NASA CR.
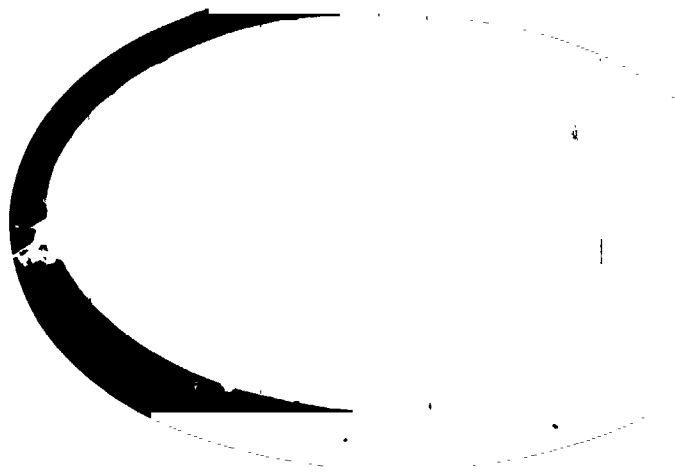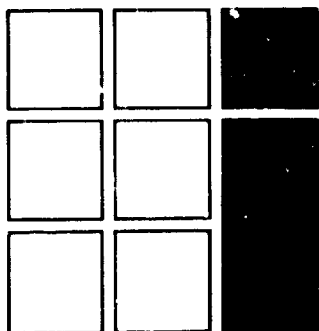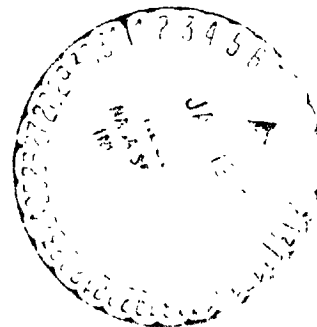
(NASA-CR-140389)   HAL/S PROGRAMMER'S GUIDE   N75-14479
(Intermetrics, Inc.)   240 p HC $7.50
                                      CSCL 09B
                                                Unclas
                                        G3/61   06027

INTERMETRICS

HAL/S

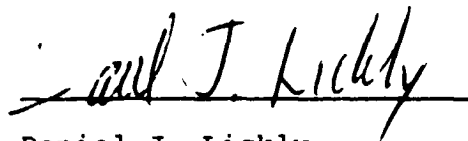## PROGRAMMER'S

GUIDE

IR-63-4

15 August 1974

Prepared by:

P.M. Newbold
R.L. Hotz

Typescript:

V.L. Cripps

Approved:

aul J. Lickly

Daniel J. Lickly
HAL Language/Compiler Dept.
Head

Approved:

F. H. Martin

Dr. F. H. Martin
Shuttle Program Manager

FOREWORD

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

CONTENTS OF PART I

INTRODUCTION

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

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

REVIEW OF THE LANGUAGE

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

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

DATA TYPES AND COMPUTATIONS

HAL/S provides facilities for manipulating a number of different data types. Its integer, scalar, vector, and matrix types, together with the appropriate operators and built-in functions provide an extremely powerful tool for the implementation of guidance and control algorithms. Bit and character types are also incorporated.

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

REAL TIME CONTROL

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

ERROR RECOVERY

HAL/S contains an elaborate run time error recovery facility
which allows the programmer freedom (within the constraints
of safety) to define his own error processing procedures, or
to leave control with the operating system.

SYSTEM LANGUAGE

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

PROGRAM RELIABILITY

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

ABOUT THE PROGRAMMER'S GUIDE

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

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

The Programmer's Guide is divided into three parts:

- PART I is aimed at the new HAL/S user and contains
  enough information on the compiler language constructs
  to enable him to begin programming.

- PART II describes other, more complex, HAL/S constructs
  which will be used regularly in applications programming.

- PART III presents programming examples designed to
  illustrate and clarify important complex HAL/S Language
  constructs. Some of the examples are constructs too
  advanced to be described in PARTS I and II, but which
  are formally defined in the HAL/S Language Specification.

PART I

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

1. STRUCTURE OF HAL/S

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

1.1 STRUCTURING AND HIGHER ORDER LANGUAGES

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

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

---

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

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

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

1.2   THE BLOCK STRUCTURE OF HAL/S

The structure of a HAL/S compilation, as indicated below,
generally consists of a program block with so-called
procedure and function blocks nested within it.
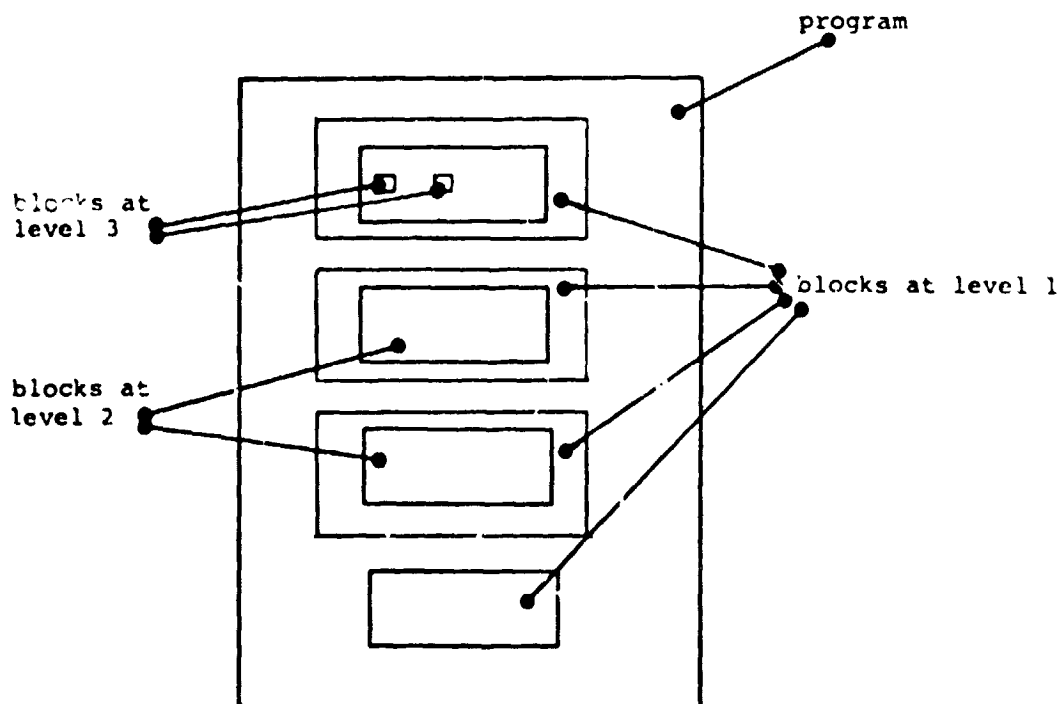
Function and procedure blocks constitute the HAL/S
interpretation of the "subroutines" of Section 1.1.
The more deeply such a block is nested, the greater
the depth of detail of the problem solution it is
supposed to handle. The blocks at each level contain
executable code implementing the appropriate part of
the problem solution.

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

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

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

---

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

1-3

SCOPING OF DATA

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

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

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

program

inner blocks

1-4
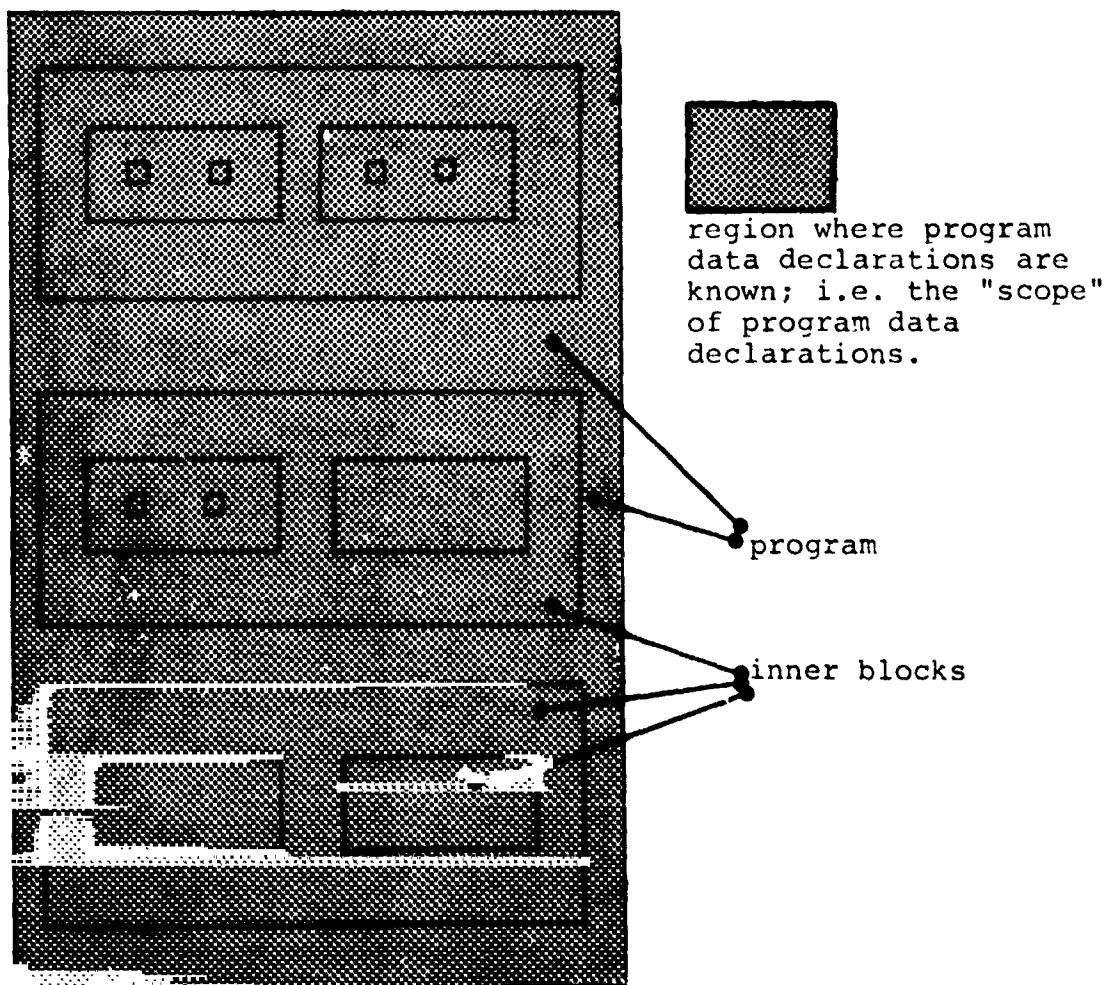
In addition, any procedure or function block nested within a program block may declare local data - data known only in that particular block and in blocks nested within it - as indicated below:

region where data declared local to X are known

region where data declared local to Y are known
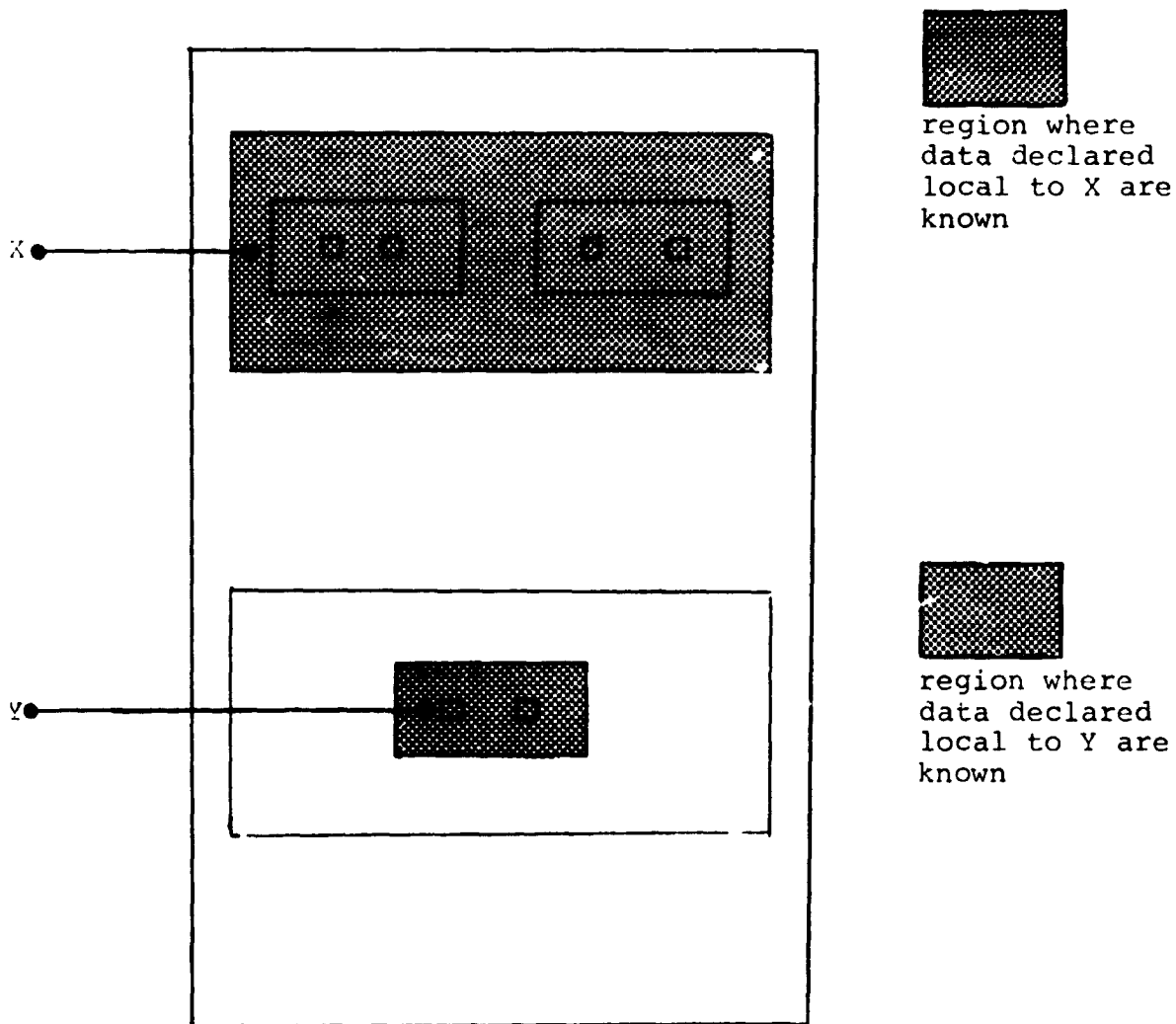
SCOPING OF BLOCK NAMES

The program block, and every procedure or function within it are named: block names have scoping rules identical with the data scoping rules already described. The name of any procedure or function block is deemed to have been "declared" in the surrounding block in which the procedure or function is nested. This bounds the region where its name is known, and therefore determines where it may be invoked. Thus, the name of any procedure or function nested at the program level is known anywhere in the program. However, since in HAL/S recursion is not allowed, such a procedure or function may be <u>invoked</u> from anywhere in the program <u>except inside itself</u>, as indicated:
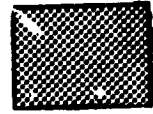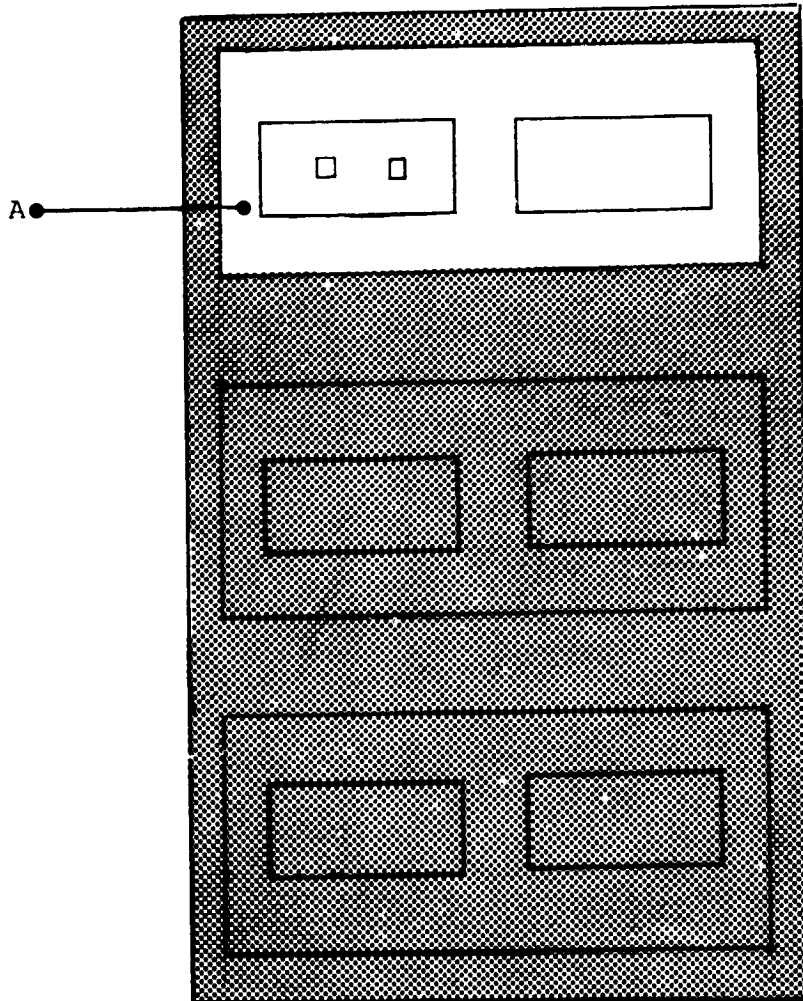
region where
block A may be
invoked

A•

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

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

region where block B may be invoked

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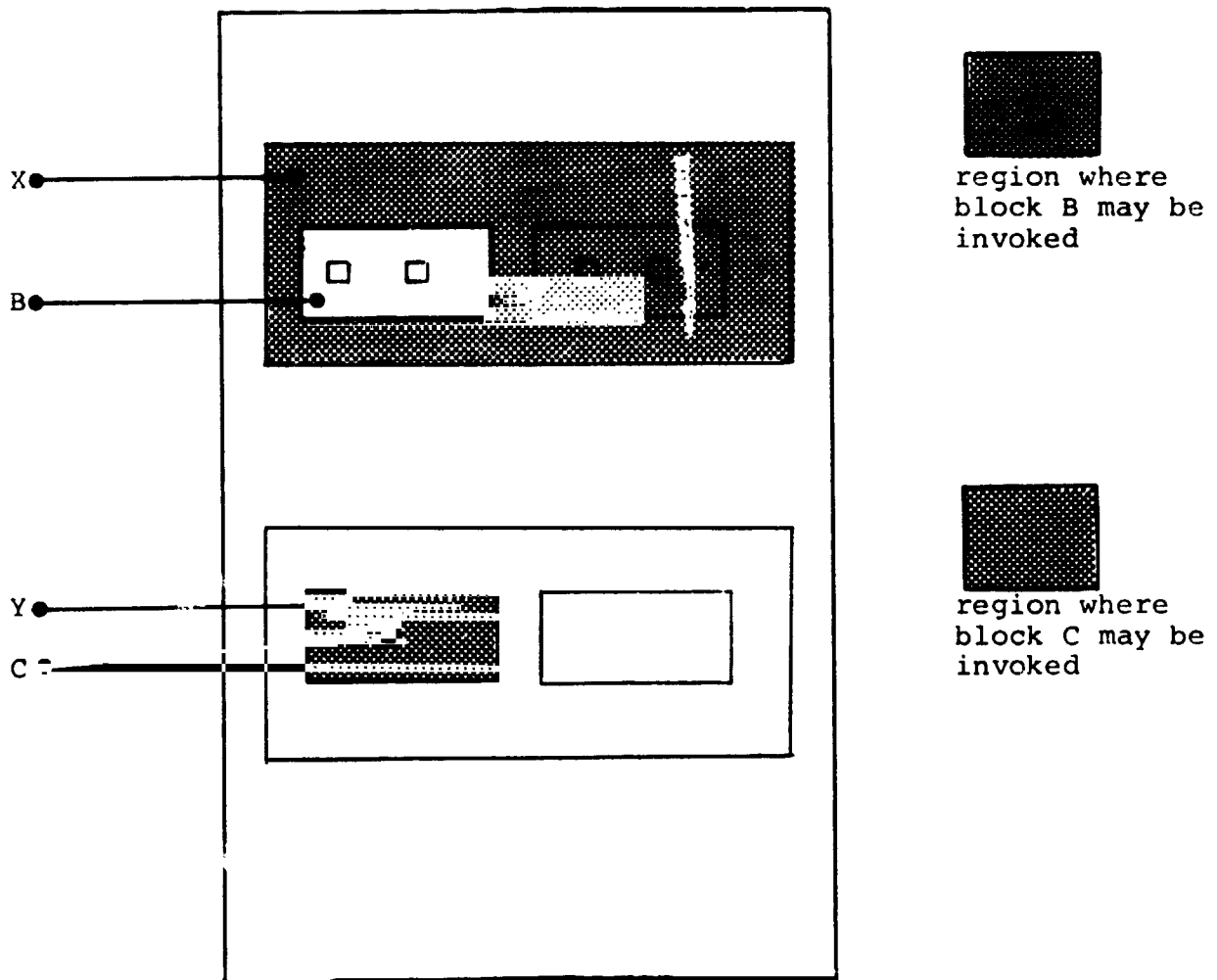
It should be noted that all forms of recursion in HAL/S
are illegal. The form of recursion not prevented by
the rules given above is that in which procedures P and
Q are not contained in each other, but P calls Q and Q
calls P.

It is also possible for a program
(or any block within it) to in-
voke entities outside the compila-
tion unit; i.e. other compilation
units. Procedures and functions
may be compiled independently for
this purpose.
See: (tbd)

1-8

1.3 STATEMENT GROUPING IN HAL/S

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

- the sequence may be executed once only;

- the sequence may be executed repetitively until specified
  conditions are met;

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

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

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

STATEMENT GROUPS AND GO TO STATEMENTS

The design of HAL/S minimizes the dangers of "GO TO" statements
by limiting the regions which can be branched to by them, in a way
analogous to the limits imposed on data by the scoping rules
described in Section 1.2.

Consider a program containing nested groups of executable statements as shown below:
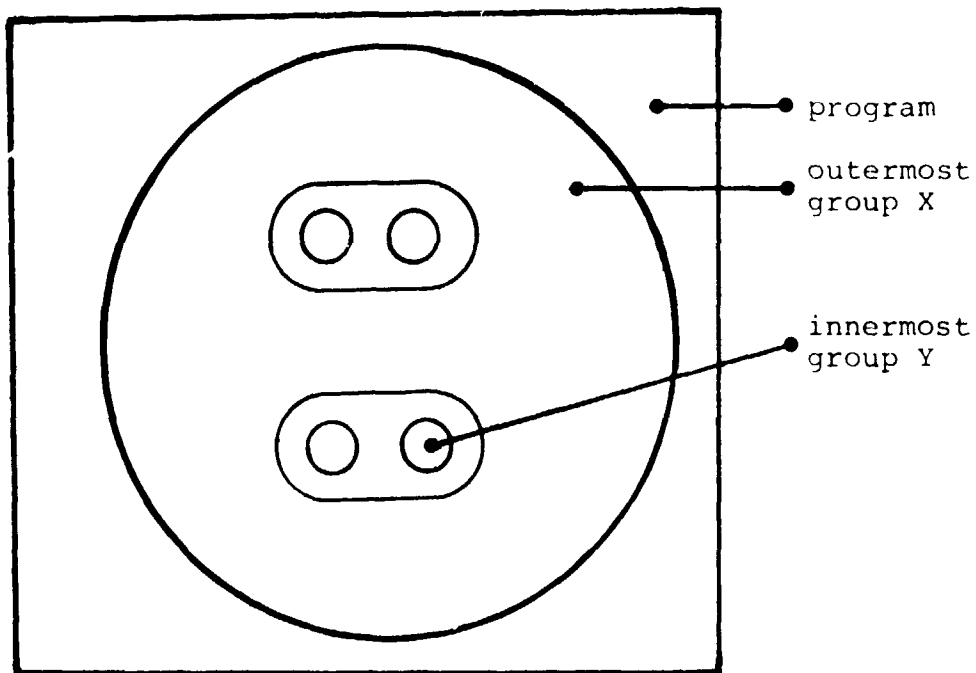
The region of legal destinations of "GO TO" statements contained in group X are as indicated below:
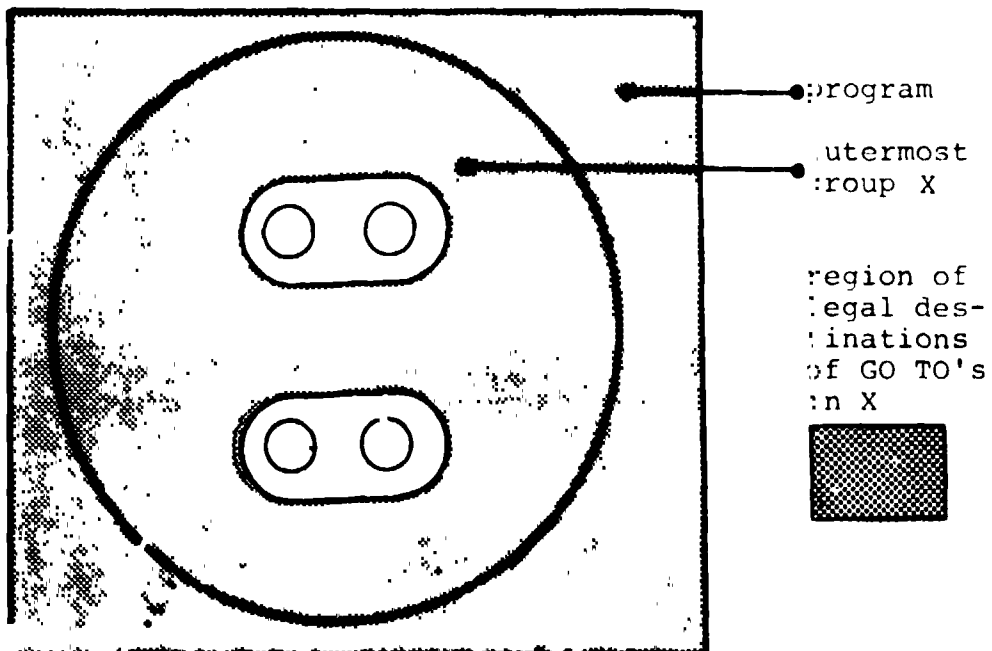
The region of legal destinations of "GO TO" statements contained
in group Y are as indicated below:
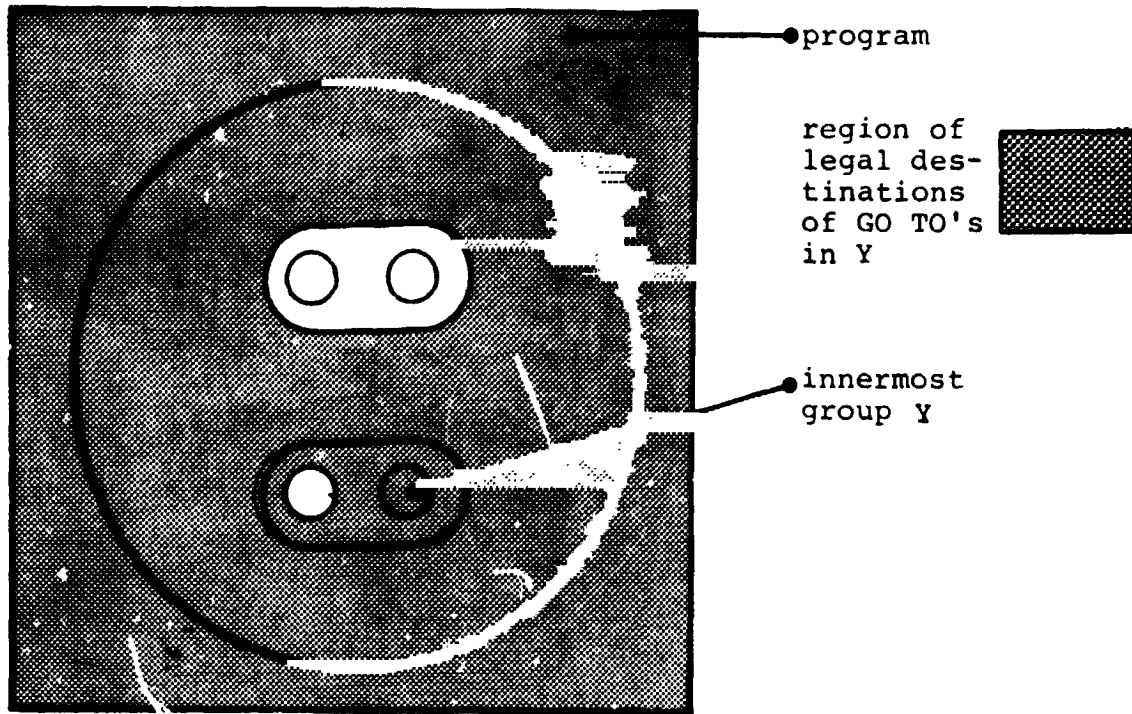
It is evident from the examples that while groups can be branched
out of, or branched within, they may not be branched into.

INTERACTION WITH BLOCK STRUCTURE

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

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

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

1-11

1.4 SUMMARY

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

2. HAL/S SYMBOLOGY

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

2.1 THE CHARACTER SET

The HAL/S language uses the following character set:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz

0123456789

+-*./|¬&=<>#@$,;:'")(_%¢

(blank)
```

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

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

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

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

2.2 RESERVED WORDS, IDENTIFIERS, AND LITERALS

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

- RESERVED WORDS are a fixed part of the language and consist
  ·of combinations of upper case alphabetic characters;

- IDENTIFIERS are user-defined names used for data or labels,
  and consist of combinations of the alphanumeric characters;

- LITERALS express actual values, and can consist of any of the
  symbols in the character set;

- SPECIAL CHARACTERS serve as delimiters, separators or
  operators, and consist of the non-alphanumeric
  characters of the HAL/S set.

RESERVED WORDS

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

- KEYWORDS are used to express parts of HAL/S statements, for
  example: GO TO, DECLARE, CALL, and so on. A complete
  list can be found in Appendix .

- BUILT-IN FUNCTION NAMES are used to identify a library of
  common mathematical and other routines, for example :
  SINE, SQRT, TRANSPOSE, and so on. A complete list can
  be found in Appendix .

IDENTIFIERS

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

---

1. The total number of characters in the name must not exceed 32;

2. The first character must be alphabetic;

3. The remaining characters may be either alphabetic or numeric;

4. Any character except the first or last may be an underscore (_).

Examples:

ELEPHANT_AND_CASTLE
A1   } legal
P

1B   } illegal
X_X_

---

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

LITERALS

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

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

```
mantissa ─────┐        ┌─exponent
              ‿‿‿      ‿‿‿
       ±ddd.ddd±ddd
```

1. ddd represents an arbitrary number of decimal digits.

2. The exponent is optional.

3. The + signs are optional.

4. The decimal point is optional. If absent, it is considered to be to the right of the least significant digit of the mantissa. If the decimal point is present, it may appear anywhere in the mantissa.

5. The minimum number of digits in the mantissa, and in the exponent, if present, is one. The maximum number is implementation dependent. (See Appendix).

Examples:

        0.123E16
        45.9
        -4

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

The use of multiple exponents,
and of binary, hexadecimal or
octal exponents, is also allowed.
See: (tbd).

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

```
          'ccccccc'


    1.  The quote marks delimit the
        beginning and end of the
        literal.

    2.  cccc represents an arbitrary
        number of characters in any
        combination.

    3.  Quote marks within the literal
        must be represented by a pair
        of quote marks to avoid con-
        fusion with the delimiting
        quotes.

    4.  The minimum number of characters
        is zero (a 'null' string), the
        maximum is 255*.
```

---

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

Examples:

```
' '
'ONE two THREE'
'DOG''S'
```

If a literal consists of a single
character, or character sequence
repeated may times, a condensed
form of literal using a repeti-
tion factor may be used.
See: (tbd).

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

```
TRUE    } expressing truth, or
ON      } binary "1"


FALSE   } expressing falsehood
OFF     } or binary "0"
```

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

2-7

2.3   FORMAT OF SOURCE TEXT

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

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

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

SINGLE-LINE FORMAT

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

● EXPONENTS are denoted by the operator **

    Example:

       x^{t+2} is coded as:

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

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

    Example:

       a_{i+1} is coded as:

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

MULTI-LINE FORMAT

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

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

    Example:

    x^{t+2} is coded as:

    ```
    :E     T+2
    :M     X
    ```

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

    Example:

    a_{i+1} is coded as:

    ```
    :M     A
    :S       I+1
    ```

When using multi-line format, care must be taken to ensure that
nothing on the E- and S-lines overlaps anything on the M-line.

Exponents of exponents and sub-
scripts of subscripts use extra
subscript and exponent lines.
Special rules apply if exponents
are subscripted, or if subscripts
possess exponents.
See: (tbd).

2.4 STATEMENT DELIMITING

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

Examples:

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

2.5 COMMENTS IN HAL/S

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

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

- COMMENT LINES may appear between main, exponent and subscript lines of HAL/S text.

IMBEDDED COMMENTS

An imbedded comment takes the form:

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

2-10

Such comments may appear between HAL/S statements or imbedded in a statement. They may <u>not</u> appear in the middle of a literal, reserved word, or identifier. As far as the sense of the source text is concerned, an imbedded comment is treated as if it were a string of blank characters.

Example:

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

COMMENT LINES

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

Examples:

```
'M    X = X + 1;
'C    ADD ONE TO X
'C    THEN CARRY ON
```

2.6   SUMMARY

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

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

3. A HAL/S COMPILATION - THE PROGRAM BLOCK

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

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

3.1 OPENING AND CLOSING THE PROGRAM BLOCK

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

PROGRAM OPENING

The statement opening a program block takes the form:

```
    label : PROGRAM;


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

PROGRAM CLOSING

The program block is closed with the statement:

```
        CLOSE   label ;
```

1. The identifier *label* is optional.

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

Example:

```
  TEST: PROGRAM;



  CLOSE TEST;
```
—●body of program goes in here

3.2   POSITION OF DATA DECLARATIONS

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

Example:

```
' TEST: PROGRAM;
:
|
|                              )
|                              |—•data declaration statements
|                              )
|
:
|                              )
|                              |—•executable statements
:                              )
|
! CLOSE TEST;
```
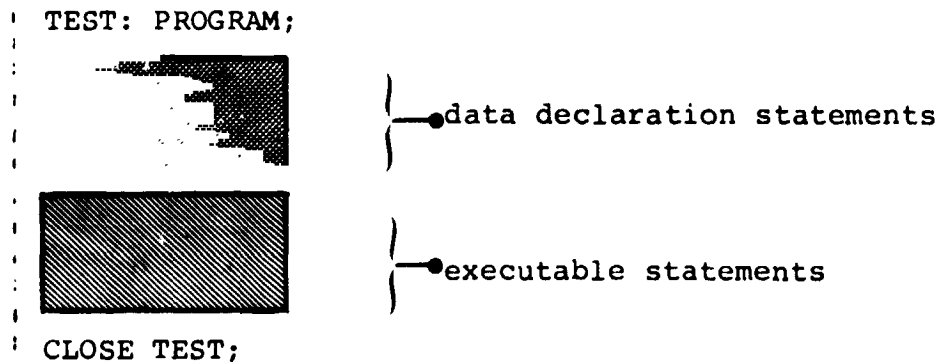
3.3 FLOW OF EXECUTION IN THE PROGRAM

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

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

---

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

Example:

TEST: PROGRAM;

data
declaration
statements

executable
statements

procedure
definition
block

path of
execution

CLOSE;

block invoked
and returned
from
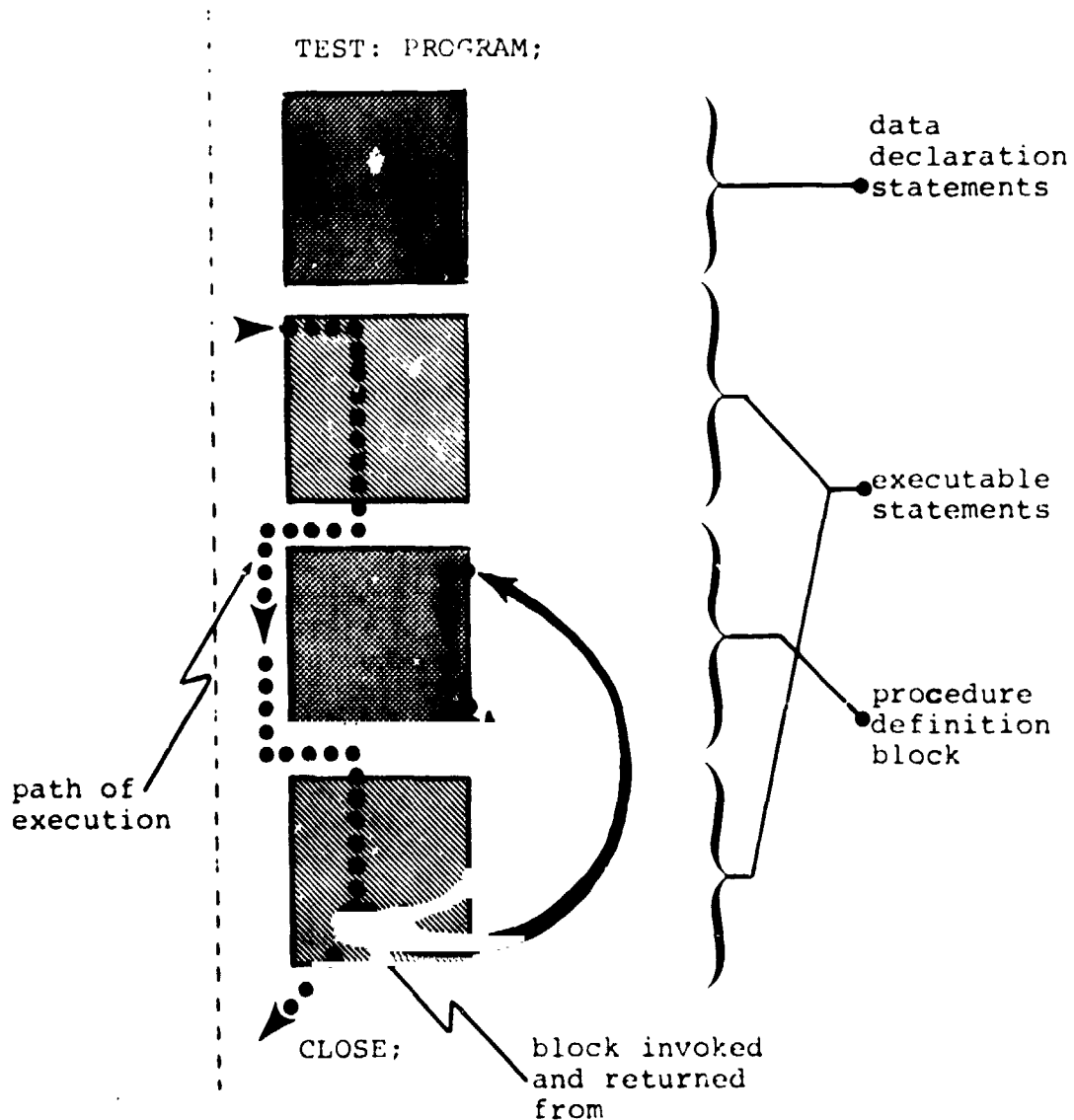
3.4 SUMMARY

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

3-4

4. DATA DECLARATION

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

Identifiers of the kind described in Section 2 are used to name
items of data. Identifier names used to represent data items
must* be defined in data declarations appearing in the appropriate
program, prodcedure or function block. The effect of placing
data in different blocks is described in Section 1. The position
of data declarations within a program block is described in
Section 3.

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

4.1 HAL/S DATA TYPES

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

- INTEGER for the representation of integer-valued quantities;

- SCALAR for the representation of "floating-point" quantities;

- VECTOR for the representation of algebraic row or column
  vectors (without distinction), and each element of which is
  a scalar quantity;

- MATRIX for the representation of algebraic matrices, and each
  element of which is a scalar quantity.

---

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

4-1

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

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

● CHARACTER for the representation of strings of text;

● BOOLEAN for the representation of binary-valued (logical) quantities.

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

HAL/S in fact allows more
data types than just those
described here. It also
allows hierarchical organ-
izations of data-types
called "structures".
See: (tbd)

4.2 SIMPLE DECLARATION STATEMENTS

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

INTEGER

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

1. In each of the forms   name   is any legal
    HAL/S identifier.

2. Presence of the keyword SINGLE specifies
    single precision.

3. Presence of the keyword DOUBLE specifies
    double precision.

4. Absence of either keyword implies default
    of single precis on.

For the integer data type, single precision usual'/ implies
halfword and double precision fullword, depending on the
implementation*.

Examples:

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

---

* See Appendix .

4-3

SCALAR

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

1. In each of the forms, *name* is any
    legal identifier.

2. Presence of the keyword SINGLE specifies
    single precision.

3. Presence of the keyword DOUBLE specifies
    double precision.

4. Absence of either keyword implies a de-
    fault of single precision.

5. The keyword SCALAR may be omitted.

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

Examples:

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

---

* See Appendix .

MATRIX

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

1. In each form *name* is any legal identifier.

2. Keywords SINGLE and DOUBLE have the same
    significance as for scalar and vector types.

3. m and n denote respectively the number of
    rows and columns in the matrix. They must
    lie in the range $1 < m, n \leq 64$*.

4. If the size specification (m,n) is absent,
    a 3x3 matrix is assumed.

Examples:

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

                        a 3x3 matrix

---

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

4-5

VECTOR

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

1. In each form name is any legal
   identifier.

2. Keywords SINGLE and DOUBLE have the
   same significance as for scalar type.

3. n specifies the length of the vector
   and must lie in the range $1 < n \leq 64$*.

4. If the length specification (n) is
   omitted a length of 3 is assumed.

Examples:

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

---

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

CHARACTER

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

1. name   is any legal identifier.

2. n specifies the maximum length of the text
    string that the data type may carry. (i.e.
    the maximum number of characters). It must
    lie in the range of $1 \leq n \leq 255$*.

3. The actual length of the string of text
    carried may vary during execution between
    zero (a "null" string) and the maximum n.

Example:

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

BOOLEAN

```
DECLARE  name   BOOLEAN;
```

1. name   is any legal identifier.

Example:

```
DECLARE B1 BOOLEAN;
```

---

* This value may be implementation dependent. See Appendix

4-7

ARRAYS

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

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

:DECLARE *name* ARRAY(n) *attributes* ;

1. *attributes*   stands for any legal form of attributes for any data type described.

2. n denotes the number of elements in the array (i.e. entries in the table) and must lie in the range $1 < n \leqslant 32768$*.

Examples:

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

---

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

COMPOUND DECLARA.IOI'S

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

Example:

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

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

Note the commas separating the declaration of each data item.

If the identifiers in a compound
declaration have some attributes
in common, a third, even more
compact, form of declaration
called a factored declaration
can be used.
See: (tbd)

4.3 INITIALIZATION OF DATA

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

● UNI-VALUED data items are those having only one element:
  unarrayed scalars, booleans, and characters.

● MULTI-VALUED data items are those having more than one
  element: unarrayed vectors and matrices, and arrayed
  data items of any type.

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

UNI-VALUED DATA ITEMS

The two variations of the form of INITIAL specification for
uni-valued data items are:

INITIAL (*value*)
CONSTANT (*value*)

1. The two forms have the same effect in
    that the data item is initialized to
    the literal indicated by *value* .

2. The form using the keyword CONSTANT is
    required only if the user wishes <u>not</u>
    to change the initial value during
    execution*.

3. The type of the literal *value* must
    be compatible with the type of the data
    item as determined from the following
    table:

data type	literal value
CHARACTER	character string
BOOLEAN	boolean
INTEGER	arithmetic
SCALAR	

---

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

Examples:

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

    Note:  initial working length of C becomes 3.
```

MULTI-VALUED DATA ITEMS

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

INITIAL($value^1_1$, $value^2_2$, ...)
CONSTANT($value^1$, $value^2$, ...)

1. The meaning of the keyword CONSTANT is
    the same as for uni-valued data items.

2. The type of each literal $value$ must be
    compatible with the type of the data item,
    as determined from the following table.

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

3. The number of literals in the list must
    equal the total number of elements implied
    by the data declaration.

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

Examples:

```
'DECLARE V VECTOR INITIAL(1,2,3.5)
'        S ARRAY(2) CONSTANT(1,0),
'        T ARRAY(2) VECTOR(2) INITIAL(4.7,-5.3,0,0);
'
'DECLARE V VECTOR INITIAL(0),
'        S ARRAY(100) INTEGER INITIAL(256);
```

all elements of these data
items are identically
initialized.

ORDER OF INITIALIZATION

To complete the specification of initialization the <u>order</u> of
initialization of the elements of multi-valued data <u>items</u>
needs to be defined.

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

- VECTOR data items are initialized in order of increasing
  index.

- MATRIX data items are initialized row by row in order of
  increasing index.

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

Example:

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

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

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

4-12

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

4.4 SUMMARY

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

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

4-13

5. REPLACE STATEMENTS

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

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

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

5.1 THE REPLACE STATEMENT

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

> REPLACE *name* BY "XXXXXXXXXXX";

1. XXXXXXX represents the HAL/S source text which
   it is desired to substitute. The text is de-
   limited by double quote marks, and must be
   written in single line format.

2. *name* is the symbolic name chosen to repre-
   sent the text. It may be any legal identifier
   name.

3. XXXXXXX may be any legal source text of arbi-
   trary length. Imbedded double quote marks
   must be represented as a pair of double quote
   marks to avoid confusion with the delimiters.

4. The text must not begin or end in the middle
   of a reserved word, identifier, literal, or
   imbedded comment.

5-1

Examples:

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

5.2   USING REPLACE STATEMENTS

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

Examples:

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

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

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

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

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

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

```
REPLACE X BY "REPLACE Y BY""Z""";
X;
DECLARE Y SCALAR;
```

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

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

Example:

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

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

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

```
┌──────────────────────────────────────┐
│  REPLACE X BY "Y";          ●──────── Program
│                                       │
│  ┌──────────────────────┐             │
│  │ DECLARE X SCALAR;     │ ●────────── Procedure block
│  │                       │             │
│  │                       │             │
│  │                       │             │
│  └──────────────────────┘             │
└──────────────────────────────────────┘
```
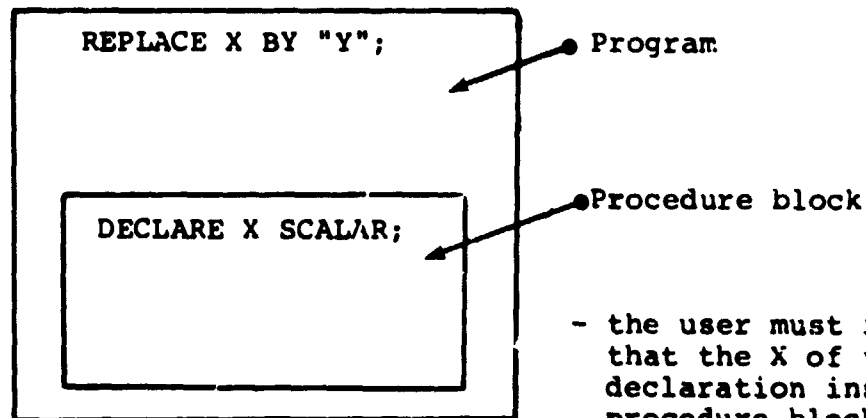
- the user must remember
  that the X of the local
  declaration inside the
  procedure block is still
  subject to the REPLACE
  statement at the program
  level.

5-3

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

Example:

REPLACE X BY "Y"; •Program

R_PLACE X BY "Z"; •Procedure block

•Procedure block

region where X is
replaced by Y

region where X is
replaced by Z
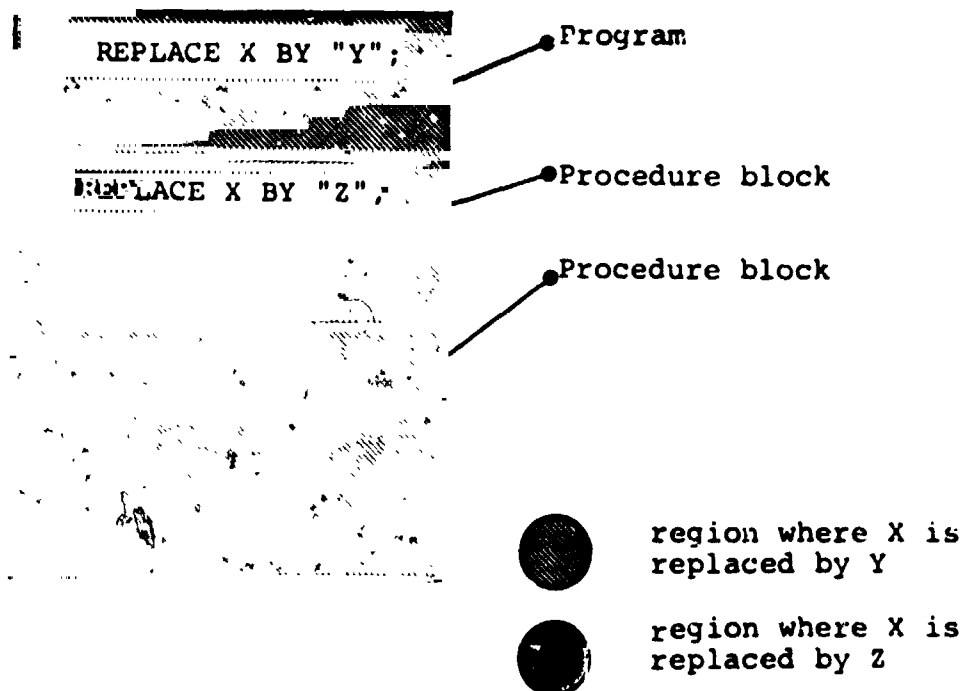
> Replace statements may also
> possess parameters, turning
> them into a sophisticated
> macro expansion facility.
> See: (tbd).

5-4

5.3 SUMMARY

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

6. DATA REFERENCING AND SUBSCRIPTING

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

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

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

- ARRAY SUBSCRIPTING allows the user to select elements or subsets of elements from arrays of any data type.

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

6.1 SUBSCRIPTS OF UNARRAYED DATA TYPES

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

---

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

CHARACTER

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

- To select the αth character from STRING:

> $$\text{STRING}_\alpha$$
>
> 1. α is an integer expression in the range $1 \leqslant \alpha \leqslant L$.

- To select α characters from STRING, starting from the βth:

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

---

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

- To select a substring starting with the αth character of STRING, and ending with the βth:

```
                    STRING
                          α  TO  β


    1.   α and β are integer expressions in
         the range 1 ⩽ α, β ⩽ L.

    2.   β ⩾ α.
```

Examples:

if the value of C is 'ABCDEF' then:

C_5 is 'E'

$C_{2\ AT\ 2}$ is 'BC'

$C_{4\ TO\ 6}$ is 'DEF'

VECTOR

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

- To select the αth element from VEC:

```
                    VEC
                       α


    1.   α is an integer expression in the
         range 1 ⩽ α ⩽ L.

    2.   The resulting data type is scalar.
```

- To select an α-vector partition starting from the βth element of VEC:

$$VEC_{\alpha \text{ AT } \beta}$$

1. α is an integer <u>literal</u> <u>value</u> in the range $2 \leq \beta \leq L$.

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

- To select a partition starting from the αth element of VEC and ending with the βth.

$$VEC_{\alpha \text{ TO } \beta}$$

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

2. $\beta > \alpha$.

Examples:

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

$V_1 = 4.5$ (scalar)

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

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

6-4

MATRIX

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

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

  $$MAT_{\alpha, \beta}$$

  1. α, β are integer expressions.

  2. α is in the range $1 \leqslant \alpha \leqslant M$,
      and β is in the range $1 \leqslant \beta \leqslant N$.

  3. The resultant data type is SCALAR.

- To select the αth row of MAT:

  $$MAT_{\alpha, *}$$

  1. α is an integer expression in the
      range $1 \leqslant \alpha \leqslant M$.

  2. The resultant data is N-vector.

  3. If the asterisk is replaced by a
      TO- or AT- subscript under the
      rules given for vector data types,
      a vector partition from the αth
      row may be selected.

- To select the βth column of MAT:

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

- To select a $\alpha \times \gamma$ matrix partition starting from the βth row and δ^{th} column of MAT:

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

6-6

Examples:

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

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

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

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

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

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

6.2   SUBSCRIPTS OF ARRAYED DATA TYPES

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

ARRAY SUBSCRIPTING ONLY

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

● To select the αth array element from TABLE:

> TABLE_α:
>
> 1. α is an integer expression in the
>   range $1 \le \alpha \le L$.
>
> 2. The colon is optional if the data
>   type of TABLE is integer or scalar.

● To select a sub-array of length α starting from the βth
   array element of TABLE:

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

- To select a sub-array starting from the αth array element of TABLE and ending with the βth.

> TABLE$_{\alpha}$ TO β:
>
> 1. α, β are integer <u>literal values</u> in the range $1 \leqslant \alpha, \beta \leqslant L$.
>
> 2. $\beta \leqslant \alpha$.
>
> 3. The colon is <u>optional</u> if the data type of TABLE is integer or scalar.

Examples:

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

$T_{2:}$ is FALSE   (unarrayed)

$T_{3 \text{ TO } 4:}$ is (TRUE,TRUE)   (still arrayed)

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

T_2 is 2   (unarrayed) $\}$ optional colon

$T_{3 \text{ TO } 4}$ is (3,4)   (still arrayed) $\}$ omitted

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

$C_{1:}$ is 'YES'   (selects first array element)

$C_{2 \text{ TO } 3:}$ is ('NO','MAYBE')   (still arrayed)

ARRAY AND COMPONENT SUBSCRIPTING

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

TABLE
array: component

1. *array* represents array sub-
   scripting of any of the forms
   previously described.

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

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

Examples:

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

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

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

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

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

> Apparently, the colon should be
> opticnal on Boolean data types
> also. It is not because the
> Boolean data type is a degener-
> ate case of a bit string data
> type which may possess com-
> ponent subscripting.
> See: (tbd).

COMPONENT SUBSCRIPTING ONLY

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

> $$TABLE_{*:\,component}$$
>
> 1. *component* represents any form
>   of component subscripting legal for
>   the data type of TABLE, as described
>   in Section 6.1.

Examples:

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

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

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

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

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

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

6-11

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

6.3   SUMMARY

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

7.   EXPRESSIONS

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

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

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

7.1   ARITHMETIC OPERATIONS

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

---

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

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

NEGATION

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

Symbolic form: - R

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

   R-type

   MATRIX
   VECTOR
   SCALAR
   INTEGER

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

Examples:

if I is an integer and I ≡ 5

then -I ≡ -5

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

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

7-2

ADDITION AND SUBTRACTION

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

Symbolic form: $L \pm R$

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

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

2. Operations on matrix and vector operands imply element-by-element addition and subtraction.

3. The operands in a matrix addition or subtraction must have the same row and column dimensions.

4. The operands in a vector addition or subtraction must have the same lengths.

5. In a mixed integer-scalar operation, the result is scalar. The integer operand is first converted to single precision scalar.

Examples:

If I is integer with $I \equiv 5$

   S is scalar with $S \equiv -4.2$

then

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

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

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

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

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

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

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

but
$V1 - V2_{1 \text{ TO } 3} \equiv \begin{bmatrix} -1.5 \\ -2.5 \\ 1.0 \end{bmatrix}$ is legal because subscripting of the R operand has produced a 3-vector.

Using S, V1 above,

   S + V1   is illegal because the types are incompatible;

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

7-4

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

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

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

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

DIVISION

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

Symbolic form: L / R

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

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

2. If the dividend is of matrix or vector type, element-by-element division by the R operand is implied.

3. If either or both operands are of integer type, they are first converted to scalar type.

7-5

Examples:

$$1/2 \equiv 0.5 \quad \text{(both integer operands converted to scalar)}$$

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

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

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

S is a scalar with S $\equiv$ 0.5

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

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

DOT PRODUCT

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

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

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

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

Symbolic form: L . R

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

L-type	R-type
VECTOR	VECTOR

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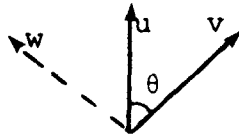
3. The result is of scalar type.

Example:

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

then V.V = 1.5

CROSS PRODUCT

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

if w is perpendicular to u, v
   as shown,
and $|w| = |u||v|\sin \theta$
then $w = u \times v$

Symbolic form: $L * R$

1. The type of the operands must be vector:

L -type	R -type
VECTOR	VECTOR

2. Both operands <u>must</u> be of length 3.

3. The result is a 3-vector.

Example:

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

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

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

MULTIPLICATION

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

- If operand types are either integer or scalar, multiplication
  in the regular arithmetic sense is implied;   ...CASE ①

- if one operand is integer or scalar, and the other vector or
  matrix, then element-by-element multiplication is implied;
                                                       ...CASE ②

- if both operands are vectors then the outer product is implied,
  the result being a matrix;   ...CASE ③

- if both operands are matrices, the matrix product is implied;
                                                       ...CASE ④

- if one operand is a matrix, and the other a vector. then
  a vector-matrix product is implied, the result being a
  vector.   ...CASE ⑤

The symbolic form for multiplication is as shown:

```
        Symbolic form:   L R

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

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

CASE ①

> **2.** The operand types are:
>
> | L-type | R-type |
> |----------|----------|
> | INTEGER | INTEGER |
> | SCALAR | SCALAR |
>
> **3.** If both operands are integer, the result is integer, otherwise it is scalar.
>
> **4.** If one operand is integer, then it it first converted to single precision scalar.

Example:

If I is integer with I = 10

then 1.5E-2 I = 0.15   (scalar result)

CASE ②

> **2.** The operand types are:
>
> | L-type | R-type |
> |----------|----------|
> | INTEGER | VECTOR |
> | SCALAR | MATRIX |
> | VECTOR | INTEGER |
> | MATRIX | SCALAR |
>
> **3.** Element-by-element multiplication of the vector or matrix is implied.
>
> **4.** If an operand is of integer type, it it first converted to single precision scalar.

7-9

Examples:

if S is scalar with S $\equiv$ 1.5

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

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

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

CASE ③

> 2. The operand types are:
>
> | L-type | R-type |
> |----------|----------|
> | VECTOR | VECTOR |
>
> 3. If the L-operand is of length m,
> and the R operand is of length n,
> the result is an m x n matrix.

Examples:

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

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

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

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

CASE ④

> 2. The operand types are:
>
> | L-type | R-type |
> |----------|----------|
> | MATRIX | MATRIX |
>
> 3. The number of columns in the L operand must equal the number of rows in the R operand.
>
> 4. If the L operand is an m x n matrix and the R operand is an n x p matrix, the result is an m x p matrix.

Examples:

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

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

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

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

Note that by using partitioning subscripts that

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

but $M2\ M1_{*,2\ TO\ 3} \equiv \begin{bmatrix} -0.25 & 0.5 \\ -0.5 & 1 \\ -0.5 & 1 \end{bmatrix}$ is still legal

7-11

CASE ⑤

> 2. The operand types are:
>
> | L-type | R-type |
> |--------|--------|
> | VECTOR | MATRIX |
> | MATRIX | VECTOR |
>
> 3. If the L operand is an m x n matrix, the R operand must be an n-vector, and the result is an m-vector.
>
> 4. If the L operand is an m x n matrix, the R operand must be an m-vector, and the result is an n-vector.

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

Examples:

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

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

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

and M V is illegal because of dimension mismatch;

however, $M\ V_{1\ TO\ 2} = \begin{bmatrix} -0.5 \\ -1.0 \\ -0.2 \end{bmatrix}$ is legal.

EXPONENTIATION, INVERSION AND TRANSPOSE

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

- If both operands are integer or scalar, then exponentiation is implied;   ...CASE (1)

- if the left operand is a square matrix, and the right is an integer-valued literal, a repeated matrix product or repeated product of inverse is implied;   ...CASE (2)

- if the left operand is a matrix, and the right operand is the character 'T', then the transpose is implied.   ...CASE (3)

These operations take the general symbolic form:

' Symbolic form: L ** R

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

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

CASE ①

> 2. The operand types are:
>
> | L -type | R -type |
> |---|---|
> | INTEGER | INTEGER |
> | SCALAR | SCALAR |
>
> 3. If the L operand is integer and the R operand is a non-negative integral-valued literal, then the result is integer, otherwise it is scalar.
>
> 4. Consistent with Rule 3, if the result is scalar, then any integer operands are first converted to single-precision scalar.

Examples:

If I is an integer with I ≡ 5

    then I ** 2 ≡   25   (integer result)

    and I**-1 ≡ 0.2   (scalar result)

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

CASE ②

<div style="border:1px solid">

2. The operand types are:

L-type	R-type
MATRIX	INTEGER

3. The L operand is a <u>square</u> matrix.

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

value	result
≤ -2	repeated product of inverse
-1	inverse
0	unit matrix
1	no-operation
≥ 2	repeated product

</div>

Examples:

If M is a 2 x 2 matrix with $M = \begin{bmatrix} 0.5 & 1 \\ -0.5 & 0 \end{bmatrix}$

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

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

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

CASE ③

> 2. The operand types are:
>
> | L-type | R-type |
> |----------|----------|
> | MATRIX | T |
>
> 3. If the L operand is an m x n matrix, then the result is an n x m matrix.
>
> 4. If R is symbolically T, then transpose is indicated <u>even if T is a declared data item</u>.

Examples:

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

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

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

then V^T is illegal because the L operand <u>is not matrix type</u>.

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

NOTE ON PRECISION CONVERSION

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

1. No precision conversion is possible in unary operations: transposition is considered a unary operation.

2. Where an operation specifies type conversion from integer to single precision scalar, this conversion is carried out first.

3. If only one operand is integer and <u>no type conversion is implied</u>, no precision conversion takes place.

4. If both operands have the same precision, the result is of the same precision (even if not of the same type).

5. If the operands have mixed precision, the single precision operand is converted to double precision. Then rule 4 is applied.

7.2   CHARACTER OPERATIONS

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

Symbol	Purpose
|| CAT }	catenation

CATENATION

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

Symbolic form: L \|\| R
                      CAT

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

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

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

7-18

Examples:

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

   I is integer with I ≡ 10

then 'TEN' || C ≡ 'TEN UNITS'

          I || C ≡ '10 UNITS'

and   I || I ≡ '1010'

7.3   BOOLEAN OPERATIONS

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

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

COMPLEMENT

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

> Symbolic form: ¬ R
>   NOT
>
> 1.   The R operand is of Boolean type.

Example:

  If B is Boolean with B $\equiv$ TRUE

  then ¬B $\equiv$ FALSE

CONJUNCTION

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

Symbolic form: $L \underset{OR}{\mid} R$

1. The L and R operands are of Boolean type.

2. The truth table for the resulting Boolean is as follows:

T=TRUE		L	
F=FALSE		T	F
R	T	T	T
	F	T	F

Examples:

If B is Boolean with B = FALSE

then B¦B = FALSE

B|TRUE = TRUE

INTERSECTION

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

Symbolic form: $L \underset{AND}{\&} R$

1. The L and R operands are of Boolean type.

2. The truth table for the resulting Boolean is as follows:

T=TRUE F=FALSE		L	
		T	F
R	T	T	F
	F	F	F

Examples:

    If B is Boolean with B = FALSE

       then B&TRUE = FALSE

          B&B   = FALSE

7.4   COMBINING OPERATIONS & PRECEDENCE

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

- The order in which operations are executed (th order
  of "precedence");

- the way in which the precedence order can be overriden.

ARITHMETIC AND CHARACTER PRECEDENCE

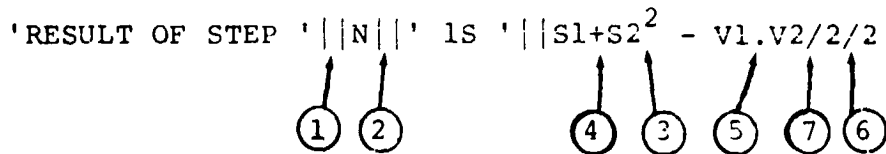The precedence of HAL/S operations on arithmetic and character
data types are shown in the following table:

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

Two rules clarify and modify this information:

- Sequences of operations of the same precedence are evaluated
  left to right, except for ** and /, which are evaluated right
  to left.

- Sequences of multiplications are sometimes reordered to minimize
  the number of elemental products required.

Examples:

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

$$\text{'RESULT OF STEP '}||N||\text{' IS '}||S1+S2^2 - V1.V2/2/2$$

```
          1  2        4  3  5  7  6
```
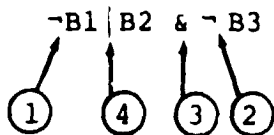
BOOLEAN PRECEDENCE

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

Symbol	Precedence	Purpose
¬, NOT &, AND |, OR	FIRST 1 2 3 LAST	complement intersection conjunction

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

Examples:

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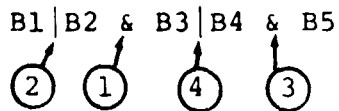order of execution of operations:

$$\neg B1 | B2 \ \& \ \neg B3$$

```
   1      4    3  2
```
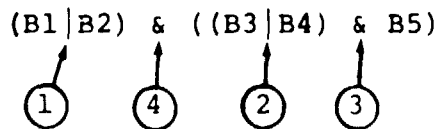
OVERRIDING PRECEDENCE ORDER

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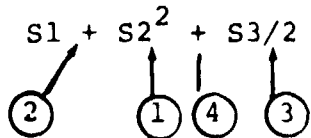the use of parentheses, nested to any arbitrary depth.
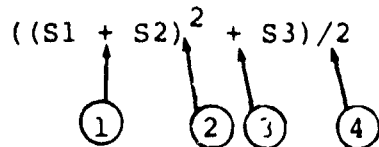
Examples:

In the following Boolean expression,

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

(2) (1) (4) (3)

parentheses may change the precedence order as shown:

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

(1) (4) (2) (3)

In the following arithmetic expression,

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

(2) (1) (4) (3)

parentheses may change the precedence order as shown:

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

(1) (2) (3) (4)

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

7-25

7.5   SOME EXPLICIT CONVERSIONS

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

PRECISION CONVERSION

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

$(expression)_{@}$ DOUBLE

$(expression)_{@}$ SINGLE

1.   In the first form, if *expression*   is
     a single precision arithmetic precision,
     it is converted to double precision.
     If it is already double precision, the
     conversion has no effect.

2.   In the second form, if *expression*   is
     a double precision arithmetic expression
     it is rounded to single precision.   If
     it is already single precision, the
     conversion has no effect.

Example:

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

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

   is double precision the type remaining unchanged.

VECTOR CONVERSION

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

$$\text{VECTOR}_n \; (\; exp^1, \quad exp^2 \; .....)$$

1. The subscript number n specifies the length of the vector to be created, and lies in the range $1 < n \leq 64^*$.

2. If n is omitted the resulting vector is assumed to be of length 3.

3. Each exp is a scalar or integer expression.

4. The number of expressions in the list __must__ match the implicit or explicit __result__ length.

5. The result of the above conversion is in single precision.

6. The matrix is assembled row by row from the list.

Examples:

    VECTOR(1, 2, 3)

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

---

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

if S is a scalar with S ≜ 0.5 then

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

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

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

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

1. The meanings of exp and n are as before.

2. If n is not specified, the preceding comma is also omitted.

Examples:

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

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

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

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

MATRIX CONVERSION

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

$$\text{MATRIX}_{m,n} (\ exp^1, \ exp^2, \ \ldots\ldots)$$

1. The subscript numbers m, n specify the
    row and column dimensions of the matrix
    to be created, and must lie in the range
    $1 < m, \ n \leq 64^*$.

2. The subscript may be omitted, in which
    case the resulting matrix is assumed to
    be 3 by 3.

3. Each exp is a scalar or integer
    expression.

4. The number of expressions must match the
    total number of elements in the resulting
    matrix.

5. The result of the above conversion is in
    single precision.

---

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

Exam les:

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

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

    $MATRIX_{2, 3}$(1.5, 0, 0, 0, 0.5, 0)

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

    Note the order of assembly in each case.

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

$$MATRIX_{@ \; DOUBLE, \; m, \; n}(exp^1 , \; exp^2 \; .....)$$

1. The meanings of m, n and exp are as
    before.

2. If the dimension subscript is omitted, the
    preceding comma is also omitted.

Examples:

    $MATRIX_{@ \; DOUBLE}$(1, 2, 3, 4, 5, 6, 7, 8, 9)

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

    $MATRIX_{A \; DOUBLE, \; 2, \; 3}$(1.5, 0, 0, 0, 0.5, 0)

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

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

7-31

7.6 BUILT-IN FUNCTIONS

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

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

● arithmetic

● algebraic

● vector-matrix

● character

● miscellaneous

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

ARITHMETIC FUNCTIONS

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

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

ALGEBRAIC FUNCTIONS

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

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

VECTOR-MATRIX FUNCTIONS

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

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

CHARACTER FUNCTIONS

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

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

MISCELLANEOUS FUNCTIONS

Some of the more important miscellaneous functions are:

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

Examples of use:

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

7.7   SUMMARY

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

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

8. ASSIGNMENTS

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

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

● Arithmetic assignments are assignments to matrix,
  vector, integer or scalar data items.

● Character assignments are assignments to character
  data items.

● Boolean assignments are assignments to Boolean
  data items.

8.1 GENERAL FORM OF ASSIGNMENT

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

Symbolic Form: L = R;

1. L is the receiving data item. It
   may be subscripted or unsubscripted.

2. Usually, R is an expression whose
   resultant value is to be used in the
   assignment. It may, of course, consist
   merely of a single operand.

8-1

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

8.2 ARITHMETIC ASSIGNMENTS

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

MATRIX

The receiving data item is a matrix.

---

1. The operand types are:

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

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

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

Examples:

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

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

then

$|M3 = -M1;$

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

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

but

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

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

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

but

$|M3 = 1;$ is illegal

VECTOR

The receiving data item is a vector.

1.	The operand types are:

L-type	R-type
VECTOR	{ VECTOR { INTEGER (rule 3)

2. The length of the R-expression must be the same as that of the receiving data item.

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

8-3

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

then

$|M3 = -M1;$

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

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

but

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

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

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

but

$|M3 = 1;$ is illegal

VECTOR

The receiving data item is a vector.

1. The operand types are:

L-type	R-type
VECTOR	{ VECTOR { INTEGER (rule 3)

2. The length of the R-expression must be the same as that of the receiving data item.

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

8-3

Examples:

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

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

then

    $V2 = -V1;$

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

    $M2 \equiv V1;$ is illegal (type mismatch),

but

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

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

    (? indicates values unchanged by assignment).

Note

    $V2 = 0;$ creates a null vector.

INTEGER/SCALAR

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

---

1. The operand types are:

L-type	R-type
INTEGER	INTEGER
SCALAR	SCALAR

2. If the L- and R-types
    do not match, type
    conversion of the result
    of the R-expression takes
    place before assignment.

3. Scalar-to-integer conversion
    implies <u>rounding</u> of the value
    of the R-expression.

Examples:

    If I is an integer,
       S is a scalar, and
       M a 2x2 matrix, then

       I = 5;   results in I ≡ 5

       I = 7.7;   results in I ≡ 8

       S = 7.7;   results in S ≡ 7.7

    Given the last values above for S, I

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

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

            (? indicates values unchanged by assignment)

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

NOTE ON PRECISION CONVERSION

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

---

1. The R-expression is converted to the
   precision of the receiving data item
   as necessary before assignment.

2. If type conversion from integer to
   single precision scalar is implied,
   it takes place before precision
   conversion.

8.3 CHARACTER ASSIGNMENTS

The receiving data item is character type.

```
1.  The operand types are:

      L-type    |R-type

                |  ( CHARACTER
      CHARACTER |  { INTEGER
                |  ( SCALAR

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

Examples:

If C is a character with C ≡ 'ABCDE' and

C2 is a character,

then

$C2 = C_3;$ results in C2 ≡ 'C'

$C2 = 1573;$ results in C2 ≡ '1573'

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

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

Examples:

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

then

    |C1 = 'ABCDE';

results in C1 ≡ 'ABCDE' of working length 5

but

    |C2 = 'ABCDE';

results in C2 ≡ 'A' of working length 1

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

Let

$$STRING_\alpha$$

denote a receiving data item of
character type:

   N is declared maximum length
   n is working length <u>before</u> assignment

1. The range of the subscript expression α
   is presumed to be in the range 1 - N;
   otherwise an error results.

2. The length of the R-expression is adjusted
   to the length implied by α, either by
   truncation of the excess from the right,
   or by padding on the right with blanks.

3. If the range of α lies inside the range
   1-n, then simple substitution of the char-
   acter positions implied takes place.

4. If the range of α lies partly beyond the
   range 1 - n, then the working length of
   STRING is increased appropriately.

5. If the range of α lies totally beyond the
   range 1 - n, the working length of STRING
   is increased appropriately, and the gap
   between the nth character and the first
   position implied by α (if any) is filled
   with blanks.

Examples:

Let Cl be character of declared maximum length 10
    with value $Cl \equiv$ 'ABCD'

Then by Rules 2 and 3:

$$Cl_{2 \text{ TO } 3} = \text{'QQ'};$$

results in $Cl \equiv$ 'AQQD'

$$Cl_{2 \text{ TO } 3} = \text{'1234'};$$

results in $Cl \equiv$ 'A12D'

$$Cl_{2 \text{ TO } 3} = \text{'X'};$$

results in $Cl \equiv$ 'AX D'

By Rules 2 and 4:

$$Cl_{4 \text{ TO } 5} = \text{'QQ'};$$

results in $Cl \equiv$ 'ABCQQ'
                (working length increased by 1)

$$Cl_{4 \text{ TO } 5} = \text{'X'};$$

results in $Cl \equiv$ 'ABCX '
                (working length increased by 1)

By Rules 2 and 5:

$$Cl_{5 \text{ TO } 6} = \text{'QQ'};$$

results in $Cl \equiv$ 'ABCDQQ'
                (working length increased by 2)

$$Cl_{7 \text{ TO } 9} = \text{'FGH'};$$

results in $Cl \equiv$ 'ABCD FGH'

$$Cl_6 = \text{'FGH'};$$

results in $Cl \equiv$ 'ABCD F'

8.4 BOOLEAN ASSIGNMENTS

The receiving data item is of a Boolean type.

1.	The operand types are:	

L-type	R-type
BOOLEAN	BOOLEAN

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

Example:

    If B is Boolean, then

        B = FALSE;

            results in B ≡ FALSE

8.5   MULTIPLE ASSIGNMENTS

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

Symbolic form:

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

1. The value of the R-expression
    is assigned to all $L^1 \ldots L^n$
    in turn.

2. Any L-type must be compatible with
    the R-type according to the rules
    stated in Sections 8.2 through 8.4.

3. No particular <u>order</u> of assignment is
    guaranteed.

Examples:

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

       M1, V1 = 0;

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

    If C is a character,
       I is an integer,

       C, I = 127.2;

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

With the above data items,

$$M1, \; C = 5;$$

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

The following example illustrates the importance of Rule 3:

If further $I \equiv 2$, then

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

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

If I is assigned before $V1_I$,

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

(? indicates values unchanged by assignments)

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

8.6 SUMMARY

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

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

9. CONDITIONAL STATEMENTS AND BRANCHES

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

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

9.1 THE CONDITIONAL STATEMENT

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

9-1

SIMPLE IF STATEMENT

The form of the simple version is:

```
    IF  exp   THEN  statement   ;
```

1. *exp*   is an expression which is
    evaluable as either TRUE or
    FALSE. It may be either a
    BOOLEAN expression or a rela-
    tional expression (these are
    described in Section 9.2).

2. *statement*   constitutes the true
    part of the conditional statement.
    Except as noted in Rule 3 it may
    be any executable statement,
    either simple or compound.

3. *statement*   may not possess a
    label, and may not be another
    conditional statement.

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

Examples:

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

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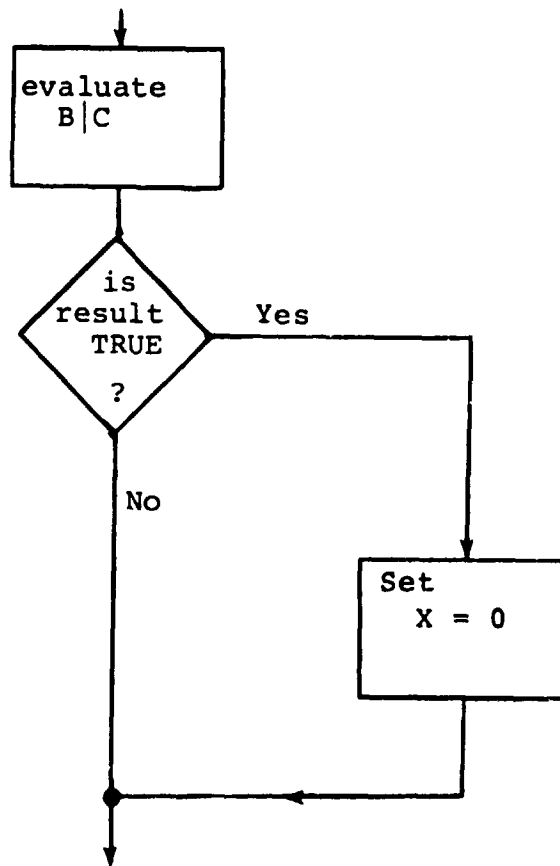

```
| IF B|C THEN DO;
|     X = X - 1;
|     Y = Y + 1;
| END;
```

   The true part is a compound statement containing
   two assignments.

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

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

9-3

AUGMENTED IF STATEMENT

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

> IF *exp* THEN *statement* ;
> ELSE *else statement* ;

1. The form of the IF clause and
   true part are the same as in
   the simple conditional state-
   ment.

2. *else statement* constitutes the
   false part of the conditional
   statement. It may be any
   <u>unlabelled</u> executable state-
   ment either simple or compound.

3. If *exp* is FALSE, execution
   proceeds to the next statement
   via *else statement* . If TRUE, it
   proceeds to the next statement
   via *statement* .

Examples:

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

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

```
                 ┌─────────────┐
                 │  evaluate   │
                 │    B|C      │
                 └─────────────┘
                        │
                       ╱ ╲
                     ╱  is  ╲
            No     ╱ result   ╲    Yes
            ──────╱   TRUE      ╲──────
                  ╲             ╱
                   ╲    ?      ╱
                     ╲       ╱
                       ╲   ╱
                        ╲ ╱
       │                               │
 ┌───────────┐                   ┌───────────┐
 │   Set     │                   │   Set     │
 │    X = 1  │                   │    X = 0  │
 └───────────┘                   └───────────┘
       │                               │
       └───────────────●───────────────┘
                       │
                       ▼
```
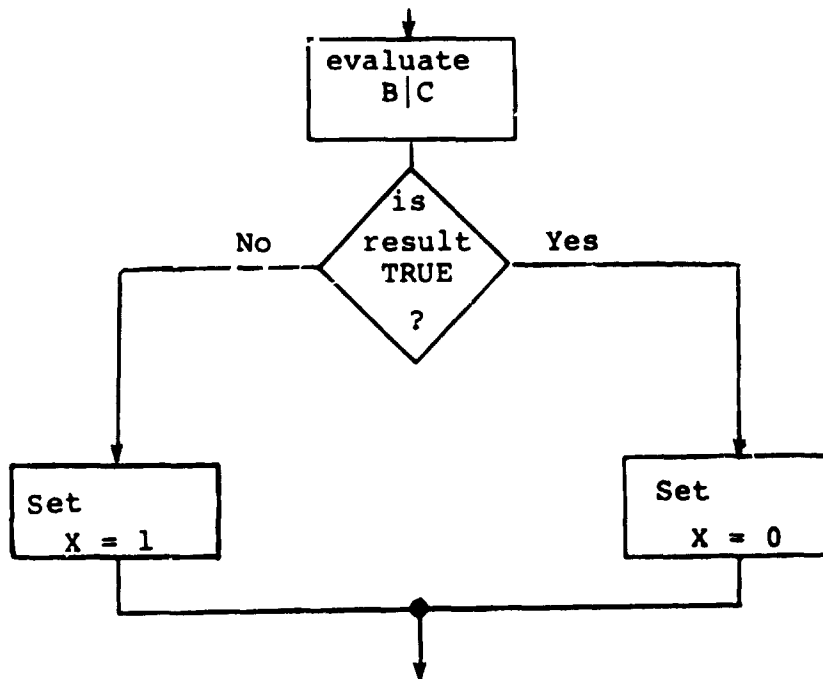
```
If B|C THEN DO;
    X = 1;
    Y = 2;
END;
ELSE DO;
    X = 2;
    Y = 1;
END;
```

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

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

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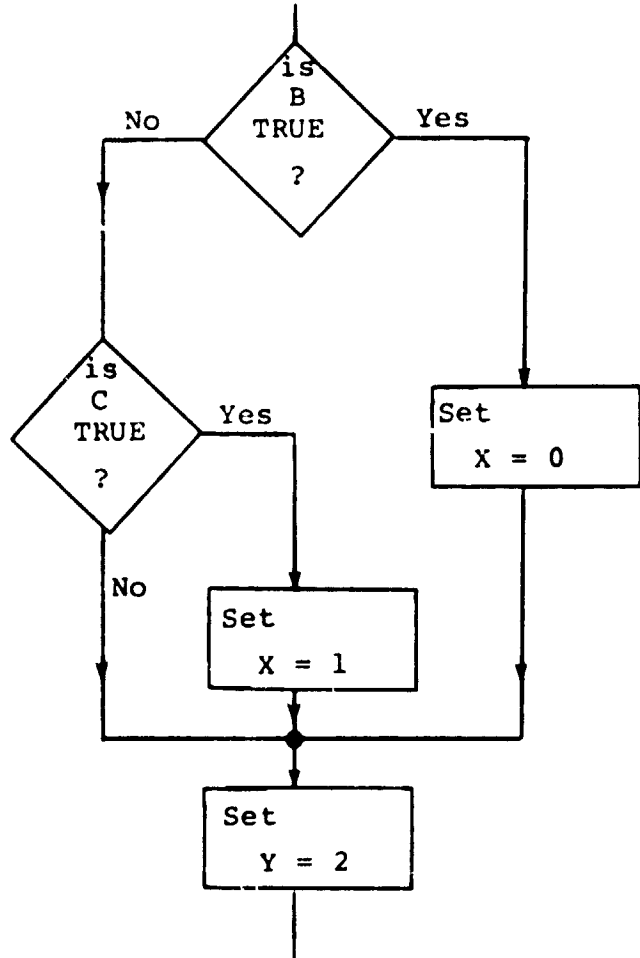
9.2 RELATICNAL EXPRESSIONS

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

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

COMPARATIVE OPERATIONS

HAL/S recognizes the following comparative operators:

Symbol	Purpose	Class
>	greater than	
<	less than	
<=	less than or equals	
NOT > ¬ >	} not greater than	I
> =	greater than or equals	
NOT < ¬ <	} not less than	
=	equals	II
NOT = ¬ =	} not equals	

9-7

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

● CLASS II ONLY

Symbolic form: L NOT $=$ R

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

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

2. Comparison of vector and matrix
    operands implies element-by-element
    comparison.

3. The operands in a vector comparison
    must be the same length.

4. The operands in a matrix comparison
    must have the same row and column
    dimensions.

Examples:

If STRING is character type with

$$STRING \equiv \text{'ABC '}$$

$$STRING = \text{'PQR'}$$

is FALSE.

$$STRING = \text{'ABC'}$$

is FALSE - character strings must be of the same length.

If V, V1 are 3-vectors with

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

then   V = V1 is FALSE,

$$V1 - V = 2 V \text{ is TRUE.}$$

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

then   V1 = V2 is illegal because of length mismatch,

but $V1_{1 \text{ TO } 2}$ = V2 is TRUE.

● CLASS I AND CLASS II

```
                                    >
                                    <
                                    >=
                                    <=
                                    NOT >
        Symbolic form:   L  ¬>  R
                                    NOT <
                                    ¬ <
                                    =
                                    NOT =
                                    ¬ =
```

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

L-type	R-type
INTEGER ⎱⎰	⎰⎱ INTEGER
SCALAR ⎰⎱	⎱⎰ SCALAR

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

Examples:

    If I is an integer with I ≡ 5

        then   I = 5 is TRUE

                I < 4 is FALSE

                I >= 5 is TRUE

NOTE ON PRECISION CONVERSION

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

---

1. Where an operation specifies type conversion from integer to single precision scalar, this conversion is carried out first.

2. If only one operand is integer and no type conversion is implied, no precision conversio.. takes place.

3. If both operands have the same precision, the result is of the same precision (even if not of the same type).

4. If the operands have mixed precision, the single precision operand is converted to double precision. Then rule 3 is applied.

COMBINING COMPARATIVE OPERATIONS

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

- Boolean expressions may contain no comparative operations.

- Relational expressions may contain no Boolean operands.

Examples:

If V1, V2 are 3-vectors with

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

and C is character with C $\equiv$ 'ABC'

then

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

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

If B is Boolean then

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

9-12

PRECEDENCE

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

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

* Any operand of this operator <u>must always</u> be parenthesized, and is evaluated immediately after evaluation of the operator itself.

Example:

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

$$\text{IF} \quad \text{S1} + \text{S2} = 0 \,|\, \neg \, (\text{S3} > 0) \, \& \, \neg \, (\text{S4} < 0 \,|\, \text{S5} > 0) \quad \text{THEN}$$

① ②⑩ ④ ③ ⑨⑧⑤ ⑦ ⑥
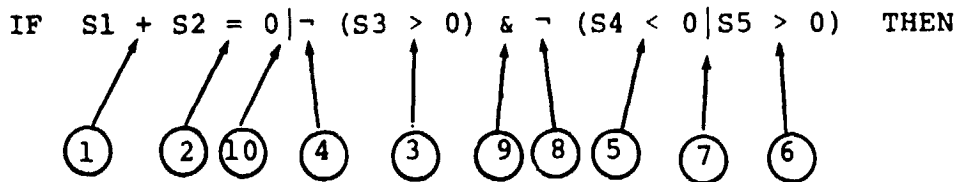
Section 9.2 ends with some more examples designed to clarify the foregoing.

Examples:

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

IF $V_1 = 1$ & $V_2 = 2$ THEN $V_3 = 0$;

S

IF $V_3 > 0$ | $V_2 < 0$ THEN $V = 0$;

S

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

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

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

$$V \equiv \begin{bmatrix} 1 \\ 2 \\ 0 \end{bmatrix} \qquad \text{as before.}$$

9.3   LABELS AND BRANCHES

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

LABELS

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

```
        |   label  :    statement   ;
        |

  1.    statement    is any executable
        statement or statement group
        (see Section 10), with two
        exceptions.

  2.     statement   may not be the
        "true part" or "false part" of
        a conditional statement.

  3.     label   is a user-defined
        identifier name (see Section
        2.2).
```

Examples:

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

The following are illegal since they violate
Rule 2:

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

However, the conditional statement itself _may_
be labelled:

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

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

Example:

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

GO TO STATEMENT

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

| GO TO *label* ;

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

Examples:

      | GO TO ONE;

      The GO TO statement itself may be labelled:

      | TWO: GO TO THREE;

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

ELIMINATING GO TO STATEMENTS

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

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

Examples:

```
        IF X > 1.5 THEN GO TO ALPHA;
        IF X < 1.5 THEN TO TO BETA;
        Y = Y + 1;
        GO TO GAMMA;
ALPHA:  X = X - 0.05;
        GO TO GAMMA;
BETA:   X = X + 0.05;
GAMMA:  .
        .
        .
        .
```

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

9-18
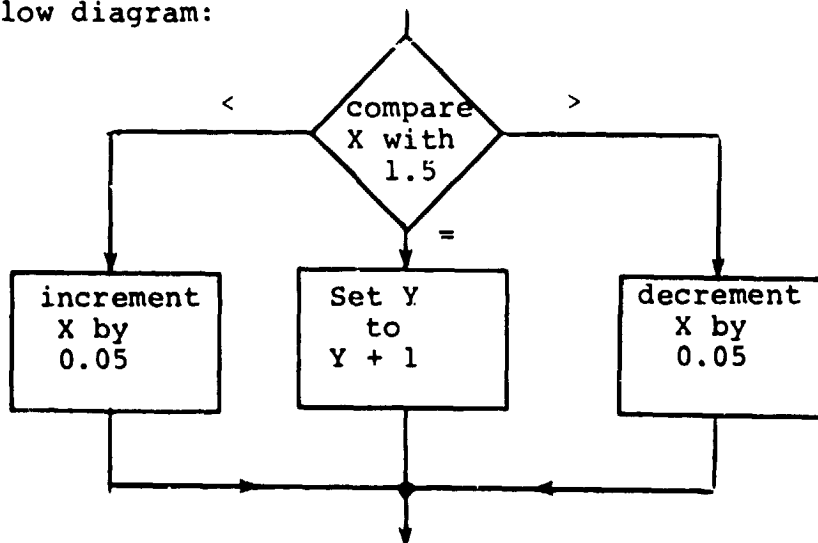
The same algorithm is more clearly programmed
as follows:

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

9.4   SUMMARY

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

10. STATEMENT GROUPS

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

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

- the sequence may be repetitively executed until some condition is satisfied;

- a single executable statement (or nest statement group) of the group, selectable at execution time, may be executed.

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

10.1 DELIMITING STATEMENT GROUPS

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

THE DO STATEMENT

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

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

1. *control* is a construct to be
    described. It specifies the manner
    in which the sequence of statements
    is to be executed.

2. *control* is optional. If it is
    absent, the sequence of statements
    is executed in its natural order*
    once only.

3. The DO statement is executable in
    that it may be labelled according
    to the Rules of Section 9.

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

---

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

THE END STATEMENT

Every statement group is closed with an END statement:

$$\text{END} \quad \textit{label} \quad ;$$

1. The END statement is executable in that it may be labelled according to the Rules of Section 9.

2. *label* is optional: if present, the opening DO statement of the group must be labelled with *label* .

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

Examples:

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

```
            DO WHILE I > 0;              opening DO statement
               I = I - 1;
               A  = 0;                   group of statements
     S         I
            END                          closing END statement


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

10-3

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

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

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

```
IF S ¬= 0 THEN GO TO NOSET;
I = 2;
C = 'RESET VALUE OF I TO '||I;
NOSET:
    •
    •
```

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

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

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

10-4

10.2 REPETITIVE EXECUTION OF STATEMENT GROUPS

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

● The DO WHILE statement, in which execution is
  repeated while a relational or boolean expression
  remains true in value;

● The DO FOR statement, in which the sequence is
  executed once for each of a set of assigned values
  of a "control variable".

THE DO WHILE STATEMENT

The form of the DO WHILE statement is:

DO WHILE *condition* ;

1. *condition* is any relational or
    BOOLEAN expression. It is
    evaluated prior to each cycle
    of execution of the statement
    sequence in the group.

2. The next cycle of execution of
    the group proceeds if the value
    of *condition* is TRUE.

3. If the value of *condition* is FALSE,
    the stopping condition is satis-
    fied. Execution proceeds to the
    statement following the END state-
    ment of the group.

10-5

Examples:

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

Here the group is executed 5 times, after which the value of I is -1. In flow diagram form, the sequence of events is:
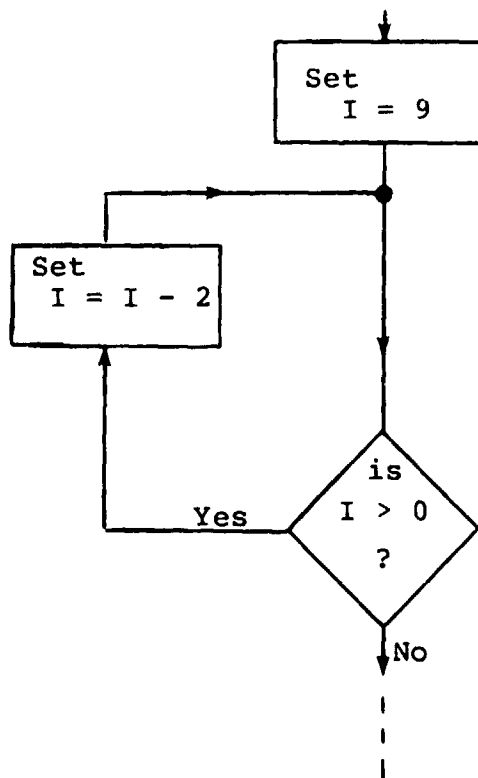
It is possible for a group never to be executed:

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

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

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

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

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

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

THE DO FOR STATEMENT

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

DO FOR var = *initial* TO *final* BY *increment*;

1. var is an unarrayed INTEGER or SCALAR
    data item (it may be subscripted if
    required). It is called the "control
    variable" of the DO FOR statement.

2. *initial*, *final* and *increment* are integer
    or scalar expressions:

    ● *initial* is the **initia**l value
      assigned to var.

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

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

    All three expressions are evaluated
    <u>once</u> prior to the first cycle of
    execution.

3. The stopping condition is met when
    the value of var lies outside the
    range bounded by *initial* and *final*.

4. *increment* may be either positive or
    <u>negative</u>   The phrase

              BY *increment*

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

10-8

Examples:

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

Here the group is executed 10 times. I is initially 1, and increments each time until 10 is reached. At the end of execution of the group, the value of I is 11. In flow diagram form, the sequence of events is:
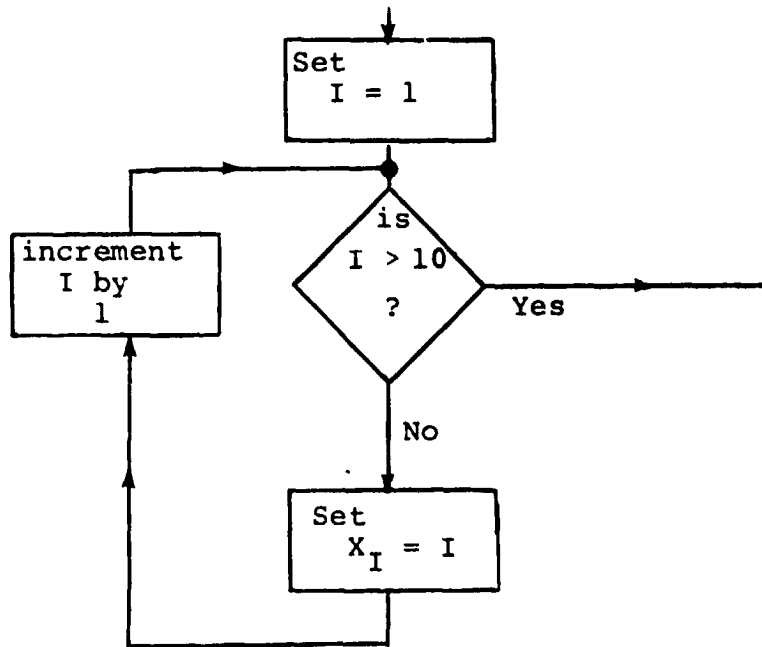
```
I = 7;
DO FOR I = I + 5 TO I - 3 BY -2;
    X = X + I;
END;
```

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

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

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

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

DO FOR var = exp^1, exp^2, ... exp^n;

1. var is the control variable as before.

2. Each exp is an integer or scalar expression. Values of the exp's are assigned to var in turn prior to the execution of each cycle, on a left-to-right basis.

3. Each exp is <u>evaluated</u> immediately prior to the cycle of execution in which it will be used.

Examples:

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

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

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

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

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

10-11

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

    12, 15, 16, 15, 12

as opposed to

    12, 10, 8, 6, 4

in the old form.

Rounding also occurs if the
control variable is integer
and any of the control expres-
sions are scalar.

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

10-12

10.3 SELECTIVE EXECUTION OF STATEMENT GROUPS

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

DO CASE exp ;

1. exp is an integer or scalar expression.

2. If its value is k (after rounding if necessary), then the kth statement of the group is selected for execution.

3. A run time error results if k < 0 or k is greater than the number of statements in the group.

The flexibility of a DO CASE statement is understood when it is realized that the selected statement may be a compound statement (i.e. it may itself be a statement group).

Example:

```
I = 3;
DO CASE I;
    X = 4;              case 1
    X = 3;              case 2
    DO;          )
        X = 7;   |
        Y = 3;   |      case 3
    END;         )
    X = 1;              case 4
    X = 0;              case 5
END;
```

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

. $X \equiv 7$, $Y \equiv 3$

An ELSE clause may be added
to the DO CASE statement which
is executed instead of an
error being signalled, if the
value of the case variable is
outside the legal range for the
statement group.
See: (tbd).

10-14

10.4 BRANCHING IN STATEMENT GROUPS

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

● The EXIT statement is, in effect, a controlled branch
  <u>out</u> of a statement group.

● The REPEAT statement only applies to statement groups
  executed repetitively, and is a controlled branch back
  to the <u>beginning</u> of the group.

THE EXIT STATEMENT

The simplest form of the EXIT statement is:

```
        EXIT;

1.  Its execution causes an immediate
    branch out of the innermost state-
    ment group in which it is enclosed.

2.  Execution is directed to the first
    statement following the END of the
    group branched out of.
```
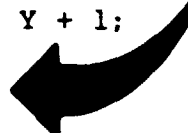
Examples:

```
DO:
   X = 1;
   Y = 2;
   IF Z = 3 THEN EXIT;
   Z = 4;
END;
X = X + 1;
```

Arrow **shows** branch in execution if Z = 3

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

Arrow shows branch in execuﾍion if Y = 3:   execution
branches to the end, but not out of DO WHILE group.
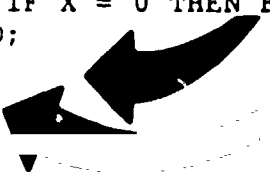
There exists a second form of the EXIT statement to allow branches
out of other than the innermost statement group:

10-16

```
┌─────────────────────────────────────────┐
│      EXIT   label   ;                     │
│                                           │
│ 1.  Its execution causes a branch out     │
│     of the enclosing stateme   group      │
│     whose DO statement possesses the      │
│     label   label  .                      │
│                                           │
│ 2.  Execution is directed to the first    │
│     statement after the END of the group  │
│     branched out of.                      │
└─────────────────────────────────────────┘
```

Example:

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

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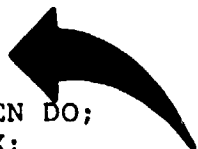
THE REPEAT STATEMENT

The simplest form of the REPEAT statement is:

```
          REPEAT;
```

1. It must be enclosed in a DO FOR
    or DO WHILE group.

2. Its execution causes an immediate
    branch to the beginning of the
    innermost enclosing DO FOR or
    DO WHILE group.

3. The next cycle of execution of
    the group then starts (unless
    of course the stopping condition
    is satisified).

Examples:

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

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

10-18

```
  X = 4;
  DO WHILE X > 1;
      X = X - 1:
      IF X = 1 THEN REPEAT;
      Y = X;
S     X
  END;
```

When X ≡ 2 the REPEAT branch is executed:
a new cycle of execution does not begin
however because the initial test shows that
the stopping condition is satisfied.

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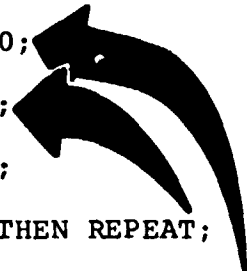
REPEAT   *label* ;

1. Its execution causes an immediate
    branch to the beginning of the
    enclosing DO FOR or DO WHILE
    group whose DO statement possesses
    the label *label* .

2. The next cycle of execution of
    the group then starts (unless the
    stopping condition is satisfied).

Example:

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

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

10.5 SUMMARY

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

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

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

11. PROCEDURES AND FUNCTIONS

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

invocation → execution → return to caller

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

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

11.1 INTRODUCTION

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

- INPUT PARAMETERS - by which values may be
  passed into a procedure only.

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

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

11-1

RELATIVE POSITION OF BLOCK DEFINITIONS

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

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

A number of rules restrict
the kind of function which
may be invoked preceding
its block definition.
See: (TBD)

11.2   BLOCK DEFINITIONS

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

11-2

PROCEDURE OPENING

The statement opening a procedure block takes the form:

$label$: PROCEDURE$(i^1,i^2,...)$ ASSIGN$(a^1,a^2,...)$;

1. $label$ is any legal identifier
    name, and constitutes the name
    of the procedure.

2. i^1, i^2,... are legal identifier
    names defining input para-
    meters. If the entire paren-
    thesized list is omitted, then
    the procedure has no input
    parameters.

3. a^1, a^2... are legal identifier
    names defining assign parameters.
    If the entire parenthesized list
    and the keyword ASSIGN are omitted,
    then the procedure has no assign
    parameters.

11-3

FUNCTION OPENING

The statement opening a function block takes the form:

$label$: FUNCTION$(i^1, i^2, \ldots)$ $attributes$;

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

2. $i^1, i^2, \ldots$ are legal identifier names defining input parameters. If the entire parenthesized list is omitted, then the function has no input parameters.

3. *attributes* defines the type of the function, and, where applicable, its precision. The form of *attributes* is the same as used in data declarations (see Section 4.2). If no *attributes* are supplied, the function is assumed to be single precision scalar.
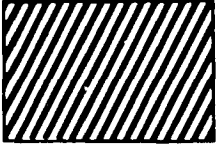
BLOCK CLOSING

Both procedure and function blocks are closed with the statement:
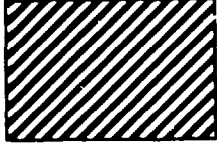
CLOSE *label*;

1. The identifier *label* is optional.

2. If supplied, it must be the name of the procedure or function block.

11-4
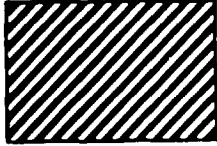
Examples:
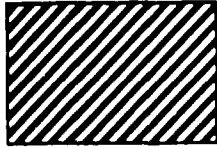
ONE: PROCEDURE;

 ⟩———— procedure body

CLOSE ONE;

TWO: PROCEDURE ASSIGN(ARG1);

↑
└── single assign parameter –
    may be used to return values
    from procedure

CLOSE;

THREE: FUNCTION MATRIX(4,4) DOUBLE;

CLOSE THREE;

FOUR: FUNCTION(ARG1,ARG2) BOOLEAN;

        ‿‿‿‿
         │
         └── two input parameters –
             for passing values into
             function only

CLOSE;

11-5

11.3 DECLARATION OF PARAMETERS AND LOCAL DATA

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

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

Examples:

    General positioning -

        ONE: PROCEDURE(ARG1) ASSIGN(ARG2);

                                parameter declarations

                                local data declarations

                                executable statements

        CLOSE ONE;

---
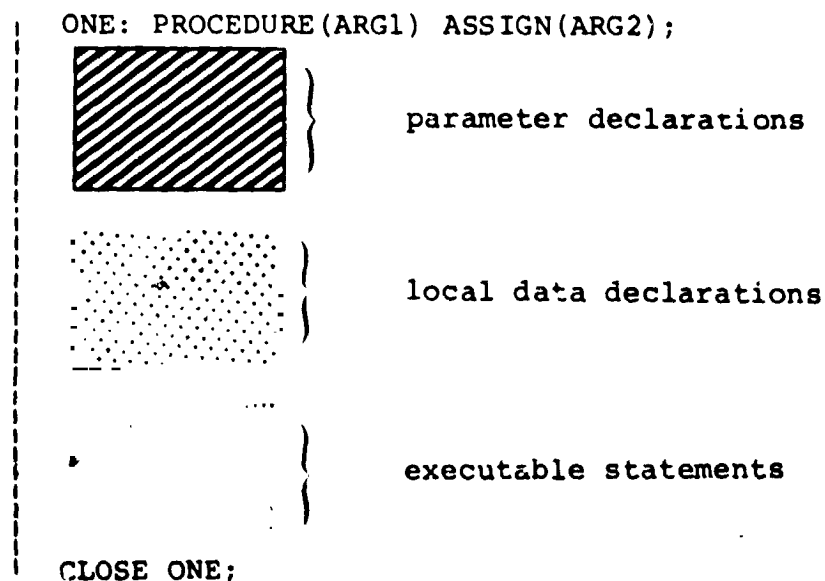
* See the User's Manual for any given implementation.

Particular instance -

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

11.4   FUNCTION INVOCATIONS

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

$label(i^1, i^2, \ldots)$

1. $label$ is the defined name of the
    function.

2. $i^1, i^2, \ldots$ is a list of arguments,
    which must correspond in number
    with the parameters of the function
    invoked. Each argument is a HAL/S
    expression.

3. If the function has no parameters,
    then the entire parenthesized argu-
    ment list must be absent.

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

MATRIX PARAMETER

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

VECTOR PARAMETER

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

11-8

INTEGER/SCALAR PARAMETER

1. The following table gives the legal argument types:

parameter	argument
INTEGER SCALAR	INTEGER SCALAR

2. Conversion of the argument takes place where necessary. Scalar-to-integer conversion implies rounding of the value of the expression.

3. Precision conversion takes place when necessary and is applied after possible type conversion.

CHARACTER PARAMETER

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

parameter	argument
CHARACTER	CHARACTER INTEGER SCALAR

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

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

BOOLEAN PARAMETER

> 1. The corresponding argument must
>   be of Boolean type.

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

Examples:

Suppose the following functions are defined:

```
ONE: FUNCTION INTEGER;
```
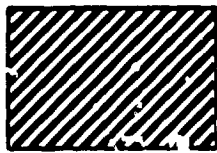


```
CLOSE;
```

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



```
CLOSE;
```

11-10
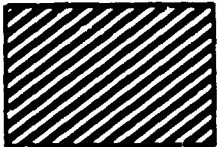
Let also the following data be declared:

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

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

```
S = S + ONE;
```

$S = S + M_{1, ONE};$ Note: subscripts may be integer expressions of any kind.
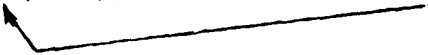
$M2 = TWO(M2,S) + M2;$ M2 is converted to single precision during transmission.

$M2 = TWO(M2,I);$ I is converted to scalar type during transmission.

11-11

The following are illegal invocations:

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


M2 = TWO(M1,'ARGUMENT'||I);
                                          transmission of character
                                          type argument to scalar
                                          parameter B incurs  an
                                          illegal type conversion.
```

Arguments may possess array-
ness. The effects of this
depend on whether or not
the corresponding parameter
is declared to be an array.
See: (TBD).

11.5 PROCEDURE INVOCATIONS

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

CALL *label* $(i^1, i^2, \ldots)$ ASSIGN $(a^1, a^2, \ldots)$;

1. *label* is the defined name of the procedure.

2. $i^1, i^2, \ldots$ is a list of input arguments which must correspond in number with the input parameters of the procedure invoked. Each input argument is a HAL/S expression.

3. If the procedure has no input parameters, then the entire parenthesized argument list must be absent.

4. $a^1, a^2, \ldots$ is a list of assign arguments which must correspond in number with the assign parameters of the procedure invoked. Each argument must be a HAL/S data item.

5. If the procedure has no assign parameters, then the entire parenthesized list of assign arguments, and the ASSIGN keyword, must be absent.

11-13

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

The transmission of the assign argument list follows stricter rules since values are passed both into and out of a procedure by this mechanism.
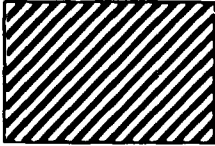
ASSIGN ARGUMENTS
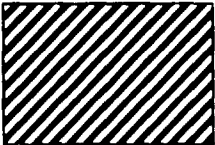
1. An assign argument must be a declared HAL/S data item.

2. An assign argument must match the corresponding assign parameter in type and precision.

3. A matrix or vector argument must match the corresponding parameter in dimension.

4. Only matrix and vector arguments may be subscripted. Such subscripting must reduce the argument to scalar type by specifying one element only.

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

Examples:

Suppose the following procedures are defined:

```
ONE: PROCEDURE;
```



```
CLOSE;
```

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



```
CLOSE;
```

Let also the following data be declared:

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

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

```
CALL ONE;
CALL ONE(I);  ◄──────────────  illegal: ONE possesses no
                                            parameters.

CALL TWO(M2ᵀ,S+1) ASSIGN(I);
                            └──────  values may be passed in
                                      and out of TWO through I.

                      └──────────  type conversion required here.

             └──────────────  precision conversion required
                               here.

CALL TWO(M3, ID) ASSIGN(S);
                          └── type conversion illegal for
                              assign arguments.

             └──────  precision conversion required.

      └──────────  dimension mismatch: parameter is
                   a 3 x 3 matrix.

CALL TWO(M1,I) ASSIGN(I);

                 └──────  appearance in both places
                          is legal.
```

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

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

11-17

11.6 RETURNS FROM PROCEDURES AND FUNCTIONS

When execution reaches the CLOSE statement of a procedure
block, an automatic return to caller takes place. How-
ever, if execution reaches the CLOSE statement of a
function block, a run time error results since the
function has no value to return to the caller. Hence
a function block needs an explicit RETURN statement
to cause the return to take place. In addition, if
returns are required from parts of the code in a
procedure block other than at the CLOSE, an explicit
RETURN statement is required.

PROCEDURE RETURN

The RETURN statement of a procedure takes the form:

```
                  |
                  |   RETURN;
                  |
```

Example:

```
| CHOICE: PROCEDURE(FLAG) ASSIGN(DIR);
|    DECLARE FLAG BOOLEAN;
|    DECLARE DIR VECTOR(3);
|    IF FLAG THEN RETURN;
|    DIR = UNIT(DIR);
| CLOSE;
|
```

If FLAG ≡ TRUE then procedure merely returns execution
at RETURN. If FLAG ≡ FALSE then 3-vector DIR is
normalized, and procedure returns execution at CLOSE.

11-18

FUNCTION RETURN

The RETURN statement of a function takes the form:

```
    |   RETURN exp ;
    |
1.  The resultant value of the
    expression exp is returned when
    the function returns to its
    caller.
```

The return of an expression by a function is similar
in nature to the transmission of an input argument of
a function to the corresponding parameter, the
function itself playing the role of parameter. During
return, type and precision conversions are possible,
and dimension matching must be ensured. The relevant
rules are the same as those described for argument
transmission in Section 11.4.

Examples:

```
FUNC1: FUNCTION(A) SCALAR;
 DECLARE A MATRIX(3,3) DOUBLE;
 DECLARE I INTEGER;
    .
    .
    .

 RETURN I+5; ◄──────────── type conversion to scalar
    .                       required.
    .
    .

 RETURN A   ; ◄─────────── conversion to single
         1,1               precision required.
    .
    .
    .

 RETURN 'I='||I; ◄──────── illegal type conversion
    .                       required.
    .
CLOSE;
```

11-19

11.7   SUMMARY

This section has explained the role of HAL/S procedures
and functions: how they are defined; how they are invoked;
and how execution is returned from them to the caller.
The mechanism of argument passage has been described in
detail.

The next section introduces the concepts of sequential
I/O in HAL/S, and describes statements for performing
input/output operations.

12.   INPUT/OUTPUT STATEMENTS

Higher order languages possess I/O statements to provide
programs with a means of communicating with their environ-
ment.   In HAL/S, simple forms of I/O statement provide
for the sequential input or output of data, including the
generation of printed listings.

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

12.1   HAL/S INPUT/OUTPUT CONCEPTS

The form of sequential I/O statements in HAL/S is based
on a specific conceptualization of the input-output process.
In this conceptualization, I/O takes place through a number
of "channels", each identified by an integer code. Each
channel is connected to an "I/O device", of which there
are two kinds, "unpaged", and "paged".

UNPAGED DEVICES

An "unpaged I/O device" can be used for both input and
output. It can be visualized as consisting of a "device
mechanism" which performs I/O on a continuous strip, across
which data is written. The data is organized in "columns"
across the strip, and in "lines" down it:

first
column — columns of data

first
line

lines of data
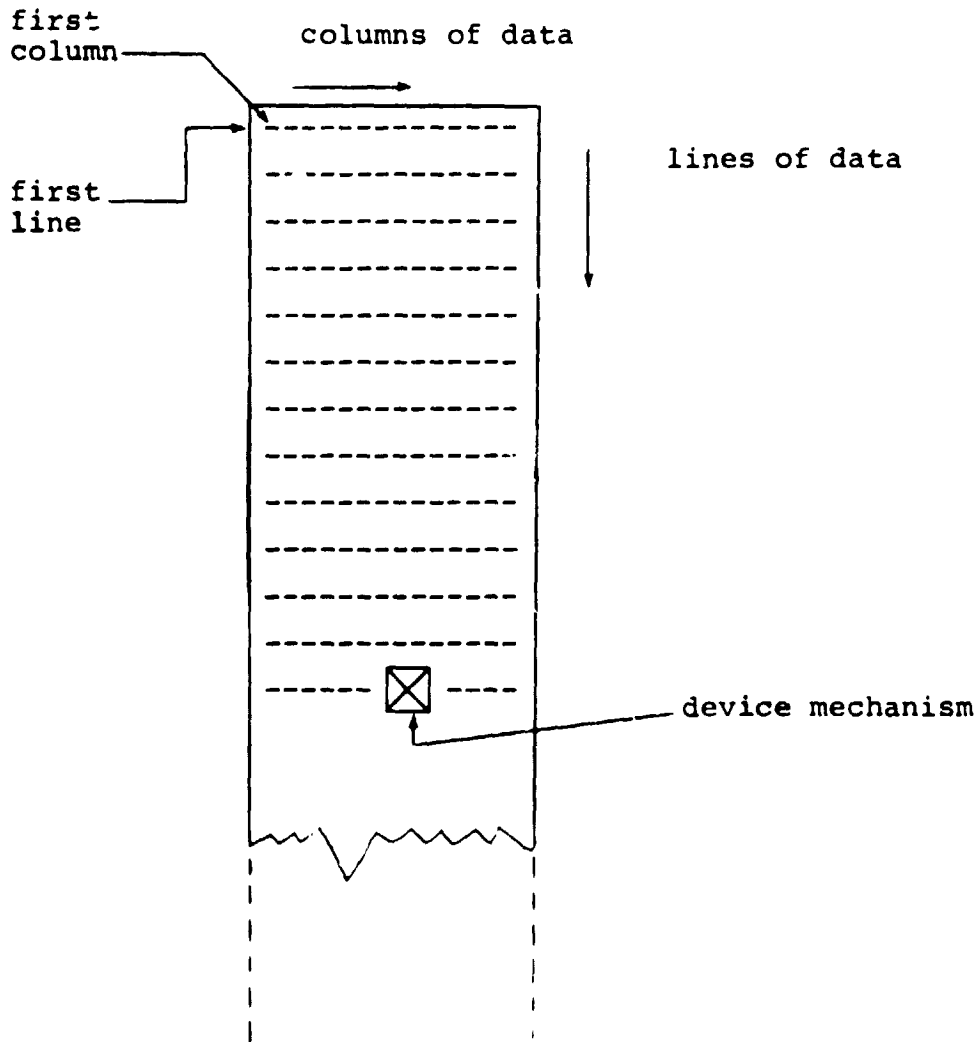
device mechanism

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

On output, the strip continually lengthens as new lines
are written on the device. On input, the strip is of
fixed length, and a run time error occurs if the device
mechanism is requested to read off the lower end.

Data output to an unpaged device is physically written
so that it may, on some future occasion, be read in again
via an unpaged device.

PAGED DEVICES

A "paged I/O device" can only be used for output. It can
be visualized in much the same way as an unpaged device,
except that the lines of data are organized into "pages":
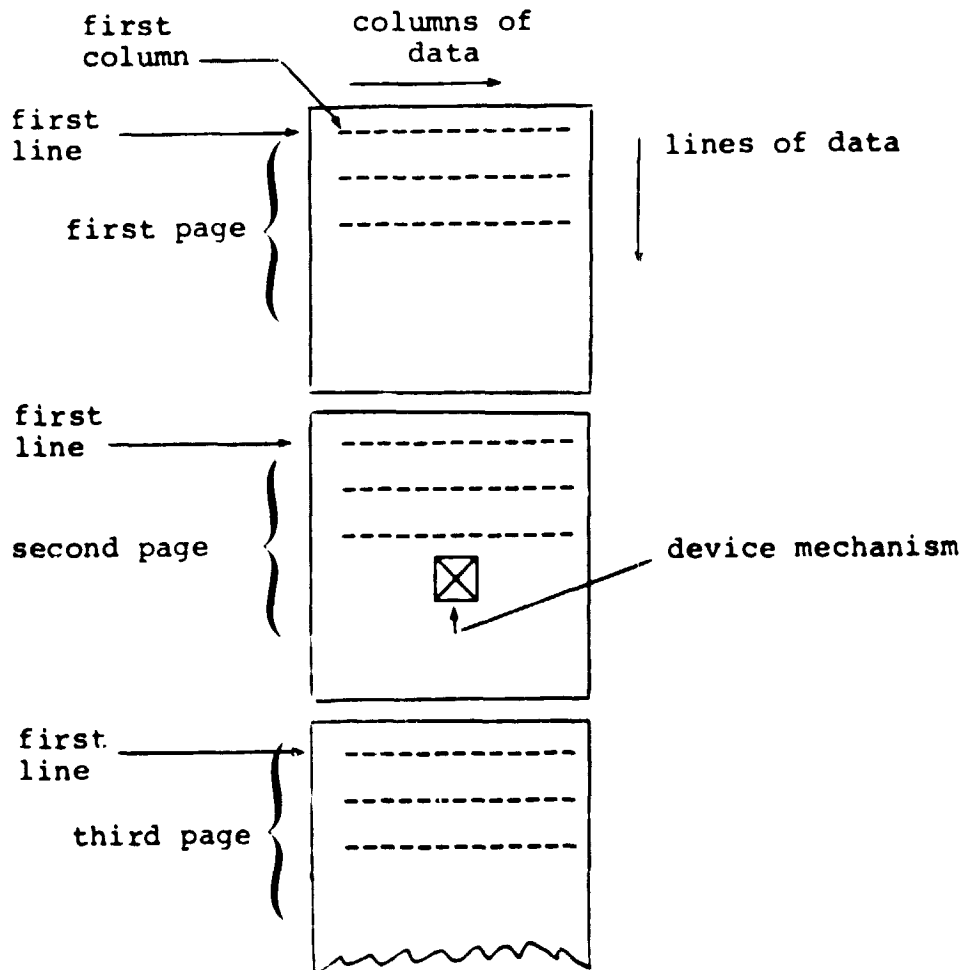
The paged device is designed to generate printed listings.
The form in which data is physically written on the device
is different from that on an unpaged device. Such data
cannot normally be read back again via an unpaged device.

DATA STORAGE

Data is conceived as being "stored" on a device, even
though in physical reality the device may be a line printer,
the data becoming inaccessible to the computer.

In HAL/S, data is written on the I/O device in "fields" which
can be separated by blank columns, or by a separator character.
The I/O process is stream-oriented: within the confines of
a single I/O statement, the column and line alignment of data
fields need be of no consequence. Data fields may even be
broken over line or page boundaries.

12.2   THE WRITE STATEMENT

The WRITE statement is an executable statement for the
output of data to a paged or unpaged I/O device. The form
of the WRITE statement is as follows:

WRITE(n) exp^1, exp^2, ... exp^n ;

1. n is the channel code number, and
    lies in the range $0 \leqslant n \leqslant 9$*.

2. Each exp is a HAL/S expression whose
    value or values are to be written on the
    device. The list of expressions may be
    arbitrarily long. Alternatively, none
    need be supplied.

3. Each expression in turn from left to
    right is evaluated, and its value (or
    values) written on the specified device.

---

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

In execution, the sequence of events is as follows:

● If the WRITE statement is the first to be executed for the specified device, the device mechanism positions itself at column 1 of line 1 (on page 1 if the device is paged). Otherwise, the device mechanism moves down one line from its current position, and repositions itself at column 1.

● Data fields are written from left to right along the line, each field being separated from the next by 5 blanks*.

● When the end of a line is reached, the device mechanism moves to column 1 of the next line and continues writing data fields. Unless the data field is of character type, the device does not attempt to break it over a line boundary if there is not room for it at the end of a line. Instead, it begins writing it on the next line.

● After finishing execution, the device mechanism is left positioned one column to the right of the end of th last data field written. Alternatively, if the dar field abuts the end of a line, it is positioned at column 1 of the next line.

● If no expressions are supplied in the WRITE statement, the device merely performs its initial positioning.

---

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

12-5

DATA FORMATS

The format of a data field depends on the type of expression whose resultant value is being written on the device, and on whether or not the device is paged. The formats are, in general, implementation dependent. Typical formats are shown in Appendix _.

Uni-valued expressions each give rise to a single data field. Multi-valued expressions each give rise to a series of data fields, which are written on the device sequentially in the following way:

● a ℓ-vector expression yields ℓ scalar data fields, one for each element. The data fields are laid out along a line, separated from each other by the standard number of blanks, and overflowing onto succeeding lines as required.

● an m x n matrix expression yields mn scalar data fields, one for each element. The matrix is laid out row by row. Each row is written as if it were an n-vector. The first element of the second and subsequent rows begin a new line, vertically aligned under the first element of the first row.

● arrays are written array element by array element, completing the requirements for one element before going on to the next. The last data field of one array element is separated from the first data field of the next element by the standard number of blanks, or starting a new line if required,

Examples:

et: M be a 3x3 matrix with $M \equiv \begin{bmatrix} 0.5 & 1.5 & 0.0 \\ 2.5 & 1.0 & 1.0 \\ 0.5 & 0.1 & 10.1 \end{bmatrix}$

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

   C be a character with $C \equiv$ 'VALUE'

   B be a Boolean with $B \equiv$ TRUE

then

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

   would result in output of the following form:

paged output: [132 columns/line]
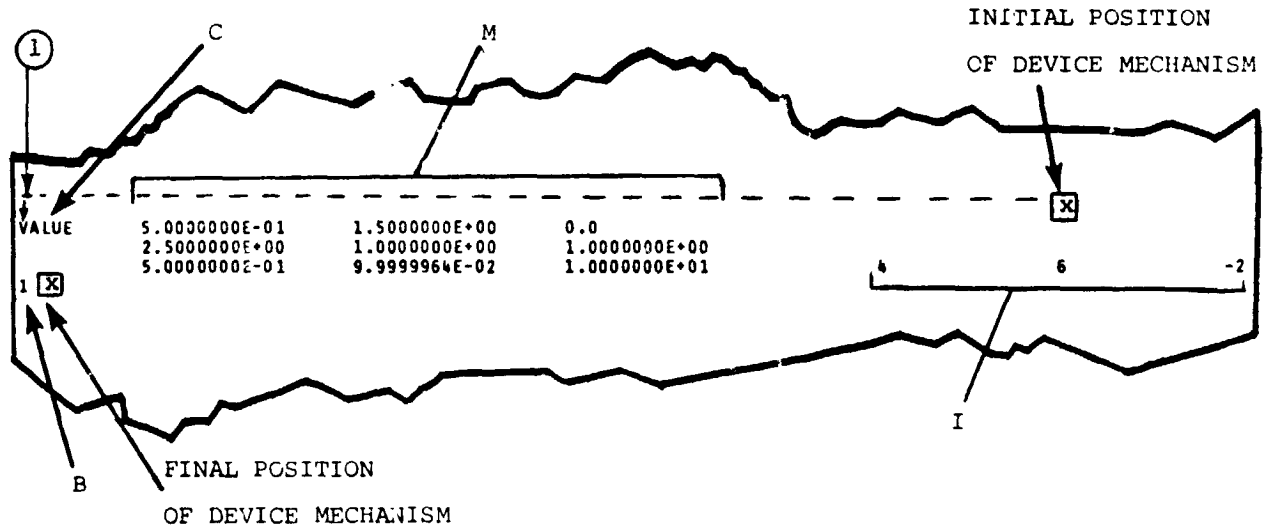
unpaged output: [80 columns/line]
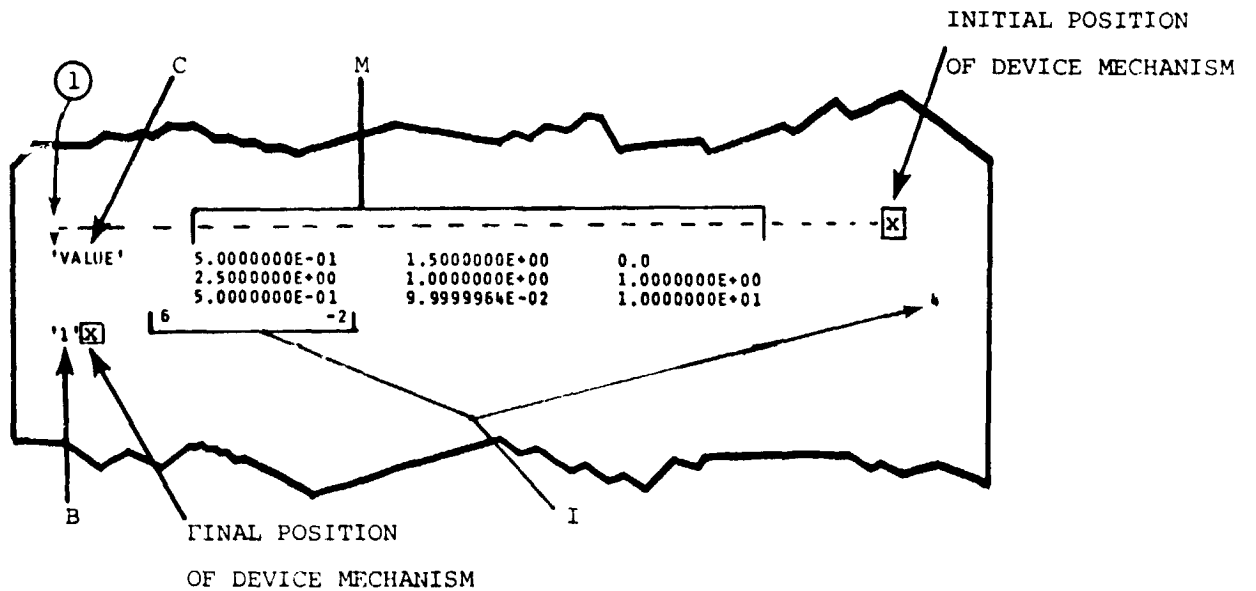
NOTES:

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

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

12-7

12.3   THE READ STATEMENT

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

READ(n) var^1 , var^2 , ... var^n ;

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

2. Each var is any type of data item,
    either subscripted or unsubscripted.
    The list of items may be arbitrarily
    long. Alternatively, none need be
    supplied.

3. The specified device reads values
    into each data item in turn from
    left to right.

In execution, the sequence of events is as follows:

● If the READ statement is the first to be executed
   for the specified device, the device mechanism positions
   itself at column 1 of line 1. Otherwise, the device
   mechanism moves down one line from its current position
   and repositions itself at column 1.

● Data fields are read from left to right along the line.
   The device expects each data field to be separated from
   the next by a comma and/or at least one blank.

● When the end of a line is reached, the device mechanism
   moves to column 1 of the next line and continues reading.
   Data fields may be broken over the line boundary.

---

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

12-8

- After finishing execution, the device mechanism
  is left positioned one column to the right of the
  end of the last data field read in. Alternatively,
  if the data field abuts the end of a line, it is
  positioned at column 1 of the next line.

- If no list of data items is supplied in the READ
  statement, the device merely performs its initial
  positioning.

- If the device reads two consecutive separating
  commas, then the value of the data item which would
  have been changed by reading a data field between
  the commas, is instead left untouched.

DATA FORMATS

The formats of data fields expected by a device on input
depend on the type of data item being read into. The
formats are, in general, implementation dependent. Typical
formats are shown in Appendix _.

Uni-valued data items cause single data fields to be read.
Multi-valued data items cause a series of data fields to be
read sequentially.

- A vector data item causes one data field per vector
  element to be read.

- A matrix data items causes one data field per matrix
  element to be read. Values are read into the matrix
  row by row.

- Arrayed data items are read into array element by
  array element, completing the read requirements for
  each element before going on to the next.

Examples:

Let M be a 3x3 matrix with initial values given

by $M \equiv \begin{bmatrix} 0.5 & 1.5 & 0.0 \\ 2.5 & 1.0 & 1.0 \\ 0.5 & 0.1 & 10.0 \end{bmatrix}$

Let I ᵇᵉ a 3-array of integers,

    C be a character data item of maximum length 10,

    B be a Boolean.

Then

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

using the following data:
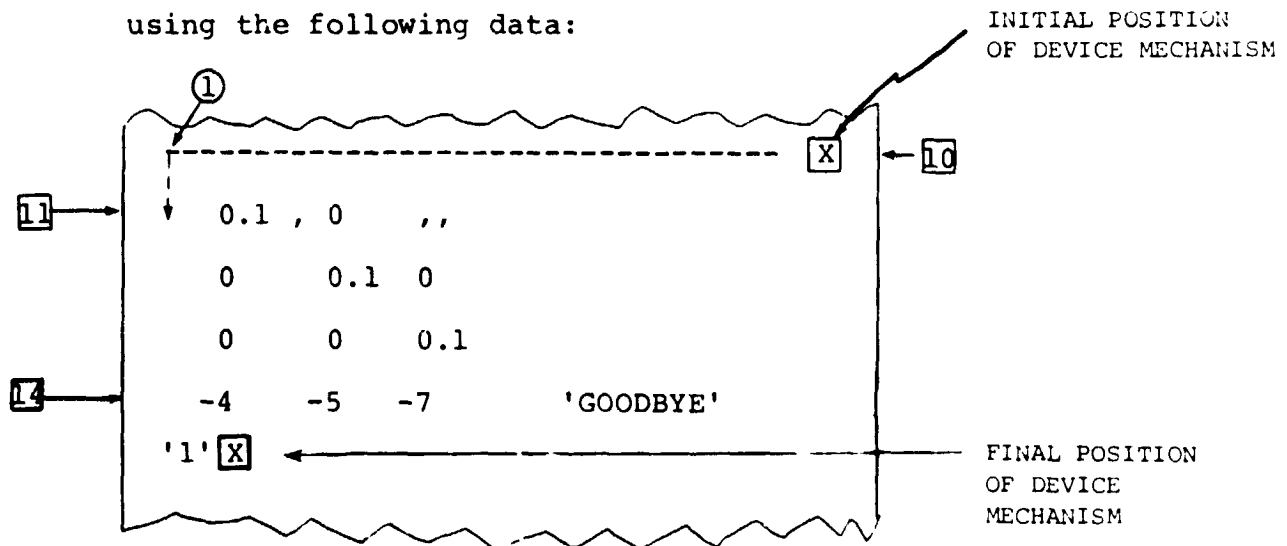
would result in:

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

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

$C \equiv$ 'GOODBYE'

$B \equiv$ TRUE

12-10

12.4 INPUT/OUTPUT FORMATTING

The formatting of I/O embraces two separate concepts:

● the shape of data fields;

● the position of data fields.

In terms of input, formatting implies that a device can be made to recognize different shapes of data fields in a variety of positions. In terms of output, formatting implies that a device can generate different shapes of data fields in a variety of positions.

Data field positioning is effected by direct movement of the device mechanism. Commands in the form of pseudo-functions can be inserted into READ and WRITE statements to cause repositioning of the mechanism.

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

There exists a second type of input statement called the READALL statement, which can be used to input arbitrary strings of characters. This can form the basis for arbitrary data field shape recogni-tion on input.
See: (tbd)

DEVICE MECHANISM POSITIONING

HAL/S possesses five pseudo-functions which can reposition a device mechanism during execution of a READ or WRITE statement. The pseudo-functions are placed in the READ or WRITE statement as if they were normal data items or expressions.

Three basic rules underlie the operation of the pseudo-functions in positioning device mechanisms:

● Horizontal and vertical positioning are separately and independently controlled.

● The operations of the pseudo-functions are independent of whether a device is being used for input or output.

● An explicit repositioning command taking effect at a particular point in execution overrides the default movement in the same direction (horizontal or vertical) which would otherwise be made by the device mechanism.

Particular instances of these rules are noted as the device positioning pseudo-functions are described below.

HORIZONTAL POSITIONING

The two pseudo-functions TAB and COLUMN serve to position a device mechanism horizontally on a line. Their form is as follows:

```
┌─────────────────────────────────────────────┐
│                  TAB(α)                      │
│                COLUMN(β)                     │
│                                              │
│   1.   α and β are integer expressions.      │
│                                              │
│   2.   TAB(α) moves the device mechanism     │
│        left or right by the number of        │
│        columns specified by α.  Negative     │
│        values of α denote movement to the    │
│        left; positive values, movement to    │
│        the right.                            │
│                                              │
│   3.   COLUMN(β) moves the device mechanism  │
│        left or right to the column indicated │
│        by β.                                 │
│                                              │
│   4.   Values of α or β must not be sucn as to│
│        try to move the device mechanism left │
│        past column 1, or right past the right-│
│        most column*.                         │
└─────────────────────────────────────────────┘
```

If a TAB or COLUMN pseudo-functio:. appears at the
beginning of a READ or WRITE statement, it overrides the
default positioning at column 1.

It does not of itself inhibit movement onto the next
line.

If a TAB or COLUMN appears between two expressions in
a WRITE statement, it overrides the standard data field
separation.

Successive TABs are cumulative in action.

---

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

Example:

If C1, C2, C3 are character data items

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

and if channel 6 is a paged device

then

    WRITE(6) TAB(-50),C1,COLUMN(5),C2,C3,TAB(2);

produces output of the following form:

INITIAL
POSITION OF
DEVICE MECHANISM

TAB LEFT 50
COLOUMNS, MOVE
DOWN 1 LINE
BY DEFAULT

SECOND   THIRD x FIRST

DEFAULT
5 BLANKS

MOVE TO
COLUMN 5

TAB RIGHT
2 COLUMNS

FINAL POSITION
OF DEVICE MECHANISM

12-14
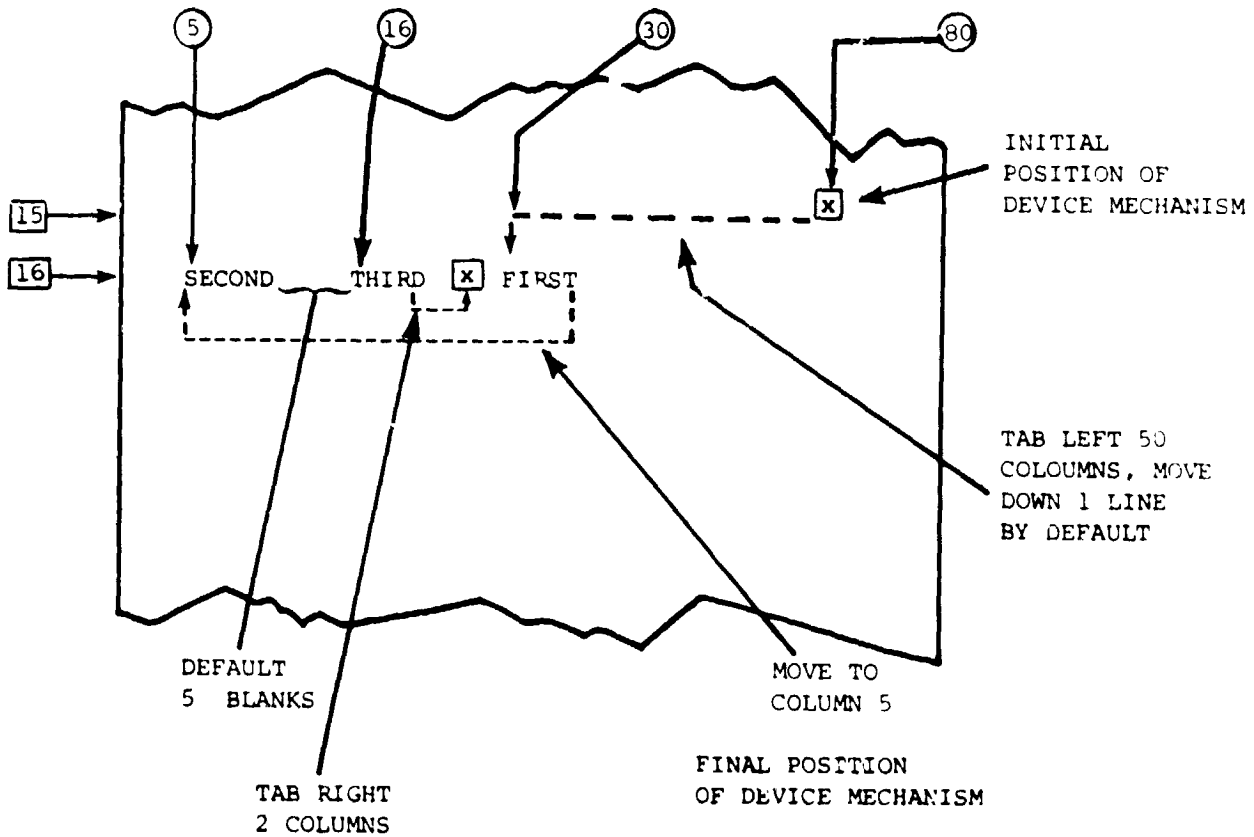
VERTICAL POSITIONING

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

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

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

1. α, β, and γ are integer expressions.

2. SKIP(α) moves the device mechanism downward by the number of lines specified by α. The value of α may be zero, in which case SKIP can suppress a default line advancement. However, α may not be negative (indicating upwards movement). SKIPs over page boundaries are allowed.

3. PAGE(β) moves the device mechanism downward by the number of pages specified by β. As in SKIP, β may **not** be negative in value. The relative line number <u>remains unchanged</u>.

4. For unpaged devices, LINE(γ) positions the device mechanism at line γ. The value of γ must not be such as to cause upwards movement of the device mechanism.

5. For paged devices, LINE(γ) has a different behaviour. Let the device mechanism be on line ℓ prior to execution of LINE(γ). If $\gamma < \ell$ then the device mechanism moves to line ℓ on the <u>next</u> page. If $\gamma \geq \ell$ then the device mechanism moves to line γ on the current page. The value of γ must lie in the range $1 \leq \gamma \leq L$, where L is the number of lines per page*.

---

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

If a SKIP, LINE, or PAGE pseudo-function appears at the beginning of a READ or WRITE statement, it overrides the default downward movement of one line.

SKIP, LINE and PAGE pseudo-functions do not of themselves inhibit the default horizontal movement to column 1. Neither does their appearance between two expressions in a WRITE statement affect the standard data field separation.

Successive SKIPs and PAGEs are cumulative in effect.

Examples:

If C1, C2, C3 are character data items

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

and if channel 6 is a paged device

then

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

produces output of the following form:

DEFAULT 5 BLANKS
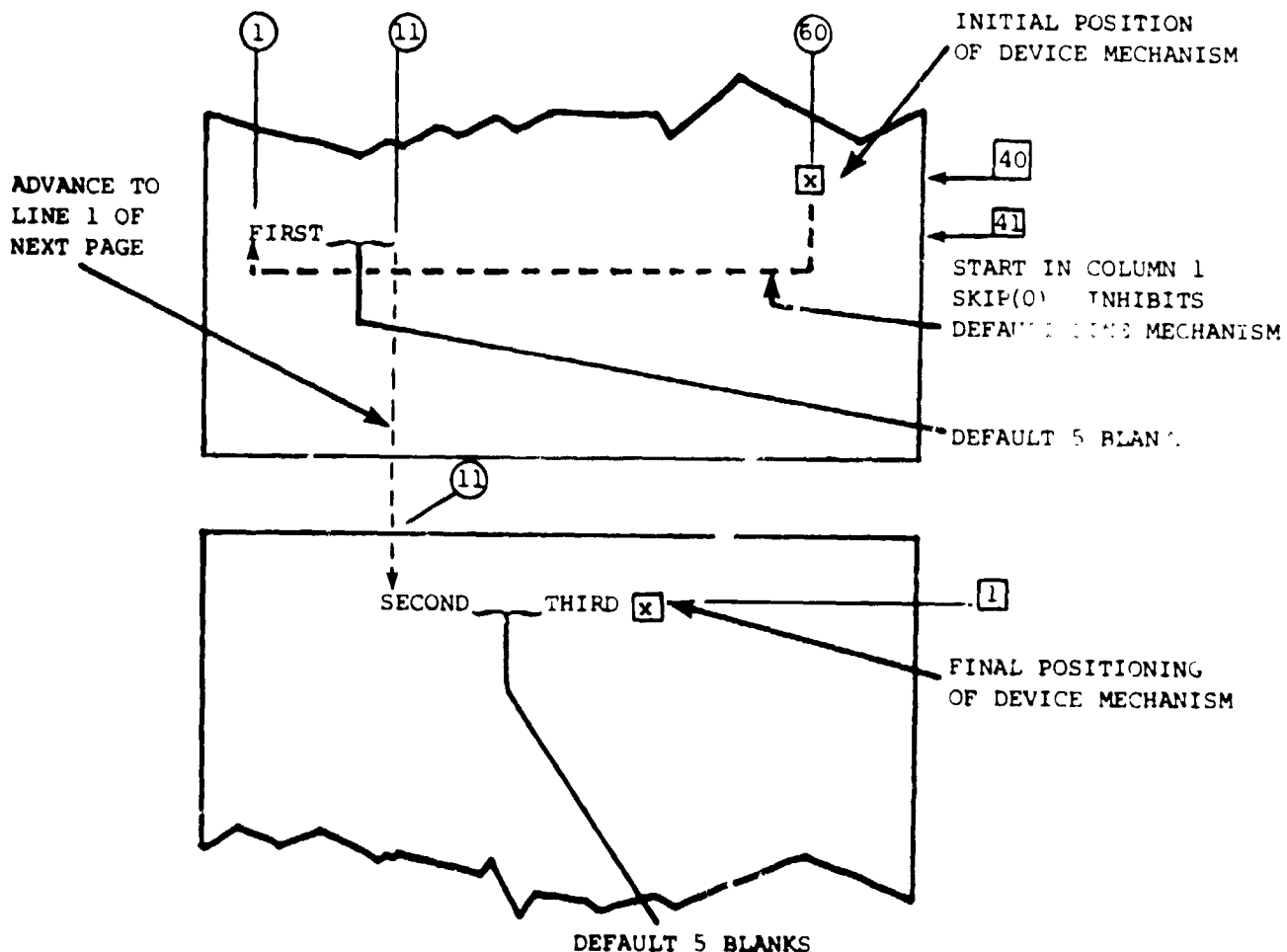
Note: If channel 6 were unpaged, the WRITE statement
would be illegal since it would be calling
for an upwards movement from line 40 to line 1.

Further,

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

produces the output of the form:

MOVE TO
LINE 41
OF NEXT PAGE

PAGE 6

FINAL POSITION OF DEVICE
MECHANISM
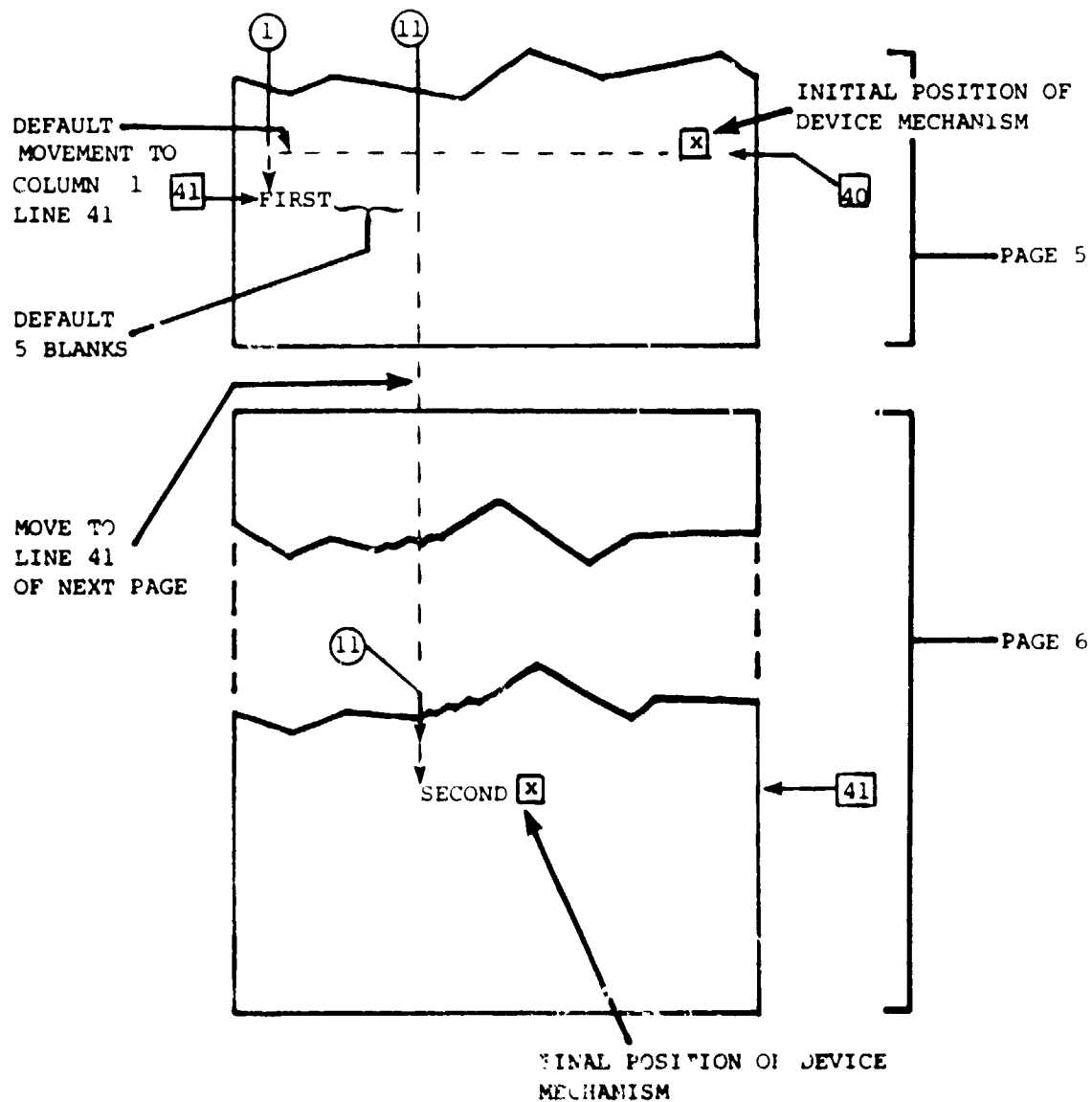
12.5   DEVICE ATTRIBUTES

In HAL/S, devices have been characterized as either paged
or unpaged. In the absence of any specific direction on
the part of a user, the following rules determine whether
a device being used is paged or unpaged.

* If only WRITE statements appear in a compilation
  for a given channel, then the device on that channel
  will be paged.

* If only READ statements appear, or if both READ and
  WRITE statements appear for a given channel, then
  the device on that channel will be unpaged.

The user may specifically direct certain channels to be
paged or unpaged, overriding these rules'.

---

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

12.6   SUMMARY

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

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

---

13. REAL TIME FEATURES OF HAL/S

So far the Guide has made no reference to the dynamic
properties of HAL/S programs. Clearly, any program will
take a finite time to execute but none of the constructs
hitherto described depend on any sense of time for their
operation.

However, the HAL/S language does contain constructs which
depend on a sense of time for their operation. This is
what is meant by the statement that HAL/S is a "real time
programming language". In other words, HAL/S programs
can be written which, when executed, cause operations to
be carried out at desired points or during desired inter-
vals in "real time".

In some implementations of HAL/S, "real time" may be just
what the phrase implies, real clock time. In others, the
"real time" may be simulated in some way by the operating
environment of a HAL/S program: in this case, it can be
referred to as "pseudo-real time".

This section of the Guide explains the basic HAL/S concepts
of real time programming, and describes some of the more
elementary real time programming language forms.

13.1 HAL/S REAL TIME CONCEPTS

The true HAL/S concept of a program at run time is an
entity executing over some interval in "real time",
directed and controlled by a Real Time Executive (RTE).
At the outset, the RTE begins execution of the program.
When program execution is completed, control is returned
to the RTE. In HAL/S terminology, the dynamic counter-
part of the static program block, which is executing
under RTE control, is called a "real time process".

MULTI-PROCESSING IN HAL/S

Multi-processing is the simultaneous handling of more
than one "real time process". With most present-day
machines, "simultaneous" really means interleaved,
because most machines can at one time only support the
execution of a single machine instruction sequence.
However, this distinction has no significance at the
higher level of the HAL/S language.

The RTE of HAL/S can simultaneously handle an arbitrary*
number of processes created by the user. A number is
attached by the user to each process, called its "priority".
The RTE maintains processes in a "process queue" ordered
by priority, and always endeavors to execute the processes
in order of priority, highest first.

The HAL/S program itself, beginning execution under the
RTE, constitutes the first or "primal process". All other
processes are brought into existence by the execution
of SCHEDULE statements coded into the program. Just as
the primal process has a static counterpart, which is
the program block coded by the user, so must the other
processes have their static counterparts. These are
so-called task blocks, which are coded inside the program
block in a very similar way to procedure blocks. Each
time a task block is invoked by execution of a SCHEDULE
statement, a new process is created and queued by the
RTE.

---

* See the User's Manual for the maximum number supported
in any given implementation.

13-2

A number of programs, indepen-
dently compiled, can be brought
together at run time. One
of them is chosen by the user
to start execution as the
primal process. Processes
can be generated from the
others by invoking them with
the same form of SCHEDULE state-
ment. Any of the programs are
allowed to contain task blocks
for which more processes in turn
can be created.
See: (TBD).

13-3

STATES OF A PROCESS

It is now possible to represent the behavior of the
RTE by a more formal description of the possible
states* in which a process can exist. This in turn
will introduce other HAL/S constructs for controlling
the activities of the RTE.

A process can be in either of the following two major
states at a given time:

- ● ACTIVE STATE: a process is in an active state
   when it exists in the RTE's process queue.
   The state actually comprises three substates
   or minor states in any one of which an active
   process may be at a given time.

- ● INACTIVE STATE: a process is defined for
   completeness as being in the inactive state
   if it does not exist in the process queue.

The minor states of an active process are as follows:

- ● EXECUTING: an active process is "executing"
   when it has actually been put into execution
   by the RTE, operating on the priority principle
   already described. The number of processes which
   can be in this state simultaneously is implementa-
   tion dependent**.

---

* The states to be defined do not correspond one-to-one
  with the RTE states described in the Language Specification
  document. The latter are defined for the convenience of
  the formal description of language constructs. The former
  are defined with user convenience in mind.

** In most implementations it is likely to be 1, but see
  the User's Manual for a given implementation.

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

- READY: an active process is "ready" if it
  is available for execution, but higher priority
  processes in execution are currently barring it.
  The occurrence of a process first entering the
  ready state will be called its "initiation".

- WAITING: an active process is "waiting" if it
  is neither ready nor executing. Some condition
  set up by the user prevents it being available
  for execution by the RTE.

When a process is created by invoking a task block by a
SCHEDULE statement, it makes a transition from the inactive
state to an active state. It is entered into the process
queue in either the ready or the waiting state, depending
on the form of the SCHEDULE statement. If it is entered in
the ready state, then depending on its priority, it may
immediately be elevated to the executing state.

A process is caused to make a transition from an active
state to the inactive state (or removed from the process
queue) by a TERMINATE statement. The process is said
to have been "terminated".

The priority of an active process may be changed by an
UPDATE PRIORITY statement.

A process may be forced into the waiting state by execution
of a WAIT statement.

The statements outlined above are among the real time
programming language forms to be described later in this
section.

PROCESS SWAPPING & BREAKPOINTS

A process swap is a pair of state transitions in which
one process leaves the executing state, and a second enters
it from the ready state. The process swap may occur because
the first process has been forced into the inactive state
or the waiting state, or because the second process has a
higher priority than the first.

The HAL/S language itself makes no assumptions on where
process swapping can occur. However, most implementations,
depending on the object machine characteristics, limit
process swapping to given places in the HAL/S code
sequences under execution by the RTE. These places are
called "breakpoints". The determination of breakpoints
is a function of the HAL/S compiler for a given implemen-
tation, and no language construct exists to modify their
existence*.

The effect of breakpoints is to superimpose a kind of time
granularity on the operation of the RTE.

PRIORITY SCALES

The number specifying the priority P of a process is
an integer in the range:

$$0 \leqslant P \leqslant 255**$$

The primal process is assigned a priority of 50** by
the RTE on beginning execution.

---

*   As an example, in the HAL/S-360 implementation, break-
     points occur at the end of every executable statement.

**   These values are, however, implementation dependent.
     See Appendix   for exceptions.

13-6

PROCESS DEPENDENCY

Suppose that there are two processes, A and B, and that A creates process B during the course of its execution. At the time of creation, B may be specified to be either "dependent" on or "independent" of A. If B is dependent, it means that it depends for its existence on the existence of A. If B is independent, then A may cease to exist without affecting B's existence.

However, an overriding rule is that all other processes are <u>always</u> dependent on the primal process for their existence.

The consequences of dependency will be seen when the flow of execution through program and task blocks is described in Section 13.3, and again when the TERMINATE statement in introduced in Section 13.5.

13.2 TASK BLOCK DEFINITIONS

A task block is a static block of code interior to a program, from whence processes can be created by means of the SCHEDULE statement. Task blocks may only be defined at the program level, and <u>not</u> nested inside procedure or function blocks defined in a program. This is illustrated as follows:

13-7

Program

Task Block

Procedure Block

Nested Function
Block
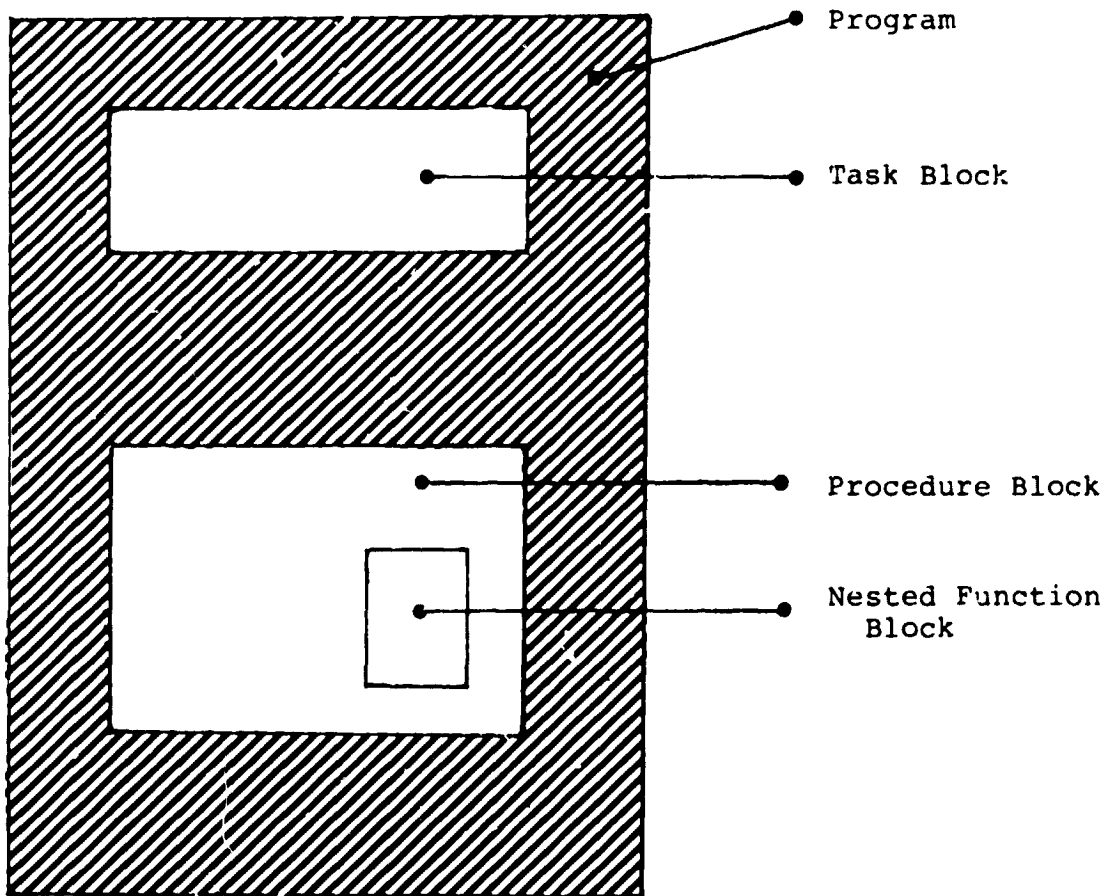
Region where Task
Blocks are legal
and may be nested.

13-8

Task block definitions are similar to program block
definitions as described in Section 3, and have similar
opening and closing statements.

RELATIVE POSITION OF TASK DEFINITIONS

Statements invoking a task block should normally follow
its block definition.

This rule is not absolute –
it can be circumvented by
the use of a task declara-
tion statement.
See: (TBD).

TASK OPENING

The statement opening a task block takes the form:

```
label:TASK;

1.   label is any legal identifier
     name, and constitutes the name
     of the task block.
```

TASK CLOSING

The statement closing a task block takes the form:

```
CLOSE label;

1.   The identifier label is optional.

2.   If supplied, it must be the name
     of the task block.
```

Example:

DISPLAY: TASK;

 } task body

CLOSE DISPLAY;

13-10

LOCAL DATA DECLARATIONS

Local data can be declared in a task block in exactly
the same way as it is declared in a procedure or
function block. The declarations appear after the
task opening statement, and before the first executable
statement of the block. The forms of the declarations
have been described in Section 4.

Examples:

general positioning -


```
DISPLAY: TASK;
```
local data declarations

executable statements

```
CLOSE DISPLAY;
```
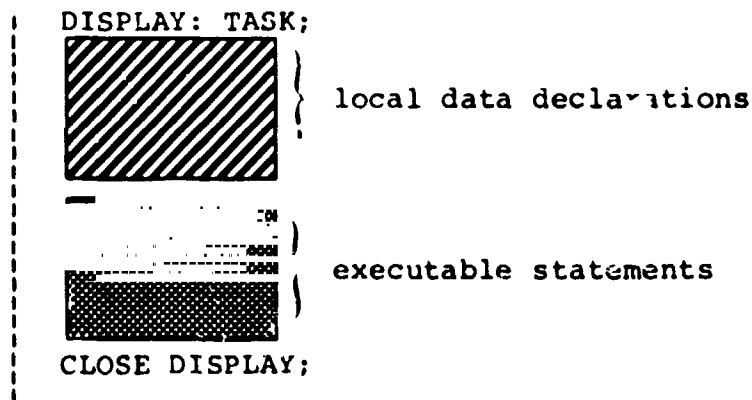
particular instance –

```
DISPLAY: TASK;
   DECLARE S CHARACTFR(10), }──── local data
           I INTEGER;       }

   .
   .
   .

CLOSE DISPLAY;
```

13.3 FLOW OF EXECUTION IN PROGRAM & TASK BLOCKS

The flow of execution through program and task blocks
is subject to a new interpretation, based on the
concepts of real time programming introduced in this
section. Programs and tasks are treated together since
their representations at run time are in both cases
real time processes.

Execution of a process begins with the first executable
statement in the corresponding static program or task
block. It continues, and if not terminated by some
other process, ends in one of the following ways:

- by execution of a TERMINATE statement
  terminating itself;

- by reaching the CLOSE statement of the
  block;

- by execution of a RETURN statement in
  the block.

If execution ends by self-termination, the process goes
into the inactive state and is removed from the process
queue. All dependents of the process are treated likewise.

If execution ends on a CLOSE or RETURN statement, the
process goes into the inactive state directly only if
it has no dependents. Otherwise, it goes into a waiting
state until the dependents have in their turn terminated.

13-12

FORM OF RETURN STATEMENT

The form of RETURN statement for programs and tasks
is the same as for procedures:

```
│
:     RETURN;
│
```

13.4 THE SCHEDULE STATEMENT

The SCHEDULE statement is an executable statement causing
a new process to be placed in the process queue. The
SCHEDULE statement specifies a task block from which
the process is to be created, and the priority which
it is to be given. A condition for the initiation of
the process can be supplied.

Only one process derived from a given task block may
be active at any given time.

The form of the SCHEDULE statement varies, depending on
whether it specifies immediate, or delayed initiation
(transition to the ready state).

IMMEDIATE INITIATION

The following variant of the SCHEDULE statement is the
simplest. It causes the creation of a process which
is placed in the process queue in the ready state. The
process is thus available for execution immediately.

13-13

```
| SCHEDULE label PRIORITY(α) DEPENDENT;
|
|   1.  A process is created from the task
|       block label and placed in the process
|       queue in the ready state.  The
|       process created is also known by
|       the name label.
|
|   2.  α is an integer expression specifying
|       the priority of the newly-created
|       process.  It must lie in the legal
|       range for a given implementation.
|
|   3.  The keyword DEPENDENT is optional.
|       Its presence denotes the dependency
|       of the process created on the
|       process executing the SCHEDULE
|       statement.  In its absence, the
|       processes are independent.
```

Examples:

```
SCHEDULE DISPLAY PRIORITY(100) DEPENDENT;
SCHEDULE RECOVER PRIORITY(255);
```

DELAYED INITIATION

The following form of the SCHEDULE statement causes
a process to be placed in the process queue in the
waiting state. The process is transferred to the ready
state on a specified time criterion being met. There
are two variants, each with a different time criterion.

● INITIATION after some duration.

SCHEDULE *label* IN *interval* PRIORITY(α) DEPENDENT;

1. A process called *label* is created from the
   corresponding task block and placed in the
   process queue in the waiting state.

2. PRIORITY(α) and DEPENDENT have the same
   meanings as described in the previous
   form of SCHEDULE statement.

3. The phrase IN *interval* indicates that the
   process is to be put in the ready state
   after a specified interval in the waiting
   state. *interval* is a scalar expression whose
   value specifies the duration in seconds.

4. If the value is negative or zero, the
   process is put in the ready state immediately.

13-15

● INITIATION at a given time.

SCHEDULE *label* AT *time* PRIORITY(α) DEPENDENT;

1. A process called *label* is created from the corresponding task block, and placed in the process queue in the waiting state.

2. PRIORITY(α) and DEPENDENT have the same meanings as described in the previous forms of SCHEDULE statement.

3. The phrase AT *time* indicates that the process is to be put in the ready state at a specified real time. *time* is a scalar expression whose value specifies the time in seconds.*

4. If the indicated time is in the past, the process is placed in the ready state immediately.

---

* The real time origin is not specified by the language. The origin is normally coincident with the initiation of the primal process. Some implementations allow its value to be preset at run time. See the User's Manual for a given implementation.

Examples:

```
SCHEDULE ALPHA AT 1.25E4 PRIORITY(I+5);
SCHEDULE BETA IN S+15.5 PRIORITY(20);
```

SCHEDULE statements can also
specify the cyclic execution
of a process until a stopping
criterion is met. An explicit
specification of the interval
between cycles can also be
given.
See: (TBD).

13-17

13.5   OTHER REAL TIME FEATURES OF HAL/S

Three other real time programming statements which
have already been mentioned are now described. These
are the TERMINATE, WAIT, and UPDATE PRIORITY statements.
Certain other useful constructs are also introduced.

TERMINATE STATEMENT

A process is forced to the inactive state (removed from
the process queue) by means of the TERMINATE statement.
Its form is shown below:

> TERMINATE *label* ;
>
> 1. The appearance of *label* is optional.
>   If present, the statement terminates
>   an active process called *label* .
>
> 2. If *label* is absent, then the process
>   executing the TERMINATE statement is
>   terminating itself.

In order to make independent processes truly independent,
HAL/S places an added restriction on the operation of
the TERMINATE statement. A process is only allowed to
use it to terminate <u>itself</u> or its <u>dependents</u>.

Note that when a process is terminated by execution of
a TERMINATE statement, all its dependents are automatically
terminated at the same time.

13-18

Examples:

```
|  TERMINATE; ——————————— self termination
|  TERMINATE BETA; ——————— termination of dependent
|
```

If a number of processes are to be terminated
simultaneously, the TERMINATE statement can
specify a list of process names:

```
|
|  TERMINATE ALPHA, BETA, GAMMA;
|
```

WAIT STATEMENT

The WAIT statement is used to force the process executing
it into a waiting state until some condition is met, where-
upon it returns to the ready state. Three forms, each
with a different condition, are described below.

● WAIT for a duration.

```
|  WAIT  interval ;
|
```

1. The statement indicates that the
   process is to be placed in the
   waiting state for a specified
   duration.

2. *interval* is ε scalar expression
   specifying the duration in seconds.

3. A negative or zero value results in
   the process not leaving the ready
   state.

13-19

● WAIT until some time.

> WAIT UNTIL *time* ;
>
> 1. The statement indicates that the process is to be placed in the waiting state until some given time.
>
> 2. *time* is a scalar expression specifying the time of return to the ready state, in seconds*.
>
> 3. Specification of a time in the past results in the process not leaving the ready state.

● WAIT for dependents.

> WAIT FOR DEPENDENT;
>
> 1. The statement indicates that the process is to be placed in the waiting state until all its dependent processes have terminated.
>
> 2. If there are no dependents, the statement has no effect.

Examples:

```
WAIT UNTIL DELTA_T+15E2;
WAIT S/2;
WAIT FOR DEPENDENT;
```

---

* See the discussion on the SCHEDULE statement in Section 13.4 for a footnote remarking on the real time origin.

13-20

UPDATE PRIORITY STATEMENT

The UPDATE PRIORITY statement is used to change the priority of an active process. Its form is:

UPDATE PRIORITY *label* TO α;

1. The process whose priority is to be changed is specified by *label*.

2. The name *label* is optional. If omitted, the process executing the statement is indicated.

3. α is an integer expression whose value indicates the new priority value to be assigned.

Examples:

```
UPDATE PRIORITY TO 16;
UPDATE PRIORITY ALPHA TO I+20;
```

Since the RTE operates on a basis of priority, apparently a user could control the execution of a desired set of processes by manipulating their relative priorities. Although this is entirely possible, it is not recommended since the behavior of such a priority-driven scheme would depend on how many processes an RTE could bring into the executing state simultaneously, which is an implementation-dependent figure.

REAL TIME BUILT-IN FUNCTIONS

Two built-in or library functions are of utility in constructing real time programs:

Function	Comments
RUNTIME	returns the current value of real time as a scalar, in seconds.
PRIO	returns the priority of the process invoking the function as an integer.

MAJOR STATE INDICATION

There exists a way of finding out whether the current state of any process is either active or inactive (i.e. whether or not it exists).

The name of the process can be used as if it were a Boolean variable. The following tables shows the correspondence between state and truth value.

State	Value
ACTIVE	TRUE
INACTIVE	FALSE

Example:

to write a message if a process ALPHA exists –

```
    IF ALPHA THEN WRITE(6) 'ALPHA IS ACTIVE';
```

13-22

13.6   A SIMPLE REAL TIME PROGRAM

The utility and importance of the constructs defined
in this section can only be properly understood by
presenting an actual example of a real time program.

The following example is given in the form of a problem
and its solution.

PROBLEM

The problem is to write a program which, when run on a
computer facility with remote interactive terminals,
will aid users in electronic circuit design (to take
an arbitrary example). A user begins each design
session by logging onto the facility at a terminal, and
invoking execution of the circuit design program.

The program is to be set up so that, at the outset, the
user may specify the desired duration of his session.
The program is then to interrupt the user's calculations
every 10 minutes and remind him how much time he has
used. At the expiration of the specified session duration,
the program is to allow the user 10 minutes more and then
terminate the session.

SOLUTION

Only the overall features of the program from the real
time programming standpoint are illustrated here. The
actual circuit design algorithms are of no consequence.

Execution of the circuit design program implies the
existence of three real time processes.

- a SUPERVISOR process controlling the two
  others, which determines the session dura-
  tion, and makes arrangements to terminate
  the session at its expiration. Most of the
  time this process will be in the waiting state.

13-23

- a TIMER process which informs the user how much time he has used every 10 minutes. This process is also mostly in the waiting state, temporarily being in execution every 10 minutes.

- a CALCULATOR process which actually interacts with the user in his design session. This process is executing most or all of the time.

The following diagram summarizes the activities of the three processes.

START

SUPERVISOR

1. determine session length
2. schedule TIMER and CALCULATOR processes
3. wait till end of session
4. signal 10 minutes more
5. wait 10 minutes
6. signal end of session and terminate

TIMER

1. wait 10 minutes
2. signal time used

CALCULATOR

interactive execution of design algor-ithms
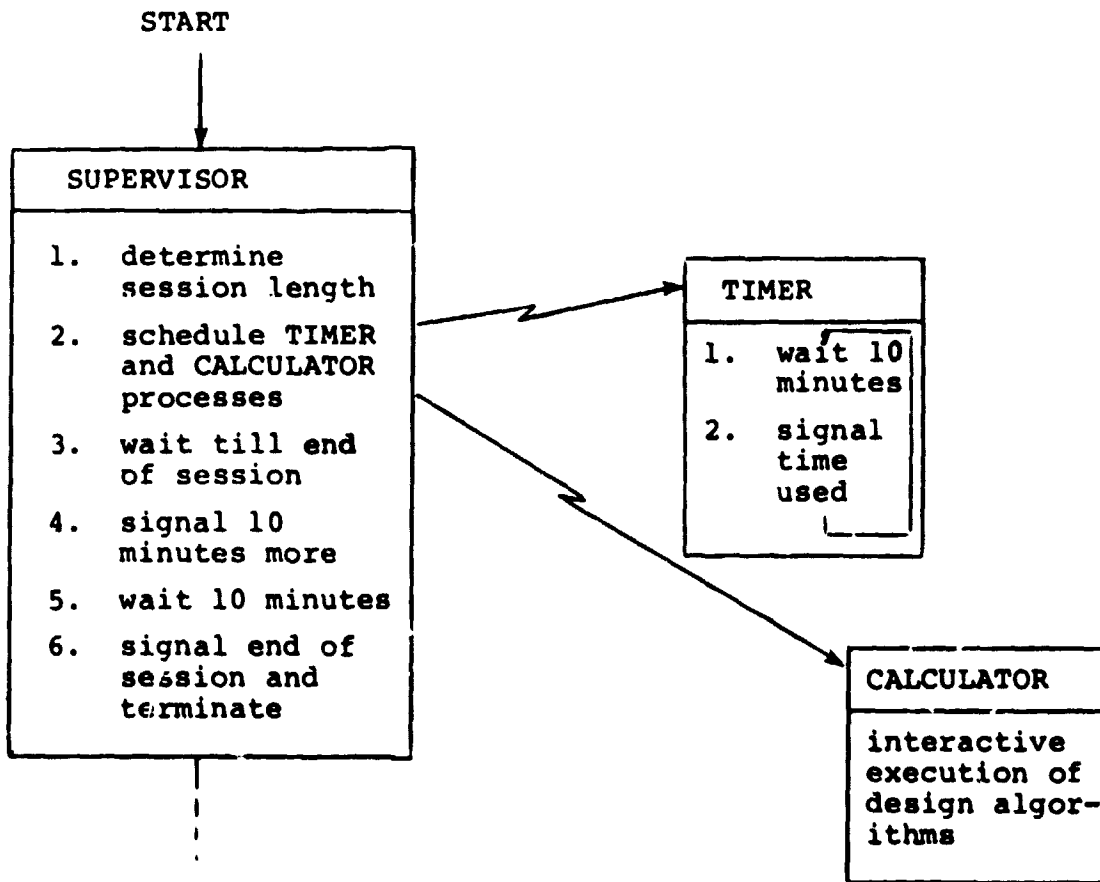
Clearly, in order for TIMER to interrupt CALCULATOR reliably every 10 minutes, it must have a higher priority than CALCULATOR. Likewise, SUPERVISOR should be of higher priority than CALCULATOR. The relative priorities of SUPERVISOR and TIMER do not matter since TIME is mostly in the waiting state anyway. The table below shows suitable priorities for each of the three processes.

process	priority
SUPERVISOR	50
TIMER	50
CALCULATOR	25

The HAL/S program corresponding to these processes
is as shown below:

```
SUPERVISOR: PROGRAM;          ←————  SUPERVISOR will be the
    DECLARE S SCALAR;                 primal process, initiated
    .                                 by the RTE at time 0.0
    .                                 with priority 50.
    .

    TIMER: TASK;      ←——————————  TIMER task block
        DO WHILE TRUE;
            WAIT 600;
            WRITE(6) 'YOU HAVE USED '||RUNTIME/60||' MINS.';
        END;   ←———————————————— infinite loop: wait 600
    CLOSE TIMER;                     seconds and signal time
    .                                used
    .

    CALCULATOR: TASK;  ←———————————  CALCULATOR task block

                                }  design algorithms

    CLOSE CALCULATOR;
    .                                 first executable state-
    .                                 ment of program
    .
    WRITE(6) 'TYPE SESSION DURATION IN MINS.';
    READ(5) S;   ←———————————————————  determine session
    SCHEDULE TIMER PRIORITY(50);          duration
    SCHEDULE CALCULATOR PRIORITY(25);  ←  Schedule TIMER &
    WAIT S 60;  ←——————————————————       CALCULATOR processes
    WRITE(6) 'TIME UP-10 MINS. MORE ALLOWED';
    WAIT 600;  ←——————————————————     Wait for session
    WRITE(6) 'END OF SESSION';            duration
    TERMINATE;                         allow 10 minutes
CLOSE SUPERVISOR;                      more

                                      signal end of
                                      session & terminate
```

13.7 SUMMARY

Section 13 has introduced the HAL/S concepts of real time and described constructs for the creation of real time programs. A concluding example shows how the constructs can be combined to perform useful functions in real time.

Section 13 completes Part I of the Programmer's Guide.

The constructs described above enable real time processes to be manipulated according to <u>time</u> criteria. Other constructs enable their manipulation according to "event" criteria. HAL/S "events" are Boolean-like data types whose values are accessible to the RTE. Their values can be set by the user, thus indirectly controlling the real time process states.
See: (TBD).

The problem of controlling the sharing of data by two or more processes is also important.
See: (TBD).

---

14. SUMMARY OF PART I

Part I of the Programmer's Guide has presented a wide
variety of the simpler constructs of the HAL/S language.
It has laid sufficient ground work for the understanding
of more complex language forms which are to be presented
in Part II.

_ low is summarized the material which has been presented
i. Part I.

SECTION 1 described, on a conceptual level, the nested
    block structure typical of HAL/S programs, and
    explained how globally and locally visible data
    could be declared. It also introduced the concept
    of nested groups of statements. The desirability
    of these hierarchical forms was expressed from the
    structured programming viewpoint.

SECTION 2 began describing the HAL/S language on the
    most fundamental level by specifying its character
    set; by explaining the forms of reserved words,
    identifiers and literals; and by introducing the
    format in which HAL/S source text is written.

SECTION 3 dealt with the HAL/S program as the basic
    unit of compilation. The delimiting statements
    of a program were defined and the positions within
    it of data declarations and executable statements
    described. The flow of execution within a program
    was pointed out.

SECTION 4 began defining the contents of HAL/S programs
    in more detail by presenting the various forms of
    declaration statements by which data could be defined.
    Ways of initializing this data were also described.

14-1

SECTION 5 turned aside from the discussion of HAL/S
    data by defining the form of REPLACE statements,
    by which symbolic HAL/S text substitutions could
    be made.

SECTION 6 returned to the discussion of HAL/S data
    by describing in detail how each of the HAL/S
    data types could be referenced, using subscripting
    to reference their specific component parts.

SECTION 7 began building a body of information towards
    the introduction of executable statements by describing
    how expressions of various types could be built up
    by combining operators with data items, literals and
    functions as operands. The topics of precedence
    and type conversion were addressed in the course of
    the section.

SECTION 8 introduced the assignment sta.ement, the first
    executable statement to be described in Part I.
    Each type of assignment was individually treated.

SECTION 9 expanded the repertoire of executable statements
    by presenting the IF statement, by whose means condi-
    tional execution of HAL/S statements could be effected.
    Its use in conjunction with statement labels and
    branches was discussed.

SECTION 10 formalized the idea of a statement group and
    stressed the importance of the idea from the structured
    programming standpoint. Various forms of statement
    group were introduced, including versions which caused
    repetitive or selective execution of the delimited
    statements.

SECTION 11 developed the concept of procedure and function
    blocks as callable entities. The forms of procedure
    and function block definitions were introduced,
    and the use of input and assign parameters explained.
    The manner of invoking procedures and functions was
    presented, and rules for matching argument and para-
    meter lists defined. Lastly, the form and purpose
    of the RETURN statement was pointed out.

14-2

SECTION 12 concluded the presentation of the HAL/S
program as a static entity by describing in
detail how input/output statements are constructed.

SECTION 13 introduced the new idea of HAL/S as a real
time programming language. The concept of real time
processes executing at run time under the control
of a Real Time Executive was presented. The form
of the task block, the static counterpart of a real
time process was described, and the SCHEDULE state-
ment for the creation of real time processes defined.
Other constructs for the handling of processes, among
them the TERMINATE, WAIT, and UPDATE PRIORITY state-
ments, were explained. Finally, a complete example
showing the assembly of the constructs into a viable
real time program was described.

INDEX