar result.

1. SUM

   Result is sum of the array elements.

2. PRODUCT, PROD

   Result is product of the array elements.

3. MAXIMUM, MAX

   Result is the maximum value of the array elements.

4. MINIMUM, MIN

   Result is the minimum value of the list elements.

5. POLY

   POLY forms a polynomial from a linear-array and from an independent variable. The elements of the array form the polynomial coefficients. The array may be of (B,I,S) data types. The general format is

   POLY(<independent-variable>,<linear-array>)

   POLY defines the following polynomial:

$$a_1 + a_2 v + a_3 v^2 + \ldots a_n v^{n-1}$$

   where a_n are the elements of the <linear-array>, and v is the <independent variable>.

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. TIME

Returns current time.

3. DATE

Returns current date

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 BIT. See Sec. 7.1.2.3.

APPENDIX B
Standard Defaults

B.1 DEFAULTS WITH DATA DECLARATIONS

B.1.1 Within DECLARE Statements

B.1.1.1 Specifications (See Sec. 5.1.1). If no <specifications> are provided; i.e. no <array-spec>, <type-spec> and <attribute list>, the following defaults apply to the declared name(s):

1. At the COMPOOL level,

   SCALAR PRECISION(6)

2. At the PROGRAM level,

   SCALAR PRECISION(6); initial value is unspecified.

3. At other levels,

   SCALAR PRECISION(6) AUTOMATIC; initial value is unspecified.

4. For a function (Secs. 5.1.1.5, 7.4.2.1), if <type-spec> is not provided,

   SCALAR PRECISION(6)

Note: For fixed point machine PRECISION default is single precision, with zero integer bits.

B.1.1.2 Precision, Dimensions and Length (Sec. 5.1.1.2).

1. If scalar, vector, or matrix PRECISION is not provided, the precision default is the same as in B.1.1.1 above.
2. If vector <length> is not provided, a length = 3 is presumed.
3. If matrix <rows> and <columns> are not provided, 3 rows and 3 columns are presumed.

4. If bit <length> is not provided, a length = 1 is presumed.

5. If fixed character <length> is not provided, a length = 8 is presumed.

B.1.1.3 Attributes (See Sec. 5.1.1.3)

B.1.1.3.1 Initialization Attributes. If INITIAL or CONSTANT is not provided, the identifier is presumed to be a variable with unspecified initial value.

B.1.1.3.2 Storage Class Attributes. If STATIC or AUTOMATIC is not provided, the STATIC storage class is used.

B.1.1.3.3 Dynamic Sharing Control Attributes. If LOCK_TYPE(<n>) is not specified, for a variable, no controlled sharing is provided.

B.1.1.3.4 Storage Optimization Attributes. If DENSE or ALIGNED is not provided, the ALIGNED attribute is presumed.

B.1.1.3.5 Structure Qualification. If QUALIFIED or NOT_QUALIFIED is not provided in a structure declaration NOT_QUALIFIED is presumed.

B.1.2 Implicit Declarations (See Sec. 5.3)

For the implicit declaration of SCALAR, VECTOR, MATRIX, BIT and CHARACTER the default characteristics of length, precision, initialization, sharing class, and storage optimization are the

same as described in B.1 above for the explicit declaration of these data types.

B.2 WITHIN EXPLICIT CONVERSION FUNCTIONS (See Sec. 6.2.2)

B.2.1 Data-type

B.2.1.1 BIT (<single-operand>). If BIT is not subscripted, the result is a full word bit string. Integers and scalars are converted to full word bit strings; character operands are converted to the bit length representing the total character string.

B.2.1.2 CHARACTER (<single-operand>). If CHARACTER is not subscripted, an integer or scalar operand is converted to a character representation; a bit string is first converted to an integer, and then to a character representation.

B.2.1.3 VECTOR (<list>). If VECTOR is not subscripted, the dimension and the number of elements in the list equals one or three, the dimension is presumed equal to 3. If the list consists of a single-operand, each vector component is assigned the value of this operand.

B.2.1.4 MATRIX (<list>). If MATRIX is not subscripted, and the number of elements in the list equals one or nine, the dimensions are presumed to be 3x3. If the <list> consists of a <single-operand>, each matrix element is assigned the value of this operand.

INTERMETRICS INCORPORATED · 380 GREEN STREET · CAMBRIDGE, MASSACHUSETTS 02139 · (617) 868-1840

B.2.2 Array-Type

B.2.2.1 INTEGER(<list>) and SCALAR (<list>). If these functions are not subscripted with <array-shape>, a one dimensional array of n elements is presumed, where n is the number of elements in the list.

B.2.2.2 BIT (<list>) and CHARACTER (<list>). If these functions are not subscripted with <array-shape>, one dimensional arrays of length n are presumed, where n is the number of elements in the list.

If these functions are not subscripted with <index-expression> then a full word bit string is presumed for BIT and a character string of length 8 for CHARACTER.

B.2.2.3 VECTOR(<list>). If VECTOR is not subscripted with <dimension> and:

1. the list consists of a single operand, a dimension of 3 is presumed. Each vector component is assigned the value of this operand.
2. the list consists of more than a single element, the dimension of each vector array-element will be equal to the number of elements in the list.

Note that the array-type function, VECTOR, must always be subscripted by <array-shape> to distinguish it from the data-type conversion function, VECTOR.

B.2.2.4 MATRIX(<list>). If MATRIX is not subscripted with <dimension> and:

1. the list consists of a single operand, dimensions of 3 x 3 are presumed. Each matrix element is assigned the value of this operand.
2. the list consists of nine elements, dimensions of 3 x 3 are presumed. Each matrix array-element will be assigned this list values.
3. the list contains multiples of 9 elements (18, 27, etc.), dimensions of 3x3 are presumed. (In this case the number of list elements must be consistent with the <array-shape> and the default, 3x3.)

Note that the array-type function, MATRIX, must always be subscripted by <array-shape> to distinguish it from the data-type conversion function, MATRIX.

APPENDIX C

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
BIT_ARRAY
BIT LENGTH
BY
CALL
CASE
CAT
CHAR
CHARACTER
CHAR_ARRAY
CHAR_LENGTH
CLOSE
COLUMN
CONSTANT
DECLARE
DO
ELSE
END
ERROR
ER_RUPT
EVENT
EXCLUSIVE
EXIT
FALSE
FILE
FOR
FUNCTION
GO
HEX
ID_CODE
IF
IN
INCLUDE
INDEPENDENT
INITIAL
INTEGER
LABEL
LATCHED
LINE
MATRIX
MATRIX_DIM
NOT
NOT_QUALIFIED
OCT
OFF
ON
OR
OUTER
PAGE
PRECISION
PRIO
PRIO_CHANGE
PRIORITY
PROCEDURE
QUALIFIED
READ
READ_ALL
REPLACE
RETURN
SCALAR
SCHEDULE
SIGNAL
SKIP
STATIC
SYSTEM
TAB
TASK
THEN
TERMINATE
TO
TRUE
UNTIL
UPDATE
VARYING
VECTOR
VECTOR_LENGTH
WAIT
WHILE
WRITE