NASA

(NASA-CR-151872) PROGRAMMING IN HAL/S
(Intermetrics, Inc.) 326 p HC A15/MF A01
CSCL 09B

N79-14808

Unclas
G3/61 41884

Programming in HAL/S

PROGRAMMING IN HAL/S

MICHAEL J. RYER
Intermetrics Inc

This document was prepared under NASA Contract NAS9-13864

PREFACE

This manual is intended as an introduction to programming in HAL/S The reader is presumed to have some experience using one or more procedure-oriented languages such as FORTRAN or PL/I The book may be used either as part of a self-study program or in conjunction with a course of twenty to forty classroom hours over a period of one to two weeks

The material is organized as a tutorial rather than as a reference book Furthermore, it is intended as an introduction to HAL/S rather than as a definitive exposition After completing the course, the reader should refer to the *HAL/S Language Specification* or the *HAL/S Programmer's Guide* for a more detailed and complete description of the language

It is impossible to give proper credit to all the people at NASA, IBM, and Intermetrics who have contributed to this book Special recognition must go to Josephine Jue, John Schwartz, and Al Mandelin for their detailed review of several drafts of the manuscript, to Gary Singer for performing the final editing and page layout, and to Valerie Censabella who typed all of the manuscripts and got the majority of the exercises through the HAL/S-360 compiler

Support of the HAL/S language, compilers, and documentation is an ongoing effort of NASA and Intermetrics Comments on this manual will be appreciated and will be incorporated into subsequent editions All comments or inquiries should be addressed to

HAL/S Language Group
NASA- Johnson Space Center
FR/Spacecraft Software Division
Houston, Texas 77058
(713) 483-2151

Michael J Ryer
September 1978

TABLE OF CONTENTS

Preceding page blank

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

1.0 INTRODUCTION

HAL/S is a computer programming language; it is a representation for algorithms which can be interpreted by either a person or a computer. HAL/S compilers transform blocks of HAL/S code into machine language which can then be directly executed by a computer. When the machine language is executed, the algorithm specified by the HAL/S code (source) is performed. This document describes how to read and write HAL/S source.

HAL/S was developed principally for real-time aerospace programming. Its most significant use to date has been the production of the NASA Space Shuttle Flight software. This intended application imposed three major constraints on the language design: reliability, efficiency, and machine-independence. Reliability and efficiency are obvious requirements of flight software. The machine-independence requirement stems from a desire to minimize programmer training, to transfer blocks of proven code between distinct NASA projects, and to reduce the dependence on flight hardware availability.

Within these constraints, the language provides simple and intuitive constructs for functions commonly performed by aerospace applications, such as vector/matrix arithmetic. More generally, HAL/S is suitable for real-time process control applications, particularly where mathematically-oriented algorithms are involved. While the language is "tuned" for aerospace, the machine-independence and reliability aspects of HAL/S make it attractive for a variety of applications which do not perfectly match the original intent.

It may seem strange to some readers to attribute reliability to a programming language rather than to programs written in that language. This viewpoint is an outgrowth of the study of structured programming. A reliable program produces correct results for all possible combinations of inputs. Since it is usually impractical to exercise the program on all possible inputs, programs must be verified by induction. The assertion is made that if the program passes a *particular* set of tests, then the program will produce correct results for *any* set of inputs. This assertion is always based on an understanding of the program's internal workings. If the logic of a program is misunderstood, the results of verification cannot be relied upon.

Although it is difficult to assess the psychological implications, certain high order language constructs (e.g., the GOTO) are known to be symptomatic of unreliable programs. These constructs have been eliminated or highly restricted in HAL/S.

1.1 LEARNING HAL/S AFTER FORTRAN

HAL/S is similar to FORTRAN in many ways. The assignment statement is essentially the same in both languages. The FORTRAN concepts of subroutines, arrays, common blocks, and library routines all have analogues in HAL/S. Some concepts have been extended; for example, the FORTRAN statement A=B+C, can be used to add either integers or reals. The compiler generates instructions appropriate to the types of A, B, and C. In HAL/S, the same concept applies, but A, B, and C may also be vectors, matrices, or arrays of any type. HAL/S has many more data types than FORTRAN.

Every variable used in a HAL/S program must be explicity declared before it is referenced. This is done via the DECLARE statement, which specifies the name of the variable and its attributes (including its data type or "mode"). The need to declare variables results from the wide variety of data types in HAL/S It also allows the compiler to check for misuse of data and to enforce certain programming standards For example, a FORTRAN programmer might divide a variable containing alphanumeric character data by the number 256 in order to access the leftmost byte HAL/S does not allow any arithmetic operations on character data since such operations usually depend on the particular character code in use and are thus machine-dependent Instead, individual characters may be extracted from a character variable by explicit subscripting Similarly, binary (logical) data is a distinct data type The AND, OR, and NOT operators may be used with BOOLEANS or BIT strings, but not with arithmetic data

These restrictions may seem awkward at first, but with experience it will become quite natural to select the appropriate type for each variable in advance HAL/S includes constructs for data type conversions, but these conversions are needed less frequently than an experienced FORTRAN programmer might expect

Another major difference between HAL/S and FORTRAN is in the flow-control (branching) statements Structured programming research has had a major impact in this area In essence, the various forms of GOTO statement have been replaced with more reliable constructs The distinction may be characterized as "flow control by nesting of statements" rather than "flow control by branching" While this difference of philosophy may make the transition to HAL/S from FORTRAN more difficult, it can be argued that the HAL/S form is more English-like and thus move intuitive Furthermore, using the HAL/S flow-control constructs instead of GOTOs tends to result in a program which can be read sequentially (from top to bottom) Loops and decisions are expressed explicitly in HAL/S rather than implied by a convoluted arrangement of forward and backward branches In any case, most modern programming languages (including FORTRAN '77) have flow control statements of the type found in HAL/S

While the treatment of data types and flow control are the most fundamental differences between HAL/S and FORTRAN, the differences in source and listing formats are the most noticeable The source format is somewhat freer than in FORTRAN The output listing format, however, is not under programmer control at all Every HAL/S listing is put in a standard format by the compiler Each HAL/S statement is placed on a new line and automatically indented to show its relationship to other neighboring statements Exponents and subscripts are raised and lowered (respectively) in the listing, and various additional information (compiler-generated annotation) is added Thus, the work of the programmer is reduced, the indenting is always correct (since the compiler re-computes it every time), and reading a listing requires no knowledge of the individual programmer's style

Other major differences between HAL/S and FORTRAN are in the areas of Real-time interactions, and the interfacing of separately compiled units These advanced topics are thoroughly discussed in chapters eleven and twelve of the text

1 2 HAL/S CONTRASTED WITH OTHER HIGH ORDER LANGUAGES

The differences between HAL/S and other high order languages arise from the characteristics of aerospace applications, and the time-frame in which HAL/S was designed HAL/S was developed between 1970 and 1972 Since that time, changes which would invalidate existing HAL/S code have been resisted Thus, some recent advances in language design have not been incorporated Note, however, that the language did evolve from a thorough study of the existing languages. Most of the concepts which have been developed since that time have not been implemented in any *operational* (rather than experimental) language. When these concepts (e.g., data abstraction) have been proven outside of the university environment, they may be incorporated in HAL/S. There is an established language control board which continuously reviews the state of the art and suggests and/or approves changes to HAL/S.

Some features which were in common use at the time were excluded due to efficiency considerations. These include recursion and dynamic storage allocation. In addition to the overhead normally associated with these facilities, a reliability problem is avoided by their exclusion. Because of these and other exclusions, the total storage requirement of a HAL/S application can be exactly determined before execution starts. Consequently, HAL/S programs can never run out of storage during execution. This safety feature is essential in aerospace applications.

Other constructs, such as the full generality of the PL/1 error recovery system, have also been omitted for reasons of efficiency.

HAL/S also lacks sophisticated facilities for dealing with ground-based peripheral devices (printers, plotters, etc.). Character-oriented I/O statements are provided for testing and development, but many I/O facilities provided by ground-based operating systems are inaccessible from HAL/S. This is due to the design emphasis on *flight* software, and the lack of standardization of the concepts and facilities of ground-based operating systems.

HAL/S stresses readability rather than "writability". This approach acknowledges the fact that a program is written once (generally by one person), but is read many times (and often by many people). For instance, there are no abbreviations for HAL/S keywords. Furthermore, all of the keywords are "reserved". No confusion can arise from variable names which duplicate keywords, because no such re-use of a keyword is allowed.

On the other hand, HAL/S includes some facilities which other languages lack. Vector/matrix arithmetic has already been mentioned. HAL/S vectors and matrices are distinct from arrays, and are supported by a full set of operations. These include cross and dot product, as well as addition, subtraction, multiplication, division, and exponentiation. All are defined according to the usual rules of mathematics.

Although HAL/S contains features abstracted from a variety of languages, it exhibits a considerable uniformity. For instance, a portion of a variable is always selected by subscripting, whether the variable is a 3-vector, a character string, or a set of bits comprising a computer word.

Finally, there is one difference which is not exhibited in the language per se. This may be termed the "system" aspect of HAL/S. In addition to the listing and a machine-language "object module", the compiler generates a machine-readable random access file containing information about every variable and statement in the program. This file is then used by various statistics and diagnostic packages. Furthermore, some compilers can optionally insert "hooks" (diagnostic package interfaces) in the generated code. These interfaces are used in a functional simulation *(FSIM)* execution mode.

FSIM is a tool which allows flight code to be developed and tested on ground-based computers It includes a model of the flight operating system, and simulates the timing of the flight computer It also includes provisions for the simulation of avionics I/O This is done in such a way that flight code can be executed on a ground-based computer without any source-level changes whatsoever Debugging commands are entirely based on the HAL/S source, the program can be debugged without knowing any details of the ground computer hardware More information regarding the compiler and related software can be found in Appendix B of this manual

1 3 HAL/S CONTRASTED WITH THE ASSEMBLY LANGUAGE

This manual is primarily intended for experienced high order language programmers, this section presents some brief background information for programmers whose experience has been primarily in assembly language

The term "high order language" refers to languages in which a line of source produces a variable number of machine instructions Some readers may initially view HAL/S as a tool for specifying machine instructions more compactly

Many assemblers allow expressions, such as "A+B/C" in certain contexts where a number is needed The symbols used in these expressions must have values known to the assembler, i e , A, B, and C must be equated to constants in some way or must be macros which expand to constants or literals The computation is done at assembly time and the output of the assembler contains just the value of the expression

This facility is present in HAL/S There is, however, an important distinction if the values of the symbols used in a HAL/S expression are not known at compile-time, then machine instructions are generated to perform the computation at run-time *Most of the computation in a HAL/S program is specified by means of expressions* There are no ADD or SUBTRACT HAL/S statements, all arithmetic is done with operators (e g , "+", "–", etc) The "+" operator will add integers, scalars, vectors, matrices or arrays of any of these basic types The same operator performs both single and double precision arithmetic Thus, the compiler "decides" what particular machine instructions are appropriate to add the specified operands together This is one type of bookkeeping that is automated by the compiler

This approach illustrates another meaning of "high order language" the programmer is farther removed from the details of the computer hardware The programmer specifies a function (e g , addition) and the *compiler* maps it into the computer's repertoire (e g , LOAD, ADD, STORE) All addressing and instruction usage decisions are also the province of the compiler

Unlike a macro assembler, the compiler does not always generate the same instruction sequence for a given source statement It can "remember" whether a variable is still in a register from some prior statement, and, if so, avoid re-loading it The compiler may also move an entire computation out of a loop if none of the variables referenced are modified within the loop Generally, the compiler is free to make *any* re-arrangement of the program, provided that the same results will be produced from its execution This means that it is nearly impossible to predict what machine instructions will be generated when a particular HAL/S statement is compiled Hence, the best policy is to specify the desired *function* in the most intuitive way and ignore the mapping into machine instructions

There is no way to reference a particular machine register or word of memory in a HAL/S program Operations are performed on variables and constants rather than addresses and registers All such assignments are made by the compiler A large class of potential programmer errors (e g , use of the wrong register) is avoided by this approach

1 4 INTRODUCTION TO THE MAIN TEXT

The following chapters describe the HAL/S Language, a few advanced features are omitted, but most of the language is covered, including all of the frequently used constructs This manual is intended for sequential reading The HAL/S Language Specification is more appropriate for use as a reference, since it is concise, complete, and fully cross-referenced This manual, being tutorial in nature, describes each facet of the language in terms of the material presented in previous chapters interactions between separate constructs are not discussed until each construct has been described separately Each chapter is a prerequisite to the next, but no other knowledge of HAL/S is assumed

Another document, the HAL/S Programmer's Guide, is also tutorial in nature, but each chapter is self contained material is repeated instead of referenced Hence, the programmer's guide may be the best choice for "brushing up" on some particular aspect of the language

The information needed to compile (link, run and debug) a HAL/S program, once it is written, can be found in the HAL/S User's Manual for the particular compiler in use These documents also describe variations among compilers (i e , implementation dependencies)

The chapters which follow explain HAL/S primarily by example The *form* of each construct is always shown by example, the examples are so constructed that the *meanings* of new forms can be deduced Those who learn easily from examples may find portions of the English explanation redundant In every case, the examples are intended to be read from top to bottom when they are first referenced, rather than after the new constructs have been explained

The occasional tables and lists need not be memorized If the exercises can be done after one reading, further study is not needed The most important constructs are used freely in subsequent chapters, thus providing a continuous review of earlier material It would be difficult to learn HAL/S without writing any HAL/S programs, about one-half of the exercises require programming Answers to all are given in Appendix C

Computer words which are not defined herein (e g , algorithm, program) may be taken at their conventional meanings In some cases, a more precise HAL/S meaning is given later Definitions are denoted by italics as in "the form and meaning of a language construct are generally termed its *syntax* and *semantics*, respectively "

Chapter Two contains enough information to write a HAL/S program that really does something Chapter Three completes the topics introduced in Chapter Two, primarily additional forms of the arithmetic expression The remaining chapters discuss flow control, additional data types, and advanced topics such as real-time programming

2 0 READING, WRITING, AND ARITHMETIC

The basic rules for writing a HAL/S program are shown in the example below ,

```
   SIMPLE  PROGRAM,
C  CODE IN THIS TYPEFACE IS
C  HAL/S SOURCE
   DECLARE PI CONSTANT (3 14159266),
   DECLARE R SCALAR,
   READ(5) R,
   WRITE(6) PI R**2,
   CLOSE SIMPLE,
```

2 1 WRITING A HAL/S PROGRAM

The example above consists of six HAL/S statements and two comments The first statement serves to illustrate several conventions used throughout the language

1. Every program begins with a labeled PROGRAM statement
2. HAL/S statements are labeled by preceding them with an *identifier* and a colon
3. All HAL/S statements end with a semi-colon

The two lines following the PROGRAM statement are comments For further clarification, additional lines could be used Any line containing a C in column one is a comment Comment lines may be placed anywhere in a program

The next statements are DECLARE statements These statements form the *declare group*, which precedes the *executable statements* in every program variables are created via the DECLARE statement Variables must always be declared before they are used READ and WRITE are executable statements The numbers 5 and 6 in parentheses are channel numbers They control the routing to and from an external device Many other executable statements will be introduced in later chapters CLOSE, like PROGRAM, is a delimiting statement It is the last line of every program The block delimiting statements are further discussed in chapter seven This chapter stresses the DECLARE statement and the assignment statement (not shown above)

In this simple example each statement could be punched onto a card just as shown HAL/S source is free format *There are no rules about particular card columns except column one* Column one must contain one of the characters E, M, S, C, D or blank Normal statements are written with a blank in column one "C" is used for comments, the use of the other characters will be discussed later

When a program is stored on disk or tape the format is the same Column one is defined as the first character of a record or the character following an end of line code With this exception, the arrangement of HAL/S source on cards or records does not affect its interpretation by the compiler The example above could also be put as

```
    SIMPLE PROGRAM,
C   THIS IS HAL/S SOURCE
    DECLARE PI CONSTANT (3 14159266), DECLARE
    R SCALAR, READ(5) R, WRITE(6)
    PI R**2, CLOSE SIMPLE,
```

Longer programs are not always written correctly the first time Placing only one statement on a line makes later modifications much easier *

Since every statement ends with a semi-colon, no additional convention is needed for long statements *It is the semicolon rather than the end of a line that marks the end of a statement* To put a comment after a statement on the same line, the "/*" form can be used For instance

```
READ(5)R, /*OBTAIN RADIUS*/
WRITE(6) PI R**2, /* ** MEANS EXPONENTIATION */
```

This type of comment may be placed anywhere a blank is allowed (except in column one) It consists of any string of characters beginning with "/*" and ending with "*/" As the example shows, "*" and "/" may be used within the string in any combination other than "*/"

The WRITE statement could also be coded as

```
column 1
    ↓
    E
    M WRITE(6) PI R2,
```

Here, column one is used to distinguish between main and exponent lines Some implementations of HAL/S accept a two dimensional input format in which exponents and subscripts are indicated by their positions Multi-line input is generally not used however, since entering and maintaining source in this form is cumbersome under common editors or on cards The compiler produces listings in the multi-line format but all *source* in this book will be shown in the single-line form

The preceding paragraphs describe the placement of statements in a file or on cards Next we will discuss the format of individual statements

The PROGRAM and CLOSE statements contain the two keywords, an identifier, and punctuation Keywords are the "verbs" in HAL/S Each has a predefined meaning, and so cannot be used as a variable name A complete list of keywords is given in Appendix D All of the HAL/S keywords are made up of the letters A through Z Except for ARCTAN2 function, no numerals are used The underscore, or "break character" (_) is not used in any HAL/S keyword

*Some debugging systems allow a breakpoint to be set at the statement on a particular card (specified by sequence number) Placing only one statement per line also simplifies this usage

Blanks, or spaces, are significant in HAL/S For instance, DECLARER is a valid identifier It would *never* be interpreted as DECLARE R Blanks must be coded between keywords and identifiers in any combination Except in comments and character strings, however, *there is no difference between one blank and many blanks*

The compiler sees its input as a continuous stream of characters, i e , the concatenation of columns 2 through n of the entire input file This input is split into words at the punctuation blanks, commas, semi-colons, etc The punctuation is in two categories *delimiters* such as , „ and blank, and *operators* such as +, –, blank, and / When a blank appears between two identifiers or expressions it serves as the multiplication operator Otherwise, it is a delimiter

Using the punctuation, the compiler breaks its input into a series of *tokens* Tokens are of four types

1 Keywords such as DECLARE
2 Identifiers such as R
3 Operators such as ** or blank
4 *Literals* such as 3 14159265

Each HAL/S statement is defined in terms of these token types For instance, the basic DECLARE statement consists of the keyword DECLARE followed by an identifier followed by *attributes* The attributes consist of keywords and literals Like all statements, DECLARE ends with a semi-colon

Identifiers consist of variable names and labels The identifiers in the sample program are SIMPLE, PI, and R Identifiers may be from one to thirty-two characters in length, and composed from the letters A-Z, the numerals 0-9 and the underscore. *The first character must be a letter*, the last may not be an underscore Selection of names is entirely up to the user

DECLARE SIGMA CONSTANT (3 14159),

is *syntactically* correct The underscore may be used in an identifier to write an identifier composed on more than one word DELTA_V and TIME_TO_GO are valid identifiers

There is a trade-off in identifier lengths Very short identifiers, such as RLNGL, make for cryptic code, whereas very long identifiers, such as CURRENT_VEHICLE_ROLL_ANGLE, make it hard to find operators and match up parentheses in expressions Identifiers may not be started on one card and continued on the next Since the card boundary serves as a delimiter equivalent to a space, long names can be awkward

HAL/S does encourage self-documenting programs through meaningful identifier names This author's preference for a mixture of long and short names is generally displayed throughout this manual Sometimes this text uses underscores and numerals in identifiers to distinguish them from keywords The HAL/S keywords cannot be used as identifiers A few to be careful of are SUM, IN, SET, LINE and TRACE None of the keywords are less than two characters

The third type of token is an operator HAL/S includes logical and character operators as well as the arithmetic operators listed in Section 2 2

The fourth type of token is a literal There are arithmetic, character, and bit literals, though only arithmetic literals are of concern now Throughout this book, arithmetic literals are called simply *numbers*

While HAL/S has both integer and scalar *datatypes*, it does not distinguish between integer and scalar numbers "3" is completely equivalent to "3 0" "3 14159" is completely equivalent to "314159/100000", and to "314159E-5", "31415 9E-4" and so forth The character E is used in numbers to indicate scientific notation The form "3 14159E-5" is interpreted as

$$314159 \times 10^{-5}$$

or

(314159)10**(–5)

Thus, numbers can be written as a sequence of digits with or without a decimal point, optionally followed by the letter E and one or more digits The minus sign (–) is used for negative numbers and exponents The HAL/S Language Specification describes the use of other exponent letters to specify powers of two or sixteen instead of ten

No blanks may appear in a number Blanks must separate numbers from adjacent keywords, identifiers and literals

The statement,

```
DECLARE PI CONSTANT(3+1/7),
```

is completely valid "3 + 1/7" is considered a *number* rather than an expression An expression which contains only numbers, CONSTANTS, and the basic arithmetic operators is said to be ***computable at compile-time*** Instead of generating code to evaluate such an expression at runtime, the compiler will convert the expression to a simple number Only the value is kept at runtime, the addition and division in "3 + 1/7" are performed during compilation When this manual refers to numbers, any expression which can be reduced to a number during compilation is included

In summary, a HAL/S program begins with a labeled PROGRAM statement and ends with a CLOSE statement In between is a declare group followed by executable statements These statements may be arranged in any convenient way on successive cards or lines, providing that column one is blank Declaration and executable statements must end with a semi-colon Both comment lines and comments within statements are allowed Statements consist of a sequence of tokens separated by blanks or other punctuation, the tokens are of four types keywords, identifiers, operators, and literals Most of the HAL/S keywords and operators will be described later The rules for forming and recognizing tokens of each type have been presented here

Exercises

2 1A Some of the following are valid HAL/S tokens, some are not Identify the valid tokens, and state the type of each

Note Appendix D contains a complete list of HAL/S keywords

a) TEST_TIME

b) CHARACTER

c) TRY AGAIN

d) 7 1E-14

e) X

f) 1ABC

g) DEC_LARE

h) INITIAL

i) ALTITUDE_

j) TRUE

k) 4 2 1

l) QUITE'_A_LONG_STRING

m) 10000000

2 2 ARITHMETIC EXPRESSIONS

Like most high order languages, HAL/S allows numeric computations to be specified in a form very similar to ordinary mathematical notation For instance, the equations below should be quite recognizable in their HAL/S forms

```
AREA_CIRCLE = PI R**2,                     /*CIRCLE*/
AREA_TRIANGLE = 1/2 B H,                   /*TRIANGLE*/
AREA_PYTHEGORUS = (H**2 - B**2)**(1/2),    /*PYTHAGORUS*/
AREA_TRAPEZOID = H(A+B)/2,                 /*TRAPEZOID*/
```

This example illustrates the *forms* of some familiar equations in HAL/S. As in other languages, the successive assignments to AREA are not functional

This example shows four *assignment statements* as well as a number of arithmetic expressions The assignment statement is much as in other languages The value of the expression on the right of the equals sign is assigned into the variable on the left This section is primarily concerned with the evaluation of the expression on the right hand side

The example shows addition, subtraction, multiplication, division and exponentiation operators As in mathematical notation, *multiplication is indicated by adjacent factors* No special character is used to stand for multiplication Sometimes the blank is referred to as a multiplication operator, since adjacent identifiers must always be separated by a blank However, it is the *adjacency* not the blank that indicates multiplication For instance, "PI R**2" can be written without a blank as "PI(R**2)" or "(PI)R**2" or "R(PI)R"

The other basic operators contain no surprises The hyphen or minus sign is used for both subtraction and negation Parentheses control the order of valuation in the usual way The table below shows the major differences between HAL/S and mathematical conventions

Mathematical Notation	HAL/S Expression
ab	a b
$2x$	2 x
nx^{n-1}	n x**(n−1)
$-(c+d)$	−(c+d)
$\left(\frac{a+b}{c-d}\right)^{2\ 5}$	((a+b)/(c−d))**2 5
$\frac{xy}{-2ab}$	(x y)/(−2 a b)
$a(x+1)$	(a (x+1)

Mathematics defines several conventions to reduce the need for parenthesis in expressions For example,

A X + B Y

is always interpreted as the sum of two terms, (A X) + (B Y) rather than as the product of three factors A(X+B)Y These conventions are stated in terms of the *order of evaluation* of various constructs In particular, multiplication and division are performed before addition and subtraction HAL/S incorporates these rules by defining a *precedence* for each operator, as shown below

Precedence of Operators

** exponentiation first

b̸ multiplication

/ division

+, − addition / subtraction last

Note that multiplication is done before division rather than at the same time as in some languages

Given this precedence, the expression

$AX^2 + BX - C$,

is evaluated correctly when written in HAL/S without parenthesis

A X**2 + B X − C

The equivalent form with parenthesis is

((A(X**2)) + (B X)) − C

If strict left-to-right evaluation was desired, this could *only* be indicated by parentheses, as shown below

((A X)**2 + B)X − C

When an expression contains several operators of the same precedence, they are evaluated from left to right for all operators except for exponentiation and division These are evaluated right to left To see why this is true, consider the definitions below

$$X^{Y^Z} = X^{(Y^Z)}$$

$$\frac{\frac{A}{B}}{C} \equiv A\,\frac{C}{B}$$

The first expression is written

X**Y**Z

If X = 4, Y = 3, and Z = 2, this is

4**3**2 ≡ 4**(3**2) = 4^9

if the natural sequence was overridden via (4**3)**2, 16^2 would be produced Likewise, A/B/C is naturally interpreted as A / (B/C), which is indeed equal to A(C/B)

Other operators of equal precedence are evaluated from left to right Addition and multiplication are commutative and associative, so the order does not matter except for precision analysis Subtraction, however, is neither, and the order of evaluation *does* affect the results The HAL/S expression,

A − B − C

is interpreted as (A−B) −C

The distinction between numbers and expressions is somewhat blurred in HAL/S As already stated, any expression that can be computed in advance (during compilation) can be used wherever a number is required Furthermore, a negative number (e g , −1) is actually an *expression*, containing the number 1 and the negation operator *The presence of a blank between a minus sign and a literal is irrelevant* "−2A" is the product of A and −2, but "A −2" is a subtraction even though there is no space between the minus sign and the 2

The construct, "A/−2" is illegal The minus sign is seen as an operator, and *HAL/S never allows two operators in succession* This division could be written as "A/(−2)" or more sensibly as "−A/2"

To summarize precedence rules,

> HAL/S has defined the precedence of each operator to correspond to the usual mathematical conventions, BUT WHEN IN DOUBT, PARENTHESIZE

Arithmetic expressions may contain a variety of arithmetic types Integers, scalars, vectors, and matrices If one variable of each type is created as follows

```
DECLARE S SCALAR,
DECLARE I INTEGER,
DECLARE V VECTOR,
DECLARE M MATRIX,
```

The following multiplications and assignments are legal

```
S = V V,
V = V*V,
V = V M,
M = V V,
M = M M,
V = V S,
M = M S,
```

They are, respectively the dot (inner) product, the cross product, the vector matrix product, the vector outer product, the matrix product, and the scaling of a vector and a matrix They produce results of the types indicated by the *target variable* (left hand side) of these assignments This is a necessity rather than a coincidence Every expression has a datatype and assignments can only be made between like types

Identical data types are not required Since integers and scalars may be used interchangeably, the following combinations are also legal

```
I = V V,
V = V I,
M = M I,
```

as are all eight combinations of integers and scalars alone This, however, exhausts the combinations that can be written with the four variables declared above Not all operators apply to every combination of datatypes For instance, the addition of a vector to a matrix is not permitted In general, operations which are undefined in mathematics are illegal in HAL/S

By default, vectors and matrices are of size 3 and 3x3 Section 2 3 explores other possibilities and defines the operators in more detail At this point it suffices to say that wherever a variable of a given type is allowed in an expression, a parenthesized *expression* of the same type is also allowed, e g ,

```
V = V*((V S)M),
M = M(V V),
```

2 2 1 A Compiled Example

With the names (I, S, M, and V) used in the previous section, the type of each variable is apparent Most applications would require a better notation This is provided by the compiler as shown below

```
M | DATATYPES.
M | PROGRAM,
M |    DECLARE S SCALAR,
M |    DECLARE I INTEGER,
M |    DECLAPE V VECTOR,
M |    DECLAPE M MATRIX,
E |          -   -
M |    S = V   V,
E |    -   -   -
M |    V = V * V,
E |    -   - *
M |    V = V M,
E |    *   - -
M |    M = V V,
E |    *   * *
M |    M = M M,
E |    -   -
M |    V = V S;
M | CLOSE DATATYPES;
```

This listing was automatically produced from the preceding HAL/S statements by a HAL/S compiler No changes to the source were made The asterisk and hyphen overmarks appear only in the listing, they are not coded by the programmer The compiler indicates the type of each variable in a compilation via the overmarks shown below

Integer and Scalar	none
Vector	—
Matrix	*
Character	,
Bit and Boolean	/
Structure	+

Other differences between the source and the listing are

1 The compiler controls spacing, indenting, and the arrangement of statements on lines in the listing The source format is irrelevant

2 Statements in the listing always appear in multi-line format, with raised exponents and lowered subscripts

The compiler marks each line of the listing with an E, M, or S to indicate exponent, main, and subscript lines These characters, as well as "C" for comments, appear outside the box in the examples Some blank lines have been removed, and DECLARE statements are sometimes used in several examples without being repeated Any HAL/S code which appears in a box like the one preceding is extracted from an actual listing It has not been re-typed and is therefore free of *any* syntax errors

The standardized listing format produced by HAL/S compilers isolates the reader of a program from the style of its author The same listing will result whether the source was entered with minimum spacing on as few lines as possible, or was entered one token per line As a result, the listing format is a reliable source of information about a program's structure, independent of individual programmers Since the indenting in the listing is re-computed at each compilation based at the flow control statements in the source, it is always up to date, and changes to the source can be made without undue concern over spacing

This completes the discussion of HAL/S source and listing formats More information about arithmetic data will be needed to proceed with the topic of arithmetic operations

Exercises

2 2A Write HAL/S expressions equivalent to the following mathematical expressions

a) ax+by+cz

b) $\frac{a+b}{c} + \frac{d}{e+f}$

c) $\frac{2^{n-1}}{2^n-1}$

d) x^3-3x^2+3x-1

e) $(x-1)^3$

f) 10^{x^Y}

g) $(10^x)^Y$

h) $\frac{V\ W}{V\ V}$ V (V, W are vectors, ' ' means dot product)

2 2B The left-hand column contains mathematical expressions that are *incorrectly* coded in HAL/S in the right-hand column Find the errors and rewrite each expression correctly

a) mx+b	M*X+B
b) 2(x+1)	2X+1
c) $x^{-2\ 5n}$	X**(–2 5N)
d) c^{-5}	C**–5
e) $\frac{ac}{bd}$	AC/BD

2.3 DECLARING DATA

The example below is a declare group which shows the three different forms of DECLARE statements

```
M | DECLARE3
M | PROGRAM,
M |    DECLARE COUNTER INTEGER,
M |    DECLARE VECTOR,
M |               POSITION, VELOCITY, TORQUE;
M |    DECLARE NEW_CO_ORDS MATRIX,
M |            SPEED SCALAR,
M |            N INTEGER,
M |            WIND_FORCE VECTOR,
M | CLOSE DECLARE3,
```

The first form is the simple DECLARE statement used previously The next two forms are for convenience in declaring many variables The effect is the same as a number of simple declare statements The second form is a *factored declare* statement It is distinguished by the appearance of a data type *before* the variable names The data type applies to all of the identifiers in the list This example creates three 3-vectors

The third statement in DECLARE3 is a *compound declare* statement This form is used either to avoid re-typing the word DECLARE, or to show that a group of variables are related This grouping capability can aid in the attempt to document a program in the code as well as in the comments

Like all HAL/S statements, declarations may be entered in free format The example above shows how the compiler arranges the tokens in the listing

The simple declare statement consists of DECLARE, a variable name, and the attributes of that variable The factored declare statement consists of DECLARE, a set of attributes, a comma, and a list of identifiers to which the attributes apply The compound declare statement consists of DECLARE and a list of identifier-attributes pairs, separated by commas

The three forms of DECLARE are for convenience and documentation A variable of any type can be created using any form, and the type of declaration used does not affect the way the data is allocated or referenced

The *attributes* of an identifier consist of its data type, precision, dimensionality, initialization, lock group, and so on The only attribute that is *required* in a declare statement is the *data type* Several other attributes are described in Chapters three and six

The INTEGER type is used for counters, indexes, status indicators, and other applications where a variable's domain is limited to the whole numbers Integers generally occupy less storage than scalars and can be operated on more efficiently

SCALARs correspond to the real numbers They are generally stored in floating point format although this is not a language requirement In any case, they can represent numbers to "n" digits of precision, where n is constant for a given implementation In a floating point implementation, scalars may trade-off precision for a greater range by representing the number as a fraction (mantissa) and an exponent (characteristic)

VECTORs have scalar components They generally represent quantities in 3-space, such as position in cartesian coordinates Vectors can be of any length from 2 to an implementation-dependent limit The VECTOR keyword may be followed by a parenthesized number VECTOR(2) is a vector with two components VECTOR alone is an abbreviation for VECTOR(3) No distinction is made between row and column vectors

The MATRIX keyword by itself is equivalent to MATRIX(3,3) Matrices also have scalar components, but are generally viewed as linear operators on vectors, rather than as a collection of scalar or vector components

A VECTOR(n) can be multiplied by a MATRIX(x,n) yielding a VECTOR(x) When x = n = 3, this can serve as a coordinate transformation since each component of the resulting vector is equal to the dot product of the original vector and one column of the matrix A projection of the vector onto one axis

A MATRIX(x,y) can be multiplied by a MATRIX(y,z) yielding a MATRIX(x,z) The inner dimensions must match The exponentiation operator can be used to invert or transpose a matrix or to generate the identity matrix The cross product (*) only applies to 3-vectors The dot product ( ) applies only to vectors of equal lengths Addition, subtraction, and assignment require identical dimensions

These definitions of the four arithmetic data types are consistent with standard mathematical conventions Data type is the most important attribute because it determines which operations may be performed on the variable

Another important attribute of variables is initialization The INITIAL attribute specifies the value a variable will have when the program is first loaded into computer memory Its form is shown below

```
M   INITIAL_AND_CONSTANT·
M   PROGRAM,
M      DECLARE X SCALAR INITIAL(0);
M      DECLARE MAX_SPEED SCALAR INITIAL(14000);
M      DECLARE FEET_TO_MILES SCALAR CONSTANT(1 / 5280);
M      DECLARE SEC_TO_HR CONSTANT(60 (60)),
M      DECLARE MAX_MPH INITIAL(14000 FEET_TO_MILES / SEC_TO_HR),
M   CLOSE;
```

The CONSTANT attribute also causes initialization When an identifier has the CONSTANT attribute, its value cannot be changed Any attempt to assign into it results in an error message

In other respects, INITIAL and CONSTANT are the same Both are followed by a parenthesized value to which the identifier is initially set Variables of any type may be initialized For integers and scalars the value must be a number As the example indicates, this includes both arithmetic literals, and expressions which can be evaluated at compile time Since the value of a CONSTANT cannot be changed, compile-time expressions may contain references to previously declared integers and scalars with the CONSTANT attribute

This example shows two new abbreviated forms SCALAR is the default data type It can be omitted, as in the fourth declaration of the example Another omission is in the CLOSE statement The program name is optional, although good reasons for keeping it will be seen when nested code blocks are introduced in Chapter Seven

A vector or matrix is initialized in much the same way as an integer or scalar The essential difference is that a value for each of the vector or matrix components is specified in parentheses following the word INITIAL or CONSTANT The values are separated by commas and are sometimes referred to as the *initial list*

For example, the declaration

```
DECLARE VECT5 VECTOR(5) INITIAL(2 8,1 3,3 7,0,0),
```

created a vector with the following initial value

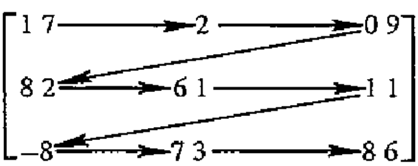
$$\begin{bmatrix} 2\ 8 \\ 1\ 3 \\ 3\ 7 \\ 0 \\ 0 \end{bmatrix}$$

Each element of the vector is initialized to the corresponding value in the initial list The first element receives the first value, the second element the second value, etc

For a matrix, the elements are initialized to the values in the initial list as follows the first *row* is initialized to the first values in the list (using enough of them to fill one row), then the second row is initialized, and so on The declaration,

```
DECLARE COORDMAT MATRIX(3,3) INITIAL(1 7,2,0 9,8 2,6 1,1 1,-8,7 3,8 6),
```

creates



The arrows indicate the order in which the matrix components are assigned from the linear series of values in the initial list

The important fact to remember about MATRIX initialization is that the order in which values are assigned is *by rows and not by columns* This row-by-row order also applies to the way matrix components are read and printed with READ and WRITE statements, and to arrays and the MATRIX shaping function, as will be shown later. This convention is commonly called row-major order

Writing an initial list as in the above examples can be cumbersome if the vector or matrix is large HAL/S offers some shortcuts

1 If only one value is specified in the initialization attribute, *all* of the components of the vector or matrix are initialized to that same value For example

```
DECLARE V VECTOR(3) INITIAL(10),
        M MATRIX(3,4) Initial(0),
```

$$\begin{bmatrix} 10 \\ 10 \\ 10 \end{bmatrix} \qquad \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

2 If several successive values in the initial list are identical, the programmer can specify a *repetition factor* and write the common component-values just once The repetition factor is a number indicating how many times the value is to be repeated, and it is separated from the value by a # symbol Using repetition factors, the initialization attribute,

```
INITIAL(1 5,1 5,1 5,2 7,2 7)
```

may be written more succinctly as,

```
INITIAL(3#1 5,2#2 7)
```

which is entirely equivalent to the longer form The repetition factor may also precede a parenthesized, comma-separated list of values, in which case the whole list is repeated Repetition factors may be nested to form a variety of patterns For example, a 3x3 matrix may be initialized to the identity matrix by the initialization attribute,

```
INITIAL(1,2#(3#0,1))
```

3 If only some components are to be initialized there are two ways to achieve the desired affect

a) A repetition factor may be specified *without* an accompanying value, in which case the specified number of components are passed over and left uninitialized, or

b) the last item in the initial list may be an asterisk, which indicates that the remaining components are not to be initialized

For example, the statement,

```
DECLARE A MATRIX(3,5) INITIAL (1,2,3,4#,8,6,3# 09,*),
```

creates the matrix

$$A \equiv \begin{bmatrix} 1 & 2 & 3 & x & x \\ x & x & 8 & 6 & 09 \\ 09 & 09 & x & x & x \end{bmatrix}$$

where x indicates an uninitialized component

The symbols # and * are used in vector and matrix initial lists as well as in other constructs They can also be used in the initial list in the declaration of an array or structure and in *shaping functions* As described later shaping functions allow the creation of vector and matrix quantities as in the following statement

```
M = MATRIX(1,2#(3#0,1)),
```

All HAL/S variables must be defined before they are referenced The DECLARE statement is the most common means of defining an identifier, but other possibilities such as use of the TEMPORARY statement will be introduced in later chapters While there are additional data types and attributes, all of the forms of the DECLARE *statement* have been presented

Exercises

2 3A Write declare statements corresponding to the table below

IDENTIFIER	TYPE	INITIAL/CONSTANT
X_DELTA	SCALAR	INITIALIZED TO 1
Y_DELTA	SCALAR	INITIALIZED TO 1
TIME_DELTA	CONSTANT	VALUE 1
DELAY_FACTOR	CONSTANT	VALUE 5
TEMP1	SCALAR	
TEMP2	SCALAR	
TEMP3	SCALAR	
COUNT	INTEGER	INITIALIZED TO 1
POINT_A	VECTOR	
ORIGIN	CONSTANT VECTOR	VALUE (0,0,0)
TRANSFORM	MATRIX	INITIALIZED TO $\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$

2.4 EXECUTABLE STATEMENTS

This chapter stresses the HAL/S source and listing formats and the arithmetic operators and data types Enough executable statements have been introduced to write simple programs The information about executable statements which will be assumed in later chapters appears below

The *assignment statement* consists of one or more target variables, an = sign, and an expression To store the value of an expression into several variables at once the *multiple assignment* is used, as in

```
I, J, K = 0,
A, B, C = (A+B+C)/3,
```

Each target variable must be of the same type as the expression on the right Conversions between integer and scalar, and single and double precision are automatically performed, however

The operands to the *READ statement* are a parenthesized channel number and a list of variables, e g ,

READ(5) ALPHA, BETA, GAMMA,

The channel number selects one of several external devices from which the variables are to read The data must be in a standard character format, so no additional control parameters need be given Chapter eight describes other options in the READ statement

The *WRITE statement* also includes an integer channel number Its remaining operands may be *expressions* of any type In the statement,

WRITE(6) M, V, M**(−1), M**(−1)V,

two matrix and two vector expressions appear Matrices can be raised to any integral power Minus one results in the "inverse" operation The output format is described in Chapter Eight along with more details of the READ, READALL, WRITE and FILE statements

The *PROGRAM and CLOSE statements* have been described in this chapter

Most of the remaining HAL/S statements alter the sequential flow of control These include statements for conditional execution (Chapter 4), looping (Chapter 5), and subroutines (Chapter 7) Error control (Chapter 10) and real-time (Chapters 11 and 12) statements complete the set

Chapter three describes additional forms of the arithmetic expression

End of Chapter Problems

2A The following program will compute the roots of the polynomial $3X^2+4X-10$ and print them out

```
ROOTS  PROGRAM;
   DECLARE SCALAR
      ROOT1, ROOT2;
   ROOT1 = (-4+(4**2-4 3 (-10)**0 5)/2;
   ROOT2 = (-4-(4**2-4 3 (-10))**0 5)/2;
   WRITE(6) ROOT1, ROOT2;
CLOSE ROOTS;
```

Modify the program to read in three scalar values A, B, and C from channel 5, and compute the roots of AX^2+BX+C

Note Assume the input values will yield real roots

2B A ball is tossed straight outward from a height of 110 feet with a horizontal velocity of 4 ft/sec Each time it hits the ground, it rebounds to 35% of its previous height

Write a HAL/S program to compute the time until the ball hits the ground for the third time, and how far it has traveled horizontally in that interval

The applicable equations of motion are

1 For an object dropping from height H to the ground or bouncing from the ground to height H, in time T,

$$H = \frac{1}{2} gT^2$$

where g = 32 ft/sec^2 is the gravitational acceleration

2 Horizontal motion is independent of vertical motion, so if D is horizontal distance traveled in time T at velocity V,

$$D = VT$$

2C An artificial satellite moves in a circular orbit of radius 4000 miles Write a HAL/S program to compute how long it takes to make 1 revolution and write the result on channel 6

Remember, $P = \dfrac{4\pi^2 R^3}{\sqrt{(\text{MASS\_OF\_EARTH})\ 6\ 670 \times 10^{-8}}}$ in CGS units

Say the MASS_OF_EARTH is 5 983 x 10^{27} grams One mile equals 160934 4 cm

2D Let $ax + by = e,$

$cx + dy = f,$

be a system of 2 equations in 2 unknowns

Write a HAL/S program to compute the solution of the system

The inputs a, b, c, d, e, and f are available on channel 5, and the solution x, y, should be written on channel 6

We are guaranteed that a solution does exist

Remember, Cramers Rule states

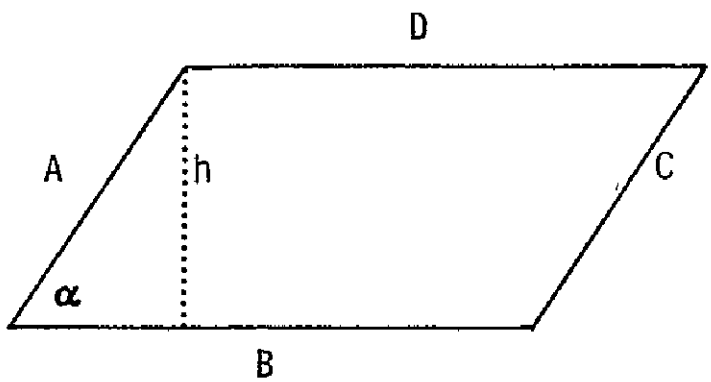
$$x = \frac{ed-bf}{ad-bc} \qquad y = \frac{af-ec}{ad-bc}$$

3.0 MORE BASICS

This chapter describes additional aspects of the arithmetic expression, including subscripting and function invocation. One new non-executable statement is also presented, so that only new data types, and *executable* statements other than assignment are left to later chapters.

3.1 BUILT-IN FUNCTIONS

In addition to the arithmetic operators, HAL/S provides a set of built-in functions. When the name of one of these functions occurs in an expression, code is generated to invoke the corresponding library routine. Built-in function names are HAL/S keywords and the run-time library routines are supplied with the compiler. Examples of several useful built-in functions can be given with the aid of a parallelogram.



The size and shape of a parallelogram are uniquely determined by the lengths of two adjacent sides and the angle between. These scalar quantities will be called LONG, SHORT and ALPHA.

Taking the lower left corner as the origin of a coordinate system with an X axis extending along B, the following program computes the coordinates of the corner points.

```
M CORNERS
M PROGRAM,
M    DECLARE SCALAR,
M                LONG, SHORT, ALPHA,
M    DECLAPE VECTOR(2),
M                AB, BC, CD, DA,
M    READ(5) LONG, SHORT, ALPHA,
E    -
M    AB = 0;
E    -
M    BC = VECTOR (LONG, 0),
S                2

E    -
M    DA = VECTOR (SHORT COS(ALPHA), SHORT SIN(ALPHA));
S                2

E    -    -    -
M    CD = BC + DA;
E             -   -   -   -
M    WRITE(6) AB, BC, CD, DA,
M CLOSE CORNERS;
```

The first assignment sets both components of the vector AB to zero Any arithmetic variable may be assigned from the literal zero Zero is the only such special case, it may be considered a typeless literal

The second assignment illustrates use of the VECTOR shaping function The expression VECTOR$ (2) (LONG,0) represents a 2-vector whose components have the values LONG and zero

In the third assignment, the arguments to the VECTOR function are arithmetic expressions As a result, the first component of DA is set to the product of the length of the short side and the cosine of the angle ALPHA The "Y" component of this vector is computed similarly, except that the sine function is used.

The fourth assignment merely illustrates the "parallelogram rule" for vector addition

SIN and COS are algebraic built-in functions, listed in Appendix A This category includes SIN, COS, TAN and their inverses (e g , ARCSIN) and the hyperbolic forms (e g , SINH, ARCCOSH) Also included are LOG, EXP, and SQRT For argument X, the latter functions are equivalent to Log_e (X), e^X, and $\sqrt{X}$

Each algebraic function returns a scalar value The arguments may be any integer or scaler expression An algebraic function name with its parenthesized argument is itself a scalar expression Thus, function invocations may be nested, as in

ARCTAN(SIN(X)/SQRT(1−SIN(X)**2))

A function's arguments are always enclosed in parenthesis As usual, the evaluation of an expression always starts at the inner-most parenthesis In the expression above, "1−SIN(X)**2" is evaluated as "1−((SIN(X))**2)" The function invocation may be viewed as of higher precedence than exponentiation Another interpretation of the same rule is that the value passed to a function is *completely* specified within the parenthesis Operators outside the parentheses apply to the value returned.

Before continuing to other classes of built-in functions, consider some general rules

1 No built-in function modifies any of its arguments
2 A function name and its argument list together comprise an expression of some data type
3 A function argument may be any expression of the specified data type
4 All trigonometric functions receive and return angles in *radians*
5 Invalid arguments (e g , SQRT(−1)) are indicated via runtime errors, as described in chapter ten

The parallelogram example also used the VECTOR shaping function Shaping functions perform conversions One function per data type is provided The arithmetic shaping functions are VECTOR, MATRIX, INTEGER and SCALAR The VECTOR and MATRIX functions will accept any number of arguments, each of which may be of any arithmetic type

The second assignment statement of the example might be entered as

```
BC = VECTOR$2(LONG,0),
```

This statement contains the first subscript used so far Whenever the VECTOR function produces a vector of dimension other than three, the dimensionality of the result must be specified as a subscript to the function HAL/S uses the dollar sign ($) to indicate a subscript, when the subscript is a single token (2), no parentheses are needed "VECTOR$2" is the HAL/S notation for "2-vector"

The MATRIX shaping function may also be subscripted A 3x2 matrix can be produced from the numbers 1–6 by

```
MATRIX$ (3,2) (1,2,3,4,5,6)
```

A three-by-three matrix can be produced without a subscript, as in

```
MATRIX (1,3#0,1,3#0,1)
```

The number of values in the argument list of a shaping function must match the subscript if one is supplied Otherwise, the number of values must be three (for a vector) or nine (for a matrix) If supplied, the subscript must be either a single compile-time expression indicating the length of a vector or two expressions, indicating a pair of matrix dimensions The product of these numbers is the number of components in the matrix The dimensions of *any* vector or matrix expression must be known at compile-time

It is the total number of *components* in a shaping function argument list that must match the subscript For instance, given

```
DECLARE M MATRIX,
        V4 VECTOR (4),
        V2 VECTOR (2),
        M22 MATRIX (2,2),
```

All of the following are legal (since each list has 9 components)

```
M = MATRIX (V4,M22,0),
M = MATRIX (V4,0V2,V2),
M = MATRIX$ (3,3) (M22,2#V2,0),
```

Whenever a data aggregate appears in the argument list of a shaping function, it is "unraveled" in the natural sequence (i e , the same order as in initial lists, row-major) The VECTOR and MATRIX functions see their argument lists as a linear stream of scalars Thus, if for example X, Y and Z are three 3-vectors, then MATRIX(X, Y, Z) is a 3x3 matrix in which the first row equals X, the second equals Y and the last contains the values from Z

Shaping functions are the only class of built-ins which accept a variable length argument list Others have a fixed number of arguments, each of a specified data type As stated above, the functions in the "algebraic" class all take one scalar argument and return a scalar result However, one basic rule in HAL/S is that wherever a scalar is expected an integer may be used, and vice-versa. In the assignment below,

```
DECLARE I INTEGER INITIAL (4),
I = TAN (I),
```

first I is converted to a scalar, then the tangent is taken and finally the result is *rounded* to the nearest integer before assignment into I

Rounding is defined in the usual way INTEGER (3 5) = 4, INTEGER (−1.4) = −1, and INTEGER (4999) = 0 As indicated, there are INTEGER and SCALAR shaping functions analogous to the VECTOR and MATRIX functions Since integer and scalar literals are written straightforwardly, and integer/scalar conversions are automatically performed, the INTEGER and SCALAR functions are less often needed than VECTOR and MATRIX More applications of these functions will arise after arrays and non-arithmetic data types have been introduced

Rounding can also be performed by the ROUND function, this function allows explicit rounding without using an integer variable, as in

```
DECLARE SCALAR, OLD, NEW,
WRITE(6) 'CHANGE IS', ROUND(100(NEW−OLD)/OLD),
   'PER CENT',
```

Character strings are described in chapter eight, character literals, such as 'per cent', are output unchanged by the WRITE statement If OLD=3 and NEW=5, the statement above would produce

```
CHANGE IS 67 PER CENT
```

The arithmetic functions include ROUND, TRUNCATE, FLOOR, and CEILING. The distinctions are shown in the following table

X =	3	5	−1.7	−1 3	1 6
ROUND (X)	0	1	−2	−1	2
TRUNCATE (X)	0	0	−1	−1	1
FLOOR (X)	0	0	−2	−2	1
CEILING (X)	1	1	−1	−1	2

In words, TRUNCATE ignores the fraction, FLOOR always rounds down, and CEILING always rounds up These functions always return an integer result

The arithmetic class also includes ABS (absolute value) and MOD (modulus) The result returned by these functions is of the same type as their argument(s) If the two arguments to MOD are of different types, the result is scalar

The remaining functions in this category, DIV, MIDVAL, ODD, REMAINDER, SIGN and SIGNUM, are described in Appendix A It should be noted here that the DIV function causes an *integer* division The remainder is discarded and the quotient is returned No rounding is performed When integers appear in a quotient written with "/", they are converted to scalars prior to the division

The only remaining category of functions to be discussed in this chapter is vector/matrix built-in functions

Name	Argument	Result	Comments
ABVAL	Vector	Scalar	Magnitude, length $\sqrt{\sum_i V_i^2}$
UNIT	Vector	Vector	Vector of length 1 in the same direction V/ABVAL(V)
INVERSE	nxn Matrix	nxn Matrix	Same as M**(−1)
TRANSPOSE	nxm Matrix	mxn Matrix	Same as M**T
DET	nxn Matrix	Scalar	Determinant
TRACE	nxn Matrix	Scalar	Sum of diagonal elements $\sum_{i=1}^{n} M_{i,i}$

The program below illustrates some of the power and convenience of HAL/S vector/matrix facilities It first reads in four 3-vectors, X, Y, Z and V, and determines whether X, Y and Z span 3-space Then it constructs an orthonormal set from X, Y, and Z yielding vectors A1, A2 and A3 Finally, these vectors are taken as the axes of a coordinate system, and V (the fourth input vector) is expressed in them

In this program, the determinant is used to find out whether X, Y and Z are linearly independent If they are not, the second assignment statement (after La Gram-Schmidt) may result in a runtime error, since unit of the zero vector is undefined Since the problem is in 3-space, A3 can be computed by a trick A1*A2 is orthogonal to both A1 and A2, A2 (A1*A2)=0), and of the length 1 (ABVAL(A1) 2 times ABVAL(A2) times sine(90°)) The transformation of V in the last assignment is conveniently done with a matrix, if, as in this program, the matrix is not saved, it may be more efficient to use the equivalent form

```
V = VECTOR(V A1,V A2,V A3),
```

The remaining built-in functions are much the same as those presented here Each is an expression of some data type, the arguments to each are of specified types, may be any expression, and so forth They will be discussed after the appropriate concepts and data types have been defined.

```
M | ORTHONORMAL
M | PROGRAM,
  |
C |  THIS PROGRAM CONSTRUCTS AN ORTHONORMAL
C |  SET FROM X,Y AND Z AND THEN EXPRESSES
C |  V IN IT
  |
M |    DECLARE VECTOR,
M |                  X, Y, Z, V, A1, A2, A3,
E |                                  -   -   -
M |    WRITE(6) DET(MATRIX(X, Y, Z)),
  |
C |  IF RESULT IS ZERO, X, Y AND Z DO NOT FORM
C |  BASIS ... EXPECT ERROR BELOW
  |
E |    -         -
M |    A1 = UNIT(X);
E |    -         -     -    -    -
M |    A2 = UNIT(Y - (Y . A1) A1),
E |    -    -    -
M |    A3 = A1 * A2,
E |    -               -    -    -    -
M |    V = MATRIX(A1, A2, A3) V;
M | CLOSE;
```

Exercises

3 1A What are the types of values of the following expressions?

a) ROUND (ABVAL(VECTORS2(SIN(0 5), COS(0 5))))

b) TRANSPOSE (MATRIX(1,3#2,3,3,4,5,6))

c) MATRIX$ (2,3) (1,0,0,1,1,1) VECTOR(1,2,3)

3 1B Write a HAL/S program to multiply the 3x3 matrix

$$\begin{bmatrix} 9 & 8 & 7 \\ 6 & 5 & 4 \\ 3 & 2 & 1 \end{bmatrix}$$

by its transpose and write the result on channel 6.

3.1C Translate these mathematical expressions into HAL/S

a) $\frac{1+\cos 2x}{2}$

b) $\tan^{-1}\left(\frac{y}{x}\right)$ (trig function is arctangent (inverse tangent))

c) $m(r\dot{z}-zr)\sin\theta-mrz\theta\cos\theta$

(use names like R_DOT, PHI, PHI_DOT, etc.)

d) $\cos^{-1}\left(\frac{m/r-ma/n}{\sqrt{2mE+\frac{m^2a^2}{n^2}}}\right)$

e) $\ln(\tan(\frac{x}{2}+\frac{\pi}{4}))$

(ln = natural logarithm, use PI for π)

3.2 SUBSCRIPTS

Subscripts are used to operate on components of larger entities. If V is a vector, V$1 refers to the first component.

Any vector or matrix variable or constant may be subscripted. This is done by appending a dollar sign ($) and a subscript expression. If the subscript expression is a single token, as in X$3, no parentheses or other punctuation is needed. Any expression *may* be parenthesized. X$ (((3))) is equivalent. Parentheses are required if the subscript involves any operators, e.g., V$(I+1).

Since matrix subscripts are written with a comma (and thus are not a single token), they are always parenthesized, as in

```
M$(I,J) = M2$(I,1) M3$(J,1) + M2$(I,2) M3$(J,2) +
          M2$(I,3) M3$(J,3);
```

Subscripting may be viewed as of higher precedence than the operators (+,–,*,**,etc.). Thus, V$I**2 is the square of the I^{th} component. This precedence is natural, since subscript computations seldom involve exponentiation.

If a subscript expression is of scalar type it is rounded. The result must be in the range 1 to N, where N is the declared dimension. *Any* integer or scalar valued expression may be used as a subscript.

A single component of a vector or matrix is a scalar, and may be used in any context where a scalar variable is allowed.

When an exponent contains a subscript, as in E**(V$1), the subscripted variable appears in the single line (source) format on the exponent line of the output listing.

$E^{V\$1}$

In all other cases, a subscript is indicated naturally by its position in the listing rather than a dollar sign When a subscript (or exponent) is lowered (or raised) in the listing, the outer parentheses (if any) are removed In A$(B$C)**(N−1), all of the parentheses are removed

```
E      N−1
M     A
S      B
S       C
```

A position in 3-space can be represented by a 3-vector in a variety of ways The program below uses subscripting to convert cartesian to polar coordinates. The results consist of bearing (angle from X axis in horizontal plane), elevation (angle from x axis in vertical plane), and total distance Angles are in radians, distance is in the original units

```
M | XYZ_TO_POLAR:
M | PROGRAM,
M |    DECLARE P VECTOR,
E |              -
M |    READ(5) P;
E |                                               -            -
M |    WRITE(6) ARCTAN(P  / P ), ARCTAN(P  / ABVAL(P       ), ABVAL(P),,
S |                     2   1           3             2 AT 1
  |
M | CLOSE XYZ_TO_POLAR;
```

This program assumes that the direction of P is in the same hemisphere as the positive x axis A more general solution can be written using the ARCTAN2 function

One new construct appears in the example P$(2 AT 1) is equal to VECTOR$2 (P$1,P$2) A 2-vector, consisting of the X and Y components of P ABVAL(P$(2 AT 1)) is the distance from the origin to a point in the horizontal plane directly beneath P

"2 AT 1" is one type of *partition subscript* It can be used to specify a slice of a vector in terms of the partition width and the number of the first included component The general form is *number* AT *expression* "Number" is any integer-scalar compile-time expression, *greater than one* and less than the corresponding declared dimension While partition widths must be known at compile-time, the starting component number may be any integer or scalar expression

Any partition of a vector is a vector A partition of length N can be used in any construct where a *declared* VECTOR(N) is allowed

P$(2 AT 1) can also be written as P$(1 TO 2) Here, the indices of the first and last components to be included are given, instead of the width and the first component

The dimension of P$(x TO y) is 1+y−x Since the dimensionality of every vector-matrix expression must be pre-determinable, *both* x and y must be known, neither may be an expression involving a variable

Given V ≡ VECTOR(10,20,30,40,50,60,70),
V$2 ≡ 20,
V$(2 TO 4) ≡ (20,30,40),
V$(3 AT 2) ≡ (20,30,40),
V$(3 AT V$3/10) ≡ (30,40,50),
V$(4 TO #) ≡ (40,50,60,70), and
V$(2 AT #–1) ≡ (60,70)

The sharp character (#) which appears in the last two partitions means "the last" V$ (4 TO #) can be read as "the fourth through last components" "2 AT #–1" is a way of specifying the last two components For the 7-vector above, any occurrence of # can be replaced by 7

A subscripted vector is either a scalar or a vector, depending on the type of subscript A subscripted matrix may be a scalar, a vector, or a matrix If both subscripts are simple (I,J) the result is scalar If one is simple and the other a partition (I,1 TO #), the result is a vector If both are partitions (2 AT 1, 1 TO 2), the result is a matrix Output listing overmarks indicate the resultant of type after subscripting

As usual, a matrix that has been subscripted down to type and dimension "X" can be used in any context where a *variable* of type and dimension "X" is allowed

The Ith row of a matrix M is M$(I, 1 TO #) This can also be written as M$(I,*) The Ith *column* is M$(*,I) The asterisk means "all of a dimension" In every case, it is equivalent to "1 TO #"

Using this form of partition subscript, the elementary row operations used in reducing matrices can be expressed compactly

```
M   ROWS
M   PROGRAM,
M      DECLARE M MATRIX,
M              C SCALAR,
M              TEMP VECTOR,
M              I INTEGER,
M              J INTEGER,

C    MULTIPLY A ROW BY A (NONZERO) CONSTANT:

E     -      -
M     M    = C M    ;
S      I,*      I,*

C    ADD A CONSTANT MULTIPLE OF ROW J TO ROW I

E     -     -        -
M     M    = M    + C M    ;
S      I,*   I,*       J,*
```

Continued

```
C | EXCHANGE ROWS I AND J:
  |
E |    -     -
M |   TEMP = M    ;
S |           I,*
  |
E |   -      -
M |   M    = M    ;
S |    I,*    J,*
  |
E |   -      -
M |   M    = TEMP;
S |    J,*
  |
M | CLOSE ROWS;
```

Before leaving the topic of subscripting, one caution is in order. HAL/S stores matrices in row-major order. This means that a *row* of matrix is stored in a contiguous block of memory. The scalars in a *column* of a matrix do *not* occupy consecutive locations. This may make operations on matrix columns less efficient than corresponding operations on rows. A few restrictions on the use of matrix columns (ASSIGN parameters, the input FILE statement and NAME variables) are described later. Matrix columns *are* acceptable in all constructs presented so far.

This section has described *component* subscripting. Most of the material also applies to *array* and *structure* subscripts, but there are some differences. These topics are discussed in chapters 6 and 9. Component subscripting applies to vectors, matrices, character strings and bit strings.

The term *subscript expression* has been used to stress the fact that there are forms which can occur only in subscripts. These are *partitions*. The forms A TO B, A AT B, *, and #±N are used only in *subscript* expressions.

An important point to remember from this section is that the set of contexts in which a variable may be used does not depend on the *presence* of subscripting, but on the *data-type which results* after the subscript has been applied.

Exercises

3 2A For the following vectors and matrices,

$$V1 = \begin{bmatrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{bmatrix} \qquad V2 = \begin{bmatrix} 10 \\ 11 \\ 12 \\ 13 \\ 14 \\ 15 \end{bmatrix} \qquad M22 = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} \qquad M35 = \begin{bmatrix} 7 & 4 & 1 & -2 & -5 \\ 6 & 3 & 0 & -3 & -6 \\ 5 & 2 & -1 & -4 & -7 \end{bmatrix}$$

a) Give the values of V1$(2), M22$(2,1), and M35$(2,3)

b) Give the values of V2$(3 AT 4), M22$(*,1), and M35$(2 TO 3, 4 AT 2)

c) Write the necessary declarations and initializations to produce V_1, V_2, M22, and M35

3 2B Write a HAL/S program that will compute the dot products of

$$\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

with each of the columns of

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

leave the results in a vector, RESULT_X, and write the results on channel 6

3.2C The diagrams below represent the values of various vectors and matrices

$$V31 = \begin{bmatrix} 0 \\ 1 \\ 2 \end{bmatrix} \quad V32 = \begin{bmatrix} 7 \\ 8 \\ 9 \end{bmatrix} \quad V33 = \begin{bmatrix} 11 \\ 12 \\ 13 \end{bmatrix} \quad M22 = \begin{bmatrix} 21 & 22 \\ 23 & 24 \end{bmatrix}$$

$$M33 = \begin{bmatrix} -1 & -2 & -3 \\ -4 & -5 & -6 \\ -7 & -8 & -9 \end{bmatrix}$$

What values will the following code print

```
V41 = VECTOR$4(M22),
M22 = MATRIX$(2,2) (M33$(2 AT 2, 2 AT 2)),
WRITE(6) V41,
WRITE(6) M22,
M33 = MATRIX$(3,3)(V31,V32,V33),
WRITE(6) M33,
M22 = MATRIX+(2,2)(V31,V32$2),
WRITE(6) M22,
```

3 3 THE REPLACE STATEMENT

The REPLACE statement provides a capability similar to the macros of other languages The REPLACE statement contains an identifier (termed the replace name or macro name) and a sequence of characters, termed the macro text The REPLACE statement instructs the compiler to substitute the macro text for every subsequent occurrence of the macro name

The REPLACE statement is not executable, it may only occur in the declare group The following represents one common use of REPLACE

```
REPLACE PRINT BY "WRITE(6)",
REPLACE PUNCH BY "WRITE(7)",
REPLACE CARDS BY "5",
```

Any occurrence of PRINT subsequent to these statements will be converted to WRITE(6) by the compiler The REPLACE statement causes the compiler to substitute the *replace text* for the replace or *macro name* wherever it occurs as a token in the following source Using the replace macros defined above,

```
READ(CARDS) X,                 becomes READ(5) X,
PRINT X, Y, Z,                 becomes WRITE(6) X, Y, Z,
```

and

```
PUNCH X, Y,                    becomes WRITE(7) X, 7,
```

The macro is not expanded in the listing Only the macro name appears Each reference to a macro is automatically underlined, however, this informs the reader that a replacement was done in order to avoid a possible mis-interpretation

The replace text is enclosed in double quotes (") This is the only use of the double-quote character in HAL/S The replace text may be *any sequence of characters not containing* " The replace name or macro name is an *identifier* and follows the conventions described in chapter two Since REPLACE is a HAL/S statement, it ends with a semi-colon

The macro name is only recognized when it appears as a token Given,

```
REPLACE A BY "1",
```

and

```
DECLARE ABLE SCALAR CONSTANT(A),
```

only one replacement is performed The other A's are part of keywords and an identifier, not complete tokens

Replace macros are commonly used to parameterize I/O channels, as indicated above, and the dimensions of variables, as in

```
REPLACE UNKNOWNS BY "6",
DECLARE AUGMENTED MATRIX(UNKNOWNS,UNKNOWNS+1),
```

HAL/S does not allow variables to be used for either channel numbers or dimensions, but since REPLACEments are done at compile-time, macro names may be used where numbers are required, provided the replace text is an expression computable at compile-time

The compiler will process the DECLARE statement above as if DECLARE AUGMENTED MATRIX(6,6+1), had been coded

Replace text is commonly a single number, but may be any string For example,

REPLACE DUMP BY "WRITE(6) X,Y,Z,GAMMA",

could be a useful abbreviation while debugging *The use of replace macros to abbreviate HAL/S keywords is strongly discouraged* HAL/S was designed to maximize readability rather than "writeability". It can be very difficult to decipher a program in which macros are used inappropriately. The time spent actually typing a program is generally insignificant compared to the time spent reading it

The program below illustrates a parameterized replace statement Here the macro is used to generate a table (for section 3 4) without writing a loop

```
M  TABLE:
M  PROGRAM,
M     REPLACE LOG2(X) BY "LOG(X)/LOG(2)";
M     REPLACE ENTRY(N) BY "WRITE(6) N, 2**(N-1),N/LOG2(10)",
E
M     ENTRY(8);
E
M     ENTRY(12);
E
M     ENTRY(16),
E
M     ENTRY(18),
E
M     ENTRY(24);
E
M     ENTRY(32);
E
M     ENTRY(36);
M  CLOSE TABLE,
```

In this example, X and N are macro arguments Wherever N appears in the replace text of the ENTRY macro, the actual parameter (8, 12, etc) is substituted Whenever the parameter, X, of the Log2 macro occurs in the text, the value 10 is substituted

The ENTRY macro generates an entire statement Note that no final semi-colon was placed inside the ending quote This produces a better listing since a semi-colon must terminate each *reference* to the macro, triggering a new listing line

The names of previously defined macros may be used in the replace text, as in LOG2 above The compiler will continue to make substitutions until no macro names remain, before any other processing An infinite expansion results if a macro's own name is used in its replace text Statements like,

REPLACE X BY "X+1",

not only cause error messages, but may abort the rest of the compilation

The above is a brief introduction to the HAL/S macro capability Additional features and more detail can be found in the Language Specification

3 4 THE PRECISION ATTRIBUTES

Most of the material so far has been concerned with the arithmetic expression Rules for forming expressions from identifiers, operators, literals, and keywords have been presented Every expression has a data type, the type is determined by the types of the identifiers and functions used, the operators which combine them, and the order of evaluation *Each expression also has a precision*

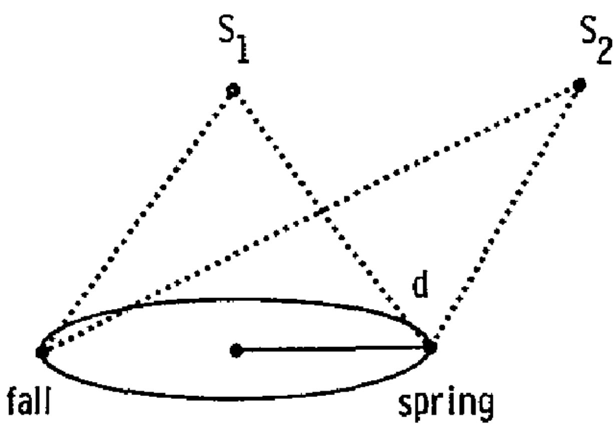
Arithmetic identifiers and expressions are of either SINGLE or DOUBLE precision All previous examples have been single precision Double precision variables represent values to more significant digits than single precision variables

Any arithmetic operation involving a double precision operand is done in double precision The result is also of double precision Thus, the usual method for specifying that a computation should be carried out to more digits is by declaring some or all of the variables to be double precision

The computation in the write statement below is performed in double precision

```
M | PARALLAX
M | PROGRAM,
M |    DECLARE EARTH_ORBIT CONSTANT(92.9E6);
M |    DECLAPE VECTOR(2),
M |                SPRING_READING, FALL_READING,
M |    DECLARE DEVIATION SCALAR DOUBLE;
E |                  -            -
M |    READ(5) SPRING_READING, FALL_READING;
E |                         -            -
M |    DEVIATION = ABVAL(SPRING_READING - FALL_READING) / 2;
M |    WRITE(6) 'DISTANCE=', EARTH_ORBIT / TAN(DEVIATION), 'MILES',
M | CLOSE PARALLAX;
```

This program could be used to compute the distance to a star based on its apparent change of position as the earth moves 180° in its orbit (93 million miles) The input data is a pair of angles in radians representing the star's direction in the Fall, and another set taken in the Spring The diagram below illustrates the algorithm in 2-space



Double precision is used in the example because a very large number is computed from a very small number using the tangent function near a zero The double precision tangent routine is invoked, and the division of 93 million by the result is performed in double precision Thus, the *expression*, "EARTH_ORBIT/TAN(DEVIATION)" is of type double precision scalar The WRITE statement outputs *all* the digits of its operands

The arithmetic in the preceding *assignment* statement is done in single precision Whether or not this is adequate depends on the provision of the measurements and the number of digits in a SCALAR SINGLE One radian is approximately 2 x 10^5 arc-seconds If the physical measurements are accurate to the nearest half second, then six decimal digits of precision would be enough * The value of the expression is converted to double precision before it is stored into deviation

The number of digits in the representation of a scalar (of either precision) is implementation-dependent These numbers are specified in the User's Guide A rule of thumb for scalars is one decimal digit for every 3 1/3 bits of mantissa

If the measurements have more significant digits than can be contained in a single precision scalar, the whole program could be done in double precision

```
DECLARE VECTOR(2) DOUBLE,S,F,
READ(5) S,F,
WRITE(6) EARTH_ORBIT/TAN(ABVAL(S—F)/2),
```

This version is written less mnemonically, and the assignment and write statements are combined These simplifications have no effect on precision

All of the computations in this form are done in double precision This is triggered entirely by the DOUBLE keyword in the declaration of S and F Note that there is only one name each for the tangent and absolute value functions, whether single or double precision The double precision form of a built-in function is automatically invoked when one or more arguments are of double precision *The value returned by a built-in function is of the same precision as its argument* Since ABVAL(S—F)/2 is a double precision expression, the double precision version of TAN is selected

Double precision expressions are formed under exactly the same rules given for single precision No restrictions apply to double precision variables that do not apply to single precision variables of the same type Precision is normally specified in declarations rather than expressions

*This program also assumes that the earth's orbit is exactly 92 9E6 miles, and that the readings are made at exactly the same time of day

The variables I, S, V, and M used in previous sections could have been declared as

```
DECLARE I INTEGER DOUBLE,
        S SCALAR DOUBLE,
        V VECTOR DOUBLE,
        M MATRIX DOUBLE,
```

This would not necessitate any changes to the expressions used

The DOUBLE attribute follows the data type in an attribute list It may be either before or after the other minor attributes such as initialization, LOCK, and AUTOMATIC, e g ,

```
DECLARE COVAR MATRIX(5,5) INITIAL(0) DOUBLE,
DECLARE V VECTOR(S) DOUBLE INITIAL(S#1),
```

Precision applies to all four arithmetic types Either SINGLE or DOUBLE may be specified in the attribute list of any integer, scalar, vector, or matrix Since single precision is the default, it need not be specified in declarations

Double precision vectors and matrices are composed of double precision scalars All of the vector-matrix operators and functions have both single and double precision implementations As before, double precision routines are selected when either operand is double, or when any *built-in* function argument is double

Since integers, double integers, single scalars and double scalars may be freely mixed and substituted for each other, these four combinations typically correspond to different sets of computer registers or machine instructions Conversions of integer to scalar and single to double are made automatically when operand types are incompatible Since integer and single precision operations are generally more efficient, data is left in the simpler forms whenever possible

The type and precision of an expression are determined solely from the expression itself Neither attribute depends on the context in which the expression is used The precision of the expression in an assignment statement is *not* determined by the precision of the target variable on the left hand side In the following, "10000 N" is a single precision expression, since neither operand of the multiplication is double

```
DECLARE D SCALAR DOUBLE,
DECLARE N INTEGER INITIAL(20),
D = 10000 N,
```

The right-hand side is of type single precision integer It will be converted to scalar double before assignment to D, but the multiplication is done in single integer mode.

Table 1 shows the range of integers with various word sizes If the code above is executed on a computer which represents single integers in 16 bits, the wrong answer will be produced The code can be corrected by adding an explicit precision specifier

```
D = 10000 N$(@DOUBLE),
```

The forms "@SINGLE" and "@DOUBLE" may be attached as subscripts to any arithmetic variable In the example above, "N$(@DOUBLE)" is of type integer double Thus, the multiplication is done in double precision and no accuracy is lost

The precision specifier may also be attached to shaping functions, as in

```
DECLARE VECTOR, V1, V2, V3,
DECLARE M MATRIX DOUBLE,
M = MATRIX$(@DOUBLE,3,3)(V1,V2,V3),
```

The precision specifier *precedes* any subscripts in a shaping function

Table 1

# of Bits	Range of Integer	# of Digits
8	128	2 4082393
12	2048	3 6123590
16	32768	4 8164796
18	131072	5 4185390
24	8388608	7 2247190
32	214748360	9 6329593
37	3435973800	10 837079

Empirically, double precision algebraic routines give better performance near zeros and singularities than their single precision counterparts These routines are generally implemented via polynomials, prefaced with code to identify the quadrant or other range of the argument The tangent routine, for an argument $0 < X < \pi/2$, might use a polynomial of the form

$$\text{Tan } x = A + Bx + CX^2 + DX^3 + EX^4 + FX^5$$

If the value DEVIATION in the parallax example has the value 1E–6 then the tangent will be

$$A + Bx10^{-6} + Cx10^{-12} + Dx10^{-18} + Ex10^{-24} + Fx10^{-30}$$

The operation $X = X + 10^{-N}\,X$, where n is greater than the number of digits contained in a scalar, does not change X

When two floating point numbers are added, the exponents are first equalized by shifting one of the mantissas It is this shifting that causes the loss of significant digits When two floating point numbers are multiplied, no shifting is required The same situation holds in fixed point, though any shifts required for addition and subtraction must be explicitly coded

In the parallax example, double precision allows the addition of more terms of the polynomial used to approximate the tangent function Double precision generally is needed when numbers of greatly different magnitudes are added or subtracted, and when a large number of output digits are needed The latter case is less common, since neither humans nor digital-analog converters can use more than a few digits directly

The arithmetic expression is summarized in the next section All of the statements made apply equally to single precision, double precision, and mixed Operations which reference one or more double precision values are done in double precision More digits are obtained, at greater expense in memory and execution time Some implementations have fixed point scalars, the Language Specification describes the explicit scaling (shifting) operators which are used in these implementations More details can be found in the appropriate User's Manual

3 5 SUMMARY OF THE ARITHMETIC EXPRESSION

An arithmetic expression has one of the following forms

1 An *identifier* This may be an integer, scalar, vector, or matrix variable or constant of either precision
2 A *literal* No sub-classes of numeric literals are defined.
3 A *subscripted identifier* Partition and simple subscripts are allowed, as well as explicit precision specifiers and scaling operators
4 A *function invocation* Both built-in and user functions may have zero or more arguments, which are themselves arithmetic expressions Shaping functions may also have subscripts
5 A further *expression prefixed by a minus sign* Any arithmetic type may be negated An expression preceeded by "+" is allowed, but functionless
6 A further *expression in parentheses* The parentheses override precedence rules, and allow scaling operators and precision specifiers to be attached to expressions
7 Two *expressions separated by an operator* Only certain combinations of operand types are allowed for each operator.

The list above is a recursive definition of the syntax of the arithmetic expression Expressions may be nested via forms three through seven

The compiler evaluates an expression outward from the most deeply-nested parentheses Within a set of parentheses, the compiler first evaluates any subscripts Operators are applied to the components selected by the subscripting

The table below shows the arithmetic operators in the order in which they are evaluated when not overridden by parentheses

	Operators in Decreasing Precedence
**	Exponentiation Applies to integers and scalars For matrices, the exponent must be either an integer or the character "T" Raising a matrix to the "T" power *always* indicates transposition of rows and columns Integer powers apply only to square matrices If I is negative, M**(I) is equal to INVERSE (M)**(–I)
multiplication	Indicated by a blank Multiplication is allowed between any two types, provided the "inner dimensions" match Resulting type given by outer dimensions
*	Cross product Applies only to 3-vectors The result is a 3-vector, given by Result = Vector($X_2Y_3-X_3Y_2, X_3Y_1-X_1Y_3, X_1Y_2-X_2Y_1$) The resulting vector is orthogonal to X and Y, and of magnitude (ABVAL(X)ABVAL(Y)SIN(θ)), where θ = the angle between X and Y
	Dot, scalar, or inner product Applies to vectors of equal dimension The result is a scalar equal to the sum of the products of corresponding components It also equals the product of the magnitudes of the vectors and cosine of the angle between
/	Division The left operand may be integer, scalar, or vector The right must be integer or scalar The result has the same dimension as the left operand, but is never integer
+, –	Addition and Subtraction If one operand is scalar, the other may be either integer or scalar Otherwise, the two operands must be of the same type and dimension
–	Negation Applies to any data type The result is of the same type

Operators of equal precedence are evaluated left to right, except for exponentiation and division which are evaluated right to left

Before non-arithmetic expressions are introduced, a number of statements which alter the sequential flow of control will be presented in chapters four and five

Exercises

3.5A HAL/S has seven infix operators

+, –, <>, *, ., /, **

Which infix operators are legal for the following pairs of data types? The characters <> represent a blank, meaning multiplication

Of what datatype is the result for each legal operation?

i)	SCALAR	SCALAR
ii)	SCALAR	INTEGER
iii)	INTEGER	SCALAR
iv)	INTEGER	INTEGER
v)	VECTOR	VECTOR
vi)	VECTOR	MATRIX
vii)	VECTOR	INTEGER/SCALAR
viii)	INTEGER/SCALAR	VECTOR
ix)	MATRIX	MATRIX
x)	MATRIX	INTEGER/SCALAR

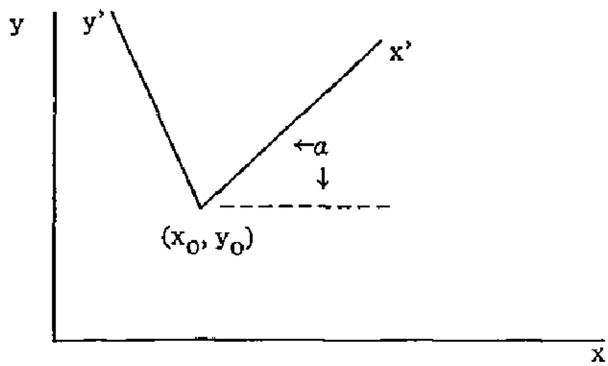
End Of Chapter Problems

3A Write a HAL/S program that will read 2 vectors from channel 5 and write the angle between them on channel 6.

Remember, $V_1 V_2 = |V_1| \; |V_2| \cos\theta$

where θ is angle between V_1 and V_2

3B There are occasions when it is necessary or advantageous to shift one's frame of reference. These occasions call for a translation and/or rotation of the coordinate system. Say the old axis (x, y) is shifted to the new axis (x', y') in the following manner, the x, y origin is shifted to (x_0, y_0) and rotated by α degrees as shown



The resulting translation equations are

$$x' = (x - x_0) \cos a + (y - y_0) \sin a$$
$$y' = -(x - x_0) \sin a + (y - y_0) \cos a$$

Write a HAL/S program that will translate 2 coordinates in the x, y system to new coordinates in x', y' where x = 54000, y_0 = 118000, a = 17° The two coordinates are available on channel 5 and should be written on channel 6

Remember that HAL/S trigonometric built-ins require angles in radians

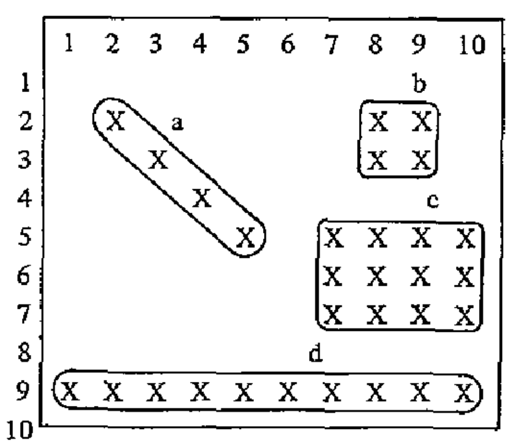
3C Write the right half of the following 4 assignments for the partitions in matrix M below

a) V4 = where V4 is a 4 vector

b) M22 = M22 is a 2x2 matrix

c) M34 = M34 is a 3x4 matrix

d) V10 = V10 is a 10 vector



4 0 CONDITIONAL EXECUTION

The statements in a program are executed sequentially, except when a flow control statement is executed The flow control statements can be loosely categorized by their use for decisions, loops, and subroutines These groups are described in chapters four, five, and seven

Although the HAL/S assignment statement is quite flexible, only a limited set of programs can be written without flow control statements The ability of digital computers to evaluate conditions and select alternatives is the essence of their power

4 1 IF THEN ELSE

A choice between two alternatives can be written with the HAL/S IF statement

```
IF A = 0 THEN WRITE(6) 'ZERO',
ELSE WRITE(6) A,
```

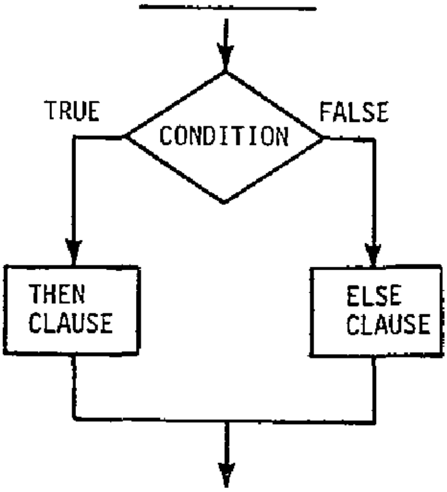
In this instance, the two alternatives are executable statements and the test is a *comparison* The first alternative is called the *then clause*, the second the *else clause*

IF is a *compound statement*, i e it is composed of further statements The concept of a statement containing "sub-statements" is common in HAL/S It will be useful to define the entire sequence, "IF *comparison* THEN *statement* ELSE *statement*" as a single statement, thereby

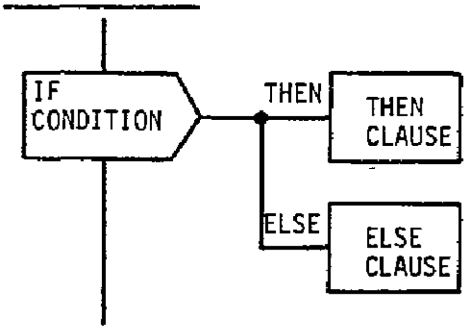
Unless the then or else clauses contain further flow control statements, control passes to the next sequential statement after an IF statement*

There are two equivalent graphical representations of the IF statement

Standard Flow



Structured Flow



*And only from the set EXIT, REPEAT, RETURN and GO TO

The form on the left illustrates the rule above by the explicit joining of two arrows at the bottom The system illustrated on the right is appropriate to structured programming languages in which complex decisions are represented through nesting of compound statements, all of which have one path in and one path out All of the HAL/S flow control statements (except GO TO) can be represented in structured flowcharts

The directions of the lines in a structured flowchart are implied Vertical lines are always traversed top to bottom Horizontal lines are always followed left to right *and back* Lines may intersect only at the points of IF and DO CASE statements There is no provision for overriding the natural direction

The above rules obviously limit the class of programs that can be represented However, the forms that have been ruled out have been shown to be symptomatic of programs that are difficult to read and maintain Any algorithm which can be expressed by a standard flowchart (where square boxes contain HAL/S assignments) is equivalent to some HAL/S program, without GO TO statements, which can be represented by a structured flowchart

The IF statement can select an alternative based on the results of a boolean combination of several comparisons A comparison consists of two expressions separated by a relational operator, as in

```
IF A = 0 THEN  .
IF N > 12 THEN   .
IF B**2 < 4 A C THEN
```

The complete list of relational operators is

=	*exact* equality
¬ = NOT =	not exactly equal
>	greater than
> =	greater than or equal
<	less than
< =	less than or equal
¬ > NOT >	not greater than (same as <=)
¬< NOT <	not less than (same as >=)

Since the character "¬" does not have a standard graphic across all systems, the keyword "NOT" may be freely substituted for it

All of the operators above may be used between any combination of integer or scalar single or double expressions When necessary, integers are automatically converted to scalars, and single precision is raised to double before the comparison

However, only the first two relational operators (= and ¬=) can be used between vectors, and matrices Two vectors or matrices may be compared for equality or inequality if they have the same dimension They are equal if each pair of components is exactly equal, and unequal otherwise

It is not generally useful to compare scalars, vectors, or matrices for equality In the statement,

```
IF A = B THEN WRITE(6) 'PURE COINCIDENCE',
```

where A and B are scalars, the WRITE statement is executed only if *every digit* in A is the same as in B Due to the finite precision of scalars and roundoff problems, if B had been set by

```
B = A/3,
B = B + 2 A/3,         /*1/3 A + 2/3. A*/
```

B would probably not be equal to A Scalars can be tested for approximate equality as in

```
IF ABS(A-B) < EPSILON THEN
```

where EPSILON is "sufficiently small", e g ,

```
DECLARE EPSILON CONSTANT( 000001),
```

or

```
EPSILON = (A+B)/16**( 25 MANTISSA_LENGTH),
```

etc

The keywords AND, OR, and NOT (or their equivalents, &, |, and ¬) may be used to combine several comparisons in one IF statement Parentheses are generally required around each simple comparison For example,

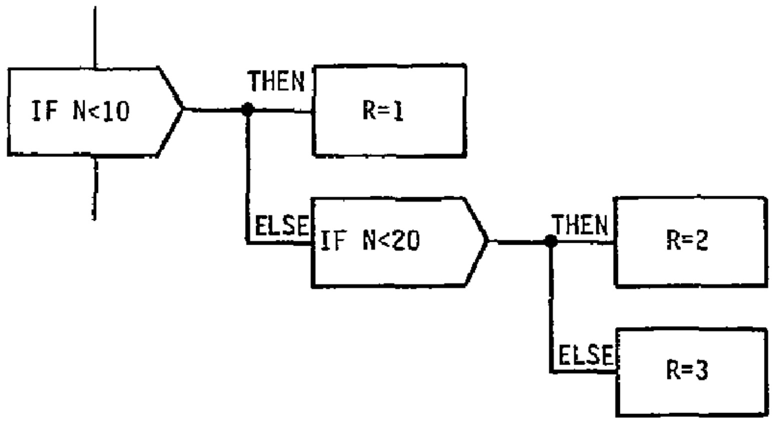
```
IF (A>0) AND (A<100) THEN
IF NOT((A<=0) OR (A>=100)) THEN
```

Both of these forms will result in the execution of the then clause if (and only if) $0 < A < 100$ The first test checks whether A is in the given range The second test is equivalent since it checks whether A is not outside the range The sense of any comparison or combination thereof can be reversed using the NOT keyword as shown in the second test This use of NOT *requires* a parenthesized argument

Suppose a number is divided into one of three ranges, as shown

```
IF N < 10 THEN R = 1,
ELSE IF N < 20 THEN R = 2,
   ELSE R = 3,
```

Here, the else clause of an IF statement is an entire IF   THEN . ELSE group It may be diagrammed as follows



The THEN clause of an IF statement may *not* be an IF   THEN   ELSE group * A four way branch can be written with a DO   END group, as described in the next section

The IF statement allows the selection of one or two alternatives based on the evaluation of a comparison When no action is required unless the test succeeds, the else clause may be omitted entirely

```
IF A > 0 THEN B = SQRT(A),
```

This statement is functionally equivalent to

```
IF A NOT > 0 THEN,
ELSE B = SQRT(A),
```

Here the then clause is just a semicolon, which is the HAL/S equivalent of a no-op or null statement

IF. . THEN   ELSE may be viewed as a single statement The then and else clauses each contain a further single statement Any executable statement is allowed in the else clause, the then clause may contain any executable statement except a further IF   THEN .ELSE The else clause may also be omitted entirely

*This rule avoids the "dangling else" problem common to ALGOL-like languages

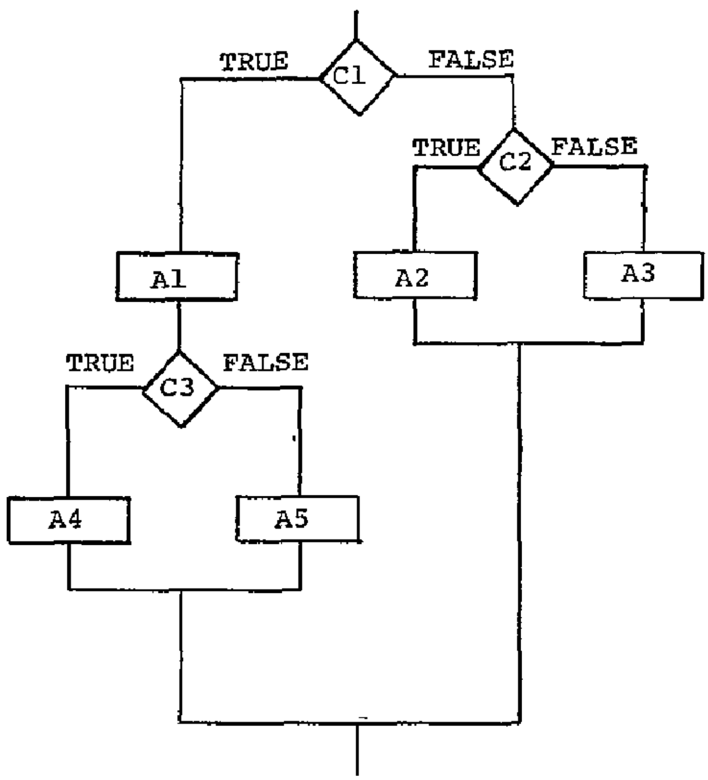
Exercises

4 1A What is wrong with the following HAL/S conditional statements (in which all variables are of SCALAR type)

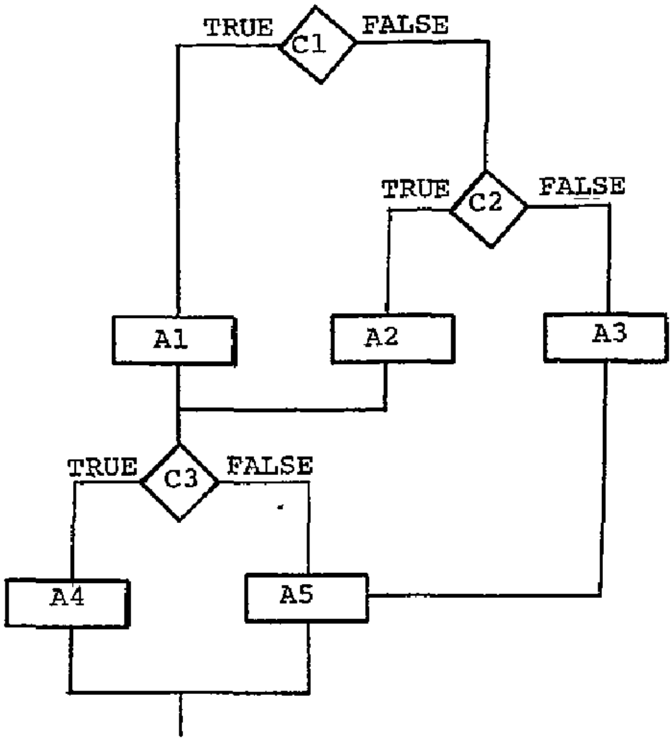
```
a) IF A < B < C THEN MIDDLE = B,
b) IF B < C THEN
      IF C < D THEN B = D,
      ELSE B = C,
   ELSE C = B,
c) IF RADIUS > 0 & NOT RADIUS > 1 THEN
      WRITE(6) PI RADIUS**2,
```

4 1B Where possible, convert these standard flowcharts to structured flowcharts, without duplicating or eliminating boxes Indicate why the others cannot be converted

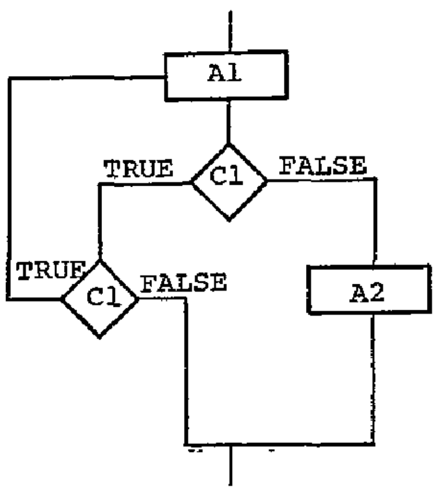
a)



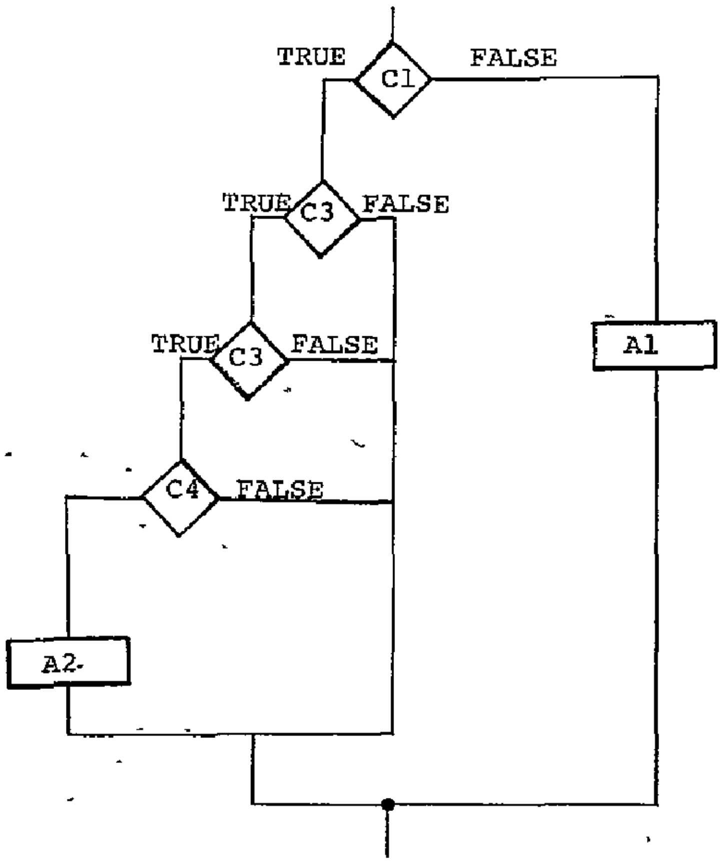
b)


TRUE
C1
FALSE
TRUE
C2
FALSE
A1
A2
A3
TRUE
C3
FALSE
A4
A5

c)


A1
TRUE
C1
FALSE
TRUE
C1
FALSE
A2

d)



4 1C Tell whether the following conditions are satisfied, not satisfied, or illegal Assume that

A, B, C, D are scalars
$\overline{V}$, $\overline{S}$ are 3-vectors
A = 7 0 C = 12 0
B = 4 0 D = 3 2
$\overline{V}$ = (2 4 6) $\overline{S}$ = (3 4 12)

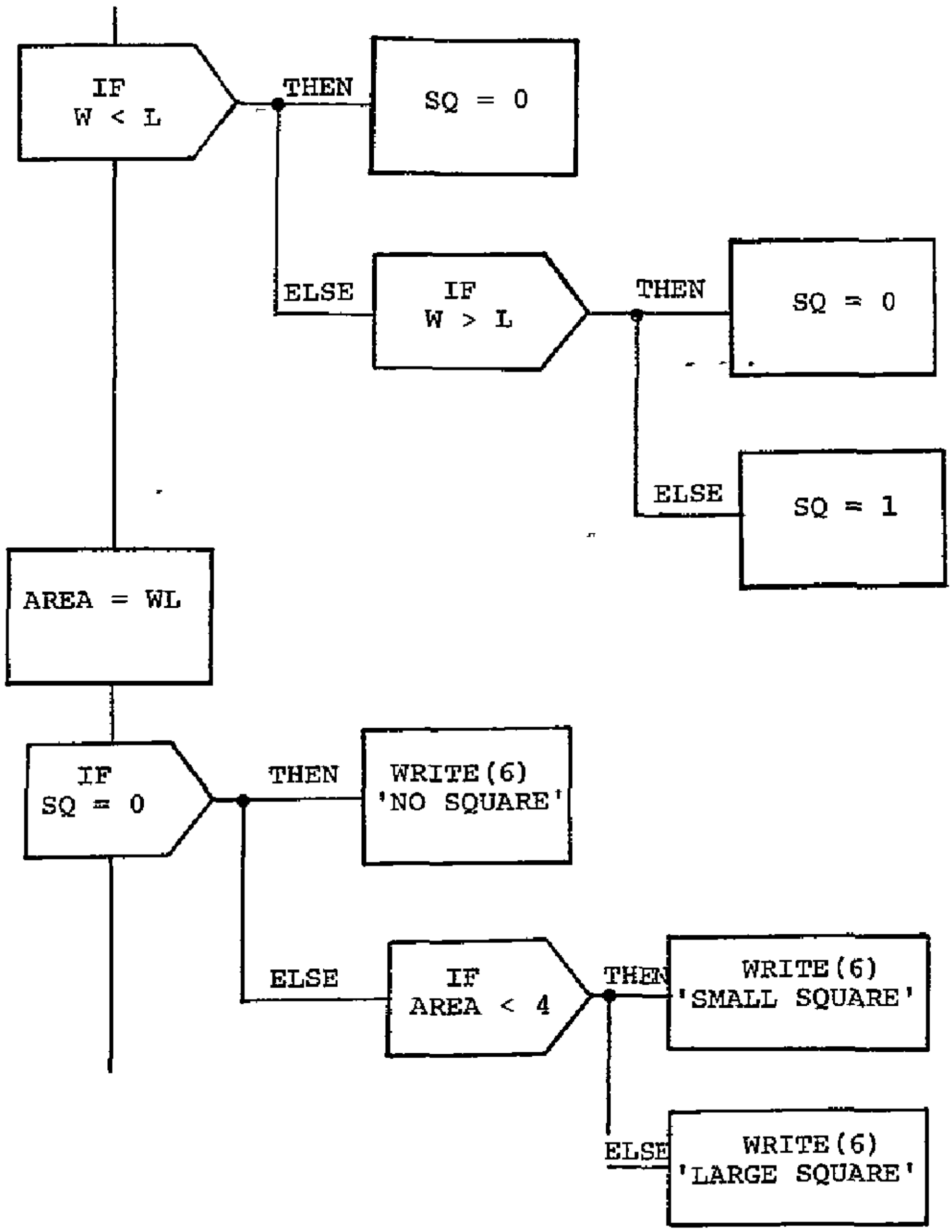
a) A < B
b) C > (NOT B)
c) (A ¬= B) & (C > = D)
d) ($\overline{S}$ ¬= $\overline{V}$) OR (B ¬> C)
e) $\overline{V}$ < $\overline{S}$
f) ($\overline{V}$ $\overline{V}$ < C) & (NOT($\overline{V}$ $\overline{S}$ < C))

Write the following descriptions in relational expressions

g) A is greater than B but less than C

h) The vector $\overline{V}$ is not equal to the vector $\overline{S}$ and C not less than D unless D is equal to 4

4 1D Write HAL/S code implementing this flowchart



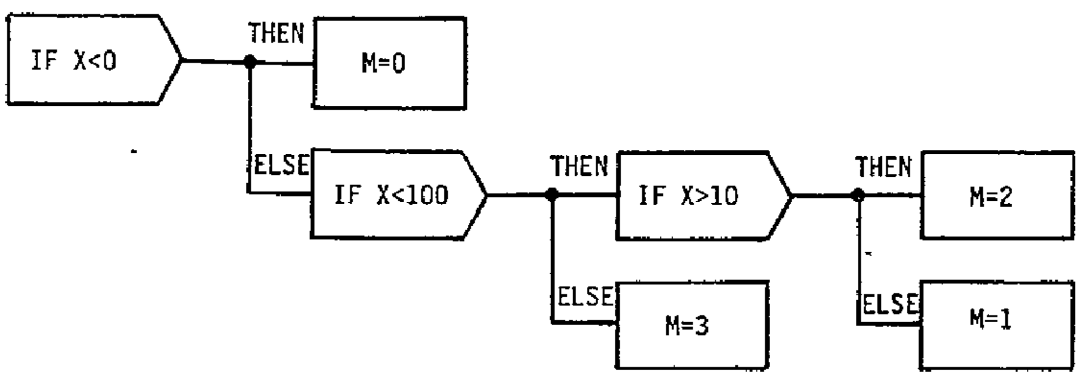
4 2 THE DO. END GROUP

A series of executable statements may be combined into a *do group*, which may then be used anywhere a single statement is required, e g , in the then clause

This allows, for example, the following coding of a four way decision

```
IF X < 0 THEN M = 0,
ELSE DO,
    IF X < 100 THEN DO,
        IF X > 10 THEN M = 2,
        ELSE M = 1,
    END,
    ELSE M = 3,
END,
```

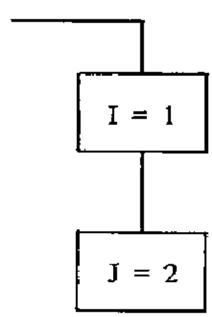
This example, which sets M to the order of magnitude of X, can be diagramed



Since it is only one statement, the entire sequence above could be further nested in IF or other compound statements

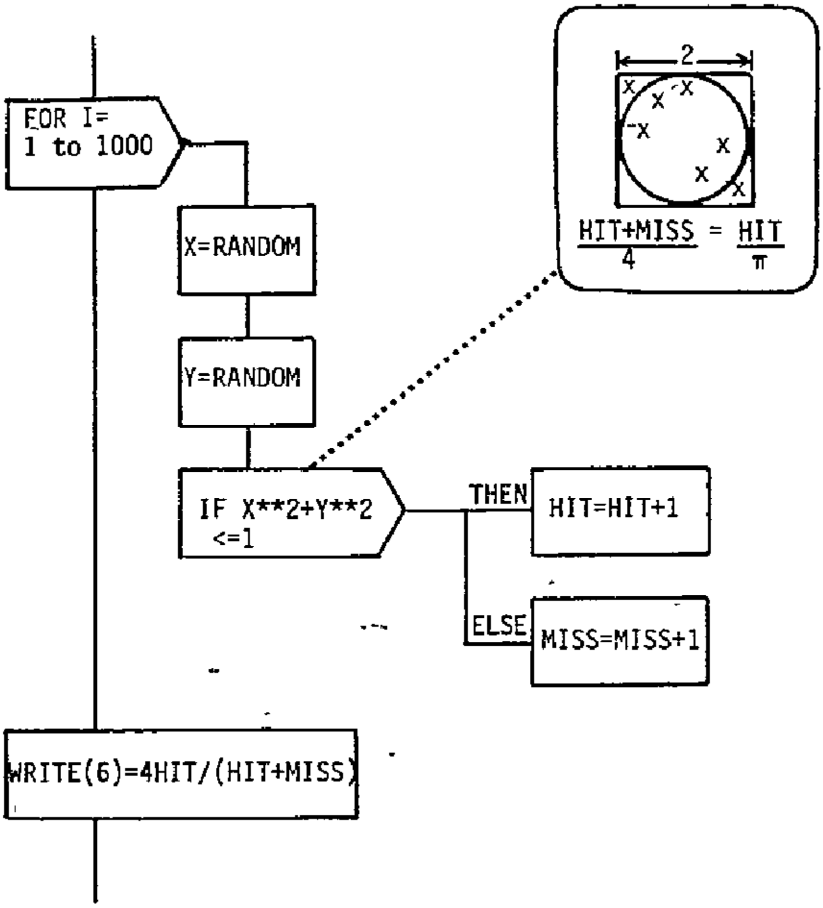
A do group consists of a DO statement, any number of executable statements* and an END statement, e g

```
DO,
    I = 1,
    J = 2,
END,
```



*Or TEMPORARY statements

The example below computes PI by an inefficient but illustrative algorithm



Here it can be seen that loops are shown with the same shaped symbol as IF statements HAL/S has several types of loops, all of which use the DO and END keywords The simplest type is shown above, and in the following compiled listing

```
M | DARTBOARD_APPROXIMATION:
M | PROGRAM;
M |    DECLARE SCALAR,
M |               X, Y;
M |    DECLARE INTEGER,
M |               I, HIT, MISS;
M |    DO FOR I = 1 TO 1000;
M |       X = RANDOM;
M |       Y = RANDOM;
E |            2    2
M |       IF X  + Y  <= 1 THEN
M |          HIT = HIT + 1;
M |       ELSE
M |          MISS = MISS + 1;
M |    END;
M |    WRITE(6) 4 HIT / MISS;
M | CLOSE;
```

Since the compiler used in preparing listings for this manual automatically indents programs to correspond to a structured flow, diagrams will not be provided for subsequent examples The same information is contained in the indenting as in the flow

The simple do group (without iteration) is classified as an *executable* statement No additional machine code is generated however An extra do group, like an extra set of parentheses, is sometimes used for clarity In the order of magnitude example, the else clause of the outer IF statement is bracketed by an unnecessary DO   END pair It is common practice to use a do group as a then or else clause even when it is not required by the syntax This allows for the possibility of later insertions

There is no way to branch into any part of a compound statement from outside the statement HAL/S has a GO TO statement, and any executable statement may be labelled, but restrictions are imposed A label inside a do group, a then clause or an else clause can only be used in GO TO statements which are themselves in the same group or clause

The do group has two uses Primarily, it allows the nesting of statements in tests and loops The secondary purpose is to define the scope of temporary data

The TEMPORARY statement is similar to the DECLARE statement It allows a temporary variable of any type to be created, as shown on the following page

```
M | EXAMPLE_2
M | PROGRAM,
M |    DECLARE VEL VECTOR,
M |            MY_FRAME MATRIX,
M |    DECLARE VECTOR,
M |               RESULT1, RESULT2, E,
  |
C |       .
  |
M |    DO,
M |       TEMPORARY V_PRIME VECTOR;
E |          -         *     -
M |       V_PRIME = MY_FRAME VEL;
E |          -             -
M |       RESULT1 = UNIT(V_PRIME);
E |          -        -      -
M |       RESULT2 = V_PRIME * E,
M |    END,
M | CLOSE EXAMPLE_2,
```

The vector, V_PRIME, exists only for the duration of the do group If the next do group contained

TEMPORARY S SCALAR,

S would probably occupy one of the storage locations that had just been used for V_PRIME

Temporary variables may be of any type and precision They may not, however, be initialized or given other minor attributes TEMPORARY statements can *only* be used within do groups Storage is allocated to temporary variables for the duration of the execution of the immediately enclosing do group The TEMPORARY statement informs the compiler of the range over which a variable will be needed The actual allocation and freeing of storage is done in an implementation-dependent manner

Very few restrictions are made on the use of temporary variables They may not be referenced at all from outside of the containing do group, otherwise, they are usable in all of the constructs introduced so far Proper use of the TEMPORARY statement can reduce a program's size without substantially increasing its execution time

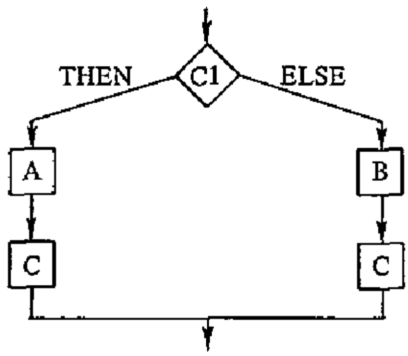
Exercises

4 2A Q A standard means of flowcharting is to use a system where

means a conditional execution along one of the paths (but not both!) depending on the condition represented by 'CX'

represents a DO END group without any conditional branches in the group

Consider the following flowhcart



A & B represent DO END groups each 5 statements long

C represents a DO END group 150 statements long

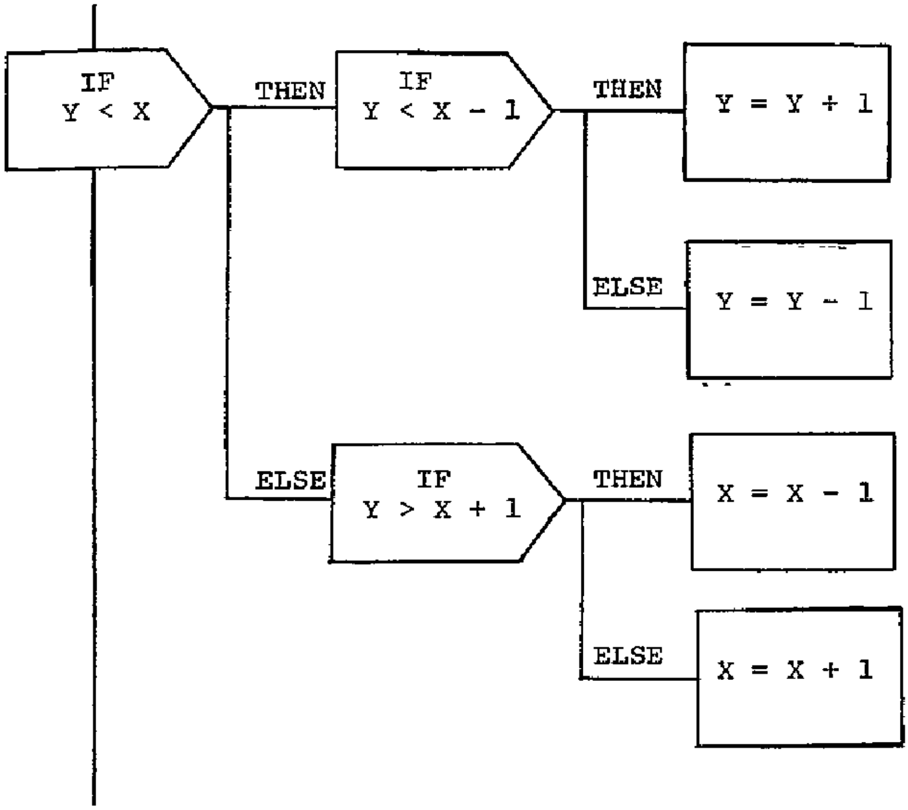
Rewrite this flowchart in a way to represent a shorter program

Can this change be made in a valid HAL/S program?

4 2B Write a HAL/S program that will solve a system of 2 equations in 2 unknowns as in problem 2–0

However, do not assume a solution exists, incorporate a test to insure that the denominator is not zero

4 2C Implement the following structured flowchart segment in HAL/S, using a few DO END groups as possible



4.2D Consider the following flowchart on the next page

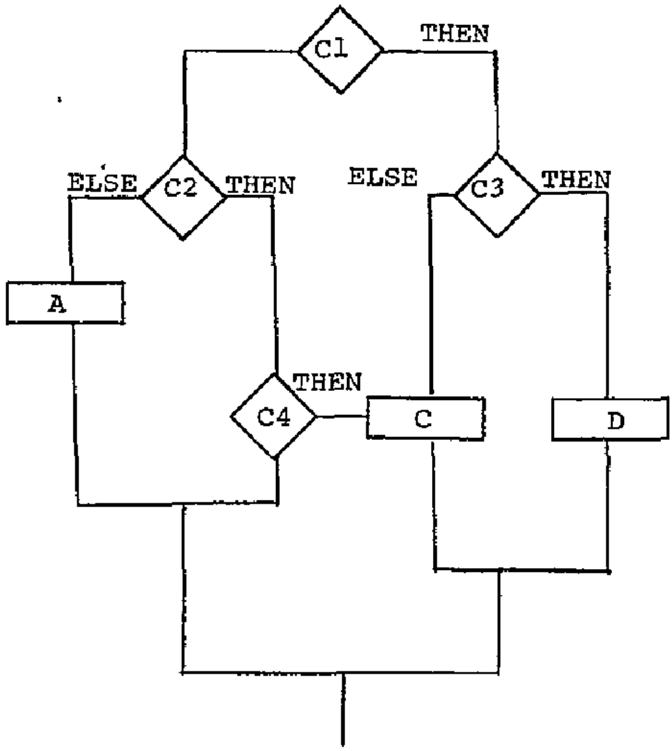
means a conditional execution on CX

means a single statement represented by M

a) There is a construct in the flowchart that is not legal in HAL/S What is it?

b) Rewrite the flowchart to eliminate the illegal construct, and write a code fragment corresponding to this structure Do not introduce or eliminate any conditions

c) How would a structured flowchart have made this mistake more easily available?



4 2E In problem 4 2D, we have seen that if the branches are to be preserved as shown, the code corresponding to

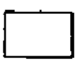
C

had to be repeated

Lets say that

C

is 250 statements long, whereas all the other



are still a single statement Rewrite the flowchart and the code to allow the code for

C

to appear only once

4 3 BOOLEANS

The test between IF and THEN in the IF statement is either a comparison or a *boolean expression* A boolean expression is a boolean variable or a combination thereof Both types of tests can be compounded using AND, OR, and NOT, but they cannot be mixed in one IF statement A boolean expression always can be converted to a comparison as in

```
M | EXAMPLE_3'
M | PROGRAM,
M |    DECLARE Q1 BOOLEAN,
E |       .
M |    IF Q1 = TRUE THEN
M |       DO,
C |          ..
C |          ..
C |
M |       END,
M | CLOSE EXAMPLE_3;
```

The IF statement can also be written IF Q1 THEN

TRUE is a boolean literal It is equivalent to BIN'1' or ON Booleans can take on one of only two possible values The other is written FALSE, BIN'0' or OFF The three different representations for each value allow mnemonic comparisons and assignments as in

```
DECLARE BOOLEAN INITIAL(OFF),
   POWER, READY,
IF READY = FALSE THEN POWER = OFF,
```

As the example shows, the form of the declare and assignment statements is the same for booleans as for other data types Booleans are annotated by the compiler with a " " on the E line

Booleans are used for flags, signal states and to optimize complex comparisons The keyword BOOLEAN is interchangeable with BIT(1) Bit strings of length greater than one are discussed in chapter 13 Since the concept of a "flag" is so common, the BOOLEAN keyword is included in the language and the applicable subset of BIT operations is presented here.

The preceding IF statement would normally be written

```
IF NOT READY THEN POWER = OFF,
```

NOT READY is a *boolean expression*, which can also be written ¬READY Boolean expressions are composed of boolean variables, the operators AND, OR, and NOT, and boolean functions The operators are defined via their truth tables below

A AND B

A \ B	TRUE	FALSE
TRUE	TRUE	FALSE
FALSE	FALSE	FALSE

A OR B

A \ B	TRUE	FALSE
TRUE	TRUE	TRUE
FALSE	TRUE	FALSE

NOT A

A: TRUE	A: FALSE
FALSE	TRUE

OR is the *inclusive* or operator. Exclusive or is provided as a built-in function,

```
IF XOR(A,B) THEN
```

but the equivalent statement,

```
IF A ¬= B THEN
```

is preferred

There are sixteen possible distinct binary operators on booleans These include AND, OR, and NOT as well as exclusive or, the bi-conditional, etc Any of them can be expressed by a combination of AND, OR and NOT *Any* boolean expression can be converted to an equivalent boolean expression using only NOT and one of the other two One such transformation is expressed by DeMorgan's rules

```
A AND B = NOT(NOT A OR NOT B)
```

and

```
A OR B = NOT(NOT A AND NOT B)
```

For another example, XOR(A,B) could also be written "A AND(NOT B) OR (NOT A) AND B"

The expression A&(¬B)|(¬A)&B is the same as "A exclusive-or B", or "A is not equal to B" Because AND has higher precedence than OR, the expression is interpreted as

```
(A&(¬B)) OR ((¬A)&B)
```

The boolean operators, AND, OR, and NOT, have considerable similarities to the arithmetic operators, multiplication, addition and negation, respectively This results in the convention that A&B | C&D is interpreted as the OR (logical sum) of two ANDs (logical products)

Consider the following example of the translation from an English statement of a condition to a boolean expression

English If the power is on and either it is not overheated or the override is set, and either switch 6 is on or it is off and switch 7 is set

HAL/S Power & (not overheated or override) & (switch 6 or (not switch 6 and switch 7))

Careful study of the English form may fail to reveal how the precedence is communicated, but most readers will see the correspondence between the two forms Symbolic logic shows that while there are a number of reliable rules for translation, much rests on the readers understanding of the situation to which an assertion applies

The boolean expression above is written with the minimum number of parentheses, taking advantage of the precedence of NOT over OR and AND The expression, (NOT SWITCH 6 and SWITCH 7), has the truth table

		SWITCH 6	
SWITCH 7	ON	FALSE	TRUE
	OFF	FALSE	FALSE

and is equivalent to

((NOT SWITCH #6) AND SWITCH 7)

In summary,

Precedence of boolean operators	
First	NOT
	AND
Last	OR

In addition to the test in an IF statement, boolean expressions may be used in assignment statements (the left hand side must also be boolean), in comparisons with other boolean expressions, and in WHILE and UNTIL loops (as described in the next chapter) Boolean expressions may appear in WRITE statements, boolean variables may be read

No other data type is automatically converted to boolean, and boolean is not automatically converted to any other type Booleans cannot be used in arithmetic expressions, and arithmetic variables cannot be used in boolean expressions The concept of precision does not apply to booleans, but *bit strings* may be viewed as sets of booleans on which operations can be performed in parallel

Both types of test in the IF statement can be written using the AND, OR, and NOT operators These operators combine either comparisons or booleans via precedence rules like those of arithmetic Parentheses can be used to override the normal precedence When *comparisons* are combined, each should be parenthesized

```
IF(I < 0) OR (I > 9) THEN
```

In boolean expressions, the precedence rules make most parentheses unnecessary, an exception is as in

```
IF A OR (NOT B) THEN
```

It is *not* possible to combine comparisons and booleans in a single expression If a statement (or group) is to be executed based on both a boolean and a comparison, the test should be written

```
IF (CHECKING = TRUE) AND (I < 0) THEN I = –I,
```

or as

```
IF CHECKING THEN IF I < 0 THEN I = –I,
```

Exercises

4 3A For each of the following, tell whether it is a boolean expression, a relational expression, or illegal For the boolean expressions, tell whether the value is TRUE or FALSE, for the relational expression, tell whether or not the condition is satisfied Assume that

A, B are INTEGER

$\overline{V}$, $\overline{S}$ are 3-vectors

UPFLG, TRFLG are booleans

A = 12 B = 6

$\overline{V}$ = (2 4 6) $\overline{S}$ = (3 4 7)

UPFLG = TRUE TRFLG = FALSE

a) UPFLG = TRFLG

b) NOT UPFLG

c) NOT($\overline{V}$ = $\overline{S}$)

d) NOT TRFLG OR A > B

e) (A < B) = TRUE

f) VPFLG = TRUE

g) TRFLG & (¬UPFLG)

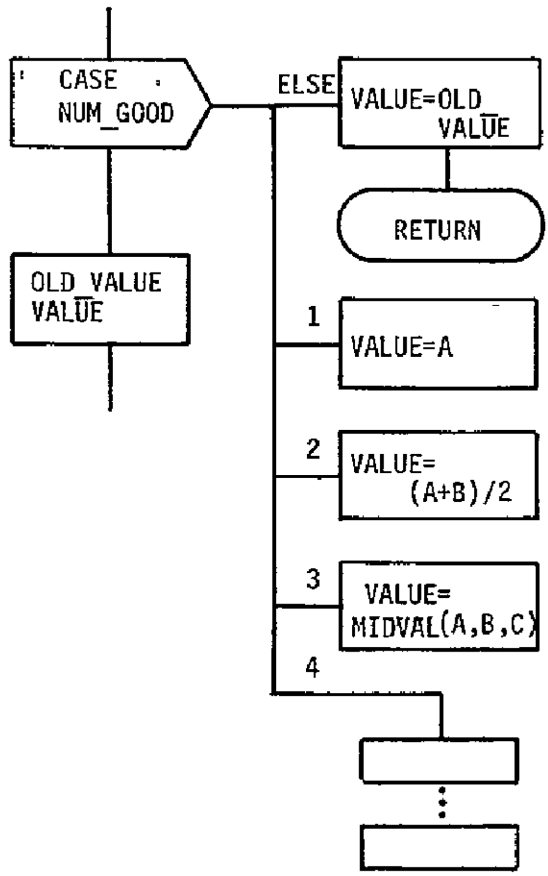
4 4 DO CASE AND GO TO

The most basic flow control constructs are loops, the IF statement, and the DO group These may be combined and compounded to implement complex structures of decisions The remaining flow control statements fill in a few gaps They are not as heavily used as the various forms of IF and DO

The IF statement allows a two-way decision based on a comparison or boolean An n-way branch based on an integer can be written with the DO CASE statement, for example

```
M   EXAMPLE_4
M   PROGRAM,
M      DECLARE SCALAR,
M                    A, B, C, D,
M      DECLARE NUM_GOOD INTEGER;
M      DECLARE SCALAR,
M                    VALUE, OLD_VALUE,

C       . . .

M      DO CASE NUM_GOOD,
M         ELSE
M            DO,
M               VALUE = OLD_VALUE,
M               RETURN,
M            END,
M         VALUE = A;
M         VALUE = (A + B) / 2,
M         VALUE = MIDVAL(A, B, C),
M         DO,

C         . . .

M         END;
M      END,
M      OLD_VALUE = VALUE,
M   CLOSE EXAMPLE_4,
```

This code sets VALUE to some combination of the variables A, B, C, and D It could be part of an algorithm for combining redundant values from a set of sensors The code is diagrammed



Any integer or scalar expression may appear after the word CASE The expression is evaluated and rounded to the nearest integer if necessary. In this example, if the expression, NUM_GOOD, is less than one or greater than four, the else clause is executed Otherwise, *one* of the four statements between the end of the else clause and the end of the DO CASE statement is executed The fourth statement (fourth *case*) is a DO group This is another instance of the use of DO END to combine several statements where one is required

Only one of the cases is executed After the selected case is done, control passes to the statement after the END statement which matches DO CASE (in this example, to the assignment of OLD_VALUE)

Each case may be any executable statement This includes assignment, IF THEN ELSE, I/O, a DO group, a loop, or a further DO CASE statement The only way to pass control to one of these nested statements is by executing the DO CASE header with an appropriate value of the expression

The compiler counts the cases and prints a case number to the extreme right of each in the listing. If an else clause is supplied, code is generated to compare the value of the case expression against the bounds, one, and the number of cases. If the expression is out of range, the else clause executes and control then continues after the END of the DO CASE. The else clause may be omitted entirely, in which case no checking is performed. Omission of the else clause may be risky, as under some circumstances, control can be passed completely out of the HAL/S program if the expression selects a missing case and no else clause is supplied.

In the example above, a RETURN statement appears in the else clause. When RETURN is used in a *program*, it is equivalent to transferring control to the close statement. It exits the program.

In chapter five, the EXIT and REPEAT statements are described. They are drawn in the same way.

Each is an unconditional transfer of control to a point defined by the structure of the program rather than to a user label. This completes the set of symbols used in a structured flow diagram.

The flow control statements include those described in this chapter, loops, and (in a sense) the statements for defining and invoking procedures and functions. Some of the real-time statements of chapter 12 may be thought of as transferring control, though there are conceptual differences.

The only other flow control statement in HAL/S is GO TO. The experience of a number of large HAL/S programming projects has shown that the GO TO statement is not necessary. It is provided chiefly for mechanical translations from other languages.

Once a degree of familiarity with the use of compound statements for flow control is achieved, it can be seen that the concept of a "conditional transfer" or branch instruction is merely a free form notation for flow diagrams: a line with an arrowhead. The restrictions on the use of GOTO correspond to the rules for a structured flow diagram presented in Section 9.1. GOTO's are not allowed at all in a proper structured flow, but HAL/S permits some exceptions:

1) between unnested statements in the same program or other block,
2) between statements nested at the same level in the same compound statement,
3) to a less deeply nested statement in the same block, *provided* that the target statement is not contained in any compound statement which does not also contain the GO TO statement.

Exercises

4 4A Rewrite the following code segment using the DO CASE statement

```
IF I = 0 THEN SCRAMBLE = 4,
    ELSE IF I = 1 THEN SCRAMBLE = 0,
        ELSE IF I = 2 THEN SCRAMBLE = 5,
            ELSE IF I = 4 THEN SCRAMBLE = 1,
                ELSE IF I = 5 THEN SCRAMBLE = 2,
                    ELSE SCRAMBLE = 3
```

5.0 LOOPS

A loop is a construct which causes a set of statements to be executed repetitively In HAL/S, a loop is a compound statement The statements to be iterated are nested within the loop Four types of loop are provided, so that the need for explicit backward branches (GO TO's) is virtually eliminated.

A loop is created in HAL/S by attaching one or more iteration control phrases to the simple DO END construct which was described in the previous chapter These iteration control phrases govern the number of times the loop is executed and may provide a counter or "loop control variable" which can be referenced from within the loop

The example below uses the most common type of loop, the iterative DO FOR, to compute the factorial of a number The number, N_MAX, is read from channel 5 and (N_MAX)! is written to channel six

```
M  FACTORIAL
M  PROGRAM,
M     DECLAPE INTEGER,
M                 RESULT, N_MAX, I,
M     READ(5) N_MAX,
M     RESULT = 1,
M     DO FOR I = 2 TO N_MAX BY 1;
M        RESULT = I RESULT,
M     END;
M     WRITE(6) RESULT,
M  CLOSE FACTORIAL,
```

Note that the body of the loop is executed repetitively until the control variable exceeds the final value specified after the keyword "TO" The example shown computes factorial (N_MAX) by doing N_MAX–1 multiplies by the control variable, which takes on the values 2, 3, 4, , N_MAX on successive iterations

In addition to the iterative DO FOR, other forms of iteration control are The discrete DO FOR, the WHILE phrase and the UNTIL phrase

These constructs probably are familiar to the reader who has used other algebraic programming languages, therefore, the remainder of the discussion in this chapter is primarily concerned with the limitations and restrictions of HAL/S loops, and the ways in which these constructs may be combined with each other and with other features of the language

5 1 THE ITERATIVE DO FOR STATEMENT

In the preceding example, the *loop body* is a single statement.

RESULT = I RESULT,

In general, the loop body may contain any number of executable statements Since the loop is constructed from a simple do group, the TEMPORARY statement may also occur in the loop body

In the phrase,

```
FOR I = 2 TO N_MAX BY 1,
```

I is termed the loop control variable, 2 is the *initial value*, N_MAX is the *final value*, and 1 is the *increment*

HAL/S places very few restrictions on these four parameters In particular, the loop control variable may be any single or double precision integer or scalar variable * For example, given the declaration

```
DECLARE A INTEGER,
        B INTEGER DOUBLE,
        C SCALAR,
        D SCALAR DOUBLE,
```

all four of the following combinations are permissable

```
DO FOR A = B TO C BY D,
DO FOR B = D TO C BY -1,
DO FOR C = D TO B/A,
DO FOR D = A-B TO A+B BY D,
```

There is one additional variation on the control variable, it may be either previously *declared* as in the previous example, *or* it may be "declared" within the DO FOR statement itself The latter is accomplished by placing the word TEMPORARY before the name of the loop control variable, as in

```
DO FOR TEMPORARY I = 2 TO N_MAX BY 1,
```

A TEMPORARY loop control variable created in this way may be used within the body of the loop in any way that a declared variable could be used, but outside of the loop the TEMPORARY variable does not exist Since the TEMPORARY control variable is effectively unDECLARED at the end of the loop, the memory locations occupied by the variable *may* be re-used, thus reducing the storage requirement of the program containing the DO FOR TEMPORARY Under some versions of the compiler a speed advantage may also result TEMPORARY control variables created in a loop are always single precision integers, their names must not duplicate declared data or other TEMPORARY variables in the same loop

The initial and final values and the increment used in an iterative DO FOR loop may be any arithmetic expression That is, each may be any expression which evaluates to a positive or negative, single or double precision integer or scalar value Each expression is evaluated only once, at entry to the loop Thus, if variables used in the expressions are modified within the loop, the iteration parameters of the loop are not affected

---

*Single precision integers are generally the most efficient

Note that in HAL/S the loop control variable may be a scalar, e g

```
DECLARE SCALAR, X, PI CONSTANT (3 14159),
DO FOR X = -PI TO PI BY 001,
   WRITE(6) X, SIN(X), COS(X), TAN(X),
END,
```

This code will produce a set of trigonemetric tables, giving sine, cosine, and tangent values for 2000 π different angles

The operation of the loop is the same as for integers On each iteration, the increment is added to the loop control variable, and if the final value is not exceeded, the loop body is executed The values taken on by X are $-\pi$, $-\pi+001$, $-\pi+002$, , etc The last value will not exactly equal π, because it is generated by a sequence of additions of 001

In the event that the result produced by adding the increment to the current value of the loop variable is not of the same type or precision as the loop variable, the usual rules for mixed mode assignment statements govern the conversion For instance, if the loop variable is an integer and the increment is less than one, rounding will occur on each pass through the loop In this case, if the increment is positive but less than 5, the value of the loop control variable would never be changed and the loop would never terminate

As previously stated, any or all of initial value, final value, and increment may be negative For instance, the loop below is functionally equivalent to the one in the original form of FACTORIAL

```
DO FOR I = N_MAX TO 2 BY -1,
   RESULT = I RESULT,
END,
```

When a negative increment is specified, the termination condition becomes "is the loop variable algebraically *less than* the final value?"

The only way that the body of a HAL/S loop may be entered is by execution of the DO statement which heads the loop, however, control may leave the loop by a variety of means other than the control variable exceeding the final value (e g , RETURN, EXIT, and GO TO statements, error conditions, etc) Since the increment has been added to the loop variable *before* the test against the final value is made, at normal exit from an Iterative DO FOR loop the loop variable will be greater than the specified final value (if the increment is positive) or less than the final value (if the increment is negative) This fact may be used to determine whether or not the loop was exited prematurely Use of this feature is illustrated in the sample below, which sets the variable NEG_PART to the number of the first negative component in a vector, or to zero if there is no negative component

```
DECLARE V VECTOR(5),
DECLARE NEG_PART INTEGER,
DO FOR NEG_PART = 1 TO 5,
   IF V$NEG_PART < 0 THEN EXIT,
END,
IF NEG_PART > 5 THEN NEG_PART = 0,
```

The EXIT statement is not fully described until later in this chapter, but in this case the meaning is intuitive If component number NEG_PART of V is less than zero, control exits from the loop (to the second IF test) Thus, NEG_PART will be greater than the 5 if only if the entire vector was examined without finding a negative value

Since it is necessary to test NEG_PART outside of the loop, a temporary loop control variable would not be appropriate in this example

To find the *second* negative component in a vector, the following loop could be added after the one above

```
DO FOR NEG_PART = NEG_PART TO 5,
   IF VSNEG_PART < 0 THEN EXIT,
END,
```

Since the initial and final values and the increment specified in an iterative DO FOR loop are evaluated only once (prior to the first iteration), there is no conflict in using NEG_PART both as a loop control value and as the initial value This new loop will continue where the first stopped

The "BY 1" cluase has been omitted above, since 1 is the most commonly used increment, it is the default and need not be specified

In summary, the iterative DO FOR takes four parameters, the first, the control variable, may be any previously declared arithmetic identifier or may be a TEMPORARY integer created within the DO FOR statement The initial value, final value and increment may be any arithmetic expression, the increment may be allowed to default to one by omitting the BY clause These expressions are evaluated prior to the first pass through the loop, and the results determine whether the loop is executed once, many times or not at all The loop terminates when the value of the control variable passes the final value specified in the TO clause Later in this chapter, we will see how the addition of a WHILE or UNTIL clause can modify the execution of a loop, but first we will examine another form of the DO FOR construct

Exercises

5 1A Consider the following code fragment where

I & N are integers,
S is scalar

```
N = 10,
S = 1,
DO FOR I = 1 TO 2 BY S,
   N = N + 1,
END,
```

What is the value of N on exit from the loop?

5 1B Consider the example where NEG_PART was set to the number of the first exponent of a vector less than zero, or zero if no elements were negative

Change the code given in the example to leave the number of the last negative component instead of the first

5.1C Consider the following code fragment where

N & I are integers

```
N = 9;
DO FOR I = 1 TO N BY 2;
    N = N + 1;
END;
```

What is the value of N on exit from the loop?

5.1D Consider the following code fragment where

A is a 5x5 matrix,
X and Y are integers

```
        X = 1;
ROWS    Y = 1;
LOOP    AS (X,Y) = 2;
        IF Y = 5 THEN GOTO OUT;
        Y = Y + 1;
        GOTO LOOP;
OUT     IF X = 5 THEN GOTO DONE;
        X = X + 1;
        GOTO ROWS;
DONE
```

a) What does this do?

b) Rewrite this using HAL/S iterative do for loops

5 2 THE DISCRETE DO FOR STATEMENT

In order to understand the utility of another type of DO FOR statement, consider the problem of recognizing prime numbers The code below sets a boolean variable, PRIME, to TRUE if NUM is prime and to FALSE otherwise (for simplicity, NUM is assumed to lie between one and one-hundred)

```
DECLARE PRIME BOOLEAN INITIAL(ON),
DECLARE INTEGER, NUM, I,
READ(5) NUM,
DO FOR I = 2 TO 10,
   IF REMAINDER(NUM,I) = 0 THEN PRIME = FALSE,
END,
```

This code produces the correct answer over the range 10 to 100, but is inefficient A better algorithm is to test the divisibility of NUM only by numbers which are themselves prime This can be conveniently expressed using the discrete DO FOR

```
DO FOR I = 2, 3, 5, 7,
   IF REMAINDER(NUM,I) - 0 THEN PRIME = FALSE,
END,
```

In this case, the loop is executed only four times, with the loop control variable, I, equal to two on the first pass, three on the second, five on the third and seven on the final iteration The reader may note that both programs contain a logical error in that the wrong result is obtained when NUM is equal to 2, 3, 5, or 7 This error will be fixed when the WHILE phrase is introduced in the next section of this chapter

The form of the discrete DO FOR is similar to the iterative version The discrete form specifies a list of values (expressions) to be assigned to the loop control variable rather than an algorithm (initial value, final value, and increment) for computing successive values

On each pass through the loop, the control variable is set to the value of one of the expression to the right of the equal sign The expressions are used from left to right on successive iterations of the loop, each one must evaluate to an integer or scalar value If the type or precision of any expression is different from that of the control variable, the usual rules for mixed code assignments are applied

Unlike the expressions in the iterative DO FOR, the expressions in the discrete DO FOR are not evaluated until the iteration of the loop on which they are to be assigned into the control variable This means that the value of the control variable on future passes through the loop can be changed by storing into variables referenced in the expressions from the body of the loop, e g

```
DO FOR I = 1, I, 2I, 3I,
```

At exit from a discrete DO FOR loop, the control variable retains the value of the last expression, unless the variable was TEMPORARY, in which case it is undefined

The remaining iteration control phrases, WHILE and UNTIL, provide for looping without the use of a control variable The next two sections of this chapter describe how to create a loop with these phrases, and show how they may be used to modify the effect of a DO FOR

5 3 THE WHILE CLAUSE

The WHILE clause may be appended to a simple DO . END group to create a loop, or it may be appended to either form of the DO FOR to introduce an additional condition for continuation of a loop The general form of the WHILE clause is

WHILE boolean expression

or

WHILE relational expression

The boolean or relational expression represents a condition for *continuation* of the loop, as long as it evaluates to the TRUE state, the loop continues For example

```
DO WHILE TRUE,
END,
```

is an infinite loop, whereas

```
DO WHILE X < 2
END,
```

continues until X≥2

The expression in the WHILE clause is evaluated prior to each execution of the first statement of the loop body If on any pass the expression evaluates to FALSE, the loop body is skipped and execution continues at the statement after the END of the DO WHILE or DO FOR . WHILE loop The DO WHILE loop is particularly useful when the number of iterations that should be made through a loop is not known in advance Consider, for example, Newton's method for computing the square root of a number, X. The method generates closer and closer approximations until the current approximation is "good enough" "Good enough" is defined as the point where the gain in accuracy from the last iteration was negligible (less than EPSILON) The example below illustrates the point

```
M   NEWTON_SQRT:
M   PROGRAM,
M      DECLARE X SCALAR;
M      DELCARE EPSILON CONSTANT(.001),
M      DECLARE SCALAR, OLD_APPROX, NEW_APPROX,
M      READ(5) X,
M      NEW_APPROX = X / 2;
M      OLD_APPROX = 0,
M      DO WHILE ABS(NEW_APPROX - OLD_APPROX) > EPSILON;
M         OLD_APPROX = NEW_APPROX;
M         NEW_APPROX = (OLD_APPROX + X / OLD_APPROX) / 2;
M      END;
M      WRITE(6) 'SQRT OF ', X, ' IS ', NEW_APPROX;
M   CLOSE NEWTON_SQRT;
```

Note that this program can be made to produce more accurate results (at the expense of greater execution time) merely by decreasing the constant EPSILON Note also that if X is equal to zero, the WHILE test will fail on the first evaluation and the correct answer will be produced but no division by zero will occur

When the WHILE clause is added to a DO FOR, a new loop is not created, but an additional condition for continuation of the existing loop is imposed This combination can be used to correct the deficiency in the PRIME program of Section 5 2 as shown below

```
DECLARE PRIME BOOLEAN INITIAL (TRUE), I INTEGER, NUM INTEGER,
READ(5) NUM,
DO FOR I = 2, 3, 5, 7 WHILE I < = SQRT (NUM),
   IF REMAINDER (NUM,I) = 0 THEN PRIME = FALSE,
END,
```

To see how the WHILE clause corrects the bug in the old version suppose X equals 3 Under the old version, REMAINDER (3,3) would be computed on the second pass through the loop, the result would be zero, and PRIME would be set to FALSE Now, however, prior to each execution of the loop body, the test "is I <= SQRT (NUM)?" is made On the first execution of the DO FOR statement, I is set to two Then I is compared with SQRT (NUM), which here is SQRT (3) or 1 732 Since it is *not* the case that 2 <= 1 732, the loop body is not executed and PRIME remains TRUE Adding the WHILE clause in this example also has the effect of determining the primeness of most numbers in fewer iterations For example, when X = 17 the loop is iterated only twice since 2 is less than or equal to SQRT (17) and 3 is less than or equal to SQRT (17), but the next number in the DO FOR, 5, is greater than SQRT (17)

EXERCISES

5 3A Change the code in the last example in Section 5 1 that finds the number of the first component < 0, eliminating the need for the line

```
IF V$NEG_PART < THEN EXIT
```

by using a WHILE clause

5 4 THE UNTIL CLAUSE

The general form of the UNTIL clause is

UNTIL boolean expression

or

UNTIL relational expression

It may be used in the same contexts as the WHILE clause with the simple DO END group or with either form of the DO FOR statement Unlike the WHILE clause, however, the UNTIL clause specifies a condition under which iteration of the loop is to *terminate* When it evalutes to TRUE, the loop terminates For example,

```
DO UNTIL 3 = 4,
END,
```

is an infinite loop, whereas

```
DO WHILE 3 = 4,
END,
```

is effectively a NO-OP (never executes) UNTIL is *not*, however, simply an inverse of WHILE for the following reason *An UNTIL clause never terminates a loop before the first pass through the loop body* This property of the UNTIL clause may be used to avoid the need to initialize variables used in the termination condition of a loop Suppose, for instance, that a program is to read vectors from channel 5 When a zero vector is read, the sum of the previous vectors is printed and another set is read The program is to run indefinitely

This could be expressed via two WHILE loops

```
DECLARE VECTOR, TOTAL, V,
DO WHILE TRUE,
   TOTAL = 0,
   V = VECTOR (1, 1, 1),
   DO WHILE V >= 0,
      READ(5) V,
      TOTAL = TOTAL + V,
   END,
   WRITE(6) TOTAL,
END,
```

In this example, the assignment

```
V = VECTOR (1, 1, 1),
```

is used to force V to be non-zero before the inner loop executes If this statement were not provided, the inner loop would not execute after the first iteration of the outer

The essential difficulty is that the inner loop written with WHILE will test the value of V *before* it has been read

If the UNTIL form is used for the inner loop, the initialization of V is not needed

```
DO WHILE TRUE,
    TOTAL = 0,
    DO UNTIL V = 0,
        READ(5) V,
        TOTAL = TOTAL + V,
    END,
    WRITE(6) TOTAL,
END,
```

Since the UNTIL clause cannot terminate the loop before the first iteration, the initial value of V is unimportant

When, as in this case, the UNTIL clause is used with a simple DO   END group, it is useful to conceive of the termination test as being done at the end of the loop (after the last statement of the loop body)

Like the WHILE clause, UNTIL may also be used as an additional condition on either type of DO FOR statement, as in

```
DO FOR I = 1 TO 10 UNTIL A$I = 0,
END,
```

This example is a loop (with no loop body) which sets I to the index of the first zero component in a vector, A However, since the UNTIL cannot terminate the loop on its first iteration, if AS1=0, the loop will continue to look for an additional zero

When used with a DO FOR statement, the UNTIL clause causes a test for termination on the second and all subsequent iterations of the loop, on the second through last iteration, the test is performed *after* the (DO FOR) loop control variable has been updated, but *before* the first statement of the loop body is executed

Exercises

5 4A Consider the problem of exercise 5 3A A proposed solution is shown below

```
DECLARE V VECTOR(5),
DECLARE NEG_PART INTEGER,
DO FOR NEG_PART = 1 TO 5 UNTIL V$NEG_PART < 0,
END,
IF NEG_PART < 5 THEN NEG_PART = 0,
```

Why is this not an acceptable solution?

5 5 EXIT AND REPEAT

The constructs already introduced in this chapter provide for the repeated execution of a loop body, and for a condition to be specified under which control is to exit from a loop These language features, however, only govern the execution of an *entire* loop body, the statements to be introduced in this section allow a *portion* of a loop to be repeated and for a termination test to be made at *any* point in the loop body rather than only at the beginning or end To see how these statements, EXIT and REPEAT, augment the other loop control statements, consider the following program

/* THIS PROGRAM READS A SERIES OF ANGLES EXPRESSED IN DEGREES, CONVERTS THEM TO RADIANS, AND KEEPS A RUNNING TOTAL ON EACH CYCLE IT PRINTS THE CURRENT TOTAL (IN RADIANS) AND THE TANGENT OF THE TOTAL ANGLE PRODUCED IT AUTOMATICALLY STOPS WHEN THE RUNNING TOTAL EXCEEDS 5 π, OR IF THE COMPUTATION OF THE TANGENT COMES TOO CLOSE TO A SINGULARITY */

```
M  TAN_SUMS
M  PROGRAM,
M     REPLACE CARDS BY "5",                          /*CARD READER IS DEVICE 5*/
M     PEPLACE LIST BY "6",                           /*PRINTER IS DEVICE 6*/
M     DECLARE SCALAR,
M                 X,
M                 TOTAL INITIAL(0),
M                 PI CONSTANT(3 1415926),
M                 RAD_PER_DEGREE CONSTANT(PI / 180),
M                 SHIFT CONSTANT(PI / 2),
M     DO UNTIL TOTAL > 5 PI,
M        READ(CARDS) X,
M        TOTAL + TOTAL + X RAD_PER_DEGREE;
M        IF MOD(TOTAL - SHIFT, PI) < .001 THEN
M           EXIT,
M        WRITE(LIST) TOTAL, TAN(TOTAL);
M     END,
M  CLOSE TAN_SUMS,
```

In this example, the statement

"IF MOD(TOTAL–SHIFT,PI) < 001 THEN EXIT,"

causes the loop to terminate if TOTAL gets within 001 of $\pi/2$, $3\pi/2$, etc If the EXIT statement is executed, control passes to the statement after the END of the loop (i e to the CLOSE statement)

The program might be more useful, however, if instead of terminating at a singularity, it allowed the user to enter another value and continued This can be accomplished by changing the EXIT statement to REPEAT as follows

IF MOD(TOTAL–SHIFT,PI) < 001 THEN REPEAT, If the REPEAT statement is executed, control will return to the top of the loop, where TOTAL will be compared with 5 PI If this test fails (TOTAL is not greater than 5 PI), the loop body will be re-executed

This example shows how EXIT may be used to insert a completion test at any point in the loop body, and how REPEAT may be used to cause iteration of a portion of the loop body

The general form of the EXIT statement is

EXIT,

or

EXIT label,

When used without a label, EXIT causes an unconditional transfer of control out of the nearest enclosing DO END group (i e to the statement following the END of the immediately enclosing loop or simple DO END group) If a label is supplied, it must match the label on some DO END group in which the EXIT statement is nested, this form causes transfer of control out of the corresponding loop or simple DO . END group Similarly, the general form of the REPEAT statement is

REPEAT,

or

REPEAT label,

Unlike the EXIT statement, however, REPEAT applies only to loops When used without a label it causes repetition of the nearest enclosing DO WHILE, DO UNTIL, or DO FOR loop Repetition, in this sense, means that the loop control variable (if any) is updated, the termination condition (if any) is re-evaluated, and if the conditions for termination are not met then control is passed to the first statement of the loop body Thus, the presence of a REPEAT statement in a loop does not change the number of iterations of the loop, but does determine which portion of the loop body is executed on each iteration

EXIT and REPEAT are controlled forms of GO TO The location to which control is transferred is defined by the structure of the program Thus, whenever these statements are used, their *functions* are what their names imply EXIT always "gets out of" a compound statement REPEAT always repeats a loop GO TO, on the other hand, has a variety of functional uses When GO TO is used, the reader must find the corresponding label to gain any idea of the effect of the GO TO

The following code fragment uses arrows to illustrate the transfer of control caused by EXIT and REPEAT

```
M   SAMPLE_FLOW
M   PROGRAM,
M      DECLARE INTEGER,
M                 I, J, K, L, M,
M    ->DO UNTIL FALSE,
M         IF I = 0 THEN
M            DO,
M               J = 0;
M    <-<-<-<-REPEAT,
M            END,
M         ELSE
M            EXIT,->->->->
M   ->LOOP2:
M         DO FOR K = 1 TO 10,
M     LOOP3  DO FOR L = M, N, M + N,
M               IF J = 0 THEN
M    <-<-<-<-REPEAT LOOP2;
M               ELSE
M    <-<-<-<-      EXIT,
M    ->END;
M         END,
M      END,<-
M   CLOSE SAMPLE_FLOW,
```

Since REPEAT applies only to loops, its effect is not changed by placing it in a simple DO ... END group. This fact can be used to make the TAN_SUM program more informative as shown below

```
IF MOD(TOTAL–SHIFT,PI) < .001 THEN DO,
   WRITE(LIST) 'TANGENT UNDEFINED',
   REPEAT, /* READ ANOTHER ANGLE */
END,
```

Exercises

5.5A Given

a) DO FOR X = 1 TO 100,

```
      EXIT,

   END,
```

and,

```
b) DO FOR X = 1 TO 100,


      REPEAT,
   END,
```

Assume that the EXIT and REPEAT are executed in some conditional branch sometime during the execution of the loop These are the only EXIT's and REPEAT's in the loops and there are no branches out of the loops

What can be said about the value of the control variable 'X' in a) and b) above when the first statement after the END is executed?

End Of Chapter Problems

5A Write a HAL/S program to use Simpson's rule to approximate the area under the curve $y = \sqrt{x}$ using smaller and smaller segments, delta The process continues until the area resulting from (delta/2) size segments differs from the result obtained using delta by less than (100 EPSILON) percent

Read the limits of integration from channel 5 in scalar form, and write the resulting area out on channel 6

Remember, Simpson's Rule is

$$\int_{INITIAL}^{FINAL} f(x)dx = \frac{delta}{2}[f(initial)+2f(INITIAL+DELTA)+ \cdots +2f(FINAL-DELTA)+f(FINAL)]$$

Include any assumptions you make

5B Consider the following code

```
       PROBLEM_PROG  PROGRAM,
       DECLARE INTEGER,
          NUMBER INITIAL(3),
          DIVIDER,
       TEST_INIT  DIVIDER = 2,
TEST   IF MOD (NUMBER, DIVIDER) = 0 THEN GO TO LOSE,
       DIVIDER = DIVIDER + 1,
       IF DIVIDER = NUMBER THEN GO TO WIN,
LOSE   NUMBER = NUMBER + 1,
       IF NUMBER = 500 THEN GO TO DONE,
       GO TO TEST,
```

```
WIN    WRITE(6) NUMBER,
       NUMBER = NUMBER + 1,
       IF NUMBER = 500 THEN GO TO TEST_INIT,
DONE CLOSE PROBLEM_PROG,
```

MOD(a,b) yields a(mod b), the remainder when the greatest integral multiple of b less than a is subtracted from a

a) What does this program do?

b) Rewrite it using do for   end loops so that the program is easier to read

6 0 ARRAY$

An ARRAY is an ordered set of variables of identical type which are accessed by a single name Arrays are completely distinct from vectors and matrices The primary uses of ARRAYs in HAL/S are

1) For performing identical operations on similar data as in

```
DECLARE IMU_ STATUS ARRAY(4) INTEGER,
DO FOR I = 1 TO 4,
   IF IMU_STATUSSI NOT = 0 THEN CALL RING_BELLS,
END,
```

2) For maintaining a history of previous data values as in

```
DECLARE ALT_HISTORY ARRAY(100) SCALAR DOUBLE,



CYCLE = CYCLE+1,
ALT_HISTORYSCYCLE = NEW_ALTITUDE,
```

and

3) For maintaining tables of all sorts, as in

```
DECLARE DAYS_PER MONTH ARRAY(12)
   INTER INITIAL(31,28,31,30,31,30,31,31,30,31,30,31),
```

HAL/S allows arrays of any data type, however, the most frequently used are single dimensioned arrays of INTEGERs and SCALARs like those in the examples above Therefore, the basic concepts of declaring and subscripting arrays will be thoroughly examined in this context before arrays of other datatypes and more advanced array operations are discussed

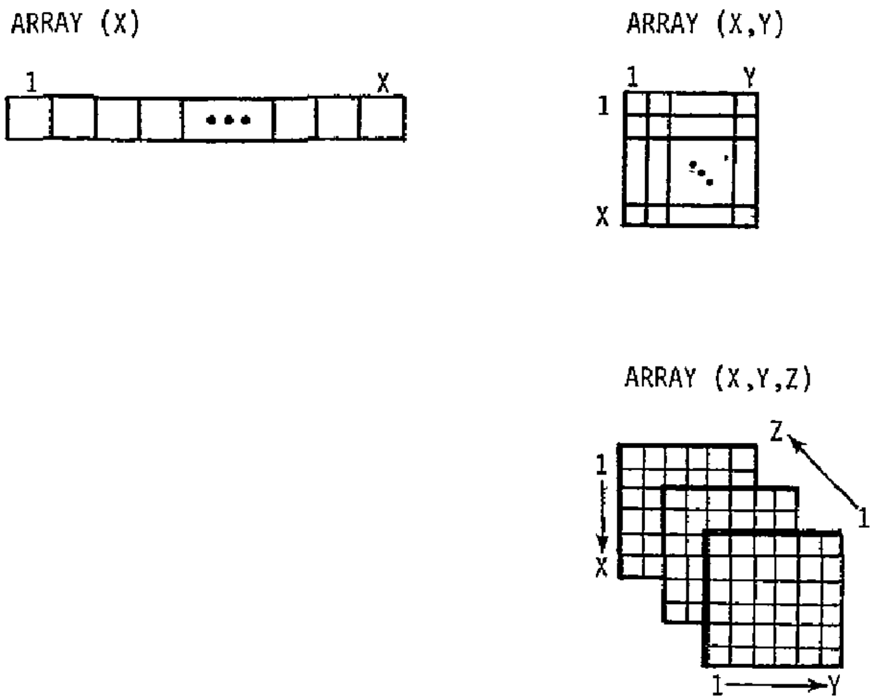
6 1 ARRAYS OF INTEGERS AND SCALARS

Arrays are created using the ARRAY keyword in the DECLARE statement, a parenthesized compile-time expression or list of expressions must follow the ARRAY keyword to denote the size of the array. Arrayness is an *attribute* of a variable of some data type rather than a new type Hence, given the statements

```
DECLARE A ARRAY(3) SCALAR,
DECLARE V VECTOR(3),
```

the datatype of A is SCALAR and the type of V is VECTOR even though both consist of three single precision SCALAR elements

Following the word ARRAY is a parenthesized list of dimensions Each dimension is described by a compile-time expression, which is the size of the dimension and the index of the last element X, Y, and Z in the next figure could be REPLACEd with any integral value up to an implementation-dependent limit



Arrays are initialized in the same manner as VECTORs and MATRIXs, a list of values is provided in parenthesis following the keyword INITIAL or CONSTANT The special characters * and # may be used for partial initialization and repetition as before. Thus,

```
DECLARE A ARRAY(5) INTEGER INITIAL(3,5,14,2,0),
```

creates

$A \equiv (3,5,14,2,0)$

and,

```
DECLARE B ARRAY(12) SCALAR INITIAL(0,1,-1,SQRT(2),
    -SQRT(2),4#2,*),
```

creates

$B \equiv (0,1,-1,\sqrt{2}, -\sqrt{2},2,2,2,2,?,?,?)$

Since it is often desirable to initialize an entire array to the same value, HAL/S also allows an initial (or constant) list to consist of only one value, in this case every element of the array is set to the value provided Thus the forms

```
DECLARE X ARRAY(5) INTEGER INITIAL(5#0),
```

and

```
DECLARE X ARRAY(5) INTEGER INITIAL(0),
```

are equivalent Finally, the ARRAY attribute may also be "factored" or specified only once in a DECLARE statement which creates multiple arrays as shown below

```
DECLARE ARRAY(3),
   GYRO_INPUT INTEGER,
   ATT_RATE SCALAR DOUBLE,
   SCALE CONSTANT(.013, 026, 013),
```

The arrays declared above might serve as the inputs and outputs of a simple program which does linear scaling of data read from an accelerometer assembly Assume that GYRO_INPUT contains three values which represent the rates of vehicle rotation along the pitch, roll, and yaw axes A simple routine to convert the data to more convenient units and data representation might be

```
DECLARE N INTEGER,
DECLARE BIAS SCALAR INITIAL(57 296),
DO FOR N = 1 TO 3,
   ATT_RATE$N = SCALE$N GYRO_INPUT$N + BIAS,
END,
```

In this example, the various arrays are subscripted in the same fashion as VECTORs, and in general, the same rules apply The subscript of a one-dimensional array may be any arithmetic expression which evaluates to a number between one and the size of the array If the expression does not produce an integral result, it is rounded to the nearest integer An array *element*, such as ATT_RATE$1* or SCALE$(N+2), may be used in any context in which a simple variable of the same data type can be used For instance, given two SCALAR ARRAY(10)'s, A and B, the following statements are all legal

```
A$1,A$2 = SIN(A$3),
A$(B$(A$3)) = 29,
DO UNTIL A$1 = A$2,
IF A$N < A$(N+1) THEN
```

*Some readers may wish to review the discussion of single and multi-line formats in Chapter 2

Another example of the use of arrays appears in example 1 This program determines the minimum, maximum, and average time required to invert a 5x5 MATRIX containing random data

```
M   EXAMPLE_1:
M   PROGRAM;
M      DECLARE M MATRIX(5, 5);
M      DECLARE N MATRIX(5, 5);
M      DECLARE TIME ARRAY(100) SCALAR INITIAL(0);
M      DECLARE SCALAR,
M                 TMIN, TMAX, TMEAN;
M      DECLARE INTEGER,
M                 I, J, K;
M      DO FOR I = 1 TO 100;
M         DO FOR J = 1 TO 5;
M            DO FOR K = 1 TO 5;
M               M    = RANDOM;
S                J,K

M            END;
M         END;
M         TIME  = RUNTIME;
S             I

E         *   *-1
M         N = M  ;
M         TIME  = RUNTIME - TIME ;
S             I                 I

M      END;

C    NOW PROCESS THE HUNDRED SAMPLES IN THE ARRAY [TIME]

M      TMAX, TMEAN, TMIN = TIME ;
S                              1

M      DO FOR I = 2 TO 100;
M         TMEAN = TMEAN + TIME ;
S                             I

M         IF TIME  > TMAX THEN
S                I

M            TMAX = TIME ;
S                       I

M         IF TIME  < TMIN THEN
S                I

M            TMIN = TIME ;
S                       I

M      END;
M      TMEAN = TMEAN / 100;
M   CLOSE EXAMPLE_1;
```

In this example, two previously undefined functions, RANDOM and RUNTIME are invoked RANDOM is used to set the matrix to a set of pseudo-random numbers, and RUNTIME returns the value of the system's real time clock

It may be noted that the min, max, and mean could have been computed within the main loop without saving all of the values in an array. Saving the data allows additional statistics, such as the median to be computed (see exercises) This method of obtaining timing data may be inaccurate if the time required to read the clock is significant

HAL/S provides for multi-dimensional arrays These are typically used for ease of subscripting and to contribute to the readability of a program by logical grouping of data For example, suppose that instead of one accelerometer assembly as described earlier, there were four of them, for reasons of fault-tolerance Then, we might declare the input data as a two-dimensional array

```
DECLARE GYRO_INPUT ARRAY(4,3) INTEGER,
```

Now, GYRO_INPUTS$(3,2) is the second measurement from the third unit, GYRO_INPUT$(1,1) is the first measurement from the first unit, and GYRO_INPUTS$(1,*) is all the data from unit one, i e the same three measurements we had before The use of an asterisk to indicate "all of a particular dimension" is the same as in VECTOR/MATRIX subscripting, the #, TO, and AT forms also apply Thus, GYRO_INPUTS$(*,1) is an array containing the first measurement from each of the four accelerometer units, and GYRO_INPUTS$(2 AT #–1,*) is a 2x3 array containing three measurements from each of the last two units In the next section we will see how these complex subscripts are used, but first we shall examine the general form of multi-dimensional arrays (and finish processing the redundant accelerometer data along the way)

The maximum number of dimensions in an array depends on the particular HAL/S compiler in use. All present HAL/S compilers allow from one to *three* dimensions In declaring an array, the number of dimensions is denoted by the number of expressions in parenthesis following the keyword ARRAY Thus,

```
DECLARE A ARRAY(5,9,4) SCALAR,
        B ARRAY(180) SCALAR,
```

creates two arrays of 180 scalars, but A is 3-dimensional while B is linear The first element of B is B$1, whereas the first element of A is A$(1,1,1) Initialization works the same as in single dimensional arrays Either a list of values containing one value per array element may be provided, or a single value may be assigned to all elements Thus, the array A may be initialized as

```
DECLARE A ARRAY(5,9,4) INITIAL(0),
```

or

```
DECLARE A ARRAY(5,9,4) INITIAL(180#0),
```

If we want A to be all zero except that A$(*,*,3) = –1,-the following initial list can be used

```
INITIAL(5#(9#(0,0,–1,0)))
```

To understand why this is correct, it is necessary to know that HAL/S stores arrays in "Row-major order" This means that the values in the initial list are assigned in the following order

A$(1,1,1) = value 1
A$(1,1,2) = value 2
A$(1,1,3) = value 3
A$(1,1,4) = value 4
A$(1,2,1) = value 5
A$(1,2,2) = value 6
et cetera

The way to remember this fact is by noting that the right-most index is incremented the most rapidly

Now, to illustrate the usefulness of multi-dimensional arrays, we will return to the examples of four accelerometer assemblies The entire set of twelve measurements could be processed as shown below

```
M   EXAMPLE_2-
M   PROGRAM,
M      DECLARE GYRO_INPUT ARRAY(4, 3) INTEGER,
M      DECLARE ATT_RATE ARRAY(4, 3) SCALAR,
M      DECLARE SCALE ARRAY(3) CONSTANT(.013, .026,  013);
M      DECLARE BIAS SCALAR INITIAL(57 296);
M      DO FOR TEMPORARY I = 1 TO 4,
M         DO FOR TEMPORARY J = 1 TO 3,
M            ATT_RATE    = GYRO_INPUT    SCALE  + BIAS,
S                    I,J             I,J      J

M         END,
M      END,
M   CLOSE EXAMPLE_2,
```

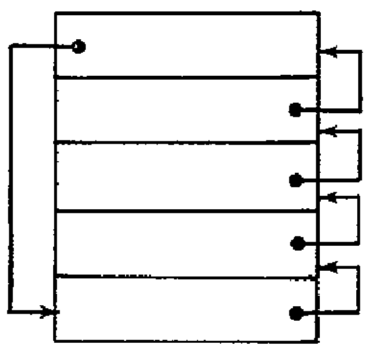
In this code, SCALE is still declared as a array of *three* Since the four instruments are identical, there is no need to keep four sets of scale factors Note, however, that if GYRO_INPUT had been declared as a linear ARRAY(12), we would have to either make the SCALE array also of size twelve, or introduce more complex code to associate the right scale factor with each of the twelve measurements Thus, a two dimensional array may be a mechanism for performing identical operations on a set of similar linear arrays just as a linear array may be used to perform identical operations on a set of similar integers or scalars

6 1 1 Additional Examples

1) Do a matrix multiply, M1 = M2 M3, with M1, M2 and M3 declared as ARRAYs rather than as matrices

```
M   EXAMPLE_3:
M   PROGRAM;
M      DECLARE ARRAY(3, 3),
M                 M1, M2, M3,
M      DECLAPE INTEGER,
M                 ROW, COL,
M      DO FOR ROW = 1 TO 3,
M         DO FOR COL = 1 TO 3;
M            M1          = M2       M3       + M2       M3       + M2       M3       ,
S              ROW,COL      ROW,1    1,COL     ROW,2    2,COL     ROW,3    3,COL
M         END;
M      END;
M   CLOSE EXAMPLE_3,
```

2) Rotate the contents of an array of five scalars as shown by the illustration



```
M   EXAMPLE_4:
M   PROGRAM,
M      DECLARE A ARRAY(5) SCALAR DOUBLE;
M      DECLARE TEMP SCALAR DOUBLE,
M      TEMP = A ,
S            1
M      DO FOR TEMPORARY T = 1 TO 4,
M         A  = A    ;
S          T    T+1
M      END,
M      A  = TEMP;
S       5
M   CLOSE EXAMPLE_4;
```

3) Find the square root of the mean of the squares of all the values in an array of 100 scalars

```
M | EXAMPLE_5:
M | PROGRAM;
M |    DECLARE A ARRAY(100),
M |    DECLARE RMS SCALAR,
M |    DECLARE TOTAL SCALAR DOUBLE INITIAL(0),
M |    DO FOR TEMPORARY N = 1 TO 100,
E |                               2
M |       TOTAL = TOTAL + A ,
S |                               N
  |
M |    END,
M |    RMS = SQRT(TOTAL / 100);
M | CLOSE EXAMPLE_5;
```

Exercises

6.1A Which of the following declarations lists are legal?

If they are legal, what do they create?

If not legal, why not?

a) DECLARE X INTEGER INITIAL(3),
   DECLARE LIST_ONE ARRAY(X) SCALAR INITIAL(6# 1),

b) DECLARE X CONSTANT(4),
   DECLARE ARRAY(X),
      LIST_ONE SCALAR INITIAL(4# 2),
      LIST_TWO INTEGER,

c) DECLARE LIST_THREE ARRAY(18) SCALAR INITIAL(10#.1,*),

d) DECLARE LIST_FOUR ARRAY(9,3) SCALAR INITIAL (3# 1,
      3#(3#.2),*),

e) DECLARE LIST_FIVE INTEGER ARRAY(6),

6 1B a) In example 1 in the text, the minimum, maximum, and mean times required to invert a 5x5 matrix are computed. Modify the code of the example to include a computation of the standard deviation, defined as follows

$$\sigma = \sqrt{\frac{\Sigma_i (X_i - \overline{X})^2}{n}}$$

where $\overline{X}$ is the mean value of the time, and n is the number of samples

b) An alternate definition for standard deviation, easily shown to be equivalent to the above, is

$$\sigma = \sqrt{\frac{\Sigma_i X_i^2}{n} - \left(\frac{\Sigma_i X_i}{n}\right)^2}$$

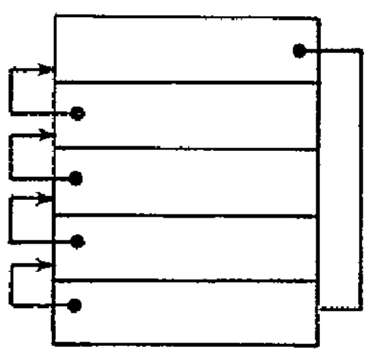
Using this formulation, it is possible to compute the standard deviation without saving all the time values in an array. Rewrite the program of part a), eliminating the array of time values. Is it possible to compute the median value without saving all the values?

6 1C In example_2, GYRO_INPUT and ATT_RATE are declared ARRAY(4,3)

The text states that if these variables were declared ARRAY(12) either SCALE would have to be declared ARRAY(12) or more complex code would be needed

Keeping SCALE declared an ARRAY(3), modify the code given for example_2 such that GYRO_INPUT and ATT_RATE are declared ARRAY(12), while still keeping the basic structure of the code given.

6 1D Instead of the modification of the array shown in EXAMPLE_4, write code that will perform the following modification of array A.



6.2 OPERATIONS ON ENTIRE ARRAYS

Most of the examples in this chapter have relied upon the iterative DO FOR loop to sequence through the elements of an array. Commonly, the loop has been used to apply one statement to each array element, i.e.

```
DO FOR I = 1 TO ARRAY_SIZE BY 1,
          (statement)
END,
```

Since this type of operation is so common, HAL/S provides a mechanism for combining these three statements into one. For example, to add one to each element of an array could be coded as follows

```
DECLARE A ARRAY(10) INTEGER,
DECLARE I INTEGER,
DO FOR I = 1 TO 10,
   A$I = A$I + 1,
END,
```

or, by eliminating the subscript and the loop, could be recoded as shown below.*

```
DECLARE A ARRAY(10) INTEGER,
A = A+1,
```

This assignment is an example of an *arrayed statement*. A statement which operates on all the elements of an array. Here the effect is the same as in the form with a loop, i.e. each element of A is incremented. In general, an arrayed assignment statement results whenever the target (left-hand side) of the assignment is an array. There are two possibilities for the expression to the right of the = sign. It may be either a simple expression (e.g. "1" or "SQRT(3)") or it may be an *arrayed expression* (e.g. "[A] + 1" or "[A]/2"). In the former case, every element of the target array is set to the value of the expression. In the latter case, one additional rule applies. The arrayness (number and size of dimensions) of an arrayed expression must be exactly the same as the arrayness of the variable to which it is assigned. This must be true because each element of the target array is set to the *corresponding* element of the arrayed expression. An arrayed expression follows the same rules as an unarrayed expression except that some or all of the variables are arrays (of identical dimensions). Thus, if

$$A = C X^2 + D X + 5,$$

is a legal HAL/S statement involving simple variables A, C, D, and X of any data type, then

$$[A] = [C]\ [X]^2 + D[X] + 5,$$

*The HAL/S compiler annotates arrays with square brackets in the output listing. Thus, the assignment statement would appear as [A] = [A] + 1, where A, C and X are identical arrays of the same data types, is also legal In general, all of the arithmetic operators (e g +, **, /, etc) will accept either two simple variables, a simple variable and an array, or two arrays of identical dimensions

Note, however, that the machine code generated to correspond to an arrayed statement still contains a loop, this fact is important when assessing the efficiency of a computation

The following shows how the partition form of array subscripting is used Given

DECLARE GRID ARRAY(6,6) SCALAR,

a variety of re-arrangements of the array can be done in a very few statements

1) Set the top half to the bottom half

   $\text{GRID}_{1 \text{ TO } 3,*} = \text{GRID}_{4 \text{ TO } 6,*}$,

2) Set the upper left quarter to the lower right corner

   $\text{GRID}_{1 \text{ TO } 3,\ 1 \text{ TO } 3} = \text{GRID}_{3 \text{ AT } 4,\ 3 \text{ AT } 4}$,

3) Set the first row to the sum of the other five

   $\text{GRID}_{1,*} = \text{GRID}_{2,*} + \text{GRID}_{3,*} + \text{GRID}_{4,*} +$

   $\text{GRID}_{5,*} + \text{GRID}_{6,*}$,

4) Set the border to zero

   $\text{GRID}_{1,*}$, $\text{GRID}_{*,6}$, $\text{GRID}_{6,*}$, $\text{GRID}_{*,1} = 0$,

This last example is a multiple assignment statement, to which one additional rule applies If one or more of the target variables in a multiple assignment statement is an array, then *all* of the target variables must be arrays and of identical dimensions

One caution is in order regarding assignments like these Consider the assignment,

GRIDS(1,2, TO #) = GRID$(1,1 TO #−1),

This statement might be intended to shift the top row one position to the right Instead, it sets GRIDS(1,2 TO #) to GRIDS(1,1), the first element is propagated throughout the row The reason can be seen when the arrayed assignment is unravelled

GRID$(1,2) = GRID$(1,1),
GRIDS(1,3) = GRID$(1,2),

This adverse effect can occur whenever a partition of an array is set from an intersecting partition of itself Such assignments should always be checked by partially expanding them by hand

Using the feature introduced in this section, we can make the redundant accelerometer example of Section 6 1 more compact

```
M   EXAMPLE_6
M   PROGRAM,
M      DECLARE ARRAY(4, 3),
M                  GYRO_INPUT INTEGER,
M                  ATT_RATE SCALAR;
M      DECLARE SCALE ARRAY(3) SCALAR CONSTANT(.013, .026,- 013),
M      DECLARE BIAS SCALAR CONSTANT(57 296),
M      DO FOR TEMPORARY DEVICE = 1 TO 4;
M         [ATT_RATE]         = [GYRO_INPUT]          [SCALE] + BIAS;
S                 DEVICE,*                DEVICE,*

M      END,
M   CLOSE EXAMPLE_6;
```

Here, we have converted an unarrayed statement in double loops to an arrayed statement in a single loop Since the SCALE array is of size 3 and the other arrays are 4x3, we cannot eliminate both loops without getting an arrayness mismatch in the assignment statement But it *is* possible to have an assignment statement with more than one dimension of arrayness as long as all of the variables match Thus, we could compute a set of four attitude arrays

```
DECLARE ATTITUDE ARRAY(4,3) SCALAR,
DECLARE ATT_RATE ARRAY(4,3) SCALAR,
```

from the attitude rates in a single statement merely by

```
[ATTITUDE] = [ATTITUDE] + [ATT_RATE] DELTA_T,
```

where DELTA_T is a SCALAR representing the time between samples This one statement is functionally the same as

```
ATTITUDE$(1,1) = ATTITUDE$(1,1) + ATT_RATE$(1,1) DELTA_T,
ATTITUDE$(1,2) = ATTITUDE$(1,2) + ATT_RATE$(1,2) DELTA_T,
ATTITUDE$(1,3) = ATTITUDE$(1,3) + ATT_RATE$(1,3) DELTA_T,
ATTITUDE$(2,1) = ATTITUDE$(2,1) + ATT_RATE$(2,1) DELTA_T,



ATTITUDE$(4,3) = ATTITUDE$(4,3) + ATT_RATE$(4,3) DELTA_T,
```

(a total of twelve simple assignments)

In addition to arrayed assignments, HAL/S also allows arrayed comparisons It is possible to compare an entire array or arrayed expression, either with a simple variable or with an identically dimensioned array or arrayed expression For example, we could create a 4 by 4 array showing mismatches between the four sets of ATTITUDE data (each an ARRAY(3) partition) as shown

```
M    EXAMPLE_7.
M    PROGRAM;
M       DECLARE ATTITUDE ARRAY(4, 3) SCALAR,
M       DECLARE MISMATCH ARRAY(4, 4) INTEGER;
M       DECLARE INTEGER,
M                  I, J,
M       DO FOR I = 1 TO 4,
M          MISMATCH     = 0;
S                  I,I

M          DO FOR J = I + 1 TO 4,
M             IF [ATTITUDE]     ¬= [ATTITUDE]     THEN
S                         I,*               J,*

M                MISMATCH    , MISMATCH     = 1;

S                         J,I         I,J

M             ELSE
M                MISMATCH    , MISMATCH     = 0;
S                         J,I         I,J

M          END,
M       END,
M    CLOSE EXAMPLE_7,
```

In this example, the statement

"IF ATTITUDE$(I,*) ¬ = ATTITUDE$(J,*) THEN "

is an arrayed comparison Each element of ATTITUDE$(I,*) is compared with the corresponding element of ATTITUDE$(J,*) If *any* of the pairs of elements is unequal, then the comparison succeeds and MISMATCH(I,J) is set to 1 Thus, this statement is functionally equivalent to

IF (ATTITUDE$(I,1) ¬ = ATTITUDE$(J,1)) OR
(ATTITUDE$(I,2) ¬ = ATTITUDE$(J,2)) OR
(ATTIDUDE$(I,3) ¬ = ATTITUDE$(J3,)) THEN

Two arrays are considered unequal if they differ in *any* element, they are equal if they do *not* differ in any element (i e they are equal if all elements are the same)

It is also possible to compare an array with an arrayed expression, for instance the statement

"IF ATTITUDE$(1,*) = (ATTITUDE$(2,*) + ATTITUDE$(3,*)) /2 THEN "

would determine whether or not the first set of readings was equal to the average of the second two. Finally, an array may be compared with a simple variable or expression, e g

IF [MISMATCH] ¬ = 0 THEN

or

IF ATTITUDE$(2 TO 4,1) = ATTITUDE$(1,1) THEN

Regardless of the data types involved, the only comparisons which may be made between arrayed operands are equal (=) and unequal (¬=) This restriction is made for the same reason as in VECTOR/MATRIX comparisons The question, "Is A ≡ (1, 57, 3) greater than B ≡ (2, 4, 3)?" has no clear answer

Exercises

6.2A Which of the following are legal arrayed statements (expressions).

Where

A ARRAY(5)	D ARRAY(5,5)
B ARRAY(5)	E ARRAY(10,10)
C ARRAY(10)	
X INTEGER	
Y SCALAR	

a) A = B,

b) A = C,

c) A = X,

d) D$(*,5) = B,

e) D$(5,*) = Y,

f) E$(5,*) = B,

g) E$(5 AT 2, 3 TO 7) = D,

h) A, B = X,

i) A, Y = X,

j) C$(5 AT 3) = A + B,

k) C$(5 AT 4) = A + X,

l) C$(B) = X,

m) DO WHILE A > X,

n) DO UNTIL A = B,

o) DO UNTIL A ¬= C,

p) DO WHILE D$(2 AT 2, 2 AT 3) = E$(2 TO 3, 3 TO 4),

q) DO WHILE D$(*,3) = A,

r) DO WHILE A$(1,1) = X,

s) DO UNTIL A = C$(5 AT 4),

t) DO UNTIL B = E$(7, 6 TO #),

6.2B What are the major benefits of the ability to do operations on entire arrays in one line of code?

6 3 ARRAYS OF OTHER DATA TYPES

So far in this book, five data types have been introduced INTEGER, SCALAR, VECTOR, MATRIX, and BOOLEAN An array of any of these types can be created in a manner completely analogous to the INTEGER/SCALAR arrays already described For instance, one array of each type can be created in a single DECLARE statement

```
DECLARE ARRAY(10),
    I INTEGER,
    S SCALAR,
    V VECTOR,
    M MATRIX,
    B BOOLEAN,
```

Each of these arrays consists of ten array elements, each element behaves in the same way as a simple variable of the same data type In the case of an array of VECTORs (e g V above), each array element in turn consists of several components (in this case, three scalars) Hence, if V were to be completely initialized, 10 x 3 = 30 values would be required As in INTEGER/SCALAR arrays, the INITIAL list may contain either a value for every array element or a "single" value (i e initialization for one VECTOR or for one MATRIX) For example

```
DECLARE A ARRAY(2) VECTOR INITIAL(1,0,0,1,0,0),
```

creates

$$A \equiv \left(\begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} , \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \right)$$

as does

```
DECLARE A ARRAY(2) VECTOR INITIAL(1,0,0),
```

and,

```
DECLARE M ARRAY(3) MATRIX(2,2) INITIAL(1,2,3,4,5,6,7,8,9,10,11,12),
```

creates

$$M \equiv \left(\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} , \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} , \begin{bmatrix} 9 & 10 \\ 11 & 12 \end{bmatrix} \right)$$

The same initial list could also be used to initialize a three by two array of 2-VECTORS

```
DECLARE X ARRAY(3,2) VECTOR(2) INITIAL(1,2,3,4,5,6,7,8,9,10,11,12),
```

But in this case, the layout of the data is significantly different

$$X \equiv \begin{pmatrix} \begin{bmatrix}1\\2\end{bmatrix} & \begin{bmatrix}3\\4\end{bmatrix} & \begin{bmatrix}5\\6\end{bmatrix} \\ \begin{bmatrix}7\\8\end{bmatrix} & \begin{bmatrix}9\\10\end{bmatrix} & \begin{bmatrix}11\\12\end{bmatrix} \end{pmatrix}$$

This is *not* merely a distinction of graphical representation The concepts of data type and arrayness are completely independent Thus given

```
DECLARE M MATRIX(2,2) INITIAL(a,b,c,d),
DECLARE N MATRIX(2,2) INITIAL (e,f,g,h),
DECLARE A ARRAY(2) VECTOR(2) INITIAL(e,f,g,h),
```

the assignment statements,

$$\overset{*}{N} = \overset{*}{M}\ \overset{*}{N},$$

and

$$[A] = \overset{*}{M}\ [\bar{A}],$$

perform very different operations "N = M N," is a simple matrix multiplication as described in Chapter 2, but "A = M A," is an arrayed statement, it does two (the arrayness) multiplications of a vector by a matrix The results would be

$$\overset{*}{N} = \begin{bmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{bmatrix}$$

$$[\bar{A}] = \left(\begin{bmatrix} ae + bf \\ ce + df \end{bmatrix}, \begin{bmatrix} ag + bh \\ cg + dh \end{bmatrix} \right)$$

As indicated above, arrayed statements may be formulated from arrays of VECTORs and/or MATRIXes according to the usual rules All of the VECTOR/MATRIX operations may be applied to two simple variables (or expressions), to an array and a simple variable, or to two arrays of identical dimensions To see how arrayed operations on these data types might be used, consider the following situation An aircraft has a position VECTOR, MY_POSN, and access to an array of five other vectors, [POSITIONS], which gives the locations of five other aircraft The code below, which executes every DELTA_T seconds, computes the velocity of each aircraft, the distance between each aircraft and MY_POSN, and the rate of approach of each toward MY_POSN

```
M | EXAMPLE_8:
M | PROGRAM,
M |    DECLARE POSITIONS ARRAY(5) VECTOR,
M |    DECLARE OLD_POSN ARRAY(5) VECTOR,
M |    DECLARE ARRAY(5),
M |                VELOCITY VECTOR,
M |                DISTANCE SCALAR,
M |                APPROACH_RATE SCALAR,                                  /*THE ANSWERS*/
M |    DECLARE MY_POSN VECTOR,
M |    DECLARE DELTA_T SCALAR,
  |
C |    OBTAIN POSITIONS FROM OUTSIDE
  |
E |      -           -                -
M |    [VELOCITY] = ([POSITIONS] - [OLD_POSN]) / DELTA_T;
E |                       -           -
M |    [DISTANCE] = ABVAL([POSITIONS] - MY_POSN);
E |                          -                -            -
M |    [APPROACH_RATE] = [VELOCITY] . UNIT([POSITIONS] - MY_POSN);
E |      -          -
M |    [OLD_POSN] = [POSITIONS];
M | CLOSE EXAMPLE_8;
```

Each of these assignment statements has an arrayness of five The second one, for instance, first subtracts MY_POSN from each of the five VECTORS in POSITIONS, producing an array of five "distance" VECTORS Then the ABVAL function operates on each VECTOR in turn producing a scalar distance which is stored into the corresponding element of DISTANCE

So far we have been deliberately avoiding any subscripts of arrays of VECTORs and MATRIXes This is because a long list of subscripts can be rather confusing For instance, a three dimensional array of MATRIXes could have up to five subscripts, Given

"DECLARE M ARRAY(2,3,4) MATRIX(5,5),"

one might expect the first MATRIX to be referenced as "M$(1,1,1,*,*)" which is fairly complicated, though more comprehensible than "M$(J+1,2 AT J–1,*,3 AT #–4,2) " To aid in dealing with these difficulties, HAL/S makes a distinction between array subscripts and component subscripts The first three subscripts of M are *array* subscripts and the last two are *component* subscripts. To make subscript expressions more readable, HAL/S enforces the following rule: Whenever both array and component subscripts are applied to a variable, they are separated by a colon instead of a comma Thus, the first MATRIX in the array M is actually "M$(1,1,1 *,*)" Using this syntax, we can re-write the second assignment statement from the example above the hard way, that is

[Distance] = ABVAL([POSITIONS]–MY_POSN),

is equivalent to

ORIGINAL PAGE IS
OF POOR QUALITY

DISTANCE$_1$ = SQRT((POSITIONS$(1 1)–MY_POSN$_1$)**2
+ (POSITIONS$(1 2)–MY_POSN$_2$)**2
+ (POSITIONS$(1 3)–MY_POSN)**2),

DISTANCE$_2$ = SQRT((POSITIONS$(2 1)–MY_POSN$_1$)**2
+ POSITIONS$(2 2)–MY_POSN$_2$)**2
+ POSITIONS$(2 3)–MY_POSN$_3$)**2),

DISTANCE$_5$ = SQRT((POSITIONS$(5 1)–MY_POSN$_1$)**2
+ POSITIONS$(5 2)–MY_POSN$_2$)**2
+ POSITIONS$(5 3)–MY_POSN$_3$)**2),

Aside from the use of the colon, all of the possibilities for subscripting still apply All of the TO, AT, and * partitions may be used on either side of the colon, any arithmetic expression may be used as a subscript, and a subscripted variable may be used in any context in which a simple variable of the same data type could be used

The data type of a *subscripted* array is not necessarily the same as the data type of the entire array For instance, given

DECLARE A ARRAY(3,2) MATRIX,,

A is a two-dimensional array of type MATRIX,*

A$(1,* *,*) is a one-dimensional array of type MATRIX,*

A$(1,* 1,*) is a one-dimensional array of type VECTOR,*

and

A$(1,1 1,1) is a single SCALAR

It is more common to reference an entire array element or sub-array than it is to reference a *component* of an array element or some sub-array of partitions, etc Therefore, HAL/S provides a more compact form for referencing an entire element of an array to which component subscripting could also apply When an entire array element is selected, the asterisks (component subscripts) to the right of the colon may be omitted Hence, the first MATRIX in the array A above can be referenced as "A$(1,1)" The convenience of this form of subscript is illustrated by the program below which processes an array of "N" 3-VECTORs and saves the three having the greatest magnitudes in a second array

*Each occurrence of A in the listing will automatically be annotated with an overpunch reflecting the effect of subscripting on A·

```
M  EXAMPLE_9:
M  PPOGRAM;
M     DECLARE V ARRAY(999) VECTOP(3),
M     DECLARE BIGTHREE ARRAY(3) VECTOR(3) INITIAL(0);
M     DECLARE N INTEGER,                                       /*NUMBER OF ACTUAL ENTRIES IN V*
M     DO FOR TEMPORARY I = 1 TO N,
M  INNER
M        DO FOR TEMPORARY J = 1 TO 3;
E                           -              -
M           IF ABVAL(V   ) > ABVAL(BIGTHREE   ) THEN
S                     I:                 J

M              DO,
M                 DO FOR TEMPORARY K = J + 1 TO 3;
E                           -              -
M                    BIGTHREE    = BIGTHREE     ,
S                            K:            K-1.

M                 END,
E                         -      -
M                 BIGTHREE    = V   ,
S                         J.     I

M                 EXIT INNER,                                    /*  TRY NEW I */
M              END,
M        END INNER,
M     END,
M  CLOSE EXAMPLE_9,
```

6 3 1 Arrays of BOOLEANs

BOOLEAN arrays are not substantially different from arrays of other data types The only attribute of BOOLEAN arrays that does not directly follow from the previous discussion is *Whenever a BOOLEAN array is subscripted, the subscript must end with a colon* The reason for this restriction is that BOOLEAN is actually a special case of BIT strings ** Like VECTORs and MATRIXes, bit strings may possess component subscripts Thus, even though a BOOLEAN has only one component (a single bit), the colon must be supplied to indicate that the subscript is an *array* subscript rather than a component subscript

Aside from this restriction, BOOLEAN arrays are used in the same way as arrays of other types, declaration and initialization take the same forms

```
DECLARE ARRAY(12) BOOLEAN,
    A,
    B INITIAL(OFF),
    C INITIAL(OFF,ON,9#ON,OFF),
```

and arrayed assignments and comparisons also function as before

```
[Ȧ]$(1 TO 6.) = [Ḃ]$(1 TO 6 ) & (ȦS(1 ) OR [Ḃ]$(7 TO 12 ),
IF[Ȧ] = TRUE THEN
```

One typical use of BOOLEAN arrays is for maintaining status tables For instance, if we had a set of redundant altimeters producing an array of altitude values

**Bit strings are fully described in Chapter 13 The word BOOLEAN is exactly equivalent to "BIT(1)"

DECLARE ALT ARRAY(4) SCALAR,,

and a "parallel" array containing the time at which each value was read

DECLARE TIMETAG ARRAY(4) SCALAR,,

then it might be useful to define a boolean array of the same size

DECLARE DATA_VALID ARRAY(4) BOOLEAN,,

each element of which indicates the validity of the corresponding altitude value One possible form of this reasonableness check is shown below

```
M | EXAMPLE_A.
M | PROGRAM;
M |    DECLARE ARRAY(4),
M |                ALT SCALAR,
M |                TIMETAG SCALAR,
M |                DATA_VALID BOOLEAN,
M |    DECLARE SCALAR INITIAL(0),
M |                TOTAL, NUMBER_GOOD;
M |    DECLARE AVERAGE SCALAR;
M |    DO FOR TEMPORARY J = 1 TO 4;
M |       IF RUNTIME - TIMETAG  > .1 OR ALT  <= 0 OR ALT  > 50000 THEN
S |                           J             J             J
  |
E |
M |          DATA_VALID  = FALSE;
S |                    J
  |
M |       ELSE
M |          DO,
E |
M |             DATA_VALID  = TRUE,
S |                       J
  |
M |             NUMBER_GOOD = NUMBER_GOOD + 1;
M |             TOTAL = TOTAL + ALT ;
S |                                J
  |
M |          END;
M |    END,
M |    AVERAGE = TOTAL / NUMBER_GOOD,
M |    DO FOR TEMPORARY I = 1 TO 4;
E |
M |       IF DATA_VALID  THEN
S |                    I
  |
M |          IF ABS(ALT  - AVERAGE) > 1 AVERAGE THEN
S |                    I
  |
E |
M |             DATA_VALID  = FALSE;
S |                       I
  |
M |    END;
  |
C | NOW WE HAVE SCREENED OUT DATA WHICH IS NEGATIVE OR ZERO,
C | OR TOO LARGE OR TOO OLD OR TOO FAR FROM THE AVERAGE
M | CLOSE EXAMPLE_A;
```

Exercises

6 3A Write out graphically the results of the following initializations

- i) DECLARE X ARRAY(3) MATRIX(3,3) INITIAL(9# 1,*)
- ii) DECLARE Y ARRAY(3,3) VECTOR(3) INITIAL(9# 1,*)
- iii) DECLARE Z ARRAY(9) VECTOR(3) INITIAL(9# 1,*)
- iv) DECLARE A ARRAY(27) SCALAR INITIAL(9# 1,*)

6 3B In the previous problem, the initializations lists were transformed into their graphical interpretations Using this data, assign the twenty-first element of the linearization of X, Y, Z, and A to a scalar variable, S

6 3C Given a variable M, declared MATRIX(3,9)

Assign the 16th through 22nd elements of the linearization of X, Y, Z, and A to the 2nd through 8th elements in the linearization of M

6 3 1A The Sieve of Eratosthenes is an ancient Greek method for computing prime numbers, but it still works today and is quite suitable for a computer The algorithm works as follows

Start with a list of integers from 2 to the largest number of interest Cross out all multiples of 2, then all multiples of 3, and so on The remaining numbers are then all prime

Write a HAL/S program to print out all primes less than 1000, using the Sieve of Eratosthenes (Hint Use an ARRAY of BOOLEAN type to indicate if a number is prime or not)

6 4 FUNCTIONS OF ARRAYS

In Section 6 2 we saw that the statement

"[A] = [B] $^{1/2}$,"

where A and B are identically dimensioned arrays, results in each element of A being set to the square root of the corresponding element of B As the reader might expect, the same result may be obtained by the statement

"[A] = SQRT([B]),".

Whenever any of the built-in functions introduced so far is applied to an array, the result is an identically dimensioned array where each element is the result of applying the function to the corresponding element of the arrayed operand Similarly, the rules for functions of two arguments, such as MOD or DIV, are the same as for infix operators (e g +, –, **, etc), both arguments may be unarrayed, or one may be arrayed and the other unarrayed, or both may be arrayed (and of identical dimensions) This usage, the *arrayed invocation* of a function, has been amply illustrated in the previous section, HAL/S also provides a set of functions that will *only* accept arrayed arguments

One of the examples in Section 6 1 gathered some statistics on the execution time of the matrix inverse operation A SCALAR ARRAY(100), TIME, was filled with 100 samples of the execution time of an assignment statement Then the variables T_MIN, T_MAX, and T_MEAN were set to the minimum, maximum and mean values from the array by means of a loop More compact code for the same function is shown below

```
T_MIN  = MIN([TIME]),
T_MAX  = MAX([TIME]),
T_MEAN = SUM([TIME])/100,
```

Here, the built-in functions, MIN, MAX, and SUM, *reduce* an array to a single unarrayed value Each of these functions (and a fourth, PROD) *requires* an arrayed operand The array may be either INTEGER or SCALAR (of either precision), and the result is an unarrayed value of the same data type and precision

The SUM function simply adds all of the array elements together

"SUM([A])"

is equivalent to

"A$1 + A$2 + + A$n"

The PROD function multiplies all of the elements together in a similar manner (A$1) (A$2) (A$3) (A$n) MIN and MAX both search through the array, and return the *value* of the array element which is algebraically smallest (MIN) or largest (MAX) All of these functions will accept a multi-dimensional array, but the result returned is always unarrayed Thus, given

```
     [A] ≡ (5,17,–3,21),
MIN([A])   = –3,
MAX([A])   = 21,
SUM([A])   = 40, and
PROD([A])  = –5355
```

The results will be exactly the same whether A is declared as

```
DECLARE A ARRAY(2,2) INITIAL(5,17,–3,21),
```

or as a liner ARRAY(4)

6 4 1 Shaping Functions

Throughout this chapter we have stressed the fact that a linear array is not the same type as a VECTOR, and that a two dimensional array is not the same type as a MATRIX Sometimes, however, it is useful to be able to convert one type to the other For instance, we might want to use arrayed statements to compute the x, y, and z components of a vehicle's position from some complex sensor, and then to treat the results as a 3-VECTOR for further computations We already know from Chapter 2 that given

```
"DECLARE A ARRAY(3) SCALAR,
          V VECTOR,"
```

the conversion can be made by

```
"V = VECTOR(A$1,A$2,A$3),"
```

In fact, the form, "V = VECTOR([A])," is completely equivalent Both the VECTOR and MATRIX conversion functions will accept any mixture of arrays and simple variables as operands, provided the total number of elements is correct When an array is specified as an operand to one of these functions, it is "unraveled", i e it is effectively replaced with a list of its elements In the same way, an array of vectors can be unraveled for assignment to a larger vector

```
DECLARE AV ARRAY(2) VECTOR(3),
DECLARE VEC6 VECTOR(6),
VEC6 = VECTOR6 ([AV]),
```

The statement above is functionally equivalent to

$$\overline{V}_{3\ AT\ 1} = \overline{AV}_1 \ ,$$
$$\overline{V}_{3\ AT\ 4} = \overline{AV}_2 \ ,$$

The MATRIX function works in much the same way, a 3 by 3 MATRIX, M, can be assigned as

$\overset{*}{M}$ = MATRIX([$\overline{AV}$],[A]),

yielding

$$M \equiv \begin{array}{lll} AV\$(1\ 1), & AV\$(1\ 2), & AV\$(1\ 3) \\ AV\$(2\ 1), & AV\$(2\ 2), & AV\$(2\ 3) \\ A\$1, & A\$2 & , A\$3 \end{array}$$

To perform the reverse conversion, the INTEGER and SCALAR functions are used These functions have already been introduced as explicit type conversions, when they are used with multiple simple arguments or any type of data aggregate (arrays, VECTORs, etc) they return an arrayed result. Thus, using the previous declaration, we can set an array to a VECTOR as

[A] = $SCALAR_3(\overline{V})$,

The SCALAR (or INTEGER) function will accept any number of arguments of any arithmetic type so long as the total number of SCALAR or INTEGER values agrees with the subscript of the function

These functions have a number of uses They may be used to convert the type of data as shown above, to initialize an array, as in

[SMALL_PRIMES] = $INTEGER_5$(1,2,3,5,7),

or, to re-arrange the elements of an array (hence the term "shaping functions")

DECLARE A12 ARRAY(12) INTEGER,
DECLARE A4X3 ARRAY(4,3) INTEGER,
DECLARE A3X4 ARRAY(3,4) INTEGER,
[A12] = $INTEGER_{12}$)([A4X3]),
[A4X3] = $INTEGER_{4,3}$([A12]),
[A3X4] = $INTEGER_{3,4}$([A4X3]),

When, as in the last two statements above, the INTEGER or SCALAR functions possess multiple subscripts, the result is a multi-dimensional array Each subscript denotes the size of one dimension of the array

Each subscript of the INTEGER or SCALAR function must be computable at compile-time (i e each must be an arithmetic expression involving only literals and CONSTANTs) In addition to the subscript, the precision specifiers, @SINGLE and @DOUBLE may be used to change the precision of the operand Just as in the VECTOR and MATRIX functions, the precision specifier is used as a subscript *and must precede the array dimensions* Thus, an ARRAY(12) SCALAR, S, can be converted to a 2x6 INTEGER DOUBLE array, I by

[I] = $INTEGER_{@DOUBLE,2,6}$([S]),

Exercises

6 4A Change the solution of the 3rd exercise in Section 6 3 from the rather unwieldy and hard to read assignments to simpler ones using the vector shaping functions

6 4B Assuming the following declarations

```
DECLARE X ARRAY(2,3) SCALAR INITIAL(2#(1.1,2 2,3 3)),
DECLARE V VECTOR(3) INITIAL(3# 1),
```

Depict graphically the results of the following shaping functions applied to X and V

i) INTEGER(X)

ii) INTEGER(X,X)

iii) SCALAR(V)

iv) INTEGERS$(2,6) (2#X)

v) MATRIX(3#V)

vi) VECTORS$(6)(X)

6 4 1A Use vector shaping functions to provide a clearer solution to exercise 6 3-C

(Note This problem requires that the reader see Section 6 5 1 of the Language Specification)

6 4 1B Given the following declarations

```
DECLARE X ARRAY(2,3) SCALAR INITIAL(2#(1 1,2 2,3 3)),
DECLARE V VECTOR INITIAL( 1)
```

State the types and depict graphically the values of the following expressions

a) INTEGER(X)

b) INTEGER(X,X)

c) SCALAR(V)

d) INTEGER$(2,6) (2#X)

e) MATRIX(3#V)

End of Chapter Problems

6A The median value of the elements of an array of odd dimension may be computed by sorting the elements in increasing order The middle element of a sorted array is, in fact, the median value Write a program to find the median value of an array of 25 integers A simple, though not very efficient, sort algorithm may be described as follows

Find the smallest element of the array If it is not the first element, exchange it with the first Then find the smallest of the remaining elements If it is not the second element, exchange it with the second Continue until the entire array is sorted

An advantage of this algorithm for the median-value problem is that it is not necessary to sort the entire array, finding the 13th smallest element is sufficient

6B We have made many timings of 3 processes A, B, and C The results of our timings are in an array TIM_VALUES declared,

TIM_VALUES ARRAY(3,25) INTEGER

We now wish to process this information, finding the sum for all 25 timings of each process A, B, and C, and the sums of the times for each set of timings for A, B, and C (i e , row and column totals) This information is to be put in an array together with the raw data, and this array is to be called TIMING_DATA

Write a segment of code that will create this new array and do the necessary information processing

Include any assumptions made and any new variables declared

7.0 PROCEDURES AND FUNCTIONS

In HAL/S, the concept of a subroutine is realized in two forms PROCEDURES and FUNCTIONS Each is a block of code delimited by a block header and a CLOSE statement These code blocks may be nested within PROGRAMS or within each other to any degree, scoping rules restrict the variables each block may reference, thus avoiding a large class of potential programming errors HAL/S PROCEDUREs and FUNCTIONs have two basic uses to share a sequence of statements among different paths through an algorithm, and to segment a programming problem into manageable parts

7 1 USER DEFINED FUNCTIONS

HAL/S includes a large assortment of built-in functions These include trigonometric routines (SIN, ARCTAN), algebraic routines (SQRT, EXP), conversion functions (INTEGER, VECTOR) and many others These functions may be used in expressions along with variables, constants and operators, they add to the power of the language by eliminating much low level coding and allowing sophisticated operations to be expressed very compactly The set of built-in functions is a part of the language, but HAL/S also allows the user to define new functions which may then be used in exactly the same way as the built-ins

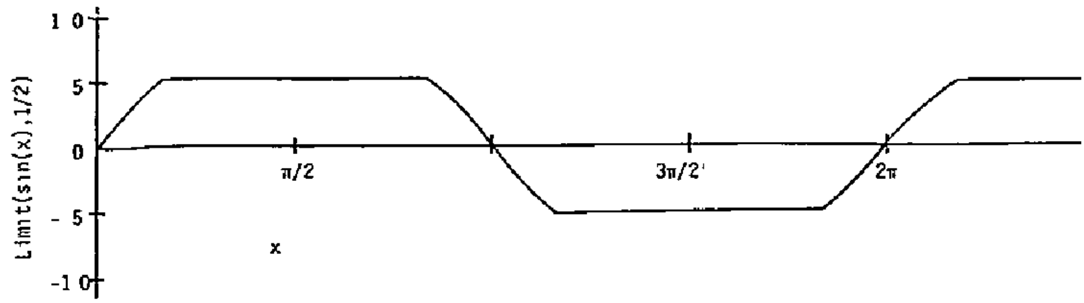
One type of operation which occurs frequently in flight software is the limiting of a variable to a given range A FUNCTION to perform this operation is shown below

```
M | LIMIT:
M | FUNCTION(VALUE, BOUND) SCALAR,
M |    DECLARE SCALAR,
M |                VALUE, BOUND,
M |    IF VALUE > BOUND THEN
M |       RETURN BOUND,
M |    IF VALUE < -BOUND THEN
M |       RETURN -BOUND;
M |    RETURN VALUE,
M | CLOSE LIMIT,
```

The function block is delimited by FUNCTION and CLOSE statements The CLOSE statement is the same as in PROGRAMs, it consists of the word CLOSE and an optional block name The FUNCTION statement contains three pieces of information the label on the statement, which defines the name of the function, the names of the formal parameters (sometimes called dummy arguments), and the return-type of the function

LIMIT is a scalar valued function of scalars This fact is denoted by the word SCALAR on the FUNCTION statement and the declaration of the formal parameters In general, a function's parameters and return value may be of any data type, hence the return type must always be specified on the FUNCTION statement and the formal parameters must always be declared Declaring the formal parameters prior to any local data is good programming practice and should be treated as a requirement

The operation of the LIMIT function may be seen from the following illustration, which is a graph of Y=LIMIT(SIN(X),1/2), for $0 \leqslant x \leqslant 5$ pi/2



Limit Function

Functions must always end by executing a RETURN statement The RETURN statement always has one operand which represents the value of the function The value returned may be a variable, as in LIMIT, or any expression of the appropriate data type Sometimes the executable code of a function consists of only the RETURN statement, for instance

```
M | MASS:
M | FUNCTION(REST_MASS, SPEED) SCALAR;
M |    DECLARE SCALAR,
M |                   REST_MASS, SPEED;
M | TAU
M | FUNCTION(V) SCALAR;
M |    DECLARE V SCALAR,
M |    DECLARE C CONSTANT(2980000),
E |                     2   2
M |    RETURN SQRT(1 - V  / C );
M | CLOSE TAU;
M |    RETURN REST_MASS / TAU(SPEED);
M | CLOSE MASS;
```

Using these functions, the apparent mass of a 100-ton vehicle moving at 20 kilometers per second can be computed by

APPARENT_MASS = MASS(100,20),

As it turns out, the MASS function is not going to be very useful Twenty kilometers per second is so slow (compared with the speed of light) that the relativistic mass increase will be lost in the round-off errors inherent in the computation To find the range over which this effect can safely be ignored, we could execute the following code

```
DECLARE V SCALAR,
DO FOR V = 250000 TO 0 BY –100 UNTIL
   ALMOST_EQUAL(1,MASS(1,V)),
END,
WRITE(6) 'THE ANSWER IS ', V,
```

This code references an additional user function, ALMOST_EQUAL, which could be written as shown below

```
M  ALMOST_EQUAL-
M  FUNCTION(A, B) BOOLEAN;
M     DECLARE SCALAR,
M                 A, B;
M     DECLARE TOLERANCE SCALAR,
M     IF B ¬= 0 THEN
M        TOLERANCE =  000001 ABS(B),
M     ELSE
M        TOLERANCE =  000001,
M     IF ABS(A - B) > TOLERANCE THEN
M        RETUPN FALSE,
M     ELSE
M        RETURN TRUE;
M  CLOSE ALMOST_EQUAL;
```

ALMOST EQUAL is a BOOLEAN-valued function of scalars, as denoted by the word BOOLEAN on the function header and the declaration of the formal parameters Hence the RETURN statements have BOOLEAN operands TRUE and FALSE

Since no other data type is automatically converted to BOOLEAN, a BOOLEAN expression is the only permissable operand to the RETURN statement of a BOOLEAN function

Likewise, the RETURN statement of a VECTOR or MATRIX function must be supplied with a VECTOR or MATRIX expression, respectively Exact matching of data type is not always required, however, the same implicit conversions that can be performed in an assignment statement can also result from a RETURN statement These conversions are

1 Single to double precision
2 Double to single precision
3 Integer to scalar
4 Scalar to integer
5 Integer or scalar to character

Aside from these exceptions, the value returned by a function must be of exactly the same type as that specified on the function header

The function header serves as a declaration of the function Variables must always be declared before they are used in expressions, the same rule applies to functions as well Therefore, function bodies are usually placed before their first invocation in a program

However, in the previous example, ALMOST_EQUAL was defined after it had been used in an UNTIL phrase In this case it is possible to make a valid HAL/S program without moving the function body, by DECLARING the function before it is used, as shown in the example below

```
M | EXAMPLE_N
M | PROGRAM,
M |    DECLARE V SCALAR;
M |    DECLARE ALMOST_EQUAL FUNCTION BOOLEAN;                          /*<---*/
M | MASS
M | FUNCTION(PEST_MASS, SPEED) SCALAR;
M |    DECLARE SCALAR,
M |                REST_MASS, SPEED;
M | TAU-
M | FUNCTION(V) SCALAR,
M |    DECLARE V SCALAR,
  |
C |           .
C |           .
C |           .
  |
M | CLOSE TAU;
  |
C |
C |           -
C |           .
  |
M | CLOSE MASS,
E |                                               .
M |    DO FOR V = 250000 TO 0 BY -100 UNTIL ALMOST_EQUAL(1, MASS(1, V));
M |    END,
M |    WRITE(6) 'THE ANSWER IS', V;
M | ALMOST_EQUAL
M | FUNCTION(A, B) BOOLEAN,
M |    DECLARE SCALAR,
M |                A, B,
  |
C |       .
C |       .
C |       .
  |
M | CLOSE ALMOST_EQUAL;
  |
M | CLOSE EXAMPLE_N;
```

The FUNCTION DECLARE statement has the same general form as a variable declaration except that the word FUNCTION (with no argument list) precedes the type specification Of course it is always possible to place a function body before its first invocation as was done with MASS and TAU above, in which case the DECLARE statement is unnecessary

Exercises

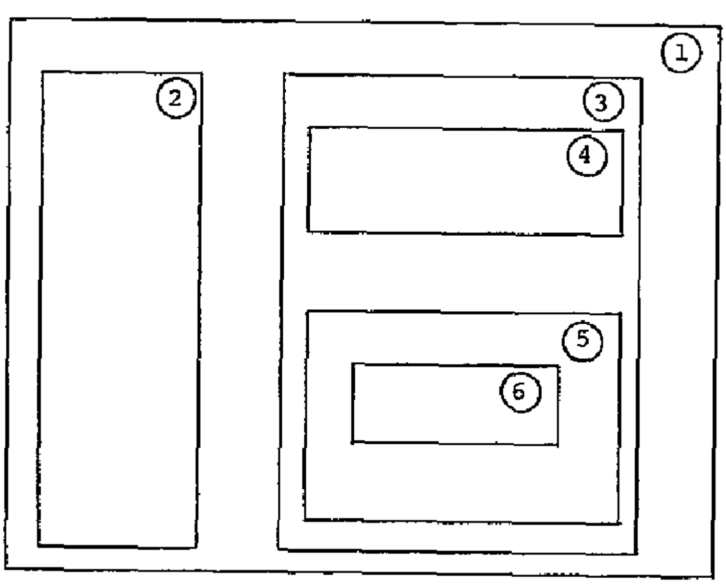
7 1A What values will be written by the following HAL/S program?

```
PROBLEM PROGRAM,
   DECLARE I INTEGER INITIAL(1),
PROC1 PROCEDURE,
   DECLARE I INTEGER INITIAL(1),
   CALL PROC2,
   I = I + 1,
   WRITE(6) I,
   CLOSE,
PROC2 PROCEDURE,
   I = I + 1,
CLOSE,
CALL PROC1,
CALL PROC2,
I = I + 1,
WRITE(6) I,
CLOSE PROBLEM,
```

7.1B What are the syntax errors in the following HAL/S program? (Line numbers are for reference only.)

```
 1) PROB2  PROGRAM,
 2)     DECLARE X INTEGER,
 3)     CALL PROC1,
 4)     CALL PROC2,
 5)     Y = Y + 1,
 6) PROC1  PROCEDURE,
 7)     DECLARE Y INTEGER,
 8)     CALL PROC1,
 9)     CALL PROC2,
10)     X = X + 1,
11) PROC2  PROCEDURE,
12)     X = X + 1,
13)     Y = Y + 1,
14) CLOSE,
15) CLOSE,
16) CLOSE PROB2
```

7.1C



Consider the above nesting diagram that depicts the scoping of blocks

For each of the procedure blocks numbered 2-6, write the numbers of the blocks from which that procedure may be invoked

7 2 ARGUMENTS AND PARAMETERS

The types of the arguments passed to a function must agree with the declaration of the formal parameters The *formal parameters* (which some languages term "dummy arguments") are declared in the function body, the function *arguments* are those expressions specified in the function invocation For example in the invocation

```
UNTIL ALMOST_EQUAL(I,MASS(1,V)),
```

(arguments)

The two arguments are scalar expressions The formal parameters are declared in the function body

```
ALMOST_EQUAL  FUNCTION(A,B) BOOLEAN,
   DECLARE SCALAR,A,B,
   DECLARE TOLERANCE SCALAR,

CLOSE,
```

(Formal parameters)

(Local variable)

Formal parameters in the functions discussed so far have all been scalars, but it is possible for them to be of any basic data type Integer, Scalar, Vector, Matrix, Boolean, Character, Structure or Bit The type of a formal parameter is determined solely by its declaration The actual arguments supplied when a function is invoked must be of the same data types as the formal parameters The exception to this rule is that under some circumstances the actual argument will be automatically (implicitly) converted to the type required by the function The conversions that are permitted are the same set that are allowed in an assignment statement Those that were listed earlier as allowable type conversions in the RETURN statement

The declaration of a formal parameter takes exactly the same form as any other DECLARE statement The INITIAL and CONSTANT attributes may not be used, but otherwise, any attribute is acceptable A function may have any number of formal parameters, including zero The following is an example of a function in which no arguments appear

```
M   ROLL
M   FUNCTION INTEGER;
M      RETURN 5 RANDOM + 1,
M   CLOSE ROLL;
```

The ROLL function returns an integer in the range 1 to 6* It may be invoked as

DO UNTIL ROLL + ROLL = 7,

Functions without parameters usually either access global data or perform some sort of input ROLL gets its "input" from the RANDOM function, though reading cards or sensors is actually more typical

A function has only a data type, but formal parameters may have other attributes In particular, a formal parameter may be arrayed The following example is a matrix-valued function of arrays of vectors The resulting matrix consists of the dot products of each pair of vectors

```
M   DOTS
M   FUNCTION(A1, A2) MATRIX(10, 10);
M      DECLARE ARRAY(10) VECTOR(3),
M                 A1, A2;
M      DECLARE RESULT MATRIX(10, 10),
M      DO FOR TEMPORARY I = 1 TO 10;
M         DO FOR TEMPORARY J = 1 TO 10,
E                             -      -
M            RESULT     = A1     A2   ,
S                  I,J    I       J;

M         END;
M      END;
E               *
M      RETURN RESULT,
M   CLOSE DOTS;
```

Before leaving the subject of functions, one more very important point must be made *No function may modify any of its formal parameters* That is, parameters are viewed as constants within the function body As a consequence, for example a formal parameter cannot be used as a loop control variable since a loop control variable is modified on each iteration

*but *not* uniformly distributed See exercises

The primary intent of this rule is to make HAL/S code easier to read and maintain In languages which do not have this restriction, it is not possible to determine which variables are being modified by inspection of a statement like "A = USERFUNC(B,C,D);" In any language, it is reasonable to *assume* that A is the only variable modified In HAL/S, this assumption will always be correct

Exercises

7.2A In example 6, ALMOST_EQUAL is declared a function in the declare group of the main-program block

With a minor modification to the program, this declaration is unnecessary What is the change?

7 2B In example 7, it is stated that while ROLL returns an integer in the range 1-6, it is not uniformly distributed

a) Why?

b) Modify the function ROLL so that it is uniformly distributed and incorporate it into a program that will count how many times a pair of "dice" must be rolled to have 7 come up 5 times

7 2C Write a HAL/S program that will read from channel 5 two arrays of 5 integers apiece, then check if corresponding elements of the two arrays are relatively prime (i e , their greatest common divisor, or GCD, is 1) If they are not relatively prime, print out the pair and their GCD

A standard algorithm for computing the GCD of two numbers is called the Euclidean algorithm, and may be described as follows

Start with integers m and n, whose GCD is desired If n = 0, then GCD(m,n) = absolute value of m Otherwise, let r be the remainder resulting from dividing m by n If r — 0, then GCD(m,n) = absolute value of n Otherwise, it is the case that GCD(m,n) = GCD(n,r) Since, by the definition of the remainder, r will decrease in absolute value on each iteration, it will eventually become zero The algorithm is thus guaranteed to terminate

Note The algorithm will work for any pair of integers, positive, negative, or zero The HAL/S built-in function REMAINDER (M,N) gives the remainder when M is divided by N, as required by the algorithm

7 3 PROCEDURES

A procedure is a code block similar to a function The primary distinction is that procedures do not return values The RETURN statement can be used in a procedure, but no operand may be provided When the RETURN statement is executed in a procedure, control is returned to the caller The RETURN statement is not required in a procedure, as procedures (unlike functions) will return if the flow of control reaches the CLOSE statement

The only way to invoke a procedure is via the call statement *Procedure* invocations are *not* used in expressions

The CALL statement consists of the keyword CALL followed by a procedure name and a list of arguments (if the procedure has defined any parameters), e g

```
CALL PROC1 (X,Y,Z),
```

X, Y, and Z are the arguments, the procedure defines its formal parameters just as in functions

```
PROC1  PROCEDURE (A,B,C),
    DECLARE SCALAR,A,B,C,
    DECLARE Q VECTOR,



    RETURN,
CLOSE PROC1,
```

Formal parameters → A, B, C

Local variable → Q

Formal parameters to procedures are like function parameters in all regards, and may not be modified within the procedure Procedures also have ASSIGN parameter, described below

Suppose that the DOTS function of section 7 2 was typically used in statements like

```
LOCAL_VAR=DOTS([V1],[V2]),
```

In this statement, the DOTS function is not used in an expression, but is directly assigned into LOCAL_VAR In such a case, some inefficiency results from coding DOTS as a function This is because when the RETURN statement is executed, the 100 scalar components of RESULT are *copied* into LOCAL_VAR A better arrangement would be to code DOTS as a procedure and invoke it by

```
CALL DOTS([V1],[V2]) ASSIGN(LOCAL_VAR),
```

The DOTS *procedure* could be coded as shown below

```
M  DOTS:
M  PROCEDURE(A1, A2) ASSIGN(PESULT);
M     DECLAPE ARPAY(10) VECTOR(3),
M                 A1, A2;
M     DECLAPE PESULT MATRIX(10, 10);
M     DO FOP TEMPORAPY I = 1 TO 10,
M        DO FOP TEMPORARY J = 1 TO 10;
E                          -       -
M           RESULT    = A1     A2  ;
S                 I,J     I      J

M        END;
M     END,
M  CLOSE DOTS,
```

Here we see an example of an *assign parameter*, RESULT The statement, "DECLARE RESULT MATRIX(10,10)," does not create a variable as it did in the function DOTS, but merely defines the data type of the assign parameter Each assignment into RESULT directly modifies LOCAL_VAR Thus, no copying of data is needed

Since variables used as assign parameters to procedures can be directly modified from the procedure body, no conversions whatsoever are permitted *The type of the variable passed as an assign parameter must agree exactly with the declaration of the assign parameter* In the program segment below, A is the only variable which may be passed to P

```
    DECLARE A INTEGER,
    B INTEGER DOUBLE,
    C SCALAR,
    D ARRAY(2) INTEGER,
P   PROCEDURE ASSIGN(X),
    DECLARE X INTEGER,
    X = 0,
CLOSE P,
```

A procedure may have any number of formal and assign parameters in any combination Thus, several values can be computed in a single procedure, as shown below

```
M | STATISTICS
M | PROCEDUPE(DATA) ASSIGN(LO_VAL, HI_VAL, MEAN),
M |    DECLARE DATA ARRAY(100) SCALAR,
M |    DECLARE SCALAR,
M |                LO_VAL, HI_VAL, MEAN,
M |    LO_VAL = MIN([DATA]),
M |    HI_VAL = MAX([DATA]),
M |    MEAN = SUM([DATA]) / 100,
M | CLOSE STATISTICS,
```

This procedure could then be used as in

```
    DECLARE SAMPLES ARRAY(100) SCALAR,
    DECLARE SUMMARY ARRAY(3) SCALAR,
    CALL STATISTICS(SAMPLES)
ASSIGN(SUMMARY$1,SUMMARY$2,SUMMARY$3),
    WRITE(6) 'Min, max and mean are ',SUMMARY,
```

Unlike formal parameters, assign parameters may also be modified, as in the following procedure which sets "AUG_LAST4" to the average of the four most recent values of INPUT

```
M   FILTER
M   PROCEDURE(INFUT) ASSIGN(AUG_LAST4, BUFF);
M      DECLARE SCALAR,
M                 INPUT, AUG_LAST4,
M      DECLAPE BUFF ARPAY(4) SCALAR,
M      [BUFF]       = [BUFF]        ,
S            1 TO 3         2 TO 4

M      BUFF  = INPUT,
S          4

M      AUG_LAST4 = SUM([BUFF]) /¯4;
M   CLOSE FILTER;
```

In this example, components of BUFF appear on the left and right sides of assignment statements BUFF is probably not used by the code which invokes FILTER It is passed as an assign parameter because a separate version must be maintained for each user of FILTER

The rules concerning arguments and parameters are summarized below

1. Arguments may be expressions of any complexity, but their types must match those specified in the formal parameter declarations The automatic conversions of precision and between integers and scalars are performed, however
2. Assign *arguments* must be variables (possibly subscripted, but not expressions in general) They must match the types of the corresponding assign *parameters* exactly
3. Formal parameters may not be modified by the procedure or function which declares them Assign parameters may be both referenced and modified
4. Copying of aggregate data (such as vectors or arrays) occurs only as a result of function returns If an argument (of any type) will not fit in a machine register or accumulator, its address is passed to the procedure or function Thus HAL/S uses "call by name" for aggregate formal parameters as well as for assign parameters, even though the restriction on modification of formal parameters gives the appearance of "call by value"

Exercises

7 3A Rewrite the improved ROLL function of exercise 7 2B as a procedure, and modify the surrounding program to invoke it properly This provides an alternate solution to 7 2B

Which of the two solutions is preferable? What general observations does this suggest about the choice between procedure and function forms, when both are possible?

7 4 SCOPING RULES

The HAL/S scoping rules for variables may be summarized as follows

1 A variable may be referenced throughout the block in which it is declared and throughout any blocks nested in that block, provided that the nested blocks do not declare another variable of the same name

2 A variable declared in a nested block cannot be referenced from an outer block

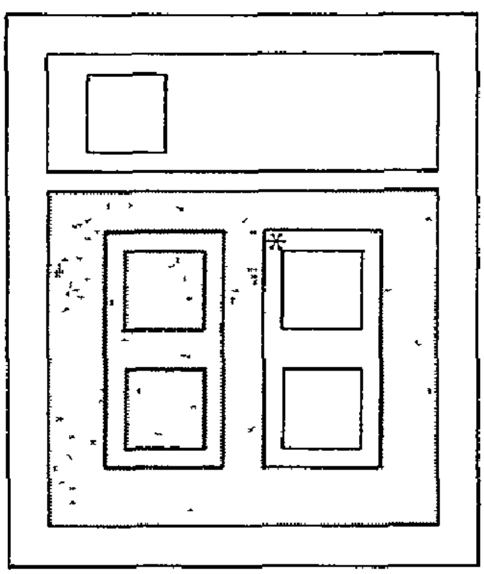
3 If variables of a given name are declared in several blocks, each reference selects the version in the nearest enclosing block

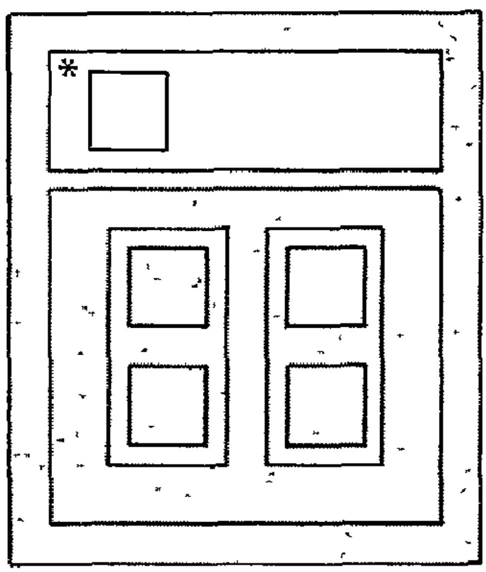
HAL/S procedures and functions may be nested within programs, or within their procedures and functions to any degree

This block structuring capability in conjunction with the scoping rules above enables a measure of functional modularity in the development of software In other words, HAL/S allows the collection of related procedures (and functions) into functional entities (themselves procedures or functions) The local resources within these entities, viz declared variables and nested procedures become unavailable, actually unseen, to 'outsiders' Communication takes place only on the highest, most visible levels

Procedure and function names are also affected by scoping rules in that a procedure or function may be invoked from the immediately enclosing block and from any other blocks which are nested in the immediately enclosing block An exception is that a procedure or function may not be referenced from within itself HAL/S does not allow recursion

The following diagrams illustrate the scoping of block names In each diagram, the shaded area indicates the region from which the block marked with an asterisk may be invoked





7 5 ARRAY(*), AUTOMATIC, AND NONHAL

In the previous section, a procedure was written to find the minimum, maximum and mean of an array of 100 scalars The STATISTICS procedure would be more general if it would accept an array of any size. The routine is rewritten as follows

```
M   STATISTICS:
M   PROCEDURE(DATA) ASSIGN(LO_VAL, HI_VAL, MEAN);
M      DECLARE DATA ARRAY(*) SCALAR;
M      DECLARE SCALAR,
M                  LO_VAL, HI_VAL, MEAN,
M      LO_VAL = MIN([DATA]);
M      HI_VAL = MAX([DATA]);
M      MEAN = SUM([DATA]) / SIZE([DATA]);
M   CLOSE STATISTICS;
```

Two changes have been made First, the formal parameter, DATA, has been declared as an ARRAY(*) DATA is still a linear array, but its size may now vary from invocation to invocation Second, the constant 100 in the computation of MEAN has been changed to the expression SIZE(DATA) SIZE is a built-in function which returns an integer denoting the number of actual elements in an ARRAY(*)

The asterisk may be used as an array dimension only in the declaration of a formal parameter An array of any data type may possess this attribute, but all such arrays must be linear (single-dimensional)

Even though a procedure or function may be written to accept an array of arbitrary size, the size of each actual argument must still be known at compile-time Thus, given the STATISTICS procedure above and the declarations

```
DECLARE A ARRAY(1000) SCALAR,
DECLARE SCALAR,X,Y,Z,
DECLARE J INTEGER INITIAL(60),
```

The statements,

```
CALL STATISTICS(A$(1 TO 60)) ASSIGN(X,Y,Z),
```

and

```
CALL STATISTICS(A$(61 TO #)) ASSIGN(X,Y,Z),
```

are both legal

But,

CALL STATISTICS(A$(I TO J)) ASSIGN(X,Y,Z),

is *not* legal because J is not a constant, i e the width of the partition (I TO J) is not known until runtime

7 5 1 Automatic Initialization

The following function will correctly sum the array of vectors, V, *only* on its first invocation


```
M   VSUM
M   FUNCTION(V) VECTOR,
M      DECLAPE V ARRAY(*) VECTOR;
M      DECLARE TOTAL VECTOR INITIAL(0);
E                                  -
M      DO FOR TEMPORARY N = 1 TO SIZE([V]),
E            -        -     -
M         TOTAL = TOTAL + V  ;
S                         N

       END;
E             -
M      RETURN TOTAL,
M   CLOSE VSUM,
```

The problem is that TOTAL is initialized to zero only on the first invocation of VSUM One way of correcting the problem is to add the statement, "TOTAL = 0," before the loop. A more convenient means of attaining the same result is to replace the declaration of TOTAL with

DECLARE TOTAL VECTOR INITIAL(0) AUTOMATIC,

The AUTOMATIC attribute controls the manner of initialization of a variable *An AUTOMATIC variable is set to its INITIAL value on each entry to the containing code block* In effect, the compiler generates an assignment statement for each automatically initialized variable immediately after the declare group of the containing block

It is important to remember that by default, initialization is STATIC (the opposite of AUTOMATIC) If the AUTOMATIC attribute is not specified, initialization occurs only once, at the time when the program is first loaded

7 5 2 The NONHAL Attribute

Sometimes it is desirable to program an application in a mixture of HAL/S and non-HAL/S code, either to capitalize on existing software or to make machine-dependent operating system interfaces which are not available in HAL/S When the non-HAL code consists of subroutines (procedures and/or functions) there is a convenient way of making them accessible to HAL/S This is the NONHAL attribute, used in a declare statement An example is

DECLARE CPU_COST FUNCTION SCALAR NONHAL(1), The form of this statement is essentially the same as the declaration of a HAL/S function that will be referenced before it is defined The only difference is the NONHAL attribute, which indicates that the function body is not included in this compilation Note that the data type of a NONHAL function must still be supplied

A similar form may be used to define a *procedure* written in some other language, e g

```
DECLARE PEARSON_CORRELATIONS PROCEDURE NONHAL(2),
```

Since a procedure has no data type, none is supplied in the declaration NONHAL procedures and functions may have formal parameters (though no assign parameters), the number and types of these parameters is not specified in the declaration, and in fact, may vary from call to call No type checking is performed on the arguments to a NONHAL procedure or function, and these blocks may even modify their input parameters Hence, great care should be taken when using the NONHAL attribute

The operand to the NONHAL attribute, which consists of a positive integer, indicates the particular language in which the subroutine was written The association of each number with a particular language is implementation dependent, and some compilers may not support NONHAL at all

These statement may *not* be used to interface separately compiled HAL/S modules A means of sharing HAL/S subroutines between separate HAL/S programs will be presented in Chapter 11

End of Chapter Problems

7A As in exercise 2B, a ball is thrown from a height of 110 feet with a horizontal velocity of 4 ft/sec Suppose that it now rebounds to 75% of its previous height on each of 10 bounces, and consider the following skeleton of a program to compute the time until the tenth bounce

```
    DO FOR I = 1 TO NUMBER_OF_BOUNCES,
       DROP_TIME = TIME_TO_DROP (HEIGHT),
       CALL HORIZ_MOTION (DROP_TIME) ASSIGN (HORIZ_DIST),
       TIME = TIME + DROP_TIME,
       WRITE(6) 'BOUNCE', I, 'TIME', TIME, 'HORIZONTAL
          DISPLACEMENT', HORIZ_DIST,
       CALL BOUNCE ASSIGN (HEIGHT, BOUNCE_TIME),
       CALL HORIZ_MOTION (BOUNCE_TIME) ASSIGN (HORIZ_DIST),
       TIME = TIME + BOUNCE_TIME,
    END
CLOSE DROP,
```

Complete the program by writing all necessary declarations, initializations, procedures, and functions

7B In exercise 5A, a program was written to compute the value of a definite integral of the SQRT function using Simpson's rule Modify that program to compute the value of a definite integral of a function of the form $f(x) = ax^3 + bx^2 + cx + d$ Assume that the quantities a, b, c, d, initial, final, and epsilon are available in that order on channel 5

7C The increased modularity and readability brought about by the use of procedures and functions is not without cost Procedure and function calls are typically somewhat expensive in terms of computer time, and their over-use can unnecessarily slow down a program

For example, in problem 7A, the function HORIZ_MOTION could easily be eliminated Furthermore, on the last bounce, the height and time of the next bounce are computed, even though they will never be used Assuming that efficient use of computer time is here of primary importance, rewrite the solution so as to eliminate these two sources of inefficiency

8.0 I/O AND CHARACTER STRINGS

The HAL/S I/O statements, READ, READALL, WRITE and FILE, are designed to provide a convenient interface to external devices used for software checkout and non-flight applications. The READ, READALL, and WRITE statements perform sequential character I/O to such devices as card readers and line printers. The file statement transfers binary (unformatted) data to and from random-access devices such as drums and disks. These statements are all designed to provide the basic capability of getting data in and out of a HAL/S program with a minimum of programmer effort.

For sophisticated ground applications, the simplicity of these statements can be a disadvantage when highly formatted output is required. To give the programmer complete control over input and output formats for those applications that require it, HAL/S provides a comprehensive set of character manipulation facilities. Any data type may be converted to a character string, operations on the resulting string can produce any desired representation of the original data.

Although most flight computers do not have interfaces to character devices such as line printers, it is common practice to use ground based computers for early checkout of HAL/S code. HAL/S I/O statements can then be used to address the wide range of external devices (peripherals) found on such computers.

8.1 THE WRITE STATEMENT

The WRITE statement has already been used in the examples of the previous chapters. A typical instance was

```
WRITE(6) 'THE ANSWER IS', V;
```

Although this statement was not fully described at the time, the assumption was made that the string "the answer is" and the *value* of V (a scalar) would come out on some sort of printer. The following paragraphs describe the manner in which the output is sent to a particular device and the format in which it is printed.

The routing of output to a particular device is controlled from outside of the HAL/S program. Each WRITE statement specifies a *channel number* (in this case, channel 6). A channel may be thought of as a virtual device or as a port between the HAL/S program and some peripheral. HAL/S defines ten channels, numbered zero through nine, which are used in READ and READALL statements, as well as in the WRITE statement. At the HAL/S level, all channels are equivalent, it is only at execution-time that the channels are associated with actual devices. This association is made in an implementation dependent manner. It is usually done through some type of "job control language" or through commands at an interactive terminal. The appropriate HAL/S User's Manual must be consulted for details. In most systems, however, channel 6 is automatically associated with a line printer.

The channel number used in HAL/S I/O statements must be an integer expression which is computable at compile time (i e., composed entirely of literals, constants, and the basic arithmetic operators). It is good practice to give a name to each channel via the REPLACE statement, as shown below

```
REPLACE PRINT BY "6",
REPLACE CARDS BY "5",
REPLACE TERMINAL BY "7",
DECLARE I INTEGER, S SCALAR, D SCALAR DOUBLE,



READ(CARDS) I, S, D,
WRITE(PRINT) I, S, D,
   etc
```

Naming channels in this way has several advantages First, if the channels are well named the program will be more readable Second, it is easier to change the number in one REPLACE statement than the channel numbers in a collection of WRITE statements Finally, it is possible to find all of the I/O statements which use a particular channel by looking up the cross reference for the channel name The naming could alternately be done by declaring integer CONSTANTs

After the channel number, the remainder of the WRITE statement consists of a series of expressions There may be any number of expressions of any datatype Any construct which has been termed an expression in this book may be used in a WRITE statement. In the previous examples, the expressions have all been simple variables, but they *may* be of any complexity Thus, values that are needed only for output need not be stored in a variable A program to compute one of the roots of a quadratic equation given scalar coefficients A, B and C, might consist only of

```
READ(5) A, B, C,
WRITE(6) (-B + SQRT(B**2-4 A C))/2 A,
```

When any type of data aggregate (e g , VECTOR, ARRAY) is written, it is first unraveled into its individual integer, scalar, character, or bit components These components or array elements are then transmitted to the external device The sequence is the same as was described in conjunction with shaping functions in chapter six For instance

```
DECLARE M ARRAY(2) MATRIX,
WRITE(6) M,
```

results in the components of M being transmitted in the sequence

```
M$(1 1,1),M$(1 1,2),M$(1 1,3),M$(1 2,1),M$(1 2,2). M$(1 3,3),
M$(2 1,1)  M$(2 3,3)
```

When a data aggregate is unraveled in a WRITE statement, the original structure may not be retained * In the absence of the I/O control functions (discussed in the next section), all of the output from a single WRITE statement is placed on as few lines as possible, with only spaces separating the operands and the elements of each operand The number of spaces placed by default between successive values (termed the *default tab*) is implementation dependent

After the operands of the WRITE statement are reduced to a sequence of Integer, Scalar, Character, or Bit components, each component is converted to its *standard* external format, which is a character representation of its value. Each of the four basic data types above has its own format

The standard external format of an integer is a string of decimal digits, preceded by a minus sign if the integer is negative Enough leading blanks are appended to make the length of the resulting string constant for all integers of a given precision. This standard length varies from compiler to compiler, but is always large enough to contain any possible integer value Leading zeros are never included in the representation of an integer The following table shows the output format of a few integer values for a compiler which assumes an integer field width of 6

Value	Standard External Format
0	0
256	256
–32,768	–32768
–2	–2

Double precision integers have the same format, except that the field width is approximately twice as large.

The standard external format of scalars is scientific notation in a fixed-width field Scalars always take the form "bd dddE±dd" or "–d dddE±dd", where each "d" represents a decimal numeral Exactly one non-zero digit always appears to the left of the decimal point and positive numbers are always presented with a leading blank The number of digits to the right of the decimal point and the number of digits in the exponent are constant for any particular version of the compiler These numbers are always chosen so that all of the precision contained in the scalar can be presented The fixed field width simplifies the writing of code to re-format scalar values as will be seen in subsequent sections The following table illustrates the output representation of various scalar values on a computer with an eight digit mantissa and a two digit exponent

Value	Standard External Format
pi	3.1415927E+00
1/2	5 0000000E–01
–3 1/8	–3 1250000E+00
0001	1 0000000E–04
–1,000,000	–1 0000000E+06
0	0 0

*Some implementations will print matrices one row per line automatically, but this is not a language requirement

Note in the table above that zero is treated as a special case Double precision scalars are presented identically except that the standard width of the mantissa is greater

The remaining data types, character and bit (including BOOLEAN), each have two standard external formats These formats are very similar, but one is more suitable for printed listings and the other is more suitable for output that is to be read back in by another HAL/S program

The programmer specifies which format is to be used for character and bit output by means of the device directive The device directive is not a HAL/S statement, it is a command to the compiler which affects the way that subsequent WRITE (and READ) statements are interpreted The device directive specifies whether the output on a particular channel is *paged* (the format suitable for printing) or *unpaged* (the machine-readable format)

Paged output is organized into lines and pages Since the WRITE statement is most frequently used to obtain printed diagnostics and results, paged output is generally the default

Unpaged output is simply a stream of data values in a format compatible with the READ statement To designate a particular channel as unpaged, the device directive is used, as shown below

```
column 1              channel number 0–9
↓                     ↓
D DEVICE CHANNEL=6 UNPAGED
                                ↑
                                no semicolon
```

Compiler directives may vary from implementation to implementation All present compilers include the device directive as shown above Other directives are described in HAL/S Users Manuals These directives should not be considered as executable statements The presence of a device directive anywhere in a compilation governs *all* uses of the specified channel

The standard external format of character strings on a paged file is simply the content of the string, with no conversions or padding On an unpaged file, the character string is enclosed in single quotes (') The output from the statement

```
WRITE(6) 'The answer is', V,
```

will be

```
THE ANSWER IS 7 5836210E+05
```

on a paged file, but will be

```
'THE ANSWER IS' 7 5836210E+05
```

on an unpaged file

The standard external format for bit strings is a series of ones and zeros As in character strings, bit output is enclosed in quotes on an unpaged file A BOOLEAN consists of a single bit, so there are only four possible outputs as shown below

Boolean Value	Paged Output	Unpaged Output
TRUE/ON	1	'1'
FALSE/OFF	0	'0'

Longer bit strings (see Chapter 13) are output with a blank between every set of four bits to enhance readability The value HEX'1234' would be output as 0001 0010 0011 0100 on a paged file, and as '0001 0010 0011 0100' on an unpaged file

For character and bit types, only the unpaged format is compatible with the READ statement Since these types are of a variable length and may contain embedded blanks, the quotes are needed to indicate the end of one value and the start of the next

In summary, the WRITE statement will evaluate a list of expressions of any data type, convert the resulting values to their standard external formats, and transmit these to the device which has been associated with the specified channel There are no restrictions on the expressions in a WRITE statement, and in no case will any data be lost in the translation to the standard external form As a result, the WRITE statement is extremely easy to use if the format of the output is of little concern, this makes it convenient for diagnostics, but less appropriate for report generation

Exercises

8 1A Why is it generally considered good programming practice to give a name to each channel for I/O functions and use the HAL/S REPLACE statement to assign the channel number?

8 1B What happens when an executing program encounters a HAL/S WRITE statement followed by a list of expressions? What limitations are there on the expressions that are legal in a WRITE statement?

8 1C Given the following declarations

```
DECLARE S SCALAR,
        I INTEGER,
        V VECTOR,
        M MATRIX,
        B BOOLEAN,
        C CHARACTER,
```

Which of these WRITE statements will produce output compatible with the HAL/S READ statement

a) On a PAGED device?

b) On a UNPAGED device?

   1) WRITE(6) S, I, V, M,
   2) WRITE(6) 'I = ', I, ', V = ', V,
   3) WRITE(6) V$1, V$3, V$2, B,
   4) WRITE(6) B, C,
   5) WRITE(6) S, M, V$(2 TO 3), I,

8 2 I/O CONTROL FUNCTIONS

When the statement

WRITE(6), M,

where M is a matrix, is executed, the three-by-three structure of M is lost The arrangement of the components of M depends only on the field width of a scalar, the amount of the default tab, and the maximum number of characters per printed line If the width of a scalar is 13, the default tab is 5 and a line is 132 characters, then seven components will be printed on the first line, and the remaining two on a second line To obtain a better arrangement, the following WRITE statement may be used

WRITE(6) M$(1,*), SKIP(1), COLUMN(1), M$(2,*), SKIP(1), COLUMN(1), M$(3,*),

This statement will cause one row of the matrix to be printed on each output line

SKIP and COLUMN are I/O control functions Syntactically, they resemble other functions, but they may only be used as arguments to the sequential I/O statements, WRITE, READ, and READALL Each has a single argument which may be any integer or scalar expression, if the expression is scalar-valued, it is rounded to the nearest integer These functions do not return a value, but only control the location in a file where subsequent data will be read or written.

The I/O control functions may be thought of as moving a read/write mechanism across a two dimensional medium The SKIP, LINE, and PAGE functions cause vertical movement and the COLUMN and TAB functions cause horizontal movement, In the example above, "SKIP(1), COLUMN(1)" moves the write mechanism to the beginning of a new line The SKIP function causes relative movement (down one line), and the COLUMN function causes absolute positioning (to the first column of the new line)

The sequence, "SKIP(1), COLUMN(1)", is implied at the beginning of each WRITE statement This automatic positioning will be overridden if the WRITE statement has explicit horizontal *and* vertical positioning functions prior to the first data operand If only horizontal *or* vertical positioning is specified, then the default movement is partially overridden In the statement

WRITE (6) COLUMN(10), M$(1,*),

the default horizontal positioning to column one is overridden, but the default vertical positioning to the next line is not. Likewise, the statement

WRITE(6) M$(1,1), TAB(12), M$(1,2), TAB(12), M$(1,3),

would leave twelve blanks between the end of one component and the start of the next. Unless overridden by explicit horizontal motion commands, a TAB function is implied between each pair of data operands to the WRITE statement. The amount of the default TAB is implementation dependent.

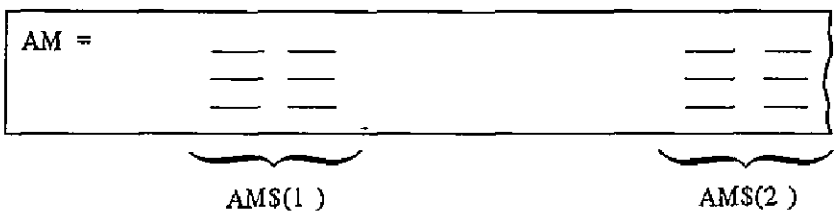
Using these functions, an array of matrices

DECLARE AM ARRAY(2) MATRIX(3,3),

can be output in a readable form by

WRITE(6), SKIP(2), COLUMN(10), AM$(1 1,*), TAB(20), AM$(2 1,*), SKIP(1),
COLUMN(1), 'AM=', COLUMN(10), AM$(1 2,*), TAB(20), AM$(2 2,*), SKIP(1),
COLUMN(10), AM$(1 3,*), TAB(20), AM$(2 3,*), SKIP(2),

yielding



The effect of the remaining I/O control functions, LINE and PAGE, depends on whether they are used on a paged or an unpaged channel. On a paged channel, the LINE function's argument must be in the range one to the maximum number of lines per page. The device mechanism is moved *forward* until the current line number is the same as that specified in the LINE function. This may cause the device mechanism to cross a page boundary. The most common use of the LINE function is to advance to the top of the next page, as in

WRITE(6) LINE(1), 'This is a page header',

When used on an unpaged channel, the LINE function causes movement to an absolute line number within the entire file.

The PAGE function may only be used on paged files. PAGE(n) results in relative movement by "n" pages. The current column and line numbers are not affected. A typical use of the PAGE function is to skip over unwanted pages of header on input.

The preceding paragraphs apply equally to all implementations of the HAL/S language. The principal variations between implementations are the number of columns per line (and lines per page) and the result of requesting backward movement of the read/write mechanism

The statement

WRITE(6) 'results follow', TAB(–14), '____ ____',

may have any of several results, depending on the compiler in use On some systems, the two character strings may both be printed in the same columns of the same line, yielding *RESULTS FOLLOW* On other systems, the second character string may overlay the first, yielding just the underscores Similarly, backwards line movement may or may not be supported and may be device dependent the effect of executing SKIP(–1) may vary from system to system The relevant User's Manual should always be consulted before requesting negative column or line movement

The following table summarizes the I/O control functions

I/O Control Function	Operation
SKIP(K)	Relative line movement Line = (Line + K) mod page size
LINE(K)	Absolute line movement Line = K
TAB(K)	Relative column movement Col = Col + K
COLUMN(K)	Absolute column movement Col = K
PAGE(K)	Relative page movement Page = Page + K

Exercises

8 2A Consider the following HAL/S statements

```
DECLARE ARRAY(3) MATRIX, MAT_ARR1, MAT_ARR2,

WRITE(6) MAT_ARR1, MAT_ARR2,
```

a) Describe what the resulting output would look like

b) Change the WRITE statement such that the resulting output will be formatted as thus

[MAT_ARR1_1]	[MAT_ARR2_1]
[MAT_ARR1_2]	[MAT_ARR2_2]
[MAT_ARR1_3]	[MAT_ARR2_3]

8 2B For each of the I/O control functions below, which of the following statements apply to its use in HAL/S WRITE statements?

a) default characteristics (implied unless overridden)

b) causes absolute vertical movement

c) causes relative vertical movement

d) causes relative horizontal movement

e) causes absolute horizontal movement

1) LINE(1)	5) COLUMN(1)
2) SKIP(1)	6) SKIP(0)
3) TAB(20)	7) SKIP(5)
4) PAGE(2)	

8 3 THE READ STATEMENT

The syntax of the HAL/S READ statement is also quite simple Some examples (e g , "READ(5) A, B, C,") have already appeared in this manual, the general form is not much more elaborate The READ statement consists of the word READ and a channel number followed by a list of variables and/or I/O control functions There are no restrictions on the variables, and the I/O control functions work the same way as in the WRITE statement

When any type of data aggregate appears in a READ statement, the components are filled in the "natural sequence", i e , in the same order in which they would be written In the code

```
DECLARE A SCALAR, V VECTOR, I ARRAY(2) INTEGER DOUBLE,
READ(5) A, V, I,
```

data from the external file will be assigned in the sequence

```
A, V$1, V$2, V$3, I$1, I$2
```

If the file was originally produced (stored on disk, punched on cards, etc), by a HAL/S WRITE statement, its contents will be in the appropriate format for the READ statement Except for character and bit strings on paged files, the standard forms produced by the WRITE statement are all acceptable on input

Input data prepared manually may be written in free format, all of the following lines are acceptable input for the READ statement above

a) 0, 0, 0, 0, 0, 0

b) 1 3E5 3 271E+06 001 24 –2

c) 1, 2 3 4, 5 6

The examples illustrate several points First, it is not necessary to distinguish between integer and scalar values Any sequence of characters which comprise a valid integer or scalar literal (as described in chapter two) is suitable to be read into either an integer or a scalar, non-integral values read into an integer will be rounded

Individual values (in this case, numbers) in the input file must be separated by blanks or other delimiters One or more blanks, a single comma, or a single comma and any number of blanks are all equivalent. Multiple commas are a special case, which indicate "missing data" If the input file contained

1, , 2, 3, 4, 5

then the value of the second scalar in the READ statement above (V$1) would not be changed

When a semicolon is encountered in the input stream, the current READ statement is terminated If the input consisted of

1 5, 2 6,

then only two values would be read, regardless of subsequent values and punctuation in the file This fact can be useful when a program must process a variable number of input values For instance, a program to sum a sequence of numbers could be coded as

```
M   ADD:
M   PROGRAM,
M      DECLARE TOTAL SCALAR INITIAL(0) AUTOMATIC;
M      DECLARE A ARRAY(100) SCALAR INITIAL(0);
M      READ(5) [A];
M      DO FOR TEMPORARY I = 1 TO 100 UNTIL A  = 0;
S                                           1

M         TOTAL = TOTAL + A ;
S                          I

M      END;
M      WRITE(6) 'TOTAL IS ', TOTAL;
M   CLOSE ADD,
```

One valid input to this program could be

–3 95, –17 31, –9 93, 572 35, –250, +1 10, – 45, +7 50,

In this case, the READ statement would terminate when the semicolon was reached, leaving the rest of the array (A$ (9 TO 100)) equal to zero

As illustrated above, a READ statement may take data from many lines of a file Lines will be processed until either a semicolon is reached or values are found for all of the operands of the READ statement The end of each line of input (e g , card column 80) serves as a delimiter equivalent to a blank Hence, individual values may not be split across lines

As in the other sequential I/O statements, WRITE and READALL, a SKIP(1), COLUMN(1) operation is implied at the beginning of each READ statement This may be overridden by the same means used in the WRITE statement, e g ,

```
READ(5) SKIP(0), TAB(0), X,
```

can be used to read data to the right of a semicolon which terminated the previous READ statement If the input data happens to be stored in fixed card columns, then the TAB and COLUMN functions can be used to skip over unwanted data

Any attempt to read past the end of a file will result in a runtime error Chapter ten describes a mechanism for recovering from this and other errors

EXERCISES

8 3A Let the program ECHO begin as follows

```
ECHO  PROGRAM,
   DELCARE INTS ARRAY(3) INTEGER, INITIAL(1),
         SCALS ARRAY(3) SCALAR, INITIAL(0),
   READ(5) INTS, SCALS,
```

What will INTS and SCALS contain given the following inputs?

a) 8, 7, 6 55, –1, 2 25E2, 4,

b) –1E–1,,,7 2,

c) 2 49,,2 51,2 49,,2 51,

8 3B Suppose input intended for the program ECHO of problem 8 3A has been formatted as follows

Col 1	Col. 8	Col 78
↓	↓	↓
INTS	3 4 5	00000001
SCALS	6 1 7 2 8 3	00000002

Modify the READ statement in ECHO to ignore the labels on the left and the sequence numbers on the right, and read in the values for INTS and SCALS properly.

8 4 CHARACTER STRINGS

A HAL/S character variable may contain a string of characters, the number of characters is allowed to vary at runtime from zero up to a maximum specified in the declaration of the variable The character datatype is declared in the same general way as other data types, e g ,

```
DECLARE STARS CHARACTER(5) INITIAL('*****'),
```

The variable STARS is a character string of maximum length five and initially containing five asterisks Each character variable has both a maximum length and a current length The current length is adjusted every time the variable is assigned, though it can never become greater than the declared maximum If the length of the string on the right-hand side of an assignment exceeds the maximum length of the target variable, characters are truncated from the right before assignment In the code below, RATING starts with a length of zero (it is initialized to the null string), but after the assignment the current length becomes three

```
DECLARE RATING CHARACTER(5) INITIAL(''),
DECLARE QUALITY INTEGER INITIAL(3),
RATING = STARS$(1 TO QUALITY),
```

As shown, the general form of character subscripting is the same as vector subscripting, except that the width of a partition does not have to be known at compile-time

In addition to subscripting a character string to pick out a single character or a substring, HAL/S provides an operator for putting two strings together This is the catenation operator, denoted by the keyword "CAT" or by the sign "| |" The effect of this operator is to append the right-hand operand to the end of the left-hand operand

```
'ABC' | | 'DEF'
```

yields

```
'ABCDEF'
```

Character strings may also be compared with each other, as in

```
IF RATING NOT = '***' THEN EXIT,
```

and may be compared for "greater than" or "less than" in order to sort them alphabetically The latter capability is affected by the collating sequence and is therefore implementation-dependent More details can be found in the appropriate Users Manual

HAL/S also provides a set of built-in character functions (listed in Appendix A) The following paragraphs describe some of these functions as well as providing some practical examples of character operations

One of the major uses of character variables and operations in HAL/S is formatting output. In the WRITE statement below, the value of the integer variable N will be inserted in a line of output

```
DECLARE N INTERGER,



WRITE(6) 'the answer is '||N||' fps',
```

If N is six, the output from this statement will look like

```
THE ANSWER IS         6 FPS
```

This statement illustrates an important rule. Whenever an integer or scalar is used in a character expression it is converted to its standard external format (a character string). The standard external format of an integer includes leading blanks. These blanks can be removed by means of the TRIM built-in function, as shown below

```
WRITE(6) 'the answer is '||TRIM(N)||' Fps',
```

This statement will produce

```
THE ANSWER IS 6 FPS
```

The TRIM function removes all leading and trailing blanks from a character string. Its argument must be a character expression, thus N is converted to character before the invocation on TRIM in the statement above.

Similar character functions are RJUST and LJUST, which *add* leading and trailing blanks, respectively. Each of these functions takes two arguments, a character expression and a field width. These functions right or left-justify the value of the character expression in a field of specified width. With N= 6, RJUST(N,2) yields ' 6' and LJUST ('XYZ',4) yields 'XYZ '.

Note that within the quotes of a character literal, blanks are treated the same as any other character. *Any* character may be used in a quoted string.

Like variables of any data type, character strings may be arrayed The following function could be used to display the value of a boolean (B) in the format specified by an integer (TYPE)

```
M   STATE:
M  .FUNCTION(B, TYPE) CHARACTER(5);
M      DECLARE B BOOLEAN,
M              TYPE INTEGER,.
M      DECLAPE YES ARRAY(4) CHARACTER(5) INITIAL('TRUE', 'ON', 'OPEN', 'VALID');
M      DECLARE NO ARRAY(4) CHARACTER(5) INITIAL('FALSE', 'OFF', 'SHUT', 'ERROR');
E         .
M      IF B THEN
E                 ,
M         RETURN YES      ;
S                   TYPE:

M      ELSE
E                ,
M         RETURN NO      ;
S                  TYPE:

M   CLOSE STATE;
```

This function could be invoked as shown below

DECLARE BOOLEAN INITIAL(OFF), VALVE, POWER,
WRITE(6) 'VALVE=',STATE(VALVE,3),'POWER=',STATE(POWER,2),

This example would produce

VALVE=SHUT   POWER=OFF

The concepts of maximum length and current length apply to *each element* of an array, and to the value returned by a character function The maximum lengths of all elements of a character array are equal, but the current lengths may vary Thus, the length of the value returned by STATE can vary from two to five The maximum length on the function header can never be exceeded, however, if "RETURN 'ABCDEFH'," was executed, the string would be truncated at the right yielding 'ABCDE'

It should be noted in the example above that the n^{th} element of a character array such as YES is represented by "YES$(N)" and not "YES$N" The trailing colon must be supplied to indicate the absence of component subscripting just as in arrays of vectors, matrices and Bit Strings (Booleans) As before, both array and component subscripts may be supplied if needed YES$(3 2) is the second character of the third element of YES 'P'

A few examples of automatic conversion to character type have appeared above It is also possible to explicitly convert to character type via the CHARACTER shaping function This function is syntactically identical to the INTEGER, SCALAR, VECTOR, and MATRIX shaping functions described previously It converts its argument or arguments to their standard external formats It has an additional form that allows conversions to octal or hexadecimal as shown below

WRITE(6) CHARACTER$(@OCT)(BIT(N)),

If the integer N is equal to 29, this statement will produce the output

'0000000035'

When the CHARACTER function is subscripted with a radix (@OCT or @HEX), its operand must be a bit string The BIT function above is not fully described until Chapter 13, but in this case it merely returns a bit pattern equivalent to its argument

Another use of the character manipulation facilities is reading data that is not in the standard HAL/S format Integer data that has been punched on cards in the format shown by the table below could be read in by the HAL/S statements which follow it

Input Format

Columns	Description
1-3	case number
4-5	age
6	1=male, 2=female
7-10	X factor

Example of Input

1152612781

```
M | AGE
M | PROGRAM,
M |    DECLARE C CHARACTER(80),
M |    DECLARE INTEGER,
M |               CASE_NUM, AGE, SEX, X,
E |               ,
M |    READALL(5) C;
E |                        ,
M |    CASE_NUM = INTEGER(C         );
S |                         1 TO 3

E |                    ,
M |    AGE = INTEGER(C        ),
S |                    4 TO 5

E |                    ,
M |    SEX = INTEGER(C ),
S |                    6

E |                  ,
M |    X = INTEGER(C         );
S |                  7 TO 10

M | CLOSE AGE,
```

This would yield the following values

```
CASE_NUM = 115
AGE      = 26
SEX      = 1
X        = 2781
```

When the argument to the INTEGER shaping function is a character string, all of the characters must be in the range 0–9 (i e , comprise a valid integer) Thus, this code would not work if the CASE_NUM field (for instance) was coded with leading blanks instead of leading zeros The TRIM function can be used to make the program more tolerant as in

```
CASE_NUM = INTEGER(TRIM(C$(1 TO 3))),
```

The READALL statement used to obtain C from channel 5 (probably a card reader) will be fully described in the next section of this chapter

Since the standard external format for scalars is not always convenient, a character function like the one below can be used to write a more readable XX YYY notation

```
M   REFORMAT:
M   FUNCTION(X, DECIMALS, WIDTH) CHARACTER(20),
M      DECLARE X SCALAR,
M              DECIMALS INTEGER,
M              WIDTH INTEGER,

C    X IS THE NUMBER TO BE CONVERTED, DECIMALS IS THE NUMBER OF
C    DIGITS TO BE PRINTED AFTER THE DECIMAL POINT, AND WIDTH IS
C    THE TOTAL LENGTH OF THE STRING RETURNED

M      DECLARE Y SCALAR;
M      DECLARE C CHARACTER(20);
M      DECLARE S CHARACTER(1);
M      DECLARE ZEROS CHARACTER(20) CONSTANT(CHAR(20)'0');
M      IF X < 0 THEN
M         DO,
M            Y = -X;
E            ,
M            S = '-',
M         END,
M      ELSE
M         DO,
M            Y = X;
E            ,
M            S = ' ';
M         END;
E      ,                                DECIMALS
M      C = CHARACTER(INTEGER        (10          Y)),
S      .                   @DOUBLE

E                 ,
M      IF LENGTH(C) < DECIMALS THEN
E         ,     ,                          ,
M         C = ZEROS                     || C,
S                  1 TO DECIMALS-LENGTH(C)

E                   ,  ,                          ,
M      RETURN RJUST(S || C              || ' ' || C              TO#), WIDTH;
S                         1 TO #-DECIMALS          #-DECIMALS-1

M   CLOSE REFORMAT,
```

With the function before,

```
WRITE(6) REFORMAT(SQRT(2), 3, 5),
```

would yield

'1 414', i e , a five character field with three decimal places

Two new features are introduced in this example First, the expression "CHAR(20)'0'" is a shorthand notation for the string consisting of twenty zeros It is a character literal which may also be used in an assignment statement such as

```
C = CHAR(80)' ', /*blank card*/
```

An additional built-in function, LENGTH, is also used LENGTH takes a character variable or expression as an argument and returns an integer representing its *current* length

The REFORMAT function shown here has one deficiency It does not check X for being too large for a field of width WIDTH A good fixup would be to return part of X in scientific notation if it is too large for the field This improvement is left as an exercise

Exercises

8 4A Which of the following expressions are legal character subscripts? Which are legal vector subscripts? (Assume all variables are of integer type)

a) (4)
b) (I+1)
c) (7 AT 3)
d) (2 TO I-2)
e) (6 AT I+J)
f) (I TO J)
g) (K TO K-1)

8 4B What will the output be from the following program?

```
PROG_B PROGRAM,
    DECLARE CH CHARACTER(15) INITIAL('ABC'),
    REPLACE PRINT BY "WRITE(6)",
    PRINT CH, CH||CH,
    CH = '123'||CH||'456',
    PRINT CH$(1 TO 5), CH$(5 TO #),
    CH = CH$(1 TO 2)||CH$(3 AT #-5),
    PRINT CH, CH(#-2 TO #),
CLOSE PROB_B,
```

8 4C Given the following declarations and assignments, which of the following comparisons are true? Assume the ‘A’ < ‘B’ < < ‘Z’

```
DECLARE C15, CHARACTER(15)
DECLARE CHARACTER(1)
    C11, C12,



C15 = 'A',
C11 = 'A',
C12 = 'B',


    .
```

a) ‘A’ = C11

b) C15 = ‘A’

c) C15 = C11

d) C15 ¬= C12

e) ‘A’ < C12

f) ‘A’ < ‘AB’

g) C11 < ‘AB’

h) C15 < C11 CAT C12

i) ‘1’ < ‘I’

j) ‘1’||‘ ’ < ‘;’

8 5 OTHER HAL/S I/O CONSTRUCTS

The READ and WRITE statements already described allow data to be transferred between a HAL/S program and a sequential character oriented file The data is always transferred in a standard format according to its type, though I/O control functions allow arbitrary positioning of the data Since character operations allow output reformatting, the addition of an unformatted read (READALL) gives the programmer complete control over sequential character files

HAL/S also supports random-access files, which do not necessarily contain character data, via the FILE statement, and provides some features which aid in transferring data to and from special purpose sensors and effectors

Exercises

8 5A What HAL/S data types may be read using the READALL statement?

8 5B How are character strings suitable for input via the READALL statement different from those suitable for input via the READ statement?

8 5 1 The READALL Statement

One example of the READALL statement,

```
DECLARE C CHARACTER(80),
READALL(5) C,
```

was used in the previous section Aside from the READALL keyword, the format of this statement is exactly that of the READ statement, although a restriction is made that all variables be of character type

The READALL statement can input up to one line of characters from a HAL/S channel, the characters read are placed directly in the character variable or variables without any special interpretation of the delimiters blank, comma, and semicolon Characters are transferred until either all of the variables have been filled to their declared maximum lengths, or the entire line has been read, whichever comes first Unless the READALL statement begins with I/O control functions (e g SKIP, LINE) the device mechanism is advanced to the beginning of a new line before the first character is transferred

When a list of variables or a character array is specified, each variable or element is filled in turn There is no automatic movement of the device mechanism between variables This allows a line of data to be broken into fields, a card could be read as eight 10-character fields by

```
DECLARE CARD ARRAY(8) CHARACTER(10),
READALL(5) CARD,
```

I/O control functions may also be used with READALL Using the declaration above, just the first and last fields could be read by

```
READALL(5) CARD$(1 ), COLUMN(71), CARD$(8 ),
```

READALL uses the same set of channels as READ and WRITE Input and output should not be mixed on the same channel, but READ and READALL may both be used on the same input file or even the same card as in the following example

```
M | OUTER:
M | PROGRAM;
  |
M |    DECLARE SCALAR,
M |                PHI, ALPHA,
M |    DECLARE INITIAL_POSN VECTOR DOUBLE;
M |    DECLAPE MODE INTEGER,
M |            PRINT BOOLEAN,
  |
C |       .
C |       .
C |       .
  |
M | INITIALIZE:
M | PROCEDURE,
M |    DECLARE V NAME CHARACTER(8);
M |    REPLACE INFILE BY "5",
M |    DO WHILE TRUE,
M |       READALL(INFILE) VNAME,
M |       VNAME = TRIM(VNAME),
M |       IF VNAME = 'PHI' THEN READ(INFILE) SKIP(0), COLUMN(9), PHI;
M |       IF VNAME = 'ALPHA' THEN READ(INFILE) SKIP(0), COLUMN(9), ALPHA;
M |       IF VNAME = 'I_POSN' THEN READ(INFILE) SKIP(0), COLUMN(9), INITIAL_POSN,
M |       IF VNAME = 'MODE' THEN READ(INFILE) SKIP(0), COLUMN(9), MODE,
M |       IF VNAME = 'PRINT' THEN READ(INFILE) SLIP (0), COLUMN(9), PRINT,
M |       IF VNAME = 'END' THEN EXIT;
M |    END,
E |
M |    IF PRINT THEN
E |                                 -
M |       WRITE(6) PHI, ALPHA, INITIAL_POSN, MODE;
M | CLOSE INITIALIZE,
  |
C |       .
C |       .
C |       .
  |
M | CLOSE OUTER;
```

The INITIALIZE procedure above could be used to read initial values for a simulation run. The input lines would consist of a variable name in the first eight columns followed by an initial value in the standard external format *for that data type*, e g

```
PHI                00137
PRINT              '1'
I_POSN             1, 1, 1
END
```

This type of initialization module takes little memory and is fairly efficient if there are not too many variables Its main advantage is that it is very easy to code, particularly if a parameterized REPLACE macro is used to abbreviate the repeated code

```
REPLACE TEST(ID, VAR) BY "
   IF VNAME = ID THEN READ(5)
   SKIP(0), COLUMN(9), VAR",


TEST('ALPHA', ALPHA),
TEST('I_POSN', INITIAL_POSN),

etc
```

8 5.2 The FILE Statement

The FILE statement is used to read and write random access files These files (which are numbered separately from *channels*) are organized into records which may be accessed in *any* sequence Generally speaking, any record may be read or written in the same amount of times as any other (hence the term "random access")

The FILE statement has two forms

FILE(number, address = expression,

and

variable = FILE(number, address),

The construct FILE(number, address) is called a *file expression* When the file expression is used on the left of the equals sign (the output file statement), the value of "expression" is written to the record specified by "address" on the file specified by "number" When the file expression is used on the right hand side (the input file statement), the record denoted by the file expression is read into "variable"

The FILE statement is highly implementation-dependent The appropriate User's Manual should be consulted before it is used

The "number" and "address" operands of the file expression may be any integer or scalar arithmetic expression "Number" must be computable at compile-time If the expression is scalar, it will be rounded to the nearest integer The legitimate ranges of these integers are implementation dependent

There are no restrictions on "expression" in the output file statement All of the following statements are legal

```
DECLARE MATRIX(10,10), M1, M2,
DECLARE A ARRAY(99) INTEGER,
DECLARE C CHARACTER(20),
DECLARE I INTEGER INITIAL(17),
REPLACE HIST BY "5",
FILE(HIST, 12) = M1,
FILE(5, I+1) = M1 + M2**T,
FILE(HIST,8) = M1$(2 TO 7,*),
FILE(HIST,9) = A+1,
FILE(HIST,10) = C || I,
```

There are, however, some restrictions on "variable" in the input file statement These are the same restrictions that apply to assign parameters of procedures "variable" must be one of the following

1 An unsubscripted variable
2 An *entire* array element
3 A *contiguous* partition of a *single* vector or matrix

The following input file statements are all legal

```
M1 = FILE(HIST,2),
C = FILE(3,3),
A$I = FILE(4,4),
M1$(I,*) = FILE(5,6),
```

It is *not* possible to read into a non-contiguous partition of a MATRIX (M1$(*,1)) or an array partition (A$(5 TO 10)) or a partition of a character string (C$(3 TO #))

Both versions of the file statement cause the transfer of unformatted binary data Thus, if the file statements are to be used reliably, *a record should always be read into a variable of the same type and organization as the expression that was written* Since the compiler cannot know how a file was originally written, it is up to the programmer to ensure compatability

8 5 3 Avionics I/O

HAL/S does not include any specific avionics I/O statements, principally due to the fact that there is currently no standardization of airborne I/O systems Some flight computers have one or more independent I/O processors or channels with their own unique instruction sets Other computers either have CPU instructions for I/O or have a section of memory that is "hard wired" to external devices (e g storing into location 5432 [octal] might lower the landing gear)

Operating systems also vary widely in this regard In some systems I/O is requested by application programs, while in others it is all done "automatically" on a periodic basis Finally, every system will have a different complement of sensors, displays, effectors, etc , each of which may have its own unique formatting and protocol requirements

Although there is presently no way to implement generalized avionics I/O as a HAL/S statement, the language does provide a number of features that allow individual systems to be tailored

1 Structure (chapter 9) and compool (chapter 11) templates allow a section of memory to be mapped into a collection of variables of assorted types

2 Procedures and functions can be coded in assembly language and interfaced to a HAL/S program (see chapter 11)

3 Bit strings (chapter 13) allow low-level formatting via subscripting and logical operators (AND, NOT, etc)

4 I/O errors may be handled via the ON ERROR statement described in chapter 10

5 Event variables (chapter 12) allows waiting for I/O completion, and may trigger transactions when signalled

The following code illustrates some of the ways that I/O might be performed in alternate systems

```
M | ASSORTEDIO
M | PROGRAM,
M |    REPLACE GEARDOWN BY "INTEGER(OCT'5432')",
M |    DECLARE DONAVREAD EVENT,
M |    DECLARE MEMNAME ARRAY(32768) BIT(16) INITIAL(NAME(NULL)),
M |    STRUCTURE IOPARM
M |       1 DEVICE INTEGER,
M |       1 STATUS BIT(16),
M |       1 BUFFER NAME ARRAY(10) INTEGER,
M |       1 WORDS INTEGER,
M |    DECLARE FWDSENSORS IOPARM-STRUCTURE INITIAL(16, HEX'0', NULL, 27),
M |    DECLARE IO PROCEDURE NORMAL(1),
M |    REPLACE OPSYS BY '1",
M |    DO CASE OPSYS,
M |       /SVC(9),                                          /*PERCENT MACRO*/
E |                  +
M |       CALL IO(FWDSENSORS),                              /*ASSEMBLY LANGUAGE*/
M |       MEM          = ON,
S |          GEARDOWN

M |       SIGNAL DONAVREAD,                                 /*EVENT VARIABLE*/
M |       ,                                                 /*NO-OP*/
M |    END,
M | CLOSE ASSORTEDIO,
```

This program only indicates a few alternatives, there are many other possibilities

End Of Chapter Problems

8A Write a HAL/S program that will read, from channel 5, 2 arrays of character strings (5 elements per array, maximum 5 characters per string), remove leading and trailing blanks from each string, reverse each string, and write the results on channel 6 in the form

Column 5	Column 15
CHAR_ARR1_1	CHAR_ARR2_1
CHAR_ARR1_2	CHAR_ARR2_2
CHAR_ARR1_5	CHAR_ARR2_5

8B Write a HAL/S program to perform the following task

Input on channel 5 contains the names of 50 people, each consisting of a first name, one blank, and a last name Names are separated by commas, the maximum length of any name is 25 characters, and there are no blanks in the input except those following the last comma in a line (no name is broken across two lines) The final name is not followed by a comma

The program should read in all 50 names into an array, and write on channel 6 all names whose last name begins with 'S'

An example of possible program input is

SAMUEL COLERIDGE,CHARLES BAVOELAIRE,EMMY NOETHER,
WILLIAM SHAKESPEARE,TYCHO BRAHE,DAVID HILBERT, etc.

8C Write a HAL/S program that will read from channel 5 a 1- to 3- digit integer, and write on channel 6 the English equivalent, e g ,

173 → ONE HUNDRED SEVENTY-THREE
0 → ZERO
15 → FIFTEEN etc

9 0 STRUCTURES

HAL/S structures provide a means of collecting a group of variables under a single name This grouping capability has a number of uses, one of which is illustrated below Suppose a utility function which requires many parameters is defined at the outer level of a program and invoked from lower level code as shown below

```
M | OUTER
M | PROGPAM,
M |    DECLARE SCALAR,
M |               G1, G2,
M | UTIL.
M | FUNCTION(A, B, C, D, E) VECTOR;
M |    DECLARE A VECTOR;
M |    DECLARE SCALAR,
M |               B, D,
M |    DECLARE C INTEGER,
M |            E BOOLEAN,
  |
C |
C |      .
C |
  |
E |          -
M |    RETUPN A,
M | CLOSE UTIL,
M | NESTED
M | PROCEDURE,
  |
C |  A PROCEDURE WHICH INVOKES UTIL
  |
M |    DECLARE RESULT VECTOR,
M |    DECLAPE V VECTOR INITIAL(0, 1, 0),
M |    DECLARE SCALAR,
M |               S1, S2,
M |    DECLARE C INTEGER INITIAL(83),
M |            E BOOLEAN INITIAL(OFF),
  |
C |
C |      .
C |      .
  |
M |    S1 = G1 / 3,
M |    S2 = SIN(G1 + G2),
E |     -          -          .
M |    RESULT = UTIL(V, S1, C, S2, E);
  |
C |
C |
C |
  |
M | CLOSE NESTED,
  |
C |
```

It is advantageous to keep the actual arguments passed to UTIL (i e V, S1, S2, etc) declared at the lowest possible level because of the protection afforded by scoping rules, and to show that these variables "belong" with the NESTED code block On the other hand, some inefficiency results from passing all five parameters separately The code in the next figure shows how structures can be used to reduce the number of UTIL parameters to one

```
M  OUTER:
M  PROGRAM,
M     DECLARE SCALAR,
M                G1, G2,
M     STRUCTURE UTIL_PARM:
M        1 V VECTOR,
M        1 S1 SCALAR,
M        1 C INTEGER,
M        1 S2 SCALAR,
M        1 E BOOLEAN;
M  UTIL:
M  FUNCTION(X) VECTOR;
M     DECLARE X UTIL_PARM-STRUCTURE,

C        .
C        .
C        .

E                -
M     RETURN X.V;
M  CLOSE UTIL,
M  NESTED
M  PROCEDURE,
M     DECLARE RESULT VECTOR;
M     DECLARE LOCAL UTIL_PARM-STRUCTURE INITIAL(0, 1, 0, 0, 83, 0, OFF);

C   NOTE THAT THE TEMPLATE IS NOT REPEATED

M     LOCAL.S1 = G1 / 3,
M     LOCAL S2 = SIN(G1 + G2),

C        .
C        .
C

E        -             +
M     RESULT = UTIL(LOCAL),

C
C
C

M  CLOSE NESTED,

C
C
C

M  CLOSE OUTER,
```

Several new language constructs are used in this example. First is the statement beginning with "STRUCTURE UTIL_PARM." This statement creates a *structure template* named UTIL_PARM which defines the layout of the UTIL_PARM-STRUCTUREs declared later. In addition to structure declaration and initialization, the example shows references to the components of a structure, *structure terminals*, such as "LOCAL S1" and an entire structure, LOCAL.

The next section describes all of the constructs used in the example, although some of the more complex forms are deferred to the end of the chapter.

9.1 DECLARING AND REFERENCING STRUCTURES

In the statement

```
DECLARE LOCAL UTIL_PARM-STRUCTURE INITIAL(0,1,0,0,83,0,OFF),
```

the phrase "UTIL_PARM-STRUCTURE" takes the position usually occupied by a data type This is actually consistent syntax because X-STRUCTURE, where X is a template name, *is* a data type Hence, a template name with the word STRUCTURE attached by a hyphen can be used in most of the constructs from previous chapters which require a data type or "type specification" Examples include factored declare statements such as

```
DECLARE UTIL_PARM-STRUCTURE,
    LOCAL,
    X,
    Y INITIAL(1,2,3,4,5,6,True),
    ZERO CONSTANT(0,0,0,0,0,0,Off),
```

and function type specification, as in

```
SHAPE  FUNCTION(A,B,C,D) UTIL_PARM-STRUCTURE,
```

It is important to note that STRUCTURE by itself is not a data type The type of a structure is entirely defined by the layout of its template From this rule, and the description of parameter passage in chapter seven it follows that *when a structure is passed to a procedure or function, the template of the actual argument passed must be identical to the template of the formal parameter*

The conditions under which two templates are identical for purposes of data type matching (in parameter passage, assignments, etc) will be discussed in Section 9.2 However, the easiest way of assuring that two structures are of the same data type is to use the same template in their declarations In the example, the STRUCTURE statement which defines the UTIL_PARM template is part of the program level declare group It can be used in the declarations of X and LOCAL in nested routines because *the scoping rules for structure templates are the same as for declared variables* Thus, a template defined at the program level is global and may be used in declarations anywhere in the program

In addition to parameter passage, entire structures may be used in assignment statements and in the various I/O statements For example, a set of ten test cases could be run through the UTIL function by executing the following code

```
M   OUTER
M   PROGRAM;
M      DECLARE SCALAR,
M                   G1, G2;
M      STRUCTURE UTIL_PARM:
M         1 V VECTOR,
M         1 S1 SCALAR,
M         1 C INTEGER,
M         1 S2 SCALAR,
M         1 E BOOLEAN,
M      DECLARE ARG UTIL_PARM-STRUCTURE,
M   UTIL-
M   FUNCTION(X) VECTOR,
M      DECLARE X UTIL_PAPM-STRUCTURE,

C         .
C         .
C         .

E              -
M      RETURN X.V,
M   CLOSE UTIL;
M      DO FOR TEMPORARY I = 1 TO 10;
E                 +
M         READ(5) ARG,
E                          +               +
M         WRITE(6) 'UTIL OF', ARG, '=', UTIL(ARG),
M      END;
M   CLOSE OUTER;
```

The statement "READ(5) ARG," is functionally equivalent to

READ(5) ARG V, ARG S1, ARG C, ARG S2, ARG E,

In other words, the components of the structure are read in the "natural sequence", which is the order in which they appear in the structure template The components are output in this same sequence when ARG appears in a WRITE statement

Similarly, given

```
DECLARE UTIL_PARM-STRUCTURE, A, B,
```

the statement

```
A = B
```

is equivalent to the sequence

```
A V  = B V,
A S1 = B S1,
A C  = B C,
A S2 = B S2,
A E  = B E,
```

Structure components, such as LOCAL V and A S1, follow *exactly* the same rules as simple variables of the corresponding data type No restrictions whatsoever are imposed on a structure component that would not also apply to a simple variable of that type Thus, the vector component, V, of a UTIL_PARM-STRUCTURE, A, can be subscripted

```
A VS1 = A VS2,
```

used in a comparison,

```
DO UNTIL A VS(2 AT 1) = 0,,
```

passed to a built-in function,

```
A S1 = ABVAL(A V),
```

read, written, or filed, or used in any other construct in which a vector is allowed Furthermore, there is no additional runtime overhead (either time or space) involved in referencing a component of a structure rather than a simple variable

Structure initialization is essentially the same as array initialization The initial list consists of a value or set of values for each component of the structure, separated by commas The CONSTANT attribute is also acceptable There is no way to write a structure *literal*, but the CONSTANT attribute may be used to obtain the same effect For example, a convenient way of setting all of the components of a structure to zero is

```
DECLARE UTIL_PARMS-STRUCTURE,
    A,
    B,
    ZERO CONSTANT(0,0,0,0,0,0,Off),
A = ZERO,
```

In addition to assignment statements, parameter passage, and I/O statements, comparison of entire structures is permitted As was the case with arrays, the only comparisons that can be made between structure operands are equal (=) and not equal (¬=)

In this section we have discussed all of the ways that entire structures can be used in executable statements and made the assertion that components of a structure may be used in any way that simple variables of the same types can be used We have discussed declaration and initialization of structures using the template names as a data type All of the examples have used the same template (UTIL_PARM), but the rules for creating templates have been omitted and the naming of structure components has only been implied by example In section 9 2 we will clear up these points and show additional examples of the use of structures This chapter concludes with the presentation of two additional attributes "Copiness", which is analogous to arrayness of other data types, and unqualified structures, which are easier to reference but more limited in capability

9 2 THE STRUCTURE TEMPLATE

A structure template describes the layout of a structure in terms of the order and data types of its components A structure template is created via the STRUCTURE statement This statement begins with the word STRUCTURE followed by the name of the template being defined and a colon The remainder of the statement is a list of component descriptions separated by commas Each component is described by a level number, a name, and a data type The statement below creates a template named SUPER_VECTOR which has three components

```
STRUCTURE SUPER_VECTOR
    1 V VECTOR,
    1 STATUS BOOLEAN,
    1 TIMETAG SCALAR,
```

The phrase "1 V VECTOR" defines a component named V of type VECTOR at level one These level numbers require some explanation, but first we will state the rules about names and data types

1) The name of a structure component may be any valid HAL/S identifier
2) The names of structure components need not be unique, provided they can be unambiguously referenced (i e. structures A and B may both have a component named X since they can be distinguished by referencing A X and B X)
3) The components of a structure may be of *any* data type They may be of single or double precision and they may be arrayed

Since SUPER_VECTOR-STRUCTURE is a data type by the definition in this chapter, le three above makes the following template legal

```
STRUCTURE STATEVEC
    1 POSITION SUPER_VECTOR-STRUCTURE,
    1 VELOCITY SUPER_VECTOR-STRUCTURE,
    1 ACCEL SUPER_VECTOR-STRUCTURE,
```

Given the following structure declaration

DECLARE STATE STATEVEC-STRUCTURE,

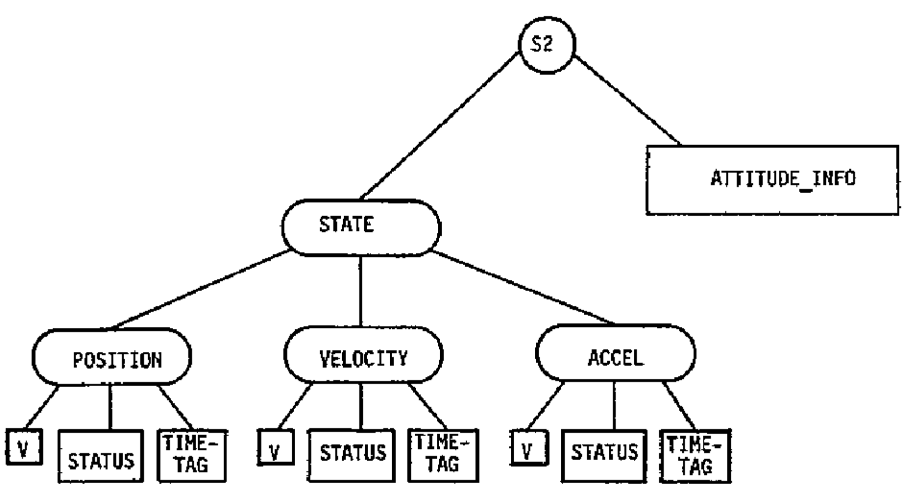
how are the low-level components referenced? The answer follows from the information already presented Since the V component of POSITION is named "POSITION V", the POSITION V component of STATE may be referenced as "STATE POSITION V" This process may be carried to any level Given,

```
STRUCTURE S2
    1 STATE STATEVEC-STRUCTURE,
    1 ATTITUDE_INFO ARRAY(3) VECTOR DOUBLE,
DECLARE STATE2 S2-STRUCTURE,
```

the components are named

```
STATE2 STATE POSITION V,
STATE2 STATE POSITION STATUS,



STATE2 STATE ACCEL TIMETAG,
STATE2 ATTITUDE INFO$(1 ),
```

and so forth The components listed above are called *structure terminals* A structure terminal is any component of a structure which itself is not a structure. Structure components which are also structures are termed *structure nodes*, this terminology stems from viewing a structure as an inverted tree, as shown below



In this diagram, rounded boxes are used to represent nodes, or forks in the tree The square boxes represent structure terminals which are the leaves of the tree

In Section 9 1 it was stated that a component of a structure may be used in any context in which a simple variable of the same type can be used This statement applies to both structure terminals and to entire nodes of a structure Since the nodes STATE2 STATE POSITION and STATE2 STATE ACCEL are of type SUPER_VECTOR-STRUCTURE, they may be read, written, filed, assigned to each other, compared, or passed as parameters to a procedure or function which expects a SUPER_VECTOR-STRUCTURE as an argument Thus, these components of STATE2 STATE might be manipulated as shown below

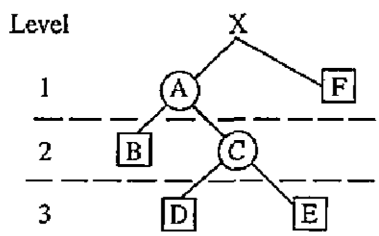
```
M  P
M  PROGRAM,
M     STRUCTURE SUPER_VECTOR
M        1 V VECTOR,
M        1 STATUS BOOLEAN,
M        1 TIMETAG SCALAR,
M     STRUCTURE STATEVEC
M        1 POSITION SUPER_VECTOR-STRUCTURE,
M        1 VELOCITY SUPER_VECTOR-STRUCTURE,
M        1 ACCEL SUPER_VECTOR-STRUCTURE,
M     DECLARE STATE STATEVEC-STRUCTURE,
M     STRUCTURE S2
M        1 STATE STATEVEC-STRUCTURE,
M        1 ATTITUDE_INFO ARRAY(3) VECTOR DOUBLE,
M     DECLARE STATE2 S2-STRUCTURE,
M     REPLACE TEST_DATA BY "1",
M     DECLARE CYCLE INTEGER INITIAL(0),
M     DECLARE DELTA_T CONSTANT(1 / 10),                          /*TIME BETWEEN SAMPLES*/
E                     +          +
M     STATE2 STATE ACCEL = READ_ACC(17),

C   ASSUME THAT 17 SELECTS THE CORRECT ACCELEROMETER

E                                          +                           +
M     CALL INTEGRATE(STATE2 STATE ACCEL) ASSIGN(STATE2 STATE VELOCITY),
E                                          +                          +
M     CALL INTEGRATE(STATE2 STATE VELOCITY) ASSIGN(STATE2 STATE POSITION),
M     CYCLE = CYCLE + 1,
E                                                +
M     FILE(TEST_DATA, CYCLE) = STATE2 STATE,                     /*SAVE FOR POST PROCESSING*/
M  INTEGRATE
M  PROCEDURE(INPUT) ASSIGN(OUTPUT),
M     DECLARE SUPER_VECTOR-STRUCTURE,
M                 INPUT, OUTPUT,
E
M     IF INPUT STATUS = FALSE THEN
M        DO,
E
M           OUTPUT STATUS = FALSE,
M           RETURN,
M        END,
M     OUTPUT TIMETAG = INPUT TIMETAG,
E              -        -       -
M     OUTPUT V = OUTPUT V + INPUT V DELTA_T,
M  CLOSE INTEGRATE,
M  CLOSE P,
```

An alternate way of coding the S2 template used in declaring STATE2 appears in the following figure This example should make the use of level numbers clear Level numbers provide the capability of creating nodes in a template without referencing other templates No change whatsoever would be required to the previous program if this S2 template was substituted for the earlier formulation

```
M   P:
M   PROGRAM;
M      STRUCTUPE SUPER_VECTOR:
M         1 V VECTOR,
M         1 STATUS BOOLEAN,
M         1 TIMETAG SCALAR,
M      STRUCTUPE S2-
M         1 STATE,
M            2 POSITION,
M               3 V VECTOR,
M               3 STATUS BOOLEAN,
M               3 TIMETAG SCALAR,
M            2 VELOCITY,
M               3 V VECTOR,
M               3 STATUS BOOLEAN,
M               3 TIMETAG SCALAR,
M            2 ACCEL SUPER_VECTOR-STRUCTURE,
M         1 ATTITUDE_INFO ARRAY(3) VECTOR DOUBLE;
M   CLOSE P,
```

By referring back to the tree diagram of the STATE2 structure, it can be seen that the level numbers represent the distances between the top of the structure and each component Another illustration of this correspondence appears below

```
STRUCTURE X
  1 A,
     2 B INTEGER,
     2 C,
        3 D INTEGER,
        3 E INTEGER,
  1 F INTEGER,
```



In these examples, the structure templates have been indented to show the contents of each node This indenting is supplied by the compiler based on the level numbers Since the HAL/S language is written in free format, the number of blanks coded on source cards is irrelevant Hence, the previous example could also be written as

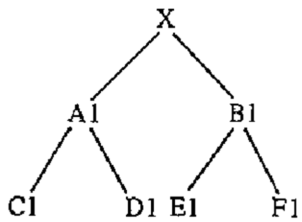
STRUCTURE X 1 A, 2 B INTEGER, 2
C, 3 D INTEGER, 3 E INTEGER, 1 F INTEGER,

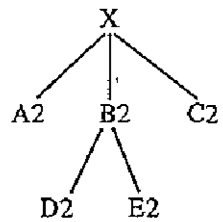
and the same output listing would result

Coding structure templates in the above form is not recommended, however Properly indented source code generally makes desk checking and subsequent modification much easier

Exercises

9 2A Write structure templates for the following trees





where

C1, E1 are 3-vectors,

D2, F1 are 3x3 matrices,

D2, E2 are arrays of length 5 of 3-vectors,

All other terminals are scalar

9 2B

a) For the following sequence of structure templates and the single declaration below, draw the tree for the declared structure TEST_DATA

```
STRUCTURE X
    1 A INTEGER,
    1 B,
        2 V1 VECTOR,
        2 V2 VECTOR,
STRUCTURE Y
    1 A,
        2 B INTEGER,
        2 V1 VECTOR,
    1 C SCALAR,
STRUCTURE DATA
    1 L,
        2 M X-STRUCTURE,
        2 N Y-STRUCTURE,
    1 I,
        2 J X-STRUCTURE,
        2 K Y-STRUCTURE,
DECLARE TEST_DATA DATA-STRUCTURE,
```

b) Write, in the natural sequence, the expressions used to reference each terminal of TEST_DATA

c) Write an alternate structure template for DATA that allows the terminals to be referenced exactly as in part (b), but does not use structures X and Y

d) Call the structure template of part (c) DATA_PRIME, and make the following declarations

```
DECLARE STRUC1 DATA-STRUCTURE,
        STRUC2 DATA_PRIME-STRUCTURE,
```

Which of these assignments are legal

1) STRUC1 L M A = STRUC2 L M A,

2) STRUC1 = STRUC2,

3) STRUC1 I K = STRUC2 I K,

4) STRUC1.L M = STRUC2 I J,

5) STRUC2 L = STRUC2 I,

9 2C Rewrite the following segment of HAL/S code, using structures to eliminate the DO FOR loop How must the procedure PROCESS be changed to allow this? Be sure the data can be read in the same order as before

```
DECLARE VEC_ARR ARRAY(5) VECTOR,
DECLARE TIM_ARR ARRAY(5) SCALAR,
DO FOR I = 1 TO 5,
   READ(5) VEC_ARR$(I ),TIM_ARR$I,
END,
CALL PROCESS(VEC_ARR,TIM_ARR),
```

9 2 1 Template Matching

Throughout this chapter, the data type of a structure has been named by referring to the template used in its declaration The statement has been made that two structures are of the same data type if their templates are identical For the purpose of matching data types, two structure templates are identical if and only if the order and data types of all of their *components* are exactly the same For structure terminals, all of the attributes including precision and arrayness must match The term "components" used above also includes structure nodes, Two nodes are of the same type if and only if their components are of the same data types and in the same order

This rule can be stated in two different ways

1) Two structure templates are identical if and only if the order, data types, and *hierarchical arrangement* of their terminals are the same
2) Two structure templates are identical if the only differences between them are the *names* of terminals and nodes

Most of the information about structures has already been presented We have seen how to declare and reference structures and their components, and how to code structure templates The use of structures to group data for parameter passage, assignment as a block, and the simplification of I/O statements has been illustrated Subsequent sections will add a few more capabilities to structure declaration and referencing by building on the basic concepts of templates, nodes, terminals, and user-defined data types presented here

9 3 MULTI-COPIED STRUCTURES

Multi-copied structures provide a capability similar to arrays of simpler data-types The uses of structure *copiness* are much the same as the uses of arrayness described in chapter six If several structures are to be processed identically, it is convenient to reference them by number within a loop An example of this usage is described below

The SUPER_VECTOR template from Section 9 1 (repeated below) might be used to contain sensed velocity data from an inertial measurement unit Since these devices are usually redundant, it is useful to define a multi-copied SUPER_VECTOR to contain the data The following figure shows how such an entity can be declared and referenced

```
M   EXAMPLE_N:
M   PROGRAM,
M      STPUCTURE SUPER_VECTOR
M         1 V VECTOR,
M         1 STATUS BOOLEAN,
M         1 TIMETAG SCALAR,
M      DECLARE VEL SUPEP_VECTOR-STRUCTURE(3),
M      DECLARE BEST INTEGER,
M      DO FOR TEMPORARY I = 1 TO 3,
E                                  +
M         CALL READ_IMU(I) ASSIGN(VEL  );
S                                  I,

M      END,
E                       +
M      CALL SELECT_BEST({VEL}) ASSIGN(BEST);
E                    +
M      CALL GUIDANCE(VEL      );
S                    BEST;

E                    +
M      CALL OTHER_SW(VEL      );
S                    BEST;
```

.
.
.

```
M   SELECT_BEST
M   PROCEDURE(V) ASSIGN(SELECTED);
M      DECLARE V SUPER_VECTOR-STRUCTURE(3),
M              SELECTED INTEGER,
M      DECLARE N INTEGER;
M      DECLARE MOST_RECENT SCALAR INITIAL(0) AUTOMATIC;
M      DO FOR N = 1 TO 3,
E                 .
M         IF V STATUS    = OFF THEN
S                    N;
M            REPEAT,
M         IF V.TIMETAG    > MOST_RECENT THEN
S                     N;
M            DO,
M               SELECTED = N,
M               MOST_RECENT = V TIMETAG   ,
S                                        N,
M            END;
M      END;
M      IF MOST_RECENT = 0 THEN
M         SELECTED = 1,                             /*ALL EQUALLY BAD*/
M   CLOSE SELECT_BEST,
M   GUIDANCE
M   PROCEDURE(BEST_VEL);
M      DECLARE BEST_VEL SUPER_VECTOR-STRUCTURE;
C            ...
M   CLOSE GUIDANCE;
M   OTHER_SW
M   PROCEDURE(V),
M      DECLARE V SUPER_VECTOR-STRUCTURE,
C
M   CLOSE OTHER_SW,
M   READ_IMU·
M   PROCEDUPE(UNIT_NUM) ASSIGN(STRUC),
M      DECLARE UNIT_NUM INTEGER,
M              STRUC SUPER_VECTOR-STRUCTURE;
C            ...
M   CLOSE READ_IMU;
M   CLOSE EXAMPLE_N;
```

Several points are illustrated by this example First, a multi-copied structure is created simply by appending a *copiness specifier* to the structure declaration The copiness specifier is a parenthesized integer which immediately follows the word STRUCTURE As with VECTOR or ARRAY dimensions, the number of copies may be specified by any arithmetic expression which can be computed at compile time*

The next new construct in the example appears in the statement

```
CALL READ_IMU(I) ASSIGN(VEL$(I,) ),
```

This statement is intended to obtain the Ith copy of [VEL] from an external device VEL$(I,) is a SUPER_VECTOR-STRUCTURE with no copiness, the fact that it is contained in a multi-copied structure does not by itself impose any restrictions on its use The semicolon in the subscript separates structure subscripts from the other types of subscripts for the same reason that the colon is used to set off array from component subscripts Structure subscripts may of course be combined with the other types For instance, the second component of V within the third copy of VEL can be referenced as VEL V$(3,2) Some of the many combinations are illustrated below Given,

```
STRUCTURE X
    1 M ARRAY(10) MATRIX,
    1 I ARRAY(3,2) INTEGER,
DECLARE BIG X-STRUCTURE(100),
```

the very first scalar component is

```
BIG M$(1,1 1,1)
```

and the last scalar is

```
BIG M$(100,10 3,3)
```

The first four integers are

```
BIG I$(1,1 TO 2,*),
```

which is a two-by-two integer array

```
BIG M$(1,* 1,*)
```

is an array of ten 3-vectors composed of the first rows of all the matrices in the first copy of BIG

*There is also an equivalent to ARRAY(*) which will be described later

Partitions are also allowed in structure subscripts, the statement

```
BIG$ (1 TO 50,) = BIG$(51 TO #,),
```

would set the first fifty copies of BIG to the values contained in the last fifty

The data type of BIG$(1 TO 50,) is "multi-copied X-structure" When the structure subscript is applied to a *terminal* (e g BIG I), the result is no longer a structure In this case, the copiness is converted to arrayness BIG M$(1 TO 50,) behaves like a 50 x 10 *array* of matrices Likewise, BIG I$(1 TO 50,1,1) behaves like an ARRAY(50) INTEGER even though all of the actual arrayness was subscripted away With respect to terminals (but not nodes), arrayness and copiness are interchangeable

Returning to the original example in which VEL was declared as a three-copied SUPER_VECTOR structure, we can see how the conversion to arrayness is used The following are arrayed statements which functions exactly as described in Section 6 2

```
[VEL STATUS] = ON,        /*set all three status booleans to TRUE*/
MOST_RECENT = MAX([VEL TIMETAG]),
AVG_Z_COMPONENT = SUM(VEL V$(*,3))/3,
AVG_Y_COMPONENT = SUM(VEL V$(*,2))/3,
VEL V = VECTOR(1,1,1),
```

In many ways, multi-copied structures are like arrays of other data types We have already seen that subscripting is essentially the same except for the use of a semicolon instead of a colon, and that terminals of multi-copied structures can participate in arrayed statements One copy of a multi-copied structure may be used in any context where a simple variable of the same structure type can be used This rule is also the same as stated previously for arrays and their elements This section has also shown that the uses of copiness are roughly the same as the uses of arrayness Identical operations on similar data, saving a set of structures in a list, and maintaining tables

Another way in which multi-copied structures resemble arrays is in initialization A multi-copied structure can be initialized by listing the initial values for each copy separated by commas, as shown

```
STRUCTURE MONTH
    1 NAMEOF CHARACTER(5),
    1 DAYS INTEGER,
    1 COLD BOOLEAN,
DECLARE YEAR MONTH-STRUCTURE(12) INITIAL('JAN', 31, TRUE, 'FEB',
28, TRUE, 'MARCH', 31, TRUE, 'APRIL', 30, FALSE, *),
```

Here, the asterisk (*) is used to indicate that only part of the structure is to be initialized The initial values of copies five through twelve are indeterminate The use of a multi-copied structure for this type of diverse table instead of a set of parallel arrays (shown below) is largely a matter of style The referencing of entries is about equally convenient, but the initial list groups all of the information about each entry in the case of a structure whereas the information for arrays must be grouped by type as shown in the alternative below

DECLARE NAMEOF ARRAY(12) CHARACTER(5) INITIAL('JAN', 'FEB', 'MARCH', 'APRIL', *),
DECLARE DAYS ARRAY(12) INTEGER INITIAL(31, 28, 31, 30, *),
DECLARE COLD ARRAY(12) BOOLEAN CONSTANT(TRUE, TRUE, TRUE, PAUSE, *),

Finally, procedures may be written to accept a structure with a variable number of copies The syntax is the same as for arrays, as shown below, which is a re-work of the example before

```
M | EXAMPLE_N
M | PROGRAM,
M |    STRUCTURE SUPER_VECTOR
M |       1 V VECTOR,
M |       1 STATUS BOOLEAN,
M |       1 TIMETAG SCALAR,
M |    DECLARE VEL SUPER_VECTOR-STRUCTURE(3);
M |    DECLARE BEST INTEGER,
M |    DO FOR TEMPOPARY I = 1 TO 3;
E |                                   +
M |       CALL READ_IMU(I) ASSIGN(VEL  ),
S |                                   I,
  |
M |    END;
E |                         +
M |    CALL SELECT_BEST({VEL}) ASSIGN(BEST),
E |                     +
M |    CALL GUIDANCE(VEL      ),
S |                     BEST,
  |
E |                     +
M |    CALL OTHER_SW(VEL      );
S |                     BEST;
  |
M | SELECT_BEST:
M | PROCEDURE(V) ASSIGN(SELECTED),
M |    DECLAPE V SUPER_VECTOP-STRUCTURE(*);
M |    DECLARE SELECTED INTEGER,
E |                                     +
M |    DO FOR TEMPOPARY N = 1 TO SIZE({V}),
E |                 .
M |       IF V.STATUS   = OFF THEN
S |                  N;
  |
M |          REPEAT,
  |
C |
  |
M |    END;
M | CLOSE SELECT_BEST;
M | GUIDANCE
M | PROCEDUPE(BEST_VEL);
M |    DECLARE BEST_VEL SUPER_VECTOR-STRUCTURE,
```

```
C  |          ...
   |
M  | CLOSE GUIDANCE,
M  | OTHER_SW:
M  | PROCEDURE(V),
M  |    DECLARE V SUPER_VECTOR-STRUCTURE;
   |
C  |          ...
   |
M  | CLOSE OTHER_SW,
M  | READ_IMU
M  | PROCEDURE(UNIT_NUM) ASSIGN(STRUC);
M  |    DECLARE UNIT_NUM INTEGER,
M  |            STRUC SUPER_VECTOR-STRUCTURE,
   |
C  |          ..
   |
M  | CLOSE READ_IMU;
M  | CLOSE EXAMPLE_N;
```

Note, however, that there are a few ways in which multi-copied structures are different from arrays

1) Only one dimension of structure copiness is allowed
2) Arrays may be used as structure components, but multi-copied structures may not
3) There are no operators or built-in functions for processing structures

Exercises

9 3A Rewrite the solution of problem 9 2C using multi-copied structures

9 3B Consider the following structure template and declaration

```
STRUCTURE A1
    1 B ARRAY(5) INTEGER,
    1 C SCALAR,
    1 D VECTOR(6),
DECLARE A A1-STRUCTURE(100),
```

Write a HAL/S expression to reference the following data items, and indicate their type and arrayness/copiness

a) The 25th copy of A
b) The 3rd component of B from all copies of A
c) C from the 10th through 20th copies of A
d) D from 75th to 85th copies of A
e) The 1st element of D from the first copy of A

9 3C The followmg informatıon about a company's 100 employees ıs avaılable

a)	SS number	(ınteger)
b)	salary (scalar)	(scalar)
c)	job code	(ınteger)
d)	name	(character)

Wnte a HAL/S program to read ın all the data from channel 5 and compute the average salary Create a structure to hold all of the avaılable ınformatıon

9 4 DENSE, RIGID, AND "UNQUALIFIED"

DENSE and RIGID are mınor attnbutes that can be applıed to structures and theır nodes to gıve the user more control over the layout of structure data ın storage The term "unqualıfied" refers to a type of structure ın whıch ıt ıs not necessary to qualıfy each reference to a termınal by the name of the contaınıng structure These features may not be frequently used, but they do provıde addıtıonal capabılıtıes requıred by some applıcatıons

9.4 1 The DENSE Attnbute

The DENSE attnbute ınstructs the compıler to pack portıons of a structure ınto as lıttle storage as possıble, generally at the expense of effıcıent references to the data The DENSE attnbute ıs specıfied on a structure template or a node of a template as shown ın the figure below

```
M | P
M | PROGRAM;
M |    STRUCTURE FLAGS DENSE·
M |        1 B1 BOOLEAN,
M |        1 B2 BOOLEAN,
M |        1 NODE INTEGER,
M |        1 B3 BOOLEAN,
M |        1 C CHARACTER(5);
M |    DECLARE STATUS FLAGS-STRUCTURE INITIAL(OFF, OFF, 0, OFF, '');
M | CLOSE P,
```

The effect of the DENSE attribute is implementation dependent This is because the mapping of HAL/S data types into bits, bytes, words, double words, etc , varies according to the storage formats of individual target machines Most computers have operand alignment requirements, for instance requiring that floating point numbers be stored at an address which is a multiple of two or four The HAL/S programmer is normally isolated from these considerations Since variables are only referenced by their symbolic names, the compiler is free to re-arrange declared data to meet the requirements of the machine

Unless the DENSE attribute is specified, all data is ALIGNED (i e placed on appropriate storage boundaries) DENSE data is packed whenever there is a reasonably efficient means of bypassing the computer's operand alignment requirements Thus, the only general statement that can be made about DENSE structures is that they *tend* to require less storage but more time to access than ALIGNED structures

It turns out, though, that most compilers will pack booleans and bit strings in DENSE structures In the example above, B1, B2 *and* B3 would occupy the same amount of storage that would be allocated to a single ALIGNED boolean Note that B3 is placed in the same byte, word or other addressable unit as B1 and B2 even though an integer is between them in the template Whether or not DENSE is specified, the compiler is free to rearrange the order of structure components to minimize the number of alignment gaps or to optimize the addressing of certain components In fact, *all* declared data is subject to the rearrangement unless the RIGID attribute is specified (see Section 9 4 2)

Components of a DENSE structure are referenced in the usual way, some additional restrictions on their use apply, but where they are allowed, they behave exactly like components of a corresponding ALIGNED structure Thus, statements like

```
STATUS B1 = ON,
STATUS B2, STATUS B3 = FALSE,
   IF STATUS B1 AND STATUS B2 THEN STATUS MODE = 9,
```

work as described previously The additional restrictions* imposed on *terminals* of dense structures are

1) Bit or boolean terminals of a dense structure may not be passed as ASSIGN parameters to procedures
2) Bit or boolean terminals of dense structures may not be used on the left hand side of a FILE statement
3) Bit or boolean terminals of dense structures may not be used in NAME expressions See Chapter 13

*These restrictions avoid the need to pass both an address *and* starting bit number to Library or USER-supplied routines

These are the only restrictions imposed on the DENSE attribute, note that they apply only to bit and boolean types and do not apply to *entire* structures with the DENSE attribute even if these structures contain bit or boolean terminals Thus,

```
[STATUS] = FILE(1,1),
```

is legal, but

```
STATUS B1 = FILE(1,1),
```

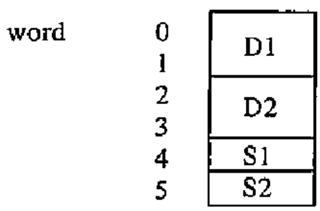
is *not* legal

9.4 2 The RIGID Attribute

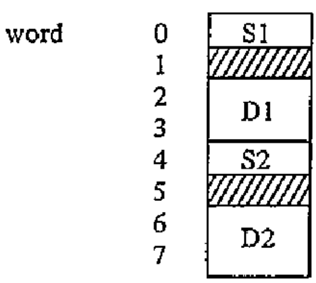
Consider the following structure

```
STRUCTURE INTEGER_LIST
    1 S1 INTEGER,
    1 D1 INTEGER DOUBLE,
    1 S2 INTEGER,
    1 D2 INTEGER DOUBLE,
DECLARE IOTA INTEGER_LIST-STRUCTURE,
```

On a computer which requires that double precision integers be stored on even addresses, the compiler would probably rearrange the data as follows



If the data was kept in the natural sequence, the following would be needed



The shaded areas indicate alignment gaps which are effectively wasted storage These diagrams show how allowing the compiler to re-arrange data can result in a substantial savings of memory

Occasionally, however, it is necessary to prevent this rearrangement, generally to interface with external devices or NONHAL routines The RIGID attribute is supplied for this purpose The second diagram shows the storage assignments that would be made if the word RIGID appeared immediately before the colon of the STRUCTURE statement. An appropriate use of the RIGID attribute appears below

```
STRUCTURE IMU_DATA RIGID
    1 DELTA_V ARRAY(3) INTEGER DOUBLE,
    1 ATTITUDE ARRAY(3) INTEGER,
    1 TIME BIT(32),
    1 STAT DENSE,
       2 F1 BOOLEAN,
       2 F2 BOOLEAN,
       2 F3 BOOLEAN,
       2 UNUSED BIT(13),
    1 OP_MODE INTEGER,
DECLARE IMU_DATA IMU-DATA-STRUCTURE,
CALL ASM_TO_ROUTINE ASSIGN(IMU_DATA),
```

In addition to the syntax for declaring a RIGID structure, this example shows the DENSE attribute applied to the STAT node IMU_DATA STAT is both RIGID and DENSE The RIGID attribute on the structure is inherited by all of its nodes If any additional nodes were defined below STAT, they would also be RIGID *and* DENSE, unless the ALIGNED keyword was specified The RIGID attribute is always inherited (cannot be turned off) since there is no "non rigid" keyword

The RIGID attribute allows any data layout to be mapped into HAL/S data types It does not impose *any* restrictions on the use of a structure or its components However, two structures cannot be of the same data type unless neither is RIGID or both are.

In the example above, note that "IMU_DATA" is the name of the template *and* the name of the declared structure This fact makes IMU_DATA an *unqualified* structure

9 4 3 Unqualified Structures

When a structure template is to be used in *only one* declaration, it is convenient to give the structure the same name as the template This permits the name of the structure to be omitted when referencing its nodes and terminals Again referring to the structure above, the statement,

```
DO CASE IMU_DATA OP_MODE,
```

is legal, but the more convenient form,

```
DO CASE OP_MODE,
```

is also permitted

Unqualified structures differ from qualified structures (all previous examples) *only* in the form of references to their components It has already been stated that there is no execution-time penalty involved in using a structure terminal instead of a simple variable If an unqualified structure is used, no distinction has to be made in the source code either Thus, there is no disadvantage to using a rigid unqualified structure to force a collection of variables to be allocated in a particular sequence, except for possible alignment gaps

Sometimes it is useful to convert a set of declared variables to the components of an unqualified structure, since all of the variables (now structure terminals) can be transferred to or from a random-access device in a single FILE statement Variables are also sometimes collected in an unqualified structure for documentation purposes since this allows them to be discussed as a group under an "official" name which appears in the source code

Now that structures and their uses have been fully described, only two data types remain Bit strings, which are the general case of booleans, are discussed in Chapter 13, and event variables, which may be thought of as "real-time booleans", in Chapter 12 The material covered thus far in the text should allow most applications to be coded in HAL/S The handling of errors and exceptional conditions will be discussed in the next chapter Then we will proceed to put a collection of programs together and execute them as an integrated system in Chapters 11 and 12 Chapter 12 describes how the user may control execution rates and inter-process communication and synchronization The book concludes by discussing several constructs that are provided for writing "system programs" such as I/O device drivers and memory management routines

Exercises

9 4A Given

```
STRUCTURE A RIGID
   1 B,
      2 C INTEGER,
      2 D VECTOR,
   1 E,
      2 F,
         3 G MATRIX(4,5),
         3 H ARRAY(2,3) INTEGER DOUBLE,
      2 I INTEGER,

STRUCTURE AF
   1 G MATRIX(4,5),
   1 H ARRAY(2,3) INTEGER DOUBLE,

STRUCTURE RAF RIGID
   1 G MATRIX(4,5),
   1 H ARRAY(2,3) INTEGER DOUBLE,

DECLARE X A-STRUCTURE,
   Y AF_STRUCTURE,
   Z RAF_STRUCTURE,
DECLARE INTARR ARRAY(2,3) INTEGER DOUBLE,
```

Are the following assignments legal?

a) X E F = Y,

b) Z = X E F,

c) X E F H = Y H+Z H,

d) Y G = Z G,

e) X B C = Y H$(1,1),

9 4B Consider the following structure template and declaration

```
STRUCTURE A
   1 B SCALAR,
   1 C INTEGER,
   1 D VECTOR(6),
DECLARE A A-STRUCTURE(20),
```

What do the following HAL/S subscripted variables reference, and what are their types and arrayness/copiness

a) A$(20,)

b) A$(2 AT 10,)

c) C$(1,)

d) D$(4 TO 6,)

e) D$(*, 4 TO 6)

End of Chapter Problems

9A What are some of the capabilities that HAL/S structures give the program that would otherwise be unavailable?

9B Write a HAL/S program that will read simulated data from 3 redundant sensors on channel 5 and compute the middle value of the 3 redundant pieces of data

Read an acceleration, velocity, attitude (3-vectors), and a scalar time tag after each from each measurement unit First read from unit 1, then 2 and 3 in that order Compute the middle value of the three measured values for each quantity (using the MIDVAL built-in function or any equivalent code), and store these values with their associated time-tags in a structure with the following template

```
1 BEST_ACCEL,
   2 ACCEL VECTOR,
   2 ACCEL_TIM SCALAR,
1 BEST_VEL,
   2 VEL VECTOR,
   2 VEL_TIM SCALAR,
1 BEST_ATTITUDE,
   2 PITCH VECTOR,
   2 PITCH_TIM SCALAR,
```

10 0 ERROR RECOVERY

Each implementation of the HAL/S language defines a set of runtime errors These errors, or *exceptions*, include

1) invalid arguments to built-in functions, such as SQRT (−1),
2) I/O errors, such as reading past the end of a file,
3) hardware detected errors, such as attempting to divide by zero,
4) and other conditions which may arise while executing certain HAL/S statements, e g inverting a singular matrix and using invalid character subscripts

By default, when one of these errors occurs, a *standard fixup* is performed, on ground-based systems, an error message may be generated as well In some cases, the standard fixup is to print diagnostic information and terminate the program, but usually some innocuous value is substituted for the offending expression and execution continues For instance, if SQRT(X) is invoked with a negative X, the standard fixup is to return SQRT(ABS(X)) The standard fixups for all errors defined in a compiler are listed in the corresponding Users Guide

The standard fixup may not be appropriate for all applications Hence, HAL/S provides a mechanism that allows user-supplied HAL/S statements to gain control when an error occurs In this figure, an ON ERROR statement is used to handle an end of file error


```
M | TEST_X
M | PROGRAM,
M |    REPLACE IO BY "10",
M |    DECLARE SCALAR,
M |               INPUT, OUTPUT, EXPECTED;
M |    DECLARE INTEGER INITIAL(0),
M |               RIGHT, WRONG,
M |    ON ERROR
S |            IO 5

M |       GO TO DONE;
M |    DO WHILE TRUE,
M |       READ(5) INPUT, EXPECTED,
M |       CALL X(INPUT) ASSIGN(OUTPUT),
M |       IF OUTPUT = EXPECTED THEN
M |          RIGHT = RIGHT + 1,
M |       ELSE
M |          WRONG = WRONG + 1;
M |    END;
M | DONE:
M |    WRITE(6) 'RESULTS OF TESTING X',
M |    WRITE(6) RIGHT, ' SAMPLES COPRECT, ', WRONG, ' SAMPLES INCORRECT';
M | X:
M | PROCEDURE(I) ASSIGN(O);
M |    DECLARE SCALAR,
M |               I, O,
C |       .
C |
C |       .
M | CLOSE X;
M | CLOSE TEST_X;
```

Only one new construct is used in this example

ON ERRORS (EOF 5) GO TO DONE,

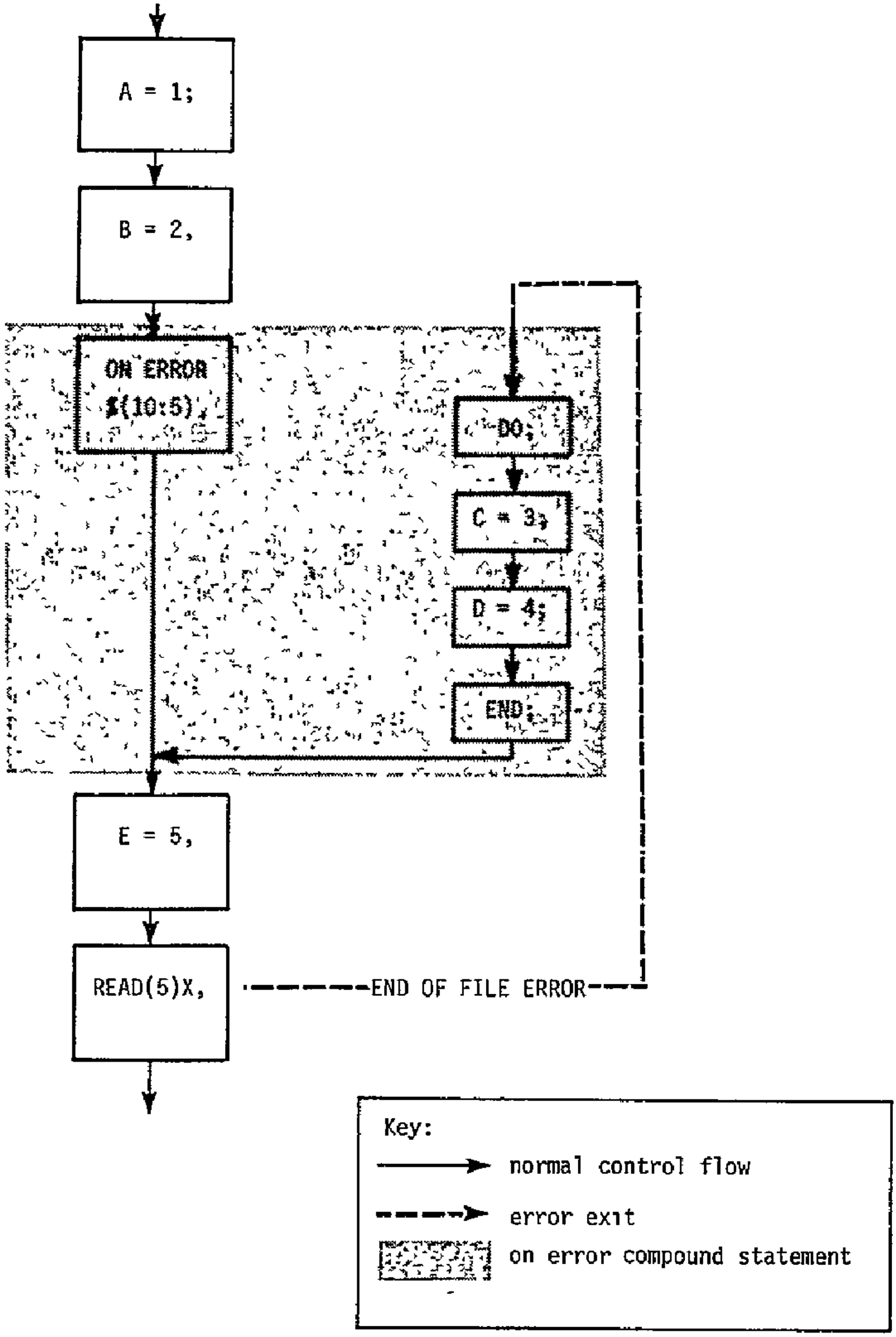
This is an *executable* statement which establishes "GO TO DONE," as a handler for the end of file error When the ON ERROR statement is executed, the default error handling (i e standard fixup) for the end of file error is replaced, by the GO TO statement supplied The function of the ON ERROR statement is to selectively replace the standard error handlers under program control

10 1 THE ON ERROR STATEMENT

Like the IF statement, ON ERROR is a compound statement (i e a statement which contains another statement) It specifies an *action* to be performed when an error occurs This action may be an executable statement, but GO TO is the most commonly used in this context In fact, the action portion of an ON ERROR statement should be the most frequent use of GO TO HAL/S The example above, however, can be re-written without a GO TO, as in this figure

```
M   TEST_X:
M   PROGRAM,
M      REPLACE IO BY "10",
M      DECLARE INTEGER INITIAL(0),
M                  RIGHT, WRONG;
C
C
C      .
M      ON ERROR
S              IO 5
M         DO;
M            WRITE(6) 'TEST RESULTS FOLLOW';
M            WRITE(6) RIGHT, WRONG;
M            RETURN,
M         END,
M      DO WHILE TRUE,
C        .
C        .
C        .
M      END;
M   CLOSE,
```

In this example, a DO . END group serves as the action of the ON ERROR statement Note that in making this change it was necessary to add a RETURN statement after the WRITE statements This is because *after the action of an ON ERROR statement has been executed, control falls through to the following statement* If the RETURN were not coded, the DO WHILE TRUE loop would be re-executed after the WRITE statements and the error probably would recur, resulting in an infinite loop The next figure illustrates the flow of control around an ON ERROR DO END group



A = 1;
B = 2,
ON ERROR
$(10:5)
DO;
C = 3;
D = 4;
END;
E = 5,
READ(5)X,
END OF FILE ERROR
Key:
normal control flow
error exit
on error compound statement

After an error occurs and a user-specified action is taken, there is no way to resume execution at the point that the error was detected, for efficiency reasons, the state of the program immediately after the error is not saved, and hence cannot be restored

The end of file example illustrates one difference between the HAL/S ON ERROR system and the system of alternate returns or "END=   " used in many languages The ON ERROR statement was coded outside of the DO WHILE loop, thus the overhead associated with defining an end of file handler is paid only once, rather than at each READ statement

The subscript in the ON ERROR statement consists of two numbers separated by a colon The left number is an *error group*, the right number is an *error code* within that group Denoting errors by both a group and a code allows entire groups of errors to be handled identically (see later) The group and code assignments of a particular error are generally the same among various implementations of the language, though this is not guaranteed by the HAL/S Language Specification The User's Manual which corresponds to the compiler in use should be consulted before using ON ERROR statements

The compiler used in producing the listings for this book follows the same convention as several HAL/S compilers All I/O errors are assigned to group 10, and codes 0-9 in this group represent end of file errors on channels 0-9 Thus, ON ERROR$ (10 5) sets up a handler for end of file on channel five Use of the macro

```
REPLACE EOF BY "10",
```

is used to improve readability

If a program reads data from several devices, an end of file handler can be created for each, e g

```
ON ERROR$ (EOF 4) GO TO NO_MORE_CARDS,
ON ERROR$ (EOF 5) GO TO END_OF_TAPE,
etc
```

It may be more convenient to write one handler for any I/O error, this can be easily done by omitting the error code as in

```
ON ERROR$ (EOF ) GO TO DONE,
```

or

```
ON ERROR$ (EOF) GO TO DONE,
```

These forms both specify "any error code with the given group" Finally, the statement

```
ON ERROR GO TO DONE,
```

sets up "GO TO DONE," as the handler for all errors (including end of file)

```
M   P:
M   PROGRAM,
M      DECLARE M MATRIX;
C         ,
C         .
C         .
M      ON ERROR
S              4:27

M         DO,
E            *
M            M = 0;
M            GO TO L1;
M         END,
E      *   *-1
M      M = M   ;
M   L1:
E                *
M      WRITE(6) M;
C         .
C
C
M   CLOSE P;
```

ON ERROR is the standard means of handling exceptions which arise from operations on invalid data For example, a runtime error will result from attempting to invert a singular matrix The standard fixup for this error is to print a message, return the identity matrix, and continue execution In the program segment above an ON ERROR statement is used to substitute a zero for the identity matrix

It should be noted that use of this form of the ON ERROR statement *replaces* the standard fixup Hence it prevents the generation of an error message Many implementations impose a limit on the number of errors that may occur before the program is terminated by the system When a user-supplied handler is invoked, the error is not counted toward this limit

Once an ON ERROR statement is executed, the specified error handler remains in effect until it is deactivated One means of deactivating an error handler is shown below

```
M   P:
M   PROGRAM;
M      DECLARE M MATRIX,
M               I INTEGER;
M      DO FOR I = 1 TO 10,
C         .
C
C         .
```

```
M       ON ERRUR
               4:27

S
M            DO;
E               *
M               M = 0,
M               GO TO L1,
M            END;
E        *   *-1
M       M = H  ;
E                  *
M    L1' WRITE(6) M,
C        .
C        .
C
M     END,
M     ON ERROR      SYSTEM,
S             4'27

C        .
C
C
M   CLOSE P;
```

Here, the keyword SYSTEM is used in place of an executable statement as the action of the ON ERROR This statement has the effect of restoring the standard fixup for ERROR$ (4 27) To see why this statement is needed, suppose that additional inverse operations were coded later in the program, and this statement was omitted If one of these operations caused an error, control would be transferred to the user handler *in the middle of a loop* This would be disasterous, since the compiler assumes that a loop can only be entered by execution of the DO statement at its head Thus, *if an error handler is coded in a loop, it should always be deactivated at exit from the loop* In general, it is good practice to deactivate error handlers as soon as they are no longer needed

The statement

ON ERROR$ (X Y) SYSTEM,

restores the default (system) recovery action for error X Y (group X, code Y) In addition to SYSTEM and an executable statement, IGNORE can be used as the action of an ON ERROR statement, as in

ON ERROR$ (4 27) IGNORE,

This statement informs the error recovery system that inverting a singular matrix is not to be considered an error, i e that the standard fixup (returning identity) is appropriate and that execution should continue without an error message or other notification Depending on the compiler in use, IGNORE may not be permitted for certain errors

When an ON ERROR statement is executed, an error recovery action is established for an error or group of errors Three recovery actions are possible

1) an executable statement to receive control, (in lieu of the standard fixup and an error message),
2) SYSTEM, which is the initial state and includes both the standard fixup and an error message, and
3) IGNORE, which requests the standard fixup without an error message

Any number of recovery actions may be in effect at one time In a sense, the actions are cumulative If the code below were executed, four recovery actions would be in effect

```
M   P
M   PROGRAM,
M      DECLARE SCALAP,
M                 A, B, C,
M      DECLARE INTEGER,
M                 X, Y, Z,
M      ON ERFOR
M         DO,
M            WRITE(6) A, B, C, X, Y, Z,
M            RETURN,
M         END,
M      ON EPROP
S              10 5

M         RETUPN;
M      ON EPPOR      IGNORE,
S              10

M      ON EPFOR      SYSTEM,
S              4 2

C
C         .
C

M   LAST_CARD
M   CLOSE P,
```

The net effect of these statements is Any end of file error, except on channel five, will be ignored, and any other error, except 4 2, will cause the WRITE and RETURN statements to be executed If error 4 2 occurs, the system action will be taken, and when 10 5 occurs, P will close This shows that the handler for error $ (10 5) takes precedence over the handler for error $ (10) The general rule that applies is *When the error specifications in several active ON ERROR statements in a single block apply to a particular error, the most specific takes precedence* Thus, as each of the last three ON ERROR statements in P is executed, the number of errors handled by the first and most general one is reduced

Note that the rule above applies only to ON ERROR statements in a single block (program, procedure, function, etc) The effect of ON ERROR statements in nested blocks will be discussed in the next section Note also that an ON ERROR statement has no effect until it is executed

Exercises

10 1A Where does the flow of control go after the action of an ON ERROR statement has been executed?

10 1B Why is it good programming practice to deactivate any error handler that is coded inside a loop when that loop is exited?

10 1C What are the three possible recovery actions in the event of a runtime error?

10 1D Write the precedence relations for the 3 general forms of subscripting for the ON ERROR statement when they occur in the same code block

10 2 DEACTIVATING ERROR HANDLERS

An error handler can be deactivated in three ways

1) by overriding it with a new handler,
2) by exiting from the containing block,
3) by using the OFF ERROR statement

All of these methods are affected by the HAL/S block structure *A procedure or function cannot make any permanent change to the error environment of its caller* This statement is a consequence of several rules which will be described with reference to the figure below

```
M   A-
M   PROGRAM,
M      ON ERROR     IGNORE;
S               1 2

M      CALL B,
M      CALL C;
M   B
M   PROCEDURE,

M      ON EPROR
S               1-2

M         GO TO X,
M      CALL C;
M   X  WRITE(6) 'GOT AN ERROR',
M   CLOSE B,
M   C:
M   PROCEDUPE,
M   CLOSE C;
M   CLOSE A;
```

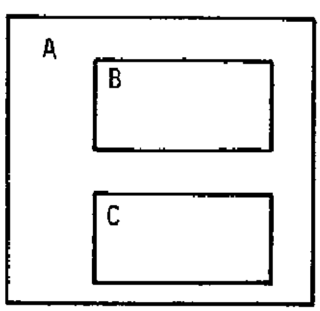
None of the statements shown can produce an error, however we will discuss what would happen if ERRORS (1 2) were caused by an additional statement inserted at various points

If the error occurs in block A proper (i e outside of B and C), the IGNORE action will be taken, even after B is called and returns This is because *any error handler defined in a block is cancelled when that block RETURNs or executes its CLOSE statement* When B returns, the error environment reverts to that in effect when B was called In this case, the IGNORE action is re-instated

When the ON ERROR statement in B is executed, the IGNORE action is temporarily overridden by the GO TO action This action then remains in effect until B returns If the error occurs in B, but before the GO TO action is set up, the IGNORE action is taken Merely invoking a block does not change the error environment When B calls C, the GO TO action is still in force, if ERRORS (1 2) occurs in block C, control will be passed to the label X *in block B* In effect, C returns to X instead of to the point of invocation When this happens, the error environment is restored to that which prevailed before C was called, just as if C had returned normally

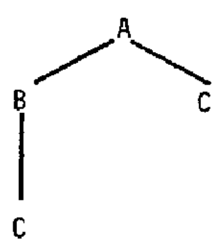
In the example, block C is also called directly from block A In this case, of course, the ON ERROR statement in B has no effect If the error occurs in C when it has been called from A, the IGNORE action is taken Thus, we see that the range over which an ON ERROR statement is active is not determined by the static block structure, but by the actual sequence of CALLs and RETURNs

The left-hand diagram below shows the static block structure of a program A, which is suitable for describing the scoping rules for variables



Block Structure

"outer" variable can be referenced



Call Tree

"upper" blocks affect error environment

The right-hand diagram illustrates the range of ON ERROR statements within A, B and C C occurs twice in the diagram, at the ends of different limbs Since all intervening blocks between a given block and the top of the tree may be scanned for handlers when an error occurs, a block's error environment depends not only on local ON ERROR statements, but those in the calling block, and in the caller's caller, and so forth Block C may be affected by B's error environment even though it cannot access B's variables

Now that the basic concepts have been illustrated, the rules for deactivation of error handlers can be stated precisely

1) When a code block exits (by RETURN, CLOSE, or due to an error) the error environment is restored to that in effect when the block was entered
2) An error handler may be replaced by execution of an *identically subscripted* ON ERROR statement *in the same block*
3) An error handler may be temporarily overriden by creating another handler in a "lesser" block (i e. lower in the call tree) which applies to the same error(s)
4) An error handler may be completely erased by execution of an *identically subscripted* OFF ERROR statement *in the same block*

These are the only ways that an error handler may be deactivated Note that there is no limit to how far up the call tree the system will search for a handler when an error occurs As stated previously, when a particular block contains several handlers that could apply to the same error, the most specific is selected Other active blocks are searched only if no handler at all for this error is found in the current block

The OFF ERROR statement may be used to cancel the error handler created by a corresponding ON ERROR statement There are only four possible forms

```
OFF ERROR,
OFF ERROR$ (n1 n2),
OFF ERROR$ (n1 ),
OFF ERROR$n1,
```

and of these, the last two are equivalent The effect is simply to cancel an identically subscripted ON ERROR statement in the same block If no such ON ERROR statement has been executed, the OFF ERROR statement has no effect

The primary use of the OFF ERROR statement is to re-instate an error handler in the calling block which had been overridden by a local ON ERROR statement An example of this usage appears in the following figure

```
M   A:
M   PROGRAM;
C
C
C       .
M     ON ERROR
M         GO TO X;
M     CALL B;
M   B·
M   PROCEDURE,
C            .
C            .
C
M     ON ERROR IGNORE;
C
C            .
C            .
M     OFF EPROR,
C
C            .
C            .
M   CLOSE B,
M
C        .
C        .
C
M   CLOSE A;
```

It should be noted that the handler cancelled by an OFF ERROR statement must not only be in the same block, but it must describe exactly the same error(s) For instance, the sequence

ON ERROR$1 IGNORE,
ON ERROR$2 IGNORE,
OFF ERROR,

would leave two handlers active, since the OFF statement is more general than the ON statements To cancel them both would require two statements

OFF ERROR$(1),
OFF ERROR$2,

Likewise, the sequence

ON ERROR$(1) IGNORE,
OFF ERROR$(1 2),

does *not* exclude ERROR$(1 2) from the handler Unless there is an *identically* (plus or minus a trailing colon) subscripted ON ERROR statement in the same block, OFF ERROR will do nothing

Exercises

10 2A In what ways is it possible for an error handler to be deactivated?

10 2B In the following examples of sequences of ON ERROR and OFF ERROR statements, which handlers are left active after the sequence?

a)
```
ON ERROR$1 IGNORE,
ON ERROR$(1 2) IGNORE,
ON ERROR$(2 1) IGNORE,
OFF ERROR,
OFF ERROR$(1 3)
```

b)
```
ON ERROR$1 IGNORE,
ON ERROR$(1 1) IGNORE,
ON ERROR$(2.) IGNORE,
OFF ERROR$(1·),
OFF ERROR$(2 1),
```

10 3 OTHER ERROR CONTROL CONSTRUCTS

In addition to ON and OFF ERROR, which activate and deactivate error handlers, HAL/S provides the SEND ERROR statement, which annunciates an error condition, and a pair of built-in functions which allow information to be obtained from the recovery system

The SEND ERROR statement has two uses To simulate the occurrence of system-defined errors for testing and other purposes, and to allow the user to define additional error types It has only one form

```
SEND ERROR$(n1 n2),
```

where n1 and n2 are integers computable at compile-time and in the valid range of error groups and codes specified by the appropriate HAL/S User's Manual The effect of the SEND ERROR statement is merely to trigger whatever handler has been set up for the specified error

When a SEND ERROR is executed, the error environment is searched for an applicable ON ERROR handler If the action is an executable statement, control is passed to it and execution continues without an error message If the IGNORE option was specified, execution continues at the statement following the SEND ERROR, also without a message If the action is SYSTEM, or no error handler is found, then an error message is generated, and either the run is terminated, or execution continues at the statement following the SEND ERROR The User's Manual states whether execution will continue* after an error of each system-defined type Generally, if the group and code are not system-defined (i e not listed in the User's Manual) the SYSTEM action allows execution to continue Thus, it is possible to write a "standard fixup" for a user-defined error, as shown below.

```
M   LOG10:
M   FUNCTION(X) SCALAR;
M      DECLARE X SCALAR;
M      IF X > 0 THEN
M         RETURN LOG(X) / LOG(10);
M      ELSE
M         DO;
M            SEND ERROR    ;
S                       9:1
M            RETURN LOG(ABS(X)) / LOG(10);
M         END;
M   CLOSE LOG10;
```

Now, when LOG10 is invoked with a negative argument, error 9 1 will result This error may be handled by the calling routine in the usual way, e g

```
DECLARE N SCALAR INITIAL(-1),
ON ERROR$(9 1) DO,
   N = 100,
END,
WRITE(6) LOG10(N),
```

This code will write $\log_{10}(100)$ If the next two statements were

```
OFF ERROR$(9 1),
WRITE(6) LOG10(-99),
```

there would be no active handler for error 9 1, so an error message would be printed and execution would continue at the second RETURN statement in LOG10. This RETURN statement serves as a "standard fixup" for a negative argument to LOG10, in this case, $\log_{10}(99)$ would be returned by the function

*Some implementations may allow an error to occur (or be simulated) a given number of times before terminating Others may always continue or always terminate

SEND ERROR is a relatively expensive statement, when an error is sent, many machine instructions may be needed to search the error environment for an appropriate handler Hence, it should be used only to indicate *exceptional* conditions, or "errors", not conditions which are expected to occur frequently The SEND ERROR statement is most appropriately used in utility routines (procedures and functions that are invoked from many places) to indicate invalid arguments, and in instances where a "catastrophic" condition is detected by very low level code but can only be handled in an outer block, perhaps by some sort of controlled restart

In addition to the ON, OFF, and SEND error statements, HAL/S provides two built-in functions, ERRGRP and ERRNUM, which provide information about previous errors These functions do not require any arguments, they return integers which represent the group and code, respectively, of the last error that occurred in the process* that invokes them If no errors have occurred, they return zero

These functions are used primarily when a number of errors are handled by a single ON ERROR statement, as illustrated below

```
    ON ERROR DO,
       WRITE(6) 'RUN STOPPED DUE TO ERROR'
||ERRGRP|| '.' || ERRNUM,
       RETURN,
    END,
```

One additional form of ON ERROR statement is provided This form allows *event variables* to be manipulated when an error occurs The form of this type of error recovery action is described in the language specification Event variables are discussed in Chapter Twelve

Exercises

10 3A What are the two uses for the HAL/S SEND ERROR construct?

10 3B Say we enter a program block, P, which calls some procedure A, which in turn calls procedure B In the code block for B, there is an ON ERROR$(1) IGNORE statement and no other error handlers Now say error (1 3) occurs during the execution of the program Does the program need to search code blocks A and P for the error handlers for error (1 3) or will it automatically ignore the error because the statement ON ERROR$1 was found in that block?

*The term process is defined in Chapter 11 Here it may be taken to mean a program *and* all of its internal blocks

End of Chapter Problems

10A Consider a HAL/S program with the following lexical structure

```
P  PROGRAM

   ON ERROR$1 IGNORE,
   ON ERROR$2 IGNORE,

      (I)

   A  PROCEDURE,

         (II)

      ON ERROR$(1 2) IGNORE,
      OFF ERROR$(1 ),

            (III)

      B  PROCEDURE,
         OFF ERROR$(1 2),
         ON ERRORS(2 1) IGNORE,
         ON ERROR$(2 1) IGNORE,
         ON ERRORS(3 ) IGNORE,

               (IV)

         OFF ERRORS(2 ),

                  (V)

      CLOSE B,

            (VI)
   CLOSE A,

         (VII)

CLOSE P,
```

Say the execution of the program procedes as follows

A → (I) executed

P calls A

A → (II) executed

A calls B

B → (IV) executed

B → (V) executed

B returns to A

A → (III) executed

A → (VI) executed

A returns to P

P → (VII) executed

execution stops

What happens if the following errors occur at these times (i e , error message or no error message)?

a) ERROR$(1 1) at (I)

b) ERRORS(3 1) at (I)

c) ERROR$(2 1) at (II)

d) ERRORS(2 2) at (IV)

e) ERROR$(1 2) at (IV)

f) ERROR$(2 1) at (V)

g) ERRORS(2 1) at (III)

h) ERROR$(1 1) at (III)

i) ERROR$(1 2) at (VI)

j) ERROR$(1 3) at (VI)

k) ERROR$(3 3) at (VII)

l) ERRORS(1 1) at (VII)

11 0 STRUCTURING LARGE APPLICATIONS

In this chapter the discussion of the HAL/S facilities for building a *program complex* consisting of many separately compiled pieces is presented First, we will describe the unit of compilation, which has been a PROGRAM in previous chapters but is not restricted to this type Then we will discuss means of putting these units together in a way that is suitable for a particular application Finally, we will introduce the concept of multi-programming and discuss some of the methods of safely sharing code and data between programs that execute "simultaneously" This discussion will lead into the real-time control statements to be presented in chapter twelve

11.1 THE UNIT OF COMPILATION

A unit of compilation is a sequence of HAL/S statements which comprise a complete, valid input to the compiler It must be either a program, a procedure, a function or a compool (common data pool) Programs have already been discussed at length, though no means of invoking them has yet been presented This is because programs receive control directly from an operating system, not from other HAL/S code

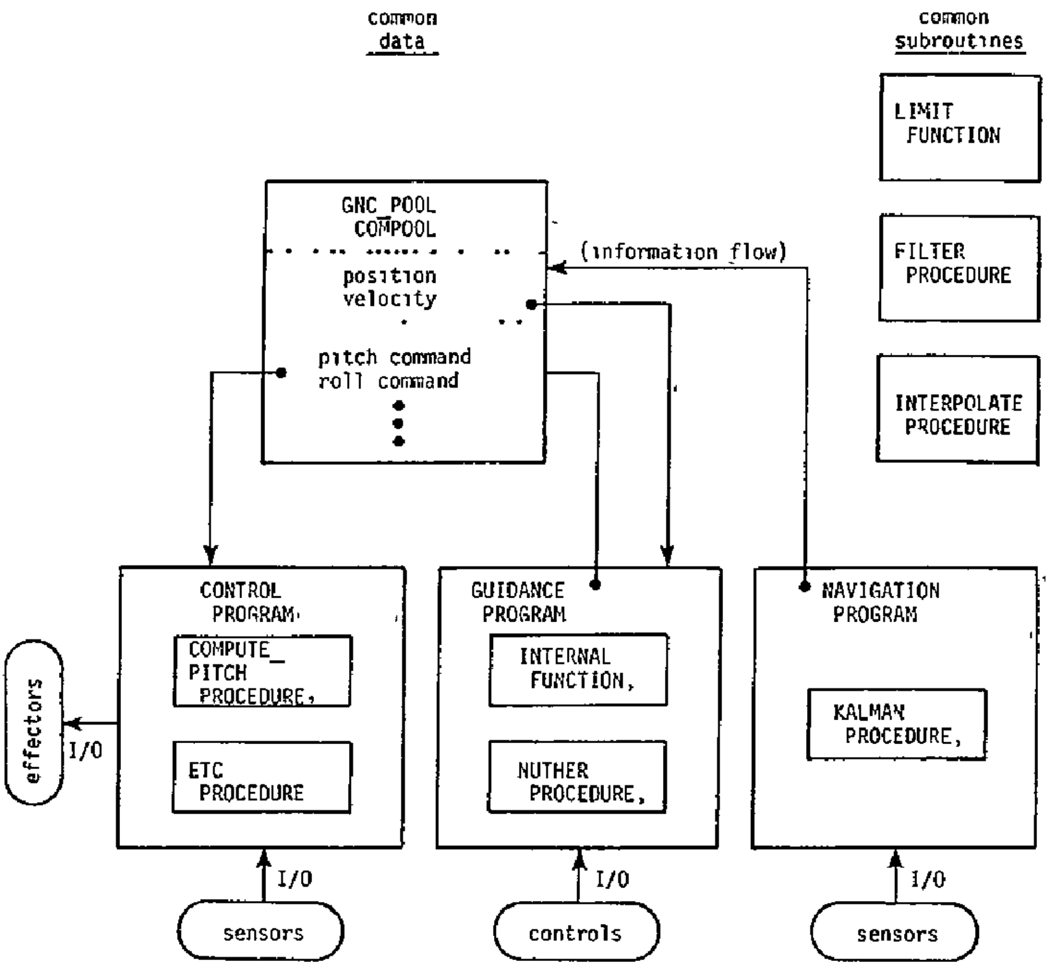
Procedures and functions can be compiled independently so they can be shared among programs, a compool is a block of data that can be shared among separately compiled units Thus, programs are the primary compilation units while the others provide global code and data

There are two major reasons for dividing a software system into separately compilable units Obviously, when several programmers collaborate on a system, it is convenient if they can compile their own work independently A more important reason stems from the way program units receive control The capabilities of the operating system in use may determine the appropriate structure for an application

Under an operating system which supports the full HAL/S real-time syntax (described in chapter 12), many programs may be "simultaneously" active and compete for the use of the computer hardware based on a user specified priority Provision is made for programs to be run cyclicly, to wait for given occurrences and to receive control when interrupts occur The operating system provides these capabilities for the invocation of PROGRAMs and TASKs (collectively called *processes*) Thus, a software system may be divided into programs to implement a desired dynamic (real-time) structure

Unlike procedures, functions and tasks, programs and compools may *not* be nested in any other blocks

The following figure shows how these blocks might be used in a simple flight application


common data
common subroutines
LIMIT FUNCTION
FILTER PROCEDURE
INTERPOLATE PROCEDURE
GNC_POOL COMPOOL
position
velocity
pitch command
roll command
(information flow)
CONTROL PROGRAM
COMPUTE_PITCH PROCEDURE,
ETC PROCEDURE
GUIDANCE PROGRAM
INTERNAL FUNCTION,
NUTHER PROCEDURE,
NAVIGATION PROGRAM
KALMAN PROCEDURE,
effectors
I/O
sensors
I/O
controls
I/O
sensors
I/O

This diagram shows the software divided into three programs, each with internal procedures and functions, and a compool and three independently compiled subroutines All together, there are seven compilable units which must be compiled in an appropriate sequence and linked together In the remainder of this section we will discuss the rules for writing the components of a program complex

The LIMIT function and the procedures, FILTER and INTERPOLATE, are compiled separately so that they can be called from any of the programs Such procedures and functions are called comsubs (from "*com*mon *sub*routines") *A comsub may be coded exactly as if it were contained in some program* For instance, the LIMIT function might be exactly as it appeared in chapter seven

```
M   LIMIT
M   FUNCTION(VALUE, BOUND) SCALAR,
M      DECLARE SCALAR,
M                 VALUE, BOUND,
M      IF VALUE > BOUND THEN
M         RETURN BOUND,
M      IF VALUE < -BOUND THEN
M         RETURN -BOUND;
M      RETURN VALUE,
M   CLOSE LIMIT,
```

Aside from the fact that a comsub is not contained in any block, and thus cannot reference outer variables via home scoping rules, all of the statements about procedures and functions made in previous chapters also apply to comsubs

Some of the consequences of this general statement may not be immediately obvious For one, comsubs may have additional procedures and functions nested within them Scoping rules apply to blocks contained in a comsub just as they would to blocks contained in a program In fact, the only significant difference between an independently compiled procedure without parameters and a program is the manner of invocation Programs are never CALLed and procedures normally do not receive control directly from the operating system

It is also worth noting that the error recovery system does not distinguish between comsubs and internal procedures and functions If an error occurs in a comsub and no local ON ERROR statement applies, the error environment of the calling block is searched, whether that block is a program, another comsub, or an internal procedure of some program or comsub

Comsubs are also referenced in the same way as corresponding internal blocks There is no way to tell by inspection of a CALL statement or function invocation whether the referenced block is internal to the compilation unit or external (a comsub) Comsubs may have any number of arguments of any type, exactly as described in chapter seven. The various rules about matching data types, restrictions on ASSIGN parameters, automatic conversions, etc, still apply In order to enforce these rules the compiler needs to know the declared types of comsub's formal parameters This information is communicated via the *block template*

Under *most* implementations of the HAL/S compiler, a block template is automatically generated whenever a program, comsub, or compool is compiled The block template contains all the information needed to reference that block from another compilation unit In the case of a comsub, this information consists of its name, the sequence and types of its formal parameters, and the type of its return value, if any A comsub is made accessible to a compilation by including its template For instance, a program which uses the LIMIT comsub is shown below

```
D INCLUDE TEMPLATE LIMIT
  P PROGRAM,
     DECLARE X SCALAR INITIAL(12),
     X = LIMIT(X,10),
  CLOSE P,
```

INCLUDE is a compiler directive, as denoted by the character D in column one It instructs the compiler to merge the template for block LIMIT into the compilation at the point of the INCLUDE directive Any number of templates may be so included, the NAVIGATION program might be compiled as

```
column 1
  ↓
  D INCLUDE TEMPLATE GNC_POOL
  D INCLUDE TEMPLATE LIMIT
  D INCLUDE TEMPLATE FILTER
     NAVIGATION PROGRAM,



        .
     CLOSE NAVIGATION,
```

Note that these templates are included *prior to the program statement* This syntax emphasizes the fact that the blocks GNC_POOL, LIMIT, and FILTER are external to NAVIGATION The printed output from the compiler contains a listing of each template that was included The template for LIMIT appears below

```
LIMIT EXTERNAL FUNCTION(VALUE,BOUND) SCALAR,
   DECLARE SCALAR, VALUE, BOUND,
CLOSE LIMIT,
```

The template for a comsub consists of the header line with the word EXTERNAL inserted, the declarations of any formal and assign parameters, and the CLOSE statement These are the only portions of a procedure or function block that are relevant outside that block*

*Scoping rules make other data items irrelevant, and no way of branching into the middle of a block is provided

The format of a block template is unimportant when a compiler with automatic template generation and the include directive is used. These features are present in all current compilers, but they are not included in the HAL/S Language Specification and thus are not guaranteed to be present in all implementations. The format of a template *is* specified, however. Hence, if the template cannot be INCLUDED, it may be hand-coded as part of the source prior to the program statement.

A program may invoke a comsub if it includes the template for that comsub prior to the program statement. This mechanism provides for executable code to be shared among separate compilation units.

Programs generally need to share data as well. The only way to pass information from one program to another is via a *compool*. A compool is a named block of DECLARE, REPLACE, and STRUCTURE statements; the variables in a compool are accessible to any compilation unit which INCLUDEs the compool's template.

The diagram at the beginning of this section shows how a compool is used to interface the Guidance, Navigation, and Control programs. This compool could be coded as shown below.

```
M | GNC_POOL:
M | COMPOOL;
  |
C |  FOLLOWING DECLARES ARE NAV TO GUIDANCE INTERFACES
  |
M |    DECLARE POSITION VECTOR;
M |    DECLARE VELOCITY VECTOR;
  |
C |  FOLLOWING DECLARES ARE GUIDANCE TO CONTROL COMMANDS
  |
M |    DECLARE PITCH_COMMAND SCALAR;
M |    DECLARE POLL_COMMAND SCALAR INITIAL(0);
M | CLOSE GNC_POOL;
```

As this indicates, a compool is delimited by a block header and a CLOSE statement much like the other block types. Unlike other HAL/S blocks, however, *a compool consists only of a DECLARE group*; no executable statements or nested blocks are allowed. It may contain DECLARE and REPLACE statements and structure templates. Generally, any DECLARE statement which may appear in a program may appear in a compool. There are only two exceptions, both resulting from the lack of executable code in a compool. No AUTOMATIC data is allowed in a compool, and no label (e.g. function and NONHAL procedure) declarations are allowed in a compool. It should be noted from the example that *static* initialization is allowed, and takes the same form as in other blocks.

Compiling a compool serves two purposes To reserve a block of storage containing any specified initial values, and to generate the compool template A compool template contains all of the information present in the compool source In fact, if automatic template generation is not available, the template may be constructed from the source merely by inserting "EXTERNAL" before "COMPOOL" in the block header Normally, however, only an INCLUDE directive is needed to make compool variables accessible to another compilation unit

When a program includes a compool template, the variables in that compool may be referenced, assigned, and used in any way appropriate to their data types Placing a variable in a compool rather than at the program level does not, by itself impose any restrictions on the way that variables may be used by the program This includes references to the variable from nested blocks We will discuss the application of scoping rules to compool variables and comsubs in the next section

Exercises

11 1A What are the major reasons for building a program complex with comsubs and compools, as opposed to a single large program?

11 1B Say an error occurs in some comsub, and no ON ERROR statement that applies to the error is found in the comsub What determines the error handler in this case?

11 1C a) Since a compool contains no executable statements, why must it be compiled at all?

b) What is the purpose of a compool template?

11 2 BUILDING A PROGRAM COMPLEX

From the viewpoint of scoping rules, the templates included in a compilation comprise an outermost block in which the main compilation unit (i e. the program, comsub, or compool being compiled) is nested

Chapter seven described the HAL/S scoping rules in terms of block diagrams like the one following From these rules it follows that

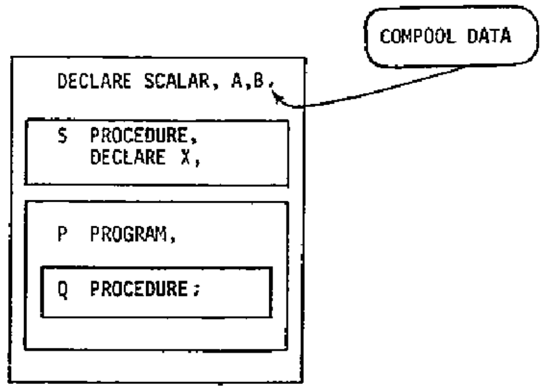
1) The comsub S can be called from anywhere within blocks P and Q
2) The variables A and B can be referenced from anywhere in blocks P and Q
3) The variable X can be referenced only from block S

This example illustrates the position of template with regard to the main compilation unit

Compilation

```
C  EXTERNAL COMPOOL,
   DECLARE SCALAR,A,B,
CLOSE C,
S  EXTERNAL PROCEDURE(X) ,
   DECLARE X SCALAR,
CLOSE S,
P  PROGRAM,
   Q  PROCEDURE,
   CLOSE Q,
CLOSE P,
```

Block Structure for Scoping Rules

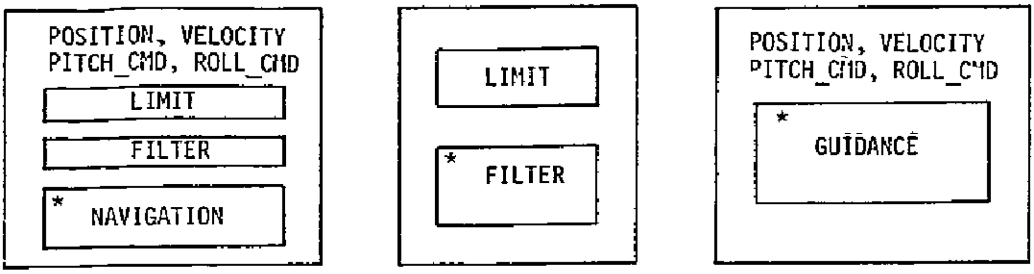


From the diagram, one might conclude that A and B can be referenced from block S This is true *if and only if* the template C is included when S is compiled Thus, the "outermost block" is not universal, its contents may appear different to each compilation unit, depending on which templates are included This mechanism support "private" compools and comsubs, as we shall see

Returning to the example of communicating Guidance, Navigation, and Control programs, suppose that the templates included by each of the seven compilation units are as indicated below

Compilation Unit	Type	Templates Included
NAVIGATION	PROGRAM	GNC_POOL, LIMIT, FILTER
GUIDANCE	PROGRAM	GNC_POOL
CONTROL	PROGRAM	GNC_POOL, FILTER, INTERPOLATE
GNC_POOL	COMPOOL	NONE
LIMIT	FUNCTION	NONE
FILTER	PROCEDURE	LIMIT
INTERPOLATE	PROCEDURE	GNC_POOL

With this structure, the contents of the "outermost block" vary considerably from compilation to compilation, as shown



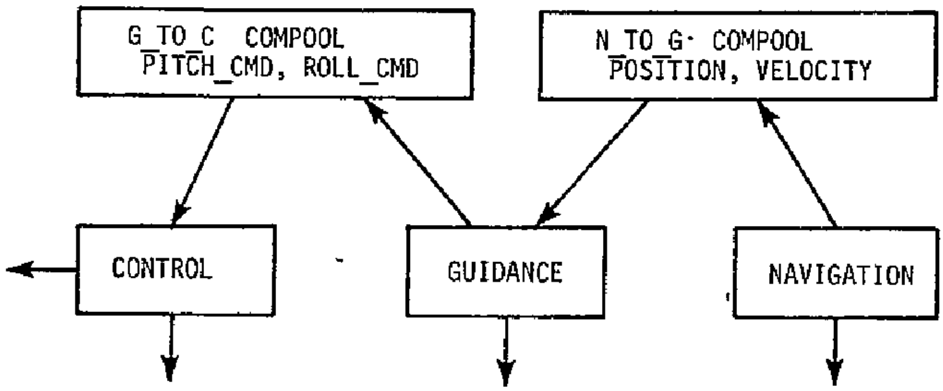
*indicates the module being compiled

As the previous table implies, any type of compilation unit may include the template of any other compilation unit. Thus, comsubs may access compool variables or call other comsubs, compools may include the templates of other compools, for instance to utilize global REPLACE statements defining array sizes. Program blocks also have templates which may be included by any type of compilation unit. We will see the utility of program templates in later sections.

From this discussion, it can be seen that access to comsubs and compool variables is controlled by the inclusion of templates. In building a particular program complex it may be desirable to set up managerial rules concerning which modules may access which data and subroutines. Comsub templates are included one at a time, but when a compool template is included, *all* of the variables in that compool become accessible. If it is desirable to partition compool data, either of two approaches may be taken. The ACCESS system may be used or multiple compools may be created.

ACCESS is a HAL/S keyword. Under some versions of the compiler, an externally maintained data base of access-rights information can augment the normal scoping rules to further restrict (not expand) the visibility of comsubs and compool data. This system is implementation dependent, somewhat complicated, and will not be discussed further in this book. However, further details are contained in the Language Specification.

The simplest method of restricting access to compool variables is via multiple compools. For instance, the following structure might be a better arrangement of the compool data for the example program complex.



Here, the interfaces between Guidance and Control are in one compool, and the interfaces between Navigation and Guidance are in another The Navigation and Control programs would include only one compool each, in this way multiple compools tend to limit the possible influences of one compilation unit on another. In this case, no data is shared between Navigation and Control

The Guidance program would have to include the templates for both compools The order in which these templates are included is irrelevant All compools are included at the same level Thus, the previous diagram of scoping rules while compiling Guidance still holds Since there is always only one scope level outside of the main unit of compilation, *the names of variables in one compool must not duplicate the names of variables in another compool if both are included by a single compilation unit*

There are, of course, other considerations in structuring an application as a set of compilation units For instance, it may be convenient to use only one compool so that all global data can be found in a single listing or so it will be contiguous in memory for telemetry purposes The addressing modes of some computers may create an efficiency trade-off between the number of compools and their average sizes Finally, in the next section, we will see that compools can be eliminated through the use of TASK blocks, this decision involves additional trade-offs

Suppose, however, that the original configuration of three programs, one compool, and three comsubs, has been chosen In this and the previous section we have described how the various compilation units are coded The remaining problem is to compile them in the appropriate order Since templates are automatically generated* when each block is compiled, the "lowest level" compilation units must be compiled first Given the table of templates included per compilation presented earlier, an appropriate sequence for this program complex is

*If automatic template generation is not available, the order of compilation is irrelevant

GNC_POOL, LIMIT, FILTER, INTERPOLATE, GUIDANCE,
NAVIGATION, CONTROL

Generally, the necessary order of compilation can be determined by inspection Starting with compools, then proceeding to "utility" comsubs, other comsubs, application programs, and finally "control" programs is usually adequate However, the following algorithm will always produce an acceptable sequence if one exists

1) Produce a list of templates included by each compilation (like the one given here)
2) Compile each module which requires no templates (except for those templates already generated)
3) Remove the modules that have been compiled from each list
4) If not done, repeat step two

It is possible that a point will be reached where every module requires at least one template If so, then there is no suitable sequence This can happen for three reasons, all of which are rare

1) Recursion If A calls B and B then Calls A, no sequence is appropriate Solution Change the structure, recursion will not work anyway
2) A pair of programs schedule or wait for each other Solution Hand-code one template or re-structure
3) Trouble with initialized NAME variables Solution Break the loop of circular references (see Chapter 13)

These difficulties almost never occur in well designed program complexes

The constructs we have discussed in this chapter are intended for putting a collection of *HAL/S* modules together A means of invoking NONHAL procedures and functions was presented in chapter seven If part of a program complex (e g special-purpose hardware interfaces) must be written in assembly language, a few additional constructs are helpful These are

1) RIGID compools, which are similar in concept to RIGID structures,
2) EQUATE EXTERNAL statements, which can make HAL/S variables accessible from assembly language, and,
3) the ability to write comsubs in assembly language A set of macros for this purpose is generally supplied with the compiler system

More detail on these features may be found in the Language Specification and the appropriate HAL/S User's Manual

Another option in designing a program complex is the use of TASK blocks instead of programs The software we have been discussing could be written as the single compilation unit shown in the figure on the next page

```
M  P:
M  PROGRAM;
M     DECLARE VECTOR,
M                 POSITION, VELOCITY;
M     DECLARE SCALAR,
M                 PITCH_CMD, ROLL_CMD;
M  LIMIT:
M  FUNCTION SCALAR;

C        ...

M  CLOSE LIMIT;
M  FILTER:
M  PROCEDURE;

C        ...

M  CLOSE FILTER;
M  INTERPOLATE:
M  PROCEDURE;

C

M  CLOSE INTERPOLATE;
M  GUIDANCE:
M  TASK;

C  -CONTENTS OF GUIDANCE PROGRAM UNMODIFIED

M  CLOSE GUIDANCE;
M  NAVIGATION:
M  TASK;

C        ...

M  CLOSE NAVIGATION;
M  CONTROL:
M  TASK;

C

M  CLOSE CONTROL;
M  CLOSE P;
```

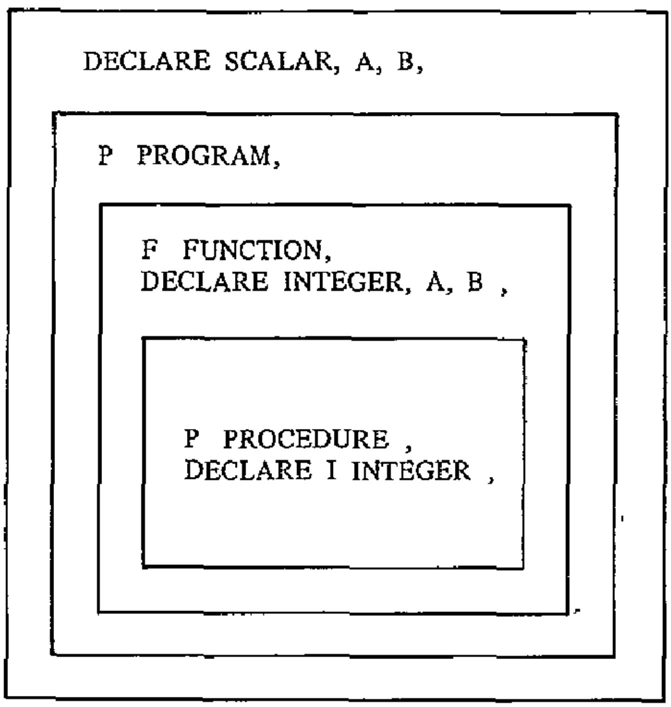
Like programs, tasks are code blocks that receive control directly from the operating system Tasks cannot be CALLed, they are used to implement real-time requirements in the same way as programs In fact, the only distinction between programs and tasks is that tasks must always be nested in programs, and may not themselves contain further program or task blocks Thus, the only change needed to convert a program to a task is in the header statement, the declare group, executable statements, and any nested procedures and functions remain exactly the same

HAL/S allows one level of nested real-time processes Tasks within programs Scoping rules treat all blocks the same Thus, a task and all of its internal procedures and functions may access data declared at the program level

Task blocks allow any real-time structure to be implemented within a single compilation unit In chapter twelve, a set of real-time control statements will be presented These statements instruct the operating system to start executing a program or task at some rate and priority, to stop cycling a process, and so forth The use of tasks as well as programs to implement-a-real-time structure tends to minimize the amount of compool data, and allows related processes to be consolidated in a single compilation unit One disadvantage of using task blocks is that they can only be SCHEDULEd, CANCELLed, etc , from within the *containing* program If a system consists of several programs, each containing tasks, then the "control" code which activates and de-activates the various processes must be distributed among the several programs

Exercises

11 2A Consider the following block structure of a program complex



From which blocks can the scalars A and B be referenced?

11 2B In the figure on page 11-2, it is shown that the compool GNC_POOL is not included in the compilation of the unit FILTER Why not?

11 2C Why is it desirable that the names of variables in a compool be unique with respect to the names of variables in other compools?

11 2D The text states that a reasonable order for compiling the various units for the example on page 11-00 is

GNC_POOL, LIMIT, FILTER, INTERPOLATE, GUIDANCE,
NAVIGATION, CONTROL

For each of the following possible orders of compilation, state whether they will necessitate the hand coding of one or more templates, and why

a) GNC_POOL, INTERPOLATE, GUIDANCE, LIMIT, NAVIGATION, FILTER, CONTROL

b) GNC_POOL, INTERPOLATE, LIMIT, CONTROL, FILTER, GUIDANCE, NAVIGATION

c) GNC_POOL, INTERPOLATE, GUIDANCE, LIMIT, FILTER, CONTROL, NAVIGATION

d) NAVIGATION, CONTROL, GUIDANCE, LIMIT, FILTER, INTERPOLATE, GNC_POOL

11 3 MULTI-PROGRAMMING CONSIDERATIONS

We have used the term "process" to refer to either a program or a task, this terminology is used throughout the HAL/S documentation The term multi-processing, however, has come to refer to the execution of software on a computer or set of-linked computers which can literally execute more than one piece of code at a time, e g programming multiple physical processors The term "multi-programming" refers to the *appearance* of this situation The use of either actual multiple processors or simulated multiple processors In the latter case, the computer's central processing unit is "time-shared" or allocated to each active process for a brief interval in succession Reallocation of the CPU may result from initiation or completion of I/O, expiration of a time limit, or other factors Since it is not possible to predict which HAL/S statement will be executing when a "process-swap" occurs*, programs must be designed so that a swap can safely occur at any point

*In fact, the timing may not be repeatable

```
M   MULTI:
M   PROGRAM;
M      DECLARE SCALAR,
M                 A, B, C,
C         .
C
C         .
M      IF A NOT = 0 THEN
M         DO;
M            B = C / A,
C         .
C         .
C         .
M         END;
M   T:
M   TASK;
M      A = 0,.
M   CLOSE T,
M   CLOSE MULTI;
```

Consider the above code Suppose that MULTI receives control and executes the IF statement, finding A not equal to zero, then, for some reason, the processor is reallocated to task T When T completes, MULTI will resume where it left off, and *divide by zero* The problem is that two processes share data (viz A) without any protection from an untimely process-swap If we could guarantee that the swap would never occur between the test for A=0 and the division by A, the problem would be solved This can be done by means of the UPDATE block and *locked* data, as shown below

```
M   BETTER:
M   PROGRAM;
M      DECLARE A SCALAR LOCK(1);
M      DECLARE SCALAR,
M                 B, C,
C
C
C
M   UPDATE,
M      IF A NOT = 0 THEN
M         DO,
M            B = C / A,
M         END,
M   CLOSE,
M   T
M   TASK,
M   UPDATE;
M      A = 0;
M   CLOSE,
M   CLOSE T;
M   CLOSE BETTER;
```

Three changes have been made in the BETTER program The variable A has been declared with the attribute LOCK(1), and both uses of A have been enclosed in UPDATE blocks The parenthesized "1" indicates the assignment of A to lock group one The use of other lock groups is discussed later in this section

Data which is used by more than one process should normally be locked Locked data can only be referenced from within an update block, the system ensures that only one update block which uses a given lock group is active at any instant of time Thus, this capability is as good as preventing process swaps over a sequence of statements A swap may occur, but the new process will not be permitted to execute an update block that pertains to the same lock group *An update block allows a process to obtain exclusive access to one or more locked variables* When an update block finishes, the locked variables become available to other processes, which also must access them via update blocks

An update block is executed when the sequential flow of control reaches it In this regard it behaves like a simple DO   END group. However, from the viewpoint of scoping rules, an update block is equivalent to any of the other block types, it may even have its own DECLARE group. An update block behaves like a procedure with respect to error recovery, except that the "calling" block is defined to be the immediately containing block An update block may be nested in a block of any other type (except compool), and may contain further procedure or function blocks There are some restrictions on the executable statements that may be used in an update block The following are prohibited

1) I/O statements,
2) Calls to procedures or invocation of functions, except for those nested in the update block, and
3) Real-time statements except for SET, RESET, and SIGNAL (see chapter twelve)

These statements are not allowed in update blocks, primarily because they potentially take a long time to execute It is desirable to minimize the time spent in an update block because while an update block is executing, other processes may be stalled even if those processes are more critical (of a higher priority)

It is almost always necessary to LOCK data which is used by more than one process The compiler does not enforce this rule, and there are cases (e g read only data) in which the protection offered by locked data is not required These cases are the exception rather than the rule For instance, the GNC_POOL compool from the earlier example should be coded as

```
GNC_POOL  COMPOOL,
    DECLARE POSITION VECTOR LOCK(1),
    DECLARE VELOCITY VECTOR LOCK(1),
    DECLARE PITCH_CMD SCALAR LOCK(2),
    DECLARE ROLL_CMD SCALAR LOCK(2),
CLOSE GNC_POOL,
```

Here,, two lock groups (1 and 2) are used Group 1 is used for the Navigation to Guidance interface, and group 2 is used for the Guidance to Control interface The selection of lock groups is entirely up to the user, the only constraint imposed by the HAL/S system is an implementation-dependent maximum number of lock groups It would be possible to use the same group for all locked data, and this may be convenient during initial development An appropriate assignment of lock groups, however, can lead to improved throughput This is because several update blocks can be active simultaneously provided that each uses a different lock group, or set of groups, with no overlap Hence, the overhead associated with a number of process swaps may be avoided Furthermore, the amount of jitter in cyclic processes may be reduced, since the chances of being stalled or suspended due to update block conflicts are lessened In our example, Control will never have to wait for Navigation since their update blocks reference variables from different lock groups

The Guidance program might begin as in the figure below As this code implies, it is sometimes preferable to copy a small amount of data (POSITION and VELOCITY) rather than extend the update block to include all of the computations involving these variables This minimizes the impact to other processes while still affording the protection against, for instance, processing a vector that has been only partially updated

INCLUDED TEMPLATE (first seven lines):

```
M | GNC_POOL
M | EXTERNAL COMPOOL,
M |    DECLAPE POSITION VECTOR(3);
M |    DECLAPE VELOCITY VECTOR(3),
M |    DECLARE PITCH_COMMAND SCALAR,
M |    DECLARE ROLL_COMMAND SCALAR INITIAL(0),
M | CLOSE,

D |  VERSION  1

M | GUIDANCE·
M | PROGRAM,
M |    DECLARE VECTOR,
M |                VEL2, POSN2,
M |    DECLARE X, Y, Z, OTHERS;
M | COPY_INPUTS
M | UPDATE,
E |     -        -
M |    VEL2 = VELOCITY,
E |     -        -
M |    POSN2 = POSITION,
M | CLOSE COPY_INPUTS,

C |     .
C |     .
C |

M | CLOSE GUIDANCE;
```

This example also shows a labelled update block The label is optional, and is used here only for self-documentation

There is one exception to the general rule that locked data may only be referenced from within an update block A locked variable may be passed as an assign parameter to a procedure. This does not defeat the protection, however, since the corresponding parameter declaration must also specify the LOCK attribute, thus it in turn can only be referenced from within an update block or passed to further procedures

The update block and locked data provide a means of safely sharing data among independent real-time processes, a similar mechanism for shared code is provided via EXCLUSIVE procedures and functions This type of protection is specified more simply Just the appearance of the word EXCLUSIVE on a procedure or function header makes that block accessible to only one process at a time To see how and why this feature is used, consider this function

```
M   MEAN
M   FUNCTION(A) SCALAR EXCLUSIVE;
M      DECLARE A ARRAY(*) SCALAR;
M      DECLARE TOTAL SCALAR INITIAL(0) AUTOMATIC,
M      DO FOR TEMPORARY I = 1 TO SIZE([A]),
M         TOTAL = TOTAL + A ,
S                         I

M      END,
M      RETURN TOTAL / SIZE([A]),
M   CLOSE MEAN;
```

Suppose the MEAN function was not exclusive If two processes invoked it, there could be a conflict in the use of TOTAL, even though it is only assigned from within MEAN If one process had executed part of the loop when the other invoked MEAN and AUTOMATICally re-initialized TOTAL, the first process would get an invalid result Thus, the problem with sharing procedures and functions among processes is a *shared-data conflict on the local data* declared in the shared block This problem can be avoided by making shared code blocks EXCLUSIVE No new construct is needed when an exclusive procedure or function is invoked, but the system will prevent multiple simultaneous users of the block by stalling the second process that tries to invoke it Exclusive routines are sometimes used for operational reasons having nothing to do with shared data For instance, a procedure to do inertial measurement unit (IMU) calibration might be made exclusive simply to avoid the risk of calibrating more than one at a time

Another keyword that can be specified instead of EXCLUSIVE is REENTRANT Neither one is the default If a procedure or function is not EXCLUSIVE *or* REENTRANT then it cannot safely be invoked from multiple processes, but no protection mechanism is present

A REENTRANT procedure or function may be executed "simultaneously" by several processes That is, if program A is executing a reentrant procedure, R, when it is interrupted by program B which also invokes R, when B completes and A resumes, there will be no adverse affect.

Simply coding the keyword REENTRANT is *not* sufficient to make a block safely "re-enterable" The following rules must also be obeyed

1) Any block invoked by the reentrant block must also be reentrant, and
2) Any local data must be declared to be AUTOMATIC whether it is initialized or not

We have already stated that the difficulty in sharing a code block is really a conflict in the use of local data *Inside a procedure or function with the REENTRANT attribute, the effect of the AUTOMATIC attribute is expanded* Each user of a reentrant procedure accesses a separate copy of the local variables if they are automatic Thus, any conflict is prevented Parameters and TEMPORARY data cannot and need not be automatic The MEAN function can be made reentrant simply by changing the EXCLUSIVE keyword to REENTRANT The necessary conditions for successful re-entrancy are described more fully in the HAL/S Language Specification

This chapter has defined the unit of compilation, and introduced the idea of a program complex, consisting of several real-time processes It has described how global code and data can be made accessible to these processes, and how the adverse effects of "simultaneous" access can be avoided In chapter twelve, we will describe the HAL/S statements for creating and controlling these processes and further discuss multi-programming concepts and their application to aerospace systems

Exercises

11.3A A bank runs several programs to modify savings and checking accounts in a multiprogramming environment The procedure MOVE_SAVE_TO_CHECK, used to move money from a savings account to a checking account, is shared by all the programs, and looks like this

```
MOVE_SAVE_TO_CHECK, PROCEDURE(ID, AMOUNT),
          .
          .
          .
    SAVINGS$ID = SAVINGS$ID-AMOUNT,
    CHECKING$ID = CHECKING$ID+AMOUNT,
          .
          .
          .
CLOSE,
```

SAVINGS and CHECKING are compool variables shared by all the programs

a) What potential error is present in this system?

b) How can it be fixed?

11 3B The bank in exercise 11 3A awards interest periodically and records each interest transaction for later printing on the customer's statement The shared procedure AWARD_INTEREST performs this task

```
AWARD_INTEREST  PROCEDURE(ID),
    DECLARE INTEREST INTEGER,
```

```
                    .
                    .
    INTEREST = SAVINGS$ID INTEREST_RATE,
    SAVINGS$ID = SAVINGSSID+INTEREST,
    CALL LOG_INTEREST(ID, INTEREST),


CLOSE,
```

a) What potential error is present?

b) How can it be fixed?

12 0 REAL-TIME STATEMENTS

Most aerospace applications have a set of timing constraints which comprise a major facet of the entire problem definition Meeting these constraints generally requires interactions with an operating system

Real-time operating systems for flight or process control applications can vary in many ways Nonetheless, certain capabilities, such as invoking a code block at a specified frequency, are almost always provided By examining several operating systems, it is possible to abstract a set of primitives (i e conceptual operating system functions) in which the various facilities can be expressed Then the real-time requirements of an application can be described without referencing any *particular* operating system The HAL/S statements described in this chapter are such a set of primitives, through which real-time requirements can be expressed in a machine-independent manner

HAL/S suggests the point of view that real-time constraints are an intrinsic part of the application, i e that timing is part of the algorithm rather than something to resolve "later" As a result, real-time statements are integral to the language, and allow the programmer to express the *entire* algorithm directly and in one place

Real-time statements isolate the programmer from operating system details in the same way that arithmetic expressions isolate the programmer from details of machine instructions and data formats A standard syntax for real-time operating system interactions greatly enhances the portability of application programs In particular, it allows flight programs to be simulated on ground-based computers, since the timing interactions are expressed in HAL/S, rec-compiling is sufficient to translate the *entire* algorithm

The mechanisms for communication among real-time processes were described in chapter 11, this chapter will discuss the set of HAL/S statements which control the initiation, termination and synchronization of processes These statements are all executable, each implementation includes some technique outside of the HAL/S language for specifying one or more initial processes which can then use the real-time statements to create and control additional processes

The figure on the next page shows the use of SCHEDULE statements to create new processes As the syntax implies, these statements create cyclic processes which will receive control from the operating system at the specified intervals The intervals may be specified by any arithmetic expression in the REPEAT EVERY clause, the units are implementation dependent but generally these values are expressed in seconds In any case, the units of time values throughout any particular implementation will be consistent Seconds will be assumed in the rest of this chapter Hence, the three processes scheduled by STARTUP would repeat at the rates of once, six times, and twenty times per second

```
M | STARTUP
M | PROGRAM,
M | GUIDANCE:
M | TASK,

C |     ...

M | CLOSE GUIDANCE,
M | NAVIGATION:
M | TASK;

C |        .

M | CLOSE NAVIGATION,
M | CONTROL:
M | TASK;

C |     ...

M | CLOSE CONTROL;
M |    SCHEDULE NAVIGATION PRIORITY(60), REPEAT EVERY 1.0;
M |    SCHEDULE GUIDANCE PRIORITY(70), REPEAT EVERY 1 / 6,
M |    SCHEDULE CONTROL PRIORITY(80), REPEAT EVERY 1 / 20,
M | CLOSE STARTUP,
```

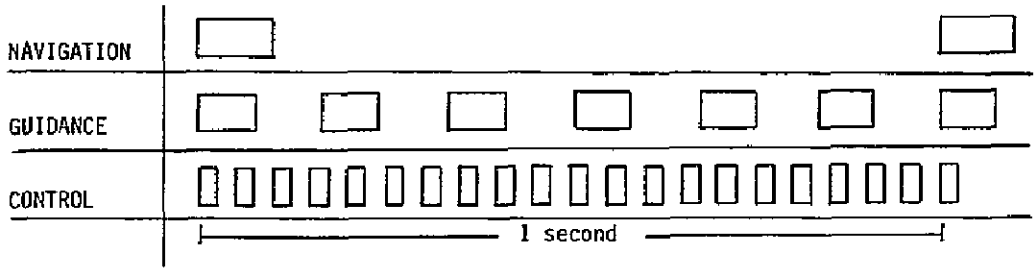
HAL/S does not impose any restrictions on the periods of cyclic processes created in this way, however, it may not be practical to provide complete generality in a flight operating system Simplifications such as rounding all time values to the nearest millisecond are to be expected in flight systems The appropriate HAL/S User's Manual and any operating system documentation should be consulted It has become common practice, however, to develop and test HAL/S software on large ground-based computers (host computers) before executing on flight (target) equipment These ground-based implementations generally do not impose any restrictions on real-time statements other than those described in the Language Specification, thus allowing a large range of operating system types to be simulated In this chapter, a complete implementation will be assumed, but the reader should not expect to find all of these capabilities in any particular flight operating system

12 1 THE SCHEDULE STATEMENT

Suppose that the average execution times of the Guidance, Navigation and Control tasks are as shown in the table below

Task	Rate	Average Time	Total Time
Guidance	6	50 ms	3 sec
Navigation	1	100 ms	1 sec
Control	20	25 ms	5 sec
		Total Time =	9 sec

Since these tasks together occupy only 9/10 of a second per second, it is clear that the specified rates are attainable. However, it would be extremely difficult to implement this structure using CALL and DO CASE statements as was done in chapter seven. The difficulty can be seen by examining a time-line of these tasks' execution.



The trouble is that no matter how the initiation of these processes is phased, a time will occur when more than one process is due to execute. If only CALL statements were used, it would be necessary to either tolerate a substantial jitter in the execution frequency of each task, or to break each task into many small procedures which would be called in a very complex sequence.

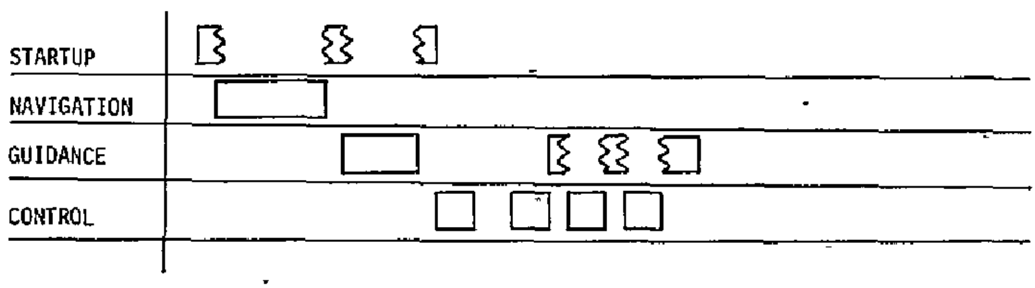
By the use of SCHEDULE statements, as shown in the example STARTUP, the timing conflicts can be automatically resolved. As we have already stated, the operating system can re-allocate the central processor at any point in the execution of a process, subject to the restrictions resulting from update blocks and exclusive procedures. *If two processes are due simultaneously, the highest priority process receives control.* The purpose of the priority clause in the SCHEDULE statement is to allow the system to resolve conflicting requests for the hardware resources. In the example, Guidance becomes ready while Control is executing about half the time. Since its priority is less than that of Control, Guidance is stalled until Control completes. Every time Guidance executes, Control comes due in the middle. Here again, the priorities govern the situation, and Guidance is stalled (interrupted) while Control runs. When Control completes, Guidance resumes at the point of interruption. As long as the shared data protection features of chapter eleven are used, this system action has no impact on the coding of either task, although some overhead is associated with the process swap.

Since Control can interrupt either of the other two processes, the jitter in its period of execution will be very small. Aside from the system overhead involved in swapping processes, delays in the execution of Control can result only from awaiting the release of locked data or an exclusive procedure by one of the other processes. Guidance can be delayed by the unavailability of a shared resource or by the execution of Control, Navigation can be interrupted by either of the others. Consequently, Navigation will generally run in very short bursts spread out through the entire second.

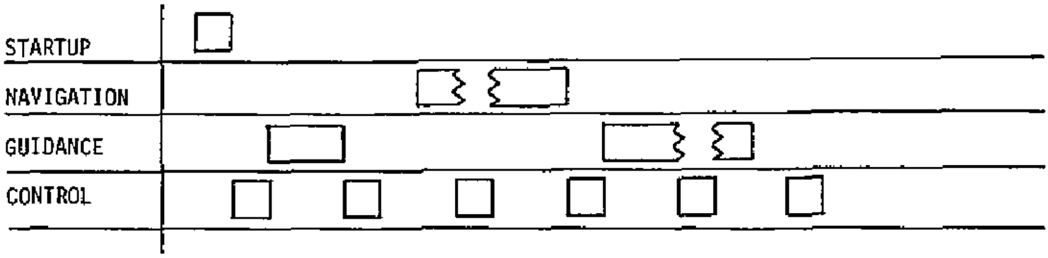
The example actually consists of four processes The three tasks and the STARTUP program The priority and other characteristics of STARTUP are determined externally, either through a SCHEDULE statement in another compilation unit or by default during system startup. Usually a HAL/S real-time executive will start a single program as a non-cyclic, process, this program must then schedule all other programs and tasks The priority of the STARTUP program affects the sequence in which the tasks are initiated If STARTUP is at priority fifty, when it schedules Navigation at priority sixty, Navigation becomes the highest priority ready process and therefore receives control immediately STARTUP is stalled until Navigation relinguishes the processor This happens when Navigation reaches its CLOSE statement, since it was scheduled to run only once per second, it enters an inter-cycle wait and ceases to be a ready process This makes STARTUP again the highest priority ready process, so it receives control and executes the second SCHEDULE statement The same situation is repeated with Guidance and Control

The effect of these SCHEDULE statements, then, seems very much like a set of CALL statements One major difference is that the Guidance, Navigation and Control tasks will continue to execute at the specified rates after STARTUP reaches its CLOSE statement, even though STARTUP executes only once Furthermore, *each HAL/S real time process has its own error environment* Any error handlers in STARTUP have no effect whatsoever on the action taken if an error occurs in one of the tasks Finally, the situation would be different if STARTUP had a higher priority

With STARTUP at priority fifty, the following time-line describes the first few cycles:



That is, Navigation and Guidance each complete a full execution uninterrupted before the higher priority task(s) are scheduled This may well simplify the system If STARTUP was at priority one hundred, however, the time-line would be completely different



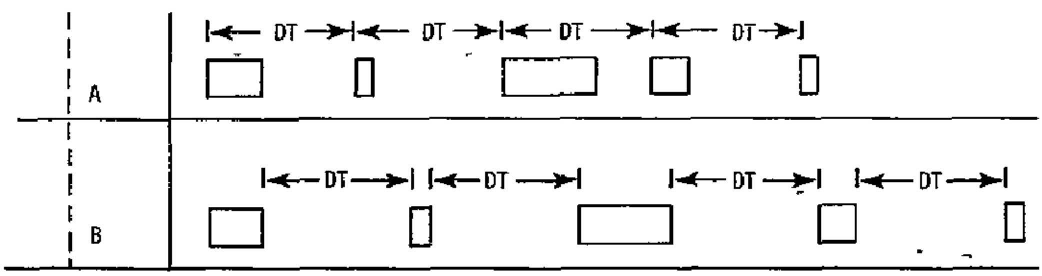
In this case, STARTUP executes all three SCHEDULE statements before any other process receives control Hence, the first cycle is not substantially different from any other

When STARTUP reaches its CLOSE statement, it enters the wait state This is similar to an inter-cycle wait, but does not result from timing considerations A program remains active as long as any of its tasks are active, due to the possibility of shared data and utility routines at the program level It is said to be "waiting for dependent processes" The memory allocated to the program cannot be released If the tasks are subsequently cancelled (i e cease to cycle), the program completes as well It is neither ready nor waiting, but simply done and forgotten In the terminology of the Language Specification, it is no longer "in the process queues"

The minimum form of the SCHEDULE statement contains only a process name and a priority, as in

SCHEDULE STARTUP PRIORITY(100),

If no repetition option is specified, the program or task executes only once The REPEAT EVERY specifies cyclic execution with a fixed interval between the beginnings of the cycles The REPEAT AFTER option is very similar, but the fixed interval is between the end of one cycle and the start of the next, as illustrated in this figure



The REPEAT AFTER form specifies the length of the inter-cycle period of waiting If REPEAT AFTER is specified, the average time between executions is the sum of DT and the average execution time whereas it is simply DT in the case of REPEAT EVERY The primary advantage of the REPEAT AFTER form is that a cycle overlap error cannot occur If process A in the previous example executes more than DT seconds in a particular cycle, it will come due again before it completes This results in a runtime error for which no ON ERROR handler can be written Process B above can execute for any length of time without an over lap, since the start of the next cycle is delayed until DT after the previous cycle completes

The primary disadvantage of the REPEAT AFTER option is that it may make system verification more difficult Use of this option tends to make the time-line of the entire system unrepeatable If the outputs of a control system depend on the sequence in which various processes are executed, a huge number of runs may be required to show that no unacceptable transients are introduced by timing fluctuations On the other hand, if REPEAT AFTER is used for less critical processes, the entire system may respond better to overload conditions

If REPEAT is specified without either AFTER or EVERY and a time

SCHEDULE X PRIORITY(17), REPEAT,

the process is immediately restarted at the end of each cycle This is equivalent to ",REPEAT AFTER 0,". This option is generally used for processes intended to use "left over" time for self-test, etc , and for processes which issue WAIT statements Use of the simple REPEAT option is not substantially different from coding an infinite loop around the task body and scheduling it as a "one-shot" The effect of the CANCEL statement is different, and under some implementations error recovery may differ as well

The SCHEDULE statement has several other options in addition to the three REPEAT forms These options allow the start of a process to be delayed until a specific condition is met, and allow cancellation criteria to be specified at the time a process is scheduled Both begin and end conditions and a repetition option may be used in a single SCHEDULE statement, as shown below

```
M | X:
M | PROGRAM;
M | P:
M | TASK;
M | CLOSE P;
M |    SCHEDULE P IN 5.4 PRIORITY(49), REPEAT EVERY .03 UNTIL RUNTIME + 100,
M | CLOSE X;
```

This statement will cause the program or task P to be initiated at priority 49 five point four seconds after the execution of the SCHEDULE statement Subsequently, it will be executed* every 03 seconds for 94 6 seconds and then be terminated

The IN and UNTIL options allow any arithmetic expression This expression is a time value in the same units as in the repeat options, generally seconds The IN option requires an interval of time whereas UNTIL expects an absolute time This is the same as the normal English usage of these words Since the RUNTIME function returns the current value of the system clock, "IN 5 4" is equivalent to "AT RUNTIME+5 4", a form which is also acceptable to the compiler

All of the arithmetic expressions in a SCHEDULE statement are evaluated only once, when the statement itself is executed Subsequent changes to the variables used in these expressions do not affect the scheduled process

The various scheduling options must be specified in the correct sequence, and only one of a given type is allowed in a single statement The sequence of phrases in a SCHEDULE statement is

1) SCHEDULE and a process name,
2) An optional begin condition IN, AT or ON,
3) A priority,
4) An optional REPEAT clause,
5) An optional end condition UNTIL or WHILE

The ON and WHILE conditions reference event variables, which will be described in Section 12 2 First a few special cases of the time options need mention

Normally, the IN or AT time used in a schedule statement is in the future If the specified time has already passed, the process is readied immediately There is one exception If AT is used with the REPEAT EVERY option and the time has already passed, phased scheduling is performed The first execution of the process occurs at the time given by the *sum* of the "AT" time and the period (REPEAT EVERY delta) of the process This allows a "synchronous" real-time structure, which is further described in the Language Specification Phased scheduling tends to minimize the number of processes that are ready at any one time

Normally, the UNTIL time specified is in the future If it is already passed, then the SCHEDULE statement has no effect *The UNTIL clause can never stop a process in mid-execution.* If the UNTIL time arrives while the process is executing, it is allowed to finish its current cycle The UNTIL and WHILE clauses can only stop a process before its first execution or during an inter-cycle wait When the end condition specified in a SCHEDULE statement is satisfied, the process is CANCELled rather than TERMINATEd, a distinction which will be explained in Section 12 3

---

*Assuming that its priority is sufficient to obtain necessary resources

Exercises

12.1A Draw a time-line for one second's execution of the processes scheduled below. Assume that each process executes for 80 ms per cycle.

```
SCHEDULE A PRIORITY(100), REPEAT EVERY 1/5;
SCHEDULE B PRIORITY(99), REPEAT EVERY 1/3;
SCHEDULE C PRIORITY(98), REPEAT EVERY 1/2;
```

12.1B Draw a time-line for the processes in exercise 12.1A, but with all occurrences of EVERY changed to AFTER.

12.1C Given two tasks, X and Y, both of which use one half second per iteration, write schedule statements that will run X continuously for two seconds, then alternate X and Y for two seconds, and then run Y half the time for two more seconds. Use only two schedule statements.

12.2 EVENT VARIABLES

The three forms of begin-condition in a SCHEDULE statement are

IN "arithmetic expression",
AT "arithmetic expression", and
ON "event expression".

Two of these forms describe a begin-condition in terms of time; the third form, ON, lets scheduling depend on conditions or occurrences which do not happen at a predetermined time. Suppose, for example, that the Guidance, Navigation and Control tasks of the previous example are used during launch of a spacecraft, but when orbit is achieved, Guidance and Control are to be replaced with another task, Freefall. If the time at which orbit will be reached is known in advance, this can be done with the AT and UNTIL clauses already presented. Otherwise, it is appropriate to declare an event variable to correspond to this occurrence as in

```
DECLARE ORBIT EVENT;
```

Then the desired transition can be specified in the SCHEDULE statements as shown in the next example. When an event variable is signalled, as in

```
SIGNAL ORBIT;
```

all *active event expressions* which reference that event are evaluated. In this case three active event expressions reference ORBIT. When the SIGNAL statement causes ORBIT to become TRUE, these expressions are all satisfied. Guidance and Control are cancelled via the UNTIL clauses, and Freefall is started via the ON clause.

An active event expression is a boolean combination of event variables used in a real-time statement which has not yet been satisfied. Event expressions are formed in the same way as boolean expressions using the AND, OR, and NOT operators However, *all variables in an event expression must be events* In the simplest case, an event expression consists of a single event variable, e g "ORBIT" in the SCHEDULE statements above A boolean combination of event variables is only considered an event expression when it is used in one of the real-time statements An *active* event expression is one that has never evaluated to TRUE since the containing real-time statement was executed Once ORBIT is signalled, the event expressions in the SCHEDULE statements are no longer active Signalling ORBIT again will have no effect unless additional real-time statements which reference it are executed

```
M | STARTUP
M | PROGRAM;
M |    DECLARE ORBIT EVENT;
M | GUIDANCE
M | TASK,
  |
C |
  |
M | CLOSE GUIDANCE;
M | NAVIGATION:
M | TASK;
  |
C |         ...
  |
M | CLOSE NAVIGATION,
M | CONTROL:
M | TASK,
  |
C |         ...
  |
M | CLOSE CONTROL;
M | FREEFALL
M | TASK,
  |
C |         ...
  |
M | CLOSE FREEFALL,
M |    SCHEDULE NAVIGATION PRIORITY(60), REPEAT EVERY 1 0;
M |    SCHEDULE GUIDANCE PRIORITY(70), REPEAT EVERY 1 / 6 UNTIL ORBIT;
M |    SCHEDULE CONTROL PRIORITY(80), REPEAT EVERY 1 / 20 UNTIL ORBIT;
M |    SCHEDULE FREEFALL ON ORBIT PRIORITY(75), REPEAT EVERY 1 / 10;
M | CLOSE STARTUP,
```

When an event expression is used in the UNTIL or WHILE clause of a SCHEDULE statement, it can cause cancellation of a process When used in the ON clause of a SCHEDULE statement or in a WAIT statement, it can cause a process to be readied or stalled Event expressions are used only in SCHEDULE and WAIT statements, and always serve as a condition under which the state of some process is to be changed

There are three types of event variables Latched and unlatched declared events, and process events All events have only two states, ON and OFF, the distinction between latched and unlatched events is that an unlatched event does not retain its state. ORBIT is an unlatched event since the LATCHED keyword was not specified in its declaration It is initially OFF or FALSE When the SIGNAL statement is executed it becomes momentarily TRUE, just long enough for all active event expressions which reference it to be evaluated SIGNAL is the only statement which can affect the value of an unlatched event

As stated above, an event expression can be a boolean combination of event variables Since an unlatched event is only true during the execution of a SIGNAL statement, and only-one event can be signalled at a time, the logical conjunction (A & B) of two unlatched events will never be satisfied This is one reason for using LATCHED events, as illustrated below

```
M | P:
M | PROGRAM;
M |    DECLARE ORBIT EVENT LATCHED INITIAL(FALSE);
M |    DECLARE ENGINE_OFF EVENT LATCHED INITIAL(FALSE);
M | GUIDANCE:
M | TASK,
M | CLOSE,
M |    SCHEDULE GUIDANCE PRIORITY(70), REPEAT EVERY 1 / 6 UNTIL ORBIT AND ENGINE_OFF,
M | CLOSE P,
```

Here, Guidance will continue to cycle until both ORBIT and ENGINE_OFF are true at the same time This can happen in several ways The sequence

SET ORBIT,
SET ENGINE_OFF,

will cause Guidance to be cancelled When a latched event variable is SET it remains true until it is RESET A latched event may also be SIGNALled In this case, the state of the event is momentarily inverted for the duration of the SIGNAL statement, just as in an unlatched event Thus,

SET ORBIT,
SIGNAL ENGINE_OFF,

will also cause Guidance to be cancelled, as will

SET ENGINE_OFF,
SIGNAL ORBIT,

However, if one event is first signalled *and then* the other set, there will be no time at which both are true, and Guidance will continue The advantages of using *unlatched* events will become clearer when the WAIT statement is introduced

The third type of event is a *process event* These events are not declared by the programmer, but automatically defined to correspond to the state of each program or task The process event has the same name as the program or task, and is true from the time the process is scheduled until it completes its last cycle *The process event of a cyclic process remains true during the inter-cycle wait,* and during any other stall or wait state Process events cannot be SET, RESET or SIGNALled, they simply reflect the state of the process of the same name

Process events can be used to solve a problem in the Guidance and Control to FREEFALL transition of the previous example Since a process cancelled via the UNTIL clause of its SCHEDULE statement is allowed to finish its current cycle, FREEFALL will start before the other tasks have finished if they are active at the time the event expression becomes true This difficulty is corrected in the following code

```
M | STARTUP
M | PROGRAM,
M |    DECLARE ORBIT EVENT LATCHED;
M | GUIDANCE:
M | TASK,
  |
C |       ...
  |
M | CLOSE GUIDANCE;
M | NAVIGATION
M | TASK,
  |
C |
  |
M | CLOSE NAVIGATION,
M | CONTROL:
M | TASK,
  |
C |
  |
M | CLOSE CONTROL;
M | FREEFALL:
M | TASK;
  |
C |       ...
  |
M | CLOSE FREEFALL;
M |    SCHEDULE NAVIGATION PRIORITY(60), REPEAT EVERY 1.0;
M |    SCHEDULE GUIDANCE PRIORITY(70), REPEAT EVERY 1 / 6 UNTIL ORBIT;
M |    SCHEDULE CONTROL PRIORITY(80), REPEAT EVERY 1 / 20 UNTIL ORBIT,
M |    SCHEDULE FREEFALL ON ORBIT PRIORITY(75), REPEAT EVERY 1 / 10,
M | CLOSE STARTUP,
```

The FREEFALL process is initiated when ORBIT is true and both other tasks have completed their last cycles In this case, ORBIT must be a latched event and it should be SET rather than SIGNALled

The effect of SET, RESET and SIGNAL on latched and unlatched events is summarized in the table on the next page As shown SET and RESET leave a latched event in the TRUE or FALSE states, respectively When a latched event is SIGNALLed, its state is *momentarily* inverted Unlatched events are always FALSE, except when SIGNAL makes them momentarily TRUE

		Set	Reset	Signal
unlatched event		illegal	illegal	Take all event actions depending on TRUE state of <event var>
latched event	old value is FALSE	1 Set event state to TRUE 2 Take all event actions depending on TRUE state of <event var>	no action	Take all event actions depending on TRUE state of <event var>
latched event	old value is TRUE	no action	1 Set event state to FALSE 2 Take all event actions depending on FALSE state of <event var>	Take all event actions depending on FALSE state of <event var>

Events can also be tested in non-real-time statements, e g

```
IF ORBIT THEN DO,
```

Booleans and events may be freely mixed in boolean expressions However, when used in any statement other than SCHEDULE or WAIT, an *unlatched* event is always false

The SCHEDULE statements allow begin and end conditions to be specified in terms of either time or event expressions, but the repetition option can only be specified in terms of a constant interval of time The WAIT statement allows a piece of code to execute at irregular intervals

Suppose a process is required to execute *whenever* ORBIT is false and ENGINE_OFF is true The schedule statement can be used to initiate a process *the first time* this combination is true, as in

```
SCHEDULE RE_IGNITE ON NOT ORBIT
    AND ENGINE_OFF PRIORITY(999),
```

A convenient means of allowing this process to execute *every time* the event expression is true is shown on the next page

```
M | P:
M | PROGRAM;
M |    DECLARE EVENT,
M |                 ENGINE_OFF,
M |                 ORBIT LATCHED;
M |    SCHEDULE RE_IGNITE PRIORITY(999);
M | RE_IGNITE
M | TASK;
M |    DO WHILE TRUE;
M |       WAIT FOR ENGINE_OFF & ~ORBIT;
  |
C |                   .
C |
C |                   .
  |
M |    END;
M | CLOSE RE_IGNITE;
M | CLOSE P;
```

When the WAIT statement is executed, if the event expression is true, execution continues at the next statement If the event expression is false when the WAIT statement is executed, *the process is stalled until the expression becomes true* as a result of event variable changes by other processes If the event expression in a WAIT statement is not immediately satisfied, it is put into the pool of active event expressions, the process containing the WAIT statement is stalled (taken out of the READY state) and the highest priority ready process receives control The process issuing the WAIT can only continue when the specified condition is satisfied

Suppose that ORBIT and ENGINE_OFF are both latched events If they are SET and RESET from some process other than RE_IGNITE, it is possible that RE_IGNITE will execute too many times Since it is of such a high priority, RE_IGNITE may finish processing and re-execute the WAIT statement before the other process has a chance to RESET ENGINE_OFF In fact, if RE_IGNITE is the highest priority process and contains no other WAIT statement, it will continue to loop to the exclusion of every other process If the RESET statement can be placed in RE_IGNITE right after the WAIT statement the problem is solved, but the situation could be avoided altogether by using a SIGNAL statement instead of SET Since SIGNAL leaves an event in the true state just long enough for all active event expressions to be evaluated, there is no possibility that RE_IGNITE will re-issue the WAIT statement while the event is still true The SIGNAL statement is generally used when an event is expected to change its state repeatedly, as there is no need to RESET* it in preparation for the next use. Note, however, that if the process which is to wait for the event has not already executed its WAIT statement, the SIGNAL has no effect

*Signal momentarily *inverts* the state of a latched event If a process waits for the false state, SIGNAL avoids the need to SET the event before the next cycle

Consider the two communicating processes below

```
P:
PROGRAM;
   DECLARE DO_SOMETHING EVENT;
   DECLARE DONE EVENT LATCHED INITIAL(OFF);
   SCHEDULE T PRIORITY(50);
   SIGNAL DO_SOMETHING;
   WAIT FOR DONE;
T:
TASK;
   WAIT FOR DO_SOMETHING;
   SET DONE;
CLOSE T;
CLOSE P;
```

In this example, if the priority of P is greater than 50, neither process will ever complete If the priority of P is less than 50, T will execute its WAIT statement before DO_SOMETHING is signalled, and both processes will complete If P is the higher priority process, it must pause before signalling DO_SOMETHING to give T a chance to execute its WAIT statement This could be done by adding

WAIT 1;

just before the SIGNAL statement

Exercises

12 2A Why does the SCHEDULE statement have both AT and ON clauses?

12 2B In the program segment below, at which of the points A-D is the event expression Q active?

```
DECLARE Q EVENT LATCHED INITIAL(OFF),

A

SCHEDULE TASK1 ON Q PRIORITY(57),

B

SIGNAL Q,

C

SET Q,

D
```

12 2C Let X be a latched event which is initially OFF How is SIGNAL X, different from the sequence SET X, RESET X,?

12 2D Re-do problem 12 1C with the two transitions based on events assume that unlatched events, tran1 and tran2 are signalled at appropriate times by another process

12 2E Is a latched or unlatched event more appropriate in each of the following situations

a) As the single operand of an ON clause

b) As part of a complex event expression

c) In a boolean expression

d) In the RESET statement

e) In a WAIT statement inside a loop

12 2F Write code that will cause the state of one event variable, COMPL, to always be the inverse of another event, MASTER, which is set and reset by some other code Do not examine the state of MASTER more often than necessary

12 3 OTHER REAL-TIME STATEMENTS

The SCHEDULE statement creates a process of some priority and possibly with some repetition rate Begin and end conditions can be specified in terms of either time or event variables These event variables may be SET, RESET and SIGNALled by other processes The WAIT statement allows a process to voluntarily release control pending some future condition This condition, like those in the SCHEDULE statement, may be either a combination of event variables or the passage of time.

In addition to the time option of the WAIT statement, this section presents the CANCEL and TERMINATE statements, which allow a process to discontinue itself or some other process, and the UPDATE PRIORITY statement, which is used to modify the priority of a process which has already been scheduled

The WAIT statement has three forms

WAIT FOR "event expression",
WAIT "delta time", and
WAIT UNTIL "time",

The effect of the statement is the same in all cases If the specified condition is already true, execution continues, otherwise, the process is stalled until the condition becomes true

As in the SCHEDULE statement, the expressions "delta time" and "time" may be any arithmetic expression, both are in the same units as time values in other real-time statements The two forms distinguish between a particular time, and an interval of time, which is the same distinction as between the IN and AT options of the SCHEDULE statement As before,

WAIT 1,

is equivalent to

WAIT UNTIL RUNTIME + 1,

These forms of the WAIT statement are generally used in "sequencing" applications, for instance to fire a vehicle control jet for a given duration or to wait between commands to some slow moving mechanical device They are also useful in testing, to generate a scenario of simulated inputs as a function of time

Note that the arithmetic expressions in the time-oriented WAIT statements are evaluated only once, when the WAIT statement is executed The expression "RUNTIME + 1" does not keep sliding into the future, but is converted to a scalar value when the WAIT statement is executed It is only event expressions that are repeatedly evaluated by the system

A further example of the WAIT statement, is shown below Here, the acceleration of a vehicle is controlled to get from HERE to THERE in minimum time by accelerating halfway and deaccelerating halfway Steering is ignored, as is any initial velocity

```
M | P-
M | PROGRAM,
M |     DECLARE VECTOR,
M |                 HERE, THERE;
M |     DECLAPE MAX_THRUST CONSTANT(1234),
M |             VEH_MASS CONSTANT(5678),
M |     DECLARE SCALAR,
M |                 A, S, T;
M |     DECLARE BOOLEAN,
M |                 ACC_CMD, DECC_CMD;
M |     A = MAX_THRUST / VEH_MASS,
E |                 -         -
M |     S = ABVAL(HERE - THERE) / 2;
M |     T = SQRT(2 A S),
E |
M |     ACC_CMD = ON,
M |     WAIT T;
E |
M |     ACC_CMD = OFF;
E |      .
M |     DECC_CMD = ON,
M |     WAIT T;
E |
M |     DECC_CMD = OFF,
M |  CLOSE P,
```

In this example, "WAIT T," introduces a delay of T seconds between setting ACC_CMD on, and back off

The WAIT statement temporarily deactivates a process, a process can also be permanently deactivated A non-cyclic process (no REPEAT clause in the SCHEDULE statement) terminates by executing its CLOSE statement, by causing a fatal runtime error, or as a result of the TERMINATE statement A cyclic process can cease executing as a result of the WHILE or UNTIL clause used when it was scheduled, the occurrence of a fatal error, or the execution of a CANCEL or TERMINATE statement

The CANCEL and TERMINATE statements are similar in form, each consisting of a keyword (CANCEL or TERMINATE) followed by a list of process names, for example

```
CANCEL GUIDANCE,
TERMINATE STARTUP,
CANCEL NAVIGATION, CONTROL, P, T,
```

The TERMINATE statement causes immediate, abrupt cessation of the listed processes Since it may stop a process at any point in its execution, its use is strongly discouraged The HAL/S Language Specification imposes additional rules on the use of TERMINATE The only use of TERMINATE which is generally considered acceptable is

```
TERMINATE,
```

When no list of processes is supplied, self-termination is implied This form of the TERMINATE statement can serve as a "super return" statement at the PROGRAM or TASK level Since the process "knows" its own state, this form is relatively safe When other processes are terminated, it is important to consider all possible points at which they might be executing to ensure safety

The CANCEL statement allows an orderly shut-down of the specified processes Like the WHILE and UNTIL clauses of the SCHEDULE statement, *CANCEL can only stop a process before its first cycle or during the inter-cycle wait* This allows processes to be stopped without the risk of leaving *partially* updated results

Since a cancelled process is allowed to finish its current cycle, the CANCEL statement may not have immediate effect Process events can be used to key on the completion of the last cycle before scheduling a "replacement" process, as shown below

```
CANCEL X, Y, Z,
WAIT FOR ¬X & ¬Y & ¬Z,
SCHEDULE XYZ_NEW PRIORITY(10), REPEAT,
```

Exercises

12 3A Surround the statement "WRITE(6) RUNTIME," with other statements so that the values 1/10, 1/8, 1/6, 1/4, 1/2, and 1 will be sent to channel 6 Use no other I/O statements Do not worry about numeric accuracy

12 3B Given

```
P  PROGRAM,
    DO WHILE TRUE,
       /*something*/
    END,
CLOSE,

SCHEDULE P PRIORITY(100),
```

What does "CANCEL P," do? How *should* this be done?

End of Chapter Problems

Part of the specification of the flight software for the XYZ aircraft might read as follows

Category	Rate	Functions
A	R_A	input processing elevon commands telemetry
B	$1/2\ R_A$	rudder commands guidance
C	$1/4\ R_A$	flight control gains
D	$1/8\ R_A$	navigation display updates

The software functions are divided into four categories as shown The category A software is to be executed at the highest possible rate consistent with the throughput of the machine and the total workload The category B software shall execute one-half as frequently as category A, the rate of category C shall be half that of category B, and the rate of category D shall be one-half that of category C (i e one-eighth the rate of category A) "

12A Implement the above example via the real-time statements Explain your choice of priorities Fix rate A at one-tenth

12B Re-do the problem under the original "as fast as possible" groundrule

13 0 SYSTEM PROGRAMMING AIDS

The information presented in earlier chapters applies equally well to any HAL/S compiler Except for numeric precision, the examples shown will produce the same results under any complete implementation of the HAL/S language This transferrability was one of the major design goals of the language It decreases the dependence on the availability of flight hardware and encourages the re-use of debugged software

In order to provide this degree of machine-independence, the language isolates the user from details of the underlying hardware, e g , the number of bits in a scalar The arithmetic data types, Integer, Scalar, Vector and Matrix correspond to mathematical abstractions For most users, the mapping of these data types into the data formats supported by a given computer is of no concern The operations that can be performed on these data types are defined in a way that is completely independent of any computer architecture The character string, boolean, and event types also are defined abstractly Users do not normally need to know how much memory is occupied by a boolean or what character code (ASCII, EBCDIC, etc) is used internally Since these low level decisions are made in the compiler, HAL/S code is usually machine-independent

While most flight code implements algorithms that are defined in machine-independent mathematical or logical terms, small portions of many projects are specified in terms much closer to the computer in use Examples of this low level code are formatting sensor data, handling interrupts, managing real-time clocks, commanding special purpose avionics, etc These functions are intrinsically machine-dependent Algorithms are designed in terms of hardware capabilities and concepts Thus, there is little chance of sharing this type of software between different projects Transferrability of "systems programs" is not a practical goal, given the diversity of flight hardware

Even though system software is generally specific to a given computer, the other advantages of high order languages still apply Also, the use of a single language for both application and system programs tends to simplify interfaces, documentation and training Hence, HAL/S provides some features for writing system software, including the use of pointers and low-level bit manipulation

These features are most frequently used in software that is intrinsically non-transferrable The restriction of bit manipulation to the BIT data type, and similar constraints on addresses, separate the possibly machine-dependent systems programs from applications code

13 1 BIT STRINGS

A bit string is a series of binary digits Each digit or bit behaves like a boolean The forms, BOOLEAN and BIT(1), are completely interchangeable A bit string of length four can be created via ,

```
DECLARE FLAGS BIT(4),
```

Like vectors, character strings and other aggregate data types, bit strings may be subscripted to select single components or partitions The first, leftmost, or most significant bit of FLAGS is denoted FLAGS$1 The last two bits would be referenced as FLAGS$(2 AT 3)

That catenation operator (| |) also applies, though bit strings differ from character strings in that bit strings are of fixed length. The AND, OR and NOT operators can be applied to entire strings as well as their boolean components.

The length of a bit string must be less than an implementation-dependent limit. This limit generally equals the maximum number of bits that can be loaded into a general purpose accumulator or register on the target machine.

Operations on single bit components of a bit string are generally *slower* than corresponding operations on BOOLEANs or entire bit strings. The machine instructions to perform these operations also tend to occupy more space.*

Because of the inefficiency of operating on a component of a bit string while leaving the other bits alone, bit strings should not routinely be used to pack the individual booleans of a program into a single word. One type of situation in which bit strings can be used effectively is illustrated below.

```
M   DECLARE I INTEGER,
M   DECLARE B BIT(8);
M   DECLARE BOOLEAN,
M                 C1, C2, C3, C4, C5, C6, C7, C8,

C

M   DO WHILE ON;
M      DO FOR I = 1 TO 100;
E                 .
M         IF B = HEX'00' THEN
M            DO,

C                ..

M            END;
M         ELSE
M            DO;

C

M            END;
M      END,
E          .
M      IF C1 THEN
E          .
M         B  = ON,
S          1

E
M      IF C2 THEN
E          .
M         B  = ON;
S          2

C      .
C
C
```

*This is because most memory units are designed to transfer many bits (a byte or word) to or from the CPU in one operation. Modifying a single bit generally requires the use of logical or shifting instructions to preserve the state of adjacent bits.

```
E              .
M        IF C8 THEN

E               .
M               B   = ON;
-S               8

M     END;
```

In this code, eight booleans are packed in a bit string called B This makes the statements, B$1=ON, B$2=ON, etc , less efficient than references to the individual booleans, C1, C2, etc However, the statement

```
IF B = HEX'00'THEN DO,
```

is *much* more efficient than

```
IF NOT (C1 | C2 | C3 | C4 | C5 | C6 | C7 | C8) THEN
    DO,
```

Since this statement is executed much more frequently than the individual assignments, the savings from making a simpler test more than offsets the cost of the component assignments Thus, one application of bit strings is to collect booleans for testing as a group

The example above tests whether all eight bits are false Other compound conditions can be tested via the AND and OR operators For instance, the following statement tests for the odd-numbered bits equal to zero

```
IF (B & BIN'1010101010') = HEX'00' THEN
    DO,
```

The test that bits 1 and 3 are on and 2 and 5 are off can be coded as

```
IF (B & BIN'11101000') = BIN'10100000' THEN
    DO,
```

When booleans are collected in a bit string, it is still possible to give symbolic names to individual components via REPLACE statements, as in

REPLACE MEANINGFUL_NAME BY "B$3",

The only comparisons that may be made between bit strings are equality and non-equality (= and ¬=) As with arrays, the components are compared in pairs Two bit strings are equal if all pairs match, and unequal if any pair mismatches If two bit strings of unequal lengths are compared, the shortest is padded on the left with binary zeros before the comparison

This left padding also occurs prior to logical operations on bit strings of unequal lengths The following assignment statements all have the effect of setting B$6 to ON while leaving the other bits alone

B_6 = ON,

B = B OR HEX'04',

B = B OR HEX'4',

B = B OR BIN'100',

Provided that the implementation dependent limit on bit string lengths is not less than twenty

B = B OR HEX'00004'

will also produce the same results A copy of B is padded to length twelve before it is ORed with the HEX'004', and the result is truncated at the left (the most significant four bits are removed) before it is stored back into B

Partitions of bit strings may be used in the same ways as entire strings, e g

```
IF B = OCT'17' THEN
    1 TO 4
    DO,
```

The width of every bit partition must be known at compile-time This means that in the form BS(X AT Y), X must be an arithmetic expression composed solely of literals, CONSTANTs, REPLACE names and the arithmetic operators In the form B$(X TO Y), *both* X and Y must be computable at compile-time *Character strings* are the only data type for which variable-width partition subscripting is allowed

As we have stated, bit strings should not be routinely used to pack booleans The overhead of referencing the boolean components generally outweighs the savings of compressing them. In the first example, a bit string was appropriate since the entire string was referenced more often that its components

It may also be appropriate to use bit strings to pack a *table* of booleans 'Since there are generally fewer HAL/S statements which reference a table than entries in the table, it is possible to save memory (at the expense of execution time) by compressing the table while expanding each reference For instance, in the table of 1000 booleans,

DECLARE INFO ARRAY(1000) BOOLEAN,

each array element can be easily referenced as in

IF INFOS (I) THEN DO,

but the table itself will occupy a lot of memory Each boolean uses a whole byte, word, or other addressable unit To save some storage, this table could be packed as shown below

```
M      DECLARE INFO ARRAY(1 + 1000 / 16) BIT(16);
M   TEST:
M   FUNCTION(I) BOOLEAN;
M      DECLARE I INTEGER;
M      DECLARE INTEGER,
M                 WORD, BITNUM,
M      WORD = DIV(I, 16);
M      BITNUM = I - 16 WORD,
E
M      RETURN INFO                ,
S                 WORD+1.BITNUM+1

M   CLOSE TEST,
```

Now the value of entry number I in the table can be referenced as TEST(I) This will be a less efficient reference, but the table size has been greatly reduced

This example assumes that the computer on which the code executes can address memory by the 16-bit unit If not, this code could be very much less efficient Thus, this example is not machine-independent It would still compile and produce the correct results on, say, a 24-bit machine, but to achieve the same efficient use of memory would require changing the four occurrences of 16 to 24 Thus, one reason why programs containing bit strings tend to be less transferrable is that bit strings are sometimes used to control the packing of information in "words" of memory

The expression INFOS(WORD BITNUM) contains both array and component subscripts As before, many combinations of simple and partition, component, array, and structure subscripts are allowed

One of the most common uses of bit strings in aerospace applications is for formatting sensor and display data For example, a sensor might produce a value in "packed decimal" format Six four-bit fields, each containing a number from 0 to 9 (BIN'0000' to BIN'1001'), packed in a 24-bit word This could be converted to a simple integer by the following code

```
M   DECLARE INPUT BIT(24);
M   DECLARE OUTPUT INTEGER INITIAL(0);
M   DO FOR I = 1 TO 21 BY 4,
E
M      OUTPUT = 10 OUTPUT + INTEGER(INPUT        ),
S                                         4 AT I

M   END,
```

Here we see that the INTEGER shaping function will accept a bit string as its operand The effect is merely to treat the string as a binary number rather than a series of booleans

Conversely, the BIT function allows an integer to be treated as a bit string The length of the string returned is always equal to the implementation-dependent maximum bit length The code below assumes that the maximum is 16

```
M   DECLARE I INTEGER,
M           B BIT(16);
M   READ(5) I;
E   .
M   B = BIT(I);
E     .
M   IF B  THEN
S       1

M      WRITE(6) 'VALUE OF I WAS NEGATIVE',
E     .
M   IF B  THEN
S       #

M      WRITE(6) 'VALUE OF I WAS ODD';
```

This example produces correct results only on a 16-bit 2's complement or sign-magnitude computer Here the machine dependence results from both the string length of 16 and the assumptions made about the interpretation of the first and last bits of an INTEGER

Conversions between bit and integer types use the BIT and INTEGER functions The BIT function will also accept a scalar argument, and the SCALAR function will accept a bit argument However, *an intermediate conversion to integer occurs in scalar-to-bit and bit-to-scalar conversions* Thus, BIT(3 5) = BIN'0000000000000100', and SCALAR(BIN '0100') = 4 0 BIT of a scalar between zero and one-half generates a string of binary zeros

The value returned by the BIT function is always of the maximum legal length for bit strings, as defined for the compiler version in use This fact must be considered when the BIT function itself is subscripted The last four bits of an integer, I, can be referenced as

BIT$(4 AT #–3) (I)

but the expression

BIT$(1 TO 4) (I)

may or may not select the first four bits of I If the number of bits in the representation of an integer is less than the bit string length limit, the BIT function will left-pad the bit pattern of I with binary zeros up to the limit The subscript applied to a BIT function selects bits from the maximum-length *result* of the conversion, rather than from the original operand, so BIT$(1 TO 4) (I) may pick out padding instead of data

The CHARACTER function can convert a bit string to its binary, octal, decimal, or hexadecimal character representation This is specified via a radix, which is written as a subscript, for example

```
M    DECLARE B BIT(8),
E
M    B = BIT(25);
E                              .
M    WRITE(6) CHARACTER       (B),
S                        @HEX

E                              .
M    WRITE(6) CHARACTER       (B);
S                        @DEC

E                              .
M    WRITE(6) CHARACTER       (B);
S                        @OCT

E                              .
M    WRITE(6) CHARACTER       (B);
S                        @BIN

E             .
M    WRITE(6) B;
```

would produce

```
'19'
'25'
'31'
'0001 1001'
'0001 1001'
```

The BIT function can convert a character string back to a bit string The radix is supplied here as well Every character in the string must be a digit in the valid range for the specified radix BIT$(@HEX) ('12') is BIN'10010', BIT$(@OCT) ('12') is BIN'1010', and BIT$(@BIN) ('12') would result in a runtime error Note that conversions between character and bit do not depend on the codes used to represent numerals within character strings

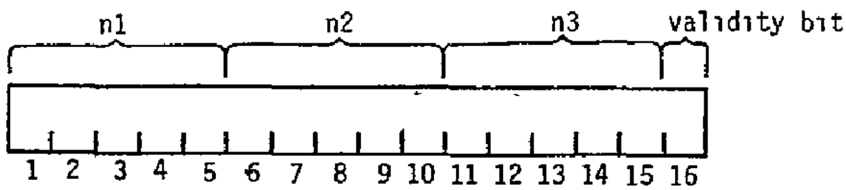
Another function, SUBBIT, allows any data type to be referenced, assigned, and subscripted *as if it were a bit string* SUBBIT obtains the internal representation of a variable with no modifications at all Since these representations of HAL/S data types vary from computer to computer, programs which use SUBBIT can not be machine-independent

The SUBBIT function is used in the code below to convert a character string containing decimal digits to the packed decimal form discussed earlier This routine assumes that the digits are represented in the EBCDIC character code In this code (which is not used in *all* implementations) the decimal digits 0–9 are represented by the binary codes HEX'F0' through HEX'F9'

```
DECLARE C CHARACTER (4) INITIAL('1234'),
DECLARE B BIT(16) INITIAL(HEX'0000'),
DO FOR TEMPORARY I = 1 TO 4,
   B = B || SUBBIT $(5 TO 8)(C$I),
END,
```

The expression SUBBIT$(5 TO 8)(C$I) selects bits five through eight of the binary representation of the Ith character of C SUBBIT can also be used to *modify* a variable as if it were a bit string The SUBBIT function is described further in the HAL/S language specification

As a final example of bit strings, consider the following problem A set of three redundant sensors produce an ARRAY(3) BIT(16), where each sensor contributes one array element containing four fields as shown below



The problem is to produce a fourth word in the same format which contains average values The five bit fields will be treated as unsigned integral numbers, the validity bit in the average will be true if and only if all three input validity bits are true

The data can be declared as

```
DECLARE DATA ARRAY(3) BIT(16),
DECLARE AVERAGE BIT(16),
```

and the computation can be done in a single statement

```
E
M    AVERAGE = BIT         (SUM(INTEGER([DATA]         )) / 3) || BIT      (SUM(INTEGER([DATA]
S                5 AT #-4                     * 1 TO 5                 5 AT #-4                    * 5 AT

E
M     )) / 3) || BIT        (SUM(INTEGER([DATA]          )) / 3) || DATA     AND DATA     & DATA    ,
S    6               5 AT #-4                    * 5 AT 11                1 16         2 16        3 16
```

Note that the bits in the diagram were numbered from one to sixteen, starting at the left (or most significant bit) HAL/S always numbers bits in this way, regardless of any conventions that may be used in hardware documentation

The expression BIT$(5 AT #-4) (   ) selects the last five bits of its operand Since the length of the string returned by the BIT function is implementation dependent, the use of "#-4" instead of "12" or "28", etc , is generally preferred

DATA$(* 1 TO 5) is an ARRAY(3) BIT(5), this expression selects a bit partition from each array element Thus, the INTEGER function is being presented with an array of "N1" fields

This example also shows the use of the catenation operator on bit strings, which operates in the same way as on character strings

In this section, two major uses of bit strings have been presented First, bit strings were used to collect booleans into a single word so that a complex boolean expression could be reduced to a simple comparison, the examples would work under any HAL/S implementation The other major use of bit strings is for manipulating quantities of less than one addressability atom Bit subscripts can be used to pick apart a word of memory This allows explicit user control over the packing of data, and provides a facility for reformatting avionics I/O data In this case, such considerations as the word size of the target machine and the internal representations of HAL/S data become important, hence, there is a degree of implementation-dependence in the use of bit strings

Exercises

13 1A Given,

```
DECLARE FLAGS BIT(12),
```

write expressions that test for each of the following conditions without using subscripts

a) bits 1 and 2 on,

b) even numbered bits off,

c) first six bits off or last six on,

d) bits 1, 3, 5, 11 on, others off, and

e) bits 1, 3, 5, 11 on, 2, 12 off, others irrelevant

13 1B Fill in the following function so it agrees with the comment

```
    FLIP  FUNCTION(B) BIT(12),
        DECLARE B BIT(12),
C   Return string of bits in reverse order,
C   i e , FLIP(HEX'001') should be HEX'800'
    CLOSE FLIP,
```

13 1C Six bits can represent an integer value between zero and 63 If a table of 200 such values were to be stored in a computer with a 24-bit word, it would be advantageous to pack four values per word Write a procedure,

```
SET_BITS  PROCEDURE(ENTRY,VALUE),
```

which can be called to set one of the 200 6-bit entries to value, and a function,

```
GET_BITS  FUNCTION(ENTRY) INTEGER,
```

which returns the value of one entry Use the declaration

```
DECLARE TABLE ARRAY(50) BIT(24),
```

13 1D A common format for floating point numbers consists of a sign bit, followed by seven exponent bits, and 24 mantissa bits The value of the number is

$\pm$ mantissa x $16^{\text{exponent} - 64}$

A non-zero number is said to be "normalized" if the first four bits of the mantissa are not all zero Write a procedure which interprets its BIT(32) argument as a floating point number, and returns a BIT(32) which has the same floating point value as the input, but is normalized If the input mantissa is 0, then return true zero (i e , all bits = 0) When would such a routine be useful?

13 1E Re-do the packed decimal to integer conversion example in the text using only one executable statement

13 1F Re-do the problem above without any arithmetic operators Hint Use character operations

13 2 NAME VARIABLES

Name variables are pointers or addresses, they allow data to be referenced indirectly Name variables are sometimes called "pointers-to", since each name variable can point only at variables of a given data type The type of the data pointed to is specified in the declaration of the name variable itself

The most prevalent use of pointers in general is to pass *the address of* a data aggregate (such as MATRIX) to a subroutine In HAL/S, this is done implicitly via ASSIGN parameters, hence, the need for name variables in application programs is almost eliminated In system programs, name variables may be used for efficiency in maintaining linked lists and queues, for buffer control and storage management, and for interfaces to non-HAL/S code or I/O hardware (e g , a DMA channel)

Another common use of name variables is to avoid a repeated structure subscript operation Suppose an inertial sensor produces data in the format indicated below

```
STRUCTURE IMU_DATA
   1 DELTA_V ARRAY(3) INTEGER DOUBLE,
   1 ATTITUDE ARRAY(3) INTEGER,
   1 STATUS BIT(16),
```

There are three of these sensors

```
DECLARE IMU_INPT IMU_DATA-STRUCTURE(3),
```

A low rate process is to select the best of the three copies of IMU data, the entire structure is to be read and the selected copy processed at a higher rate One way* to pass the selection information between the processes is as a structure subscript An integer,

```
DECLARE BEST INTEGER,
```

could be located in a compool visible to both processes It would be assigned to 1, 2 or 3 at the low rate, and the high rate would have computations involving IMU_INPT$(BEST,) No name variables are used so far, but this solution will work Individual components of the selected structure can be referenced as in

PITCH_ANGLE = SCALAR(IMU_INPT ATTITUDE$_{BEST,1}$),

*without using name variables

Every reference to the selected structure copy includes the subscripting operation This conceptually involves adding the base address of the structure to the product of the structure width and the value of BEST Multiplication is relatively slow on most computers It would generally be more efficient to compute the address of the BEST copy of IMU_INPT only once and reference it directly through this saved address Both "indexing" and "indirection" are performed in a variety of ways on different computers, but when the index requires multiplication, in this case by the width of ten integers, indirection is quicker This is *not* to say that it is always preferred Some of the risks of using name variables will be discussed later

Before giving the name variable solution, we note that the address can be computed and saved by adding an additional procedure

```
E                              +
M      CALL XTRA ASSIGN(IMU_INPT      );
S                                   BEST;

M   XTRA:
M   PROCEDURE ASSIGN(BEST_IMU);
M      DECLARE BEST_IMU IMU_DATA-STRUCTURE;

C              ...

M      PITCH_ANGLE = SCALAR(BEST_IMU.ATTITUDE );
S                                           1

C               ..

M   CLOSE XTRA;
```

Here the structure subscript is eliminated throughout the XTRA code block, since HAL/S ASSIGN parameters are a case of "call by reference" rather than "call by value" The *address* of the argument is passed to the procedure Name variables allow the same type of indirect reference without the overhead of calling an extra procedure This is shown below

```
M      STRUCTURE IMU_DATA
M         1 DELTA_V ARRAY(3).INTEGER DOUBLE,
M         1 ATTITUDE ARRAY(3) INTEGER,
M         1 STATUS BIT(16),

M      DECLARE IMU_INPT IMU_DATA-STRUCTURE(3);
M      DECLARE BEST INTEGER,
M      DECLARE PITCH_ANGLE SCALAR;

M      DECLARE BEST_IMU NAME IMU_DATA-STRUCTURE;
```

```
M   LOW_RATE:
M   TASK;
M      DECLARE BEST INTEGER,
M      CALL TBD ASSIGN(BEST);
E             +              +
M      NAME(BEST_IMU) = NAME(IMU_INPT      );
S                                  BEST;

M   CLOSE LOW_RATE,

C

M   HI_RATE
M   TASK,

C        ...

M      PITCH_ANGLE = SCALAR(BEST_IMU ATTITUDE );
S                                            1

C         ..

M   CLOSE HI_RATE;
```

This program is much the same as before In particular, the HI_RATE task is the same as when BEST_IMU was an assign parameter, except that the XTRA procedure is gone

The name variable, BEST_IMU, occurs three times in the program above First is the declaration A variable is specified to be a *name* by placing the keyword NAME before the data type The second is when it appears as an operand to the NAME function in the LOW_RATE task In this context (and only in this context) the name is treated as a pointer Here it is set to the address of the best copy of IMU_INPT The only way to "re-point" the name variable BEST_IMU is by executing a statement of the form

NAME(BEST_IMU) = NAME(   );

The only way to reference a name variable's pointer value at all is by use of the NAME function Normally, BEST_IMU is of type IMU_DATA-STRUCTURE It may be used anywhere that a non-name variable of type IMU_DATA-STRUCTURE is allowed In a normal context, outside the name function, a name variable serves as an alias for data of some other type, hence the terminology NAME instead of "pointer" This is not at all the same as the use of a REPLACE macro as in

REPLACE BEST_IMU BY "IMU_INPT$ (BEST,)",

because the replace macro results in the subscript operation performed every time In the case of name variables, changes to the value of BEST only affect which data is referenced by BEST_IMU when the

NAME(BEST_IMU) = NAME(IMU_INPT$ (BEST,),

name assignment is *executed*

Name variables may be of almost any data type, though the most useful is structure The types of data to which names cannot point are those which require more than a simple address to describe These are the same types that are disallowed as assign parameters, examples include bit partitions, matrix columns, etc

A name variable can only refer to data of *exactly* the same type as specified in its declaration This means that all of the type attributes must match, including precision, arrayness, structure hierarchy, and so on The INITIAL attribute is an exception The statement

```
E                                                            +
M   DECLARE BEST_IMU NAME IMU_DATA-STRUCTURE INITIAL(NAME(IMU_INPT  ));
S                                                             2,
```

initializes NAME(BEST_IMU), i e , the pointer value When a name variable is declared, the amount of storage reserved is just enough for one address The INITIAL attribute specifies the value to be placed in this address word The block of storage needed to contain an IMU_DATA-STRUCTURE is not allocated when the *name* is declared, thus the initial values for the structure pointed at must be specified elsewhere The statement shown causes the name variable BEST_IMU to point initially at the second copy of IMU_INPT

If the INITIAL attribute is not specified in a name declaration, the name initially points nowhere A special value is used as a null address so that all uninitialized names have the same values This null value is an address at which it is impossible to locate data and can be written either as "NULL" or as "NAME(NULL)" It is possible to determine whether or not a name variable points anywhere, as shown below

```
E                  +
M   IF NAME(BEST_IMU) = NAME(NULL) THEN
M      WRITE(6) 'BEST IMU NOT CHOSEN',
```

The basic NAME syntax has been shown in the context of one example The forms of declaring, initializing, re-pointing, and dereferencing (i e , accessing the data pointed at) have been shown The main example used is machine-independent and at least somewhat application oriented Nonetheless, there are pitfalls in the use of name variables It is difficult to find out what a name variable is pointing at by examining the code surrounding a reference to it Data which is accessed via name variables is not fully tracked in the cross reference listing Name variables allow a single location to be referenced by several identifiers, possibly resulting in obscure side-effects of assignments Name variables also tend to bypass compiler optimization, since they make it difficult to find a segment of code over which a particular variable is not modified. It is hard for either the programmer or the compiler to be certain what is being changed when name variables are assigned into. Thus, it is frequently worthwhile to use a less efficient but less dangerous construct such as structure subscripting. A common lament is "I though I understood this code until I saw these name variables!"

In most application code, name variables should be avoided. The possible gain in efficiency is generally outweighted by the loss in reliability and maintainability.* Name variables are provided in HAL/S primarily to allow the writing of system software.

Exercises

13.2A Name any three HAL/S data items which cannot appear as an operand of the NAME pseudo-function.

13.2B Which of the following can be done with name variables:

a) bypass HAL/S scoping rules,

b) declare a structure node with copiness,

c) reference a single data item by several names or identifiers,

d) reference absolute addresses, and

e) change the type of data.

13.3 LISTS AND QUEUES

The HAL/S language does not provide syntax for dynamic storage allocation. Temporary variables and space for intermediate results may be allocated and freed by the runtime code, but all decisions are made based on the *static* block structure, DO ... END grouping, etc. List processing languages can automatically release data that is not on any list and allow the space so created to be used for new lists. HAL/S does not provide this type of storage management because it is not possible to guarantee that such systems will not run out of storage: this would be an unacceptable condition in flight.

Aside from storage management, the most valuable feature of lists is that entries can be deleted or inserted in the middle without copying data. This capability is available in HAL/S through structures and name variables.

Consider the timer queue, a concept which is central to many operating systems. Each entry in the queue contains a time and an action to be taken. The queue is maintained in order of increasing time: the top entry is loaded into an interval timer. This could be coded in HAL/S as shown on the next page.

*Qualitatively speaking, a program's reliability is the probability that it has no hidden bugs. Its maintainability is the probability that it can be changed or extended without reducing reliability.

```
M    STRUCTURE TQE.
C    TIMER QUEUE ELEMENT
M       1 TIME SCALAR,
M       1 ACTION INTEGER,
M       1 AFFECTED_PROCESS NAME PROCESS_CONTROL-STRUCTURE,
M       1 NEXT NAME TQE-STRUCTURE,
M    DECLARE TQ TQE-STRUCTURE(100);
```

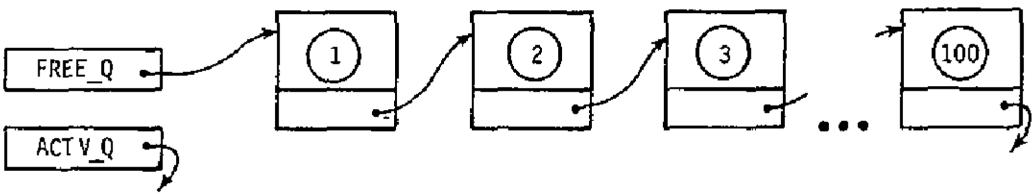
These statements create a 100-copy structure, with four fields in each copy Two fields are name variables, they are referenced in the usual manner, e g ,

TQ AFFECTED_PROCESS$(1,)

is the third field of the first copy of TQ It is of type PROCESS_CONTROL-STRUCTURE Only the address is physically contained in TQ$(1,), but the structure elsewhere is accessed when the name variable is referenced in a normal context (i e , outside of the NAME function) The other name variable points to a TQE structure The last field of TQE is the name of another TQE We will explore the implications of this later As it stands, all of the fields in TQ are null The queue could be initialized as shown below

```
M      DECLARE FREE_Q NAME TQE-STRUCTURE,
M      DECLARE ACTV_Q NAME TQE-STRUCTURE;

M   INITIALIZE
E              +               +
M      NAME(FREE_Q) = NAME(TQ   );
S                               1;

M      DO FOR TEMPORARY N = 1 TO 99,
E                +              +
M         NAME(TQ.NEXT  ) = NAME(TQ     ),
S                    N,            N+1;

M      END,
```

Now the entries in the queue are tied together with pointers, as shown below



The structure copy numbers are shown in the diagram, but each field can now be referenced without using a copy number, as indicated in the following table

Referenced Data	Pointed To By
TQ$(1,)	FREE_Q
TQ$(2,)	FREE_Q NEXT
TQ$(3,)	FREE_Q NEXT NEXT
TQ TIME$(2,)	FREE_Q NEXT TIME

Since FREE_Q NEXT is the name of a TQE structure, it also has a NEXT field This field points at the third entry in the free queue, which *at the moment* is also the third copy of TQ

The procedure below creates an entry in the active queue by removing it from the free queue and inserting it at the appropriate point in ACTV_Q based on the time field

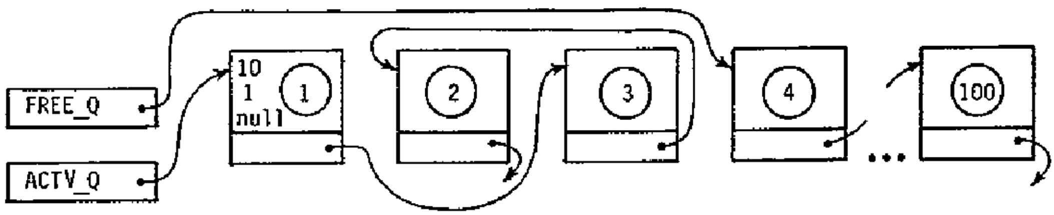
```
M   ENQUEUE
M   PROCEDURE(WHEN, WHAT, PROCNAME),
M      DECLARE WHEN SCALAR,
M              WHAT INTEGER,
M              PROCNAME NAME PROCESS_CONTROL-STRUCTURE,
M      DECLARE NEW NAME TQE-STRUCTURE,

C    THE FOLLOWING NAME VARIABLE IS USED LIKE A LOOP
C    VARIABLE IN A SEARCH

M      DECLARE ENT NAME TQE-STRUCTURE,

C     IF NO FREE ENTRY THEN AN ERROR

E                +
M      IF NAME(FREE_Q) = NULL THEN
M         RETURN,
```

```
                                        .
                                        .
C      ELSE USE TOP FREE ENTRY FOR NEW ACTIVE Q ELEMENT

E           +             +
M      NAME(NEW) = NAME(FREE_Q),

C      REMOVE NEW ENTRY FROM FREE_Q

E           +                    +
M      NAME(FREE_Q) = NAME(FREE_Q NEXT),

C     PUT INFO INTO NEW ENTRY

M      NEW TIME = WHEN,
M      NEW ACTION = WHAT,
E                     +                    +
M      NAME(NEW AFFECTED_PROCESS) = NAME(PROCNAME),

C       NOW INSERT NEW ENTRY IN APPROPRIATE POINT OF ACTV QUEUE
C             EITHER BEFORE FIRST,
C             BETWEEN ENT A'D ENT NEXT FOR SOME ENT
C             OR AT END OF QUEUE

M,     IF NEW TIME < ACTV_Q TIME THEN
M         DO,
E                  +             +
M            NAME(NEW NEXT) = NAME(ACTV_Q),                          /* PUT FIRST*/
E                 +             +
M            NAME(ACTV_Q) = NAME(NEW),
M            RETURN,
M         END,
E          +           +
M      NAME(ENT) = NAME(ACTV_Q),                                     /*START AT TOP*/
E                       +
M      DO UNTIL NAME(ENT NEXT) = NAME(NULL),                         /* SEARCH Q*/
E            +            +
M         NAME(ENT) = NAME(ACTV_Q),                                  /*START AT TOP*/
M         IF ENT NEXT TIME > NEW TIME THEN
M            DO,
E                      +                 +
M               NAME(NEW NEXT) = NAME(ENT NEXT),
E                      +              +
M               NAME(ENT NEXT) = NAME(NEW),
M               RETURN,                                              /* NEW ENTRY INSERTED */
M            END,
E             +           +
M         NAME(ENT) = NAME(ENT NEXT),                                /* TRY NEXT ENTRY*/
M      END,

C    AT THIS POINT, THE WHOLE Q WAS SEARCHED UNSUCCESSFULLY,
C      SO ADD NEW TO THE END

E              +             +
M      NAME(ENT NEXT) = NAME(NEW),
E               +
M      NAME(NEW NEXT) = NULL,
M    CLOSE ENQUEUE,
```

This procedure can insert an entry in the middle of the queue without physically moving subsequent entries down, since the sequence information is encoded in the links (name variables) rather than the position in memory (the copy number) After

CALL ENQUEUE(10, 1, NULL),

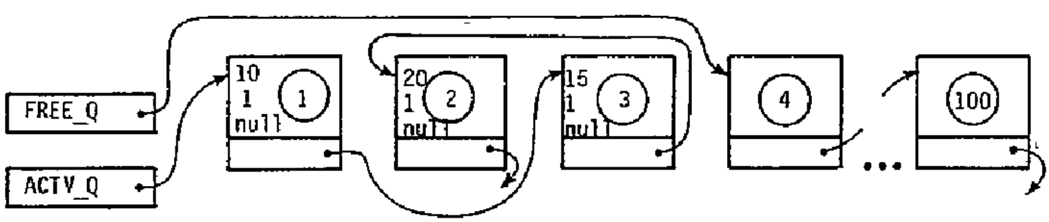
is executed, the queue looks like



If the next calls are

CALL ENQUEUE(20, 1, NULL),
CALL ENQUEUE(15, 1, NULL),

the queue looks like



Now, ACTV_Q is TQS(1,),
ACTV_Q NEXT is TQS(3,), and
ACTV_Q.NEXT NEXT is TQS(2,)

Thus when viewed as a list structure, the elements of ACTV_Q are sorted by increasing TIME, even though

TQ.TIME(2,) > TQ.TIMES(3,)

This queue could be used in implementing the HAL/S real time statements The code below illustrates how the timer queue might be used The CALL SET_CLOCK and WAIT FOR event statements are intended to load the value ACTV_Q TIME into an interval timer, and wait for the interrupt This would have to be done via assembly language or %-macros "Percent" macros are implementation-dependent They allow a *pre-defined* sequence of machine instructions to be inserted in a HAL/S program More detail is given in each Users Manual

```
M   INT_HANDLER
M   TASK,
M      DECLARE CLOCK_INTERRUPT EVENT,
M      DECLARE TEMP NAME TQE-STRUCTURE;
M      DO WHILE TRUE;
M         CALL SET_CLOCK(ACTV_Q TIME) ASSIGN(CLOCK_INTERRUPT);
M         WAIT FOR CLOCK_INTERRUPT;
M         DO CASE ACTV_Q.ACTION,
E                                     +
M            CALL RECYCLE(ACTV_Q AFFECTED_PROCESS),
E                                         +
M            CALL CANCEL_PROC(ACTV_Q.AFFECTED_PROCESS),
E                                   +
M            CALL READY(ACTV_Q AFFECTED_PROCESS),
E                                         +
M            CALL SCHEDULE_AT(ACTV_Q AFFECTED_PROCESS);
M            ;                                                        /* ETC */
M         END;

C     NOW REMOVE TQE FROM ACTIVE CHAIN

E              +              +
M         NAME(TEMP) = NAME(ACTV_Q);
E               +                    +
M         NAME(ACTV_Q) = NAME(ACTV_Q.NEXT);
E                   +              +
M         NAME(TEMP NEXT) = NAME(FREE_Q),
E               +              +
M         NAME(FREE_Q) = NAME(TEMP);
M      END,
M   CLOSE,
M   RECYCLE
M   PROCEDURE(X);
M      DECLARE X PROCESS_CONTROL-STRUCTURE,
M   CLOSE,
M   CANCEL_PROC
M   PROCEDURE(X),
M      DECLARE X PPOCESS_CONTROL-STRUCTURE;
```

With the process INT_HANDLER running, and appropriate routines to recycle, cancel, and otherwise change process states, ENQUEUE could be called as a result of several HAL/S statements "WAIT 5," executed by some process X might be translated to

CALL ENQUEUE(RUNTIME + 5, 3, NAME(X)),
CALL STALL(NAME(X)), /*enter wait state*/

Here we are assuming that X is a process_control-structure Such a structure might consist of

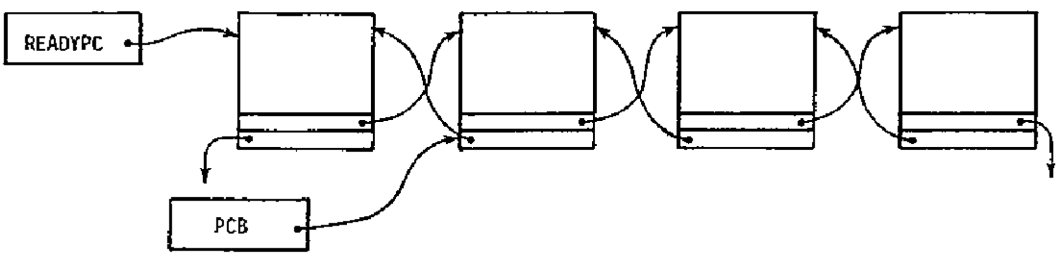
```
M   STRUCTURE PROCESS_CONTROL:
M      1 SAVE_AREA RIGID,
M         2 FIXED_REGS ARRAY(16) BIT(32),
M         2 FLOAT_REGS ARRAY(8) SCALAR DOUBLE,
M         2 OTHER BIT(32),
M      1 PRIORITIE INTEGER,
M      1 STATUS INTEGER,
M      1 NEXT NAME PROCESS_CONTROL-STRUCTURE,
M      1 LAST NAME PROCESS_CONTROL-STRUCTURE,
```

when the node, SAVE_AREA is machine dependent This is a double linked list Each entry has both forward and backward pointers To see how this is useful, suppose that there are three queues containing process control blocks (PCBs) FREEPC will be the anchor (simple name variable pointing at the first element of) of a queue of unused PCBs, READYPC will be the anchor of a queue of PCBs representing *ready* processes, (sorted by priority), and STALLED will be a queue representing *blocked* processes (e g , those in the wait state) One of these queues is diagrammed on the next page All three have the same form The STALL routine that was called above might simply remove the indicated process from the READYPC queue and add it to the STALLED queue The argument to STALL is the address of the PCB to be removed from the READYPC It could be written as

```
M      DECLARE READY_PC NAME PROCESS_CONTROL-STRUCTURE,
M      DECLARE STALLED NAME PROCESS_CONTROL-STRUCTURE,
M      DECLARE FREEPC NAME PROCESS_CONTROL-STRUCTURE,

C
C

M   STALL:
M   PROCEDURE ASSIGN(PCB);
M      DECLAPE PCB PROCESS_CONTROL-STRUCTURE,

C    REMOVE FROM READY QUEUE
```

```
                          :
E              +                 +
M    NAME(PCB LAST NEXT) = NAME(PCB NEXT),
E              +                 +
M    NAME(PCB.NEXT LAST) = NAME(PCB LAST);
C   ADD TO STALLED QUEUE AT THE BEGINNING
E           +          +
M    NAME(PCB NEXT) = NAME(FREEPC);
E         +         +
M    NAME(FREEPC) = NAME(PCB),
M  CLOSE STALL;
M  CLOSE LAST_EXAMPLE;
```

The reason a double linked list is needed is that STALL receives the address of a PCB in the middle of a chain.



To remove it, the links of both neighbors must be changed. A singly linked list would suffice if it was always searched starting from READYPC.

In this section, we have sketched portions of one possible implementation of the HAL/S real time statements. This design does not necessarily correspond to any actual operating system. The point of this section is to give a degree of familiarity with sophisticated uses of name variables, and to illustrate that large portions of "system programs" can be written in HAL/S.

This system presented is not at all complete. A routine is needed to make a process ready. It could be essentially the same as the ENQUEUE routine shown earlier. The routine that readies a cyclic process when the timer goes off should put a new entry in the queue for the next cycle. Also, some low-level control code is needed to dispatch the highest priority ready process. This process is always the one that corresponds to READYPC, since the ready queue is sorted, the top routine is always the one to receive control However, there is no HAL/S syntax for branching to a program or for loading/storing specific machine registers At some level, assembly language has to be used, though HAL/S does allow certain canned machine-instruction sequences to be generated via % macros These macros make machine dependencies highly visible in the listing If the %-macros defined for a particular implementation are not sufficient, assembly language comsubs can fill the gap

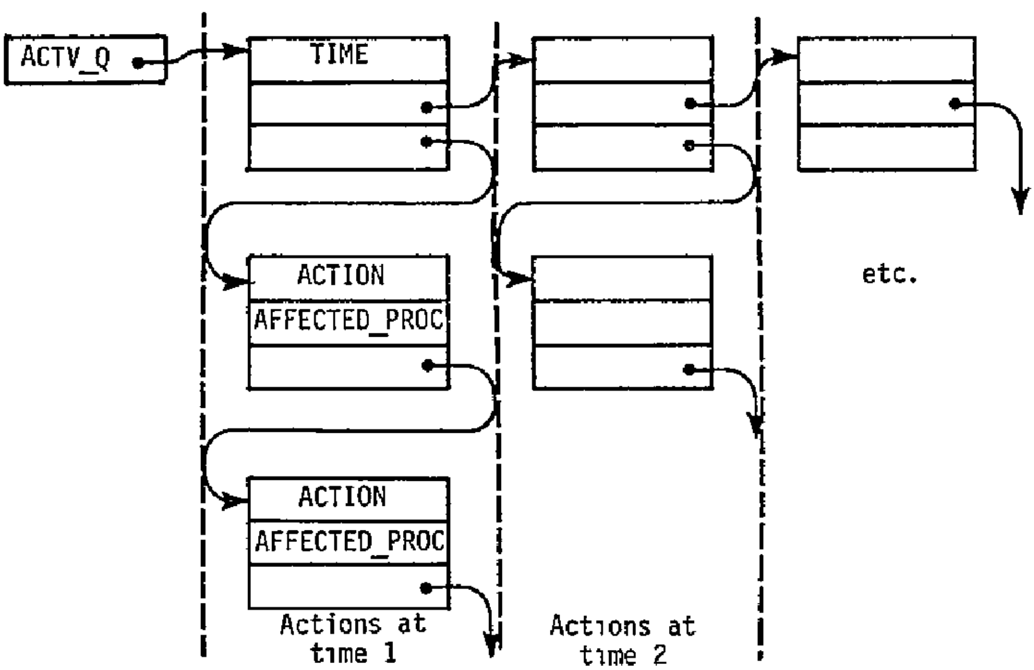
Name variables, percent macros, bit strings, EQUATE EXTERNAL*, and the ability to call assembly language routines all contribute to making HAL/S suitable for systems programming Use of these features in application programming is discouraged, nonetheless, some safety is provided by the type checking rules (as applied to name variables and bit strings) and other safeguards Even in the system-language portion of HAL/S, many forms of bad programming practice are precluded by compiler restrictions These features are designed so that reliable, readable and efficient programming is still encouraged even though it cannot be as thoroughly enforced when the system programming features are used

Exercises

13 3A Declare and initialize a structure, CIRCLE, such that the following statement is true

NAME(CIRCLE NEXT) = NAME(CIRCLE)

13 3B Change the declaration of the timer queue so that each element (TQE) is the head of an arbitrary-length list of action-affected process pairs all to be done at the same time, as illustrated



*See appropriate User's Manual for details

Change the ENQUEUE routine to either add the new element to the end of an existing list, if there is already one, or more actions at that time, or insert a new list consisting of a header and the new item

13 3C As written in the text, the procedure STALL may fail with some inputs When will this happen? Modify the procedure to remove this problem

End of Chapter Problems

13A Write a procedure which will insert a PROCESS_CONTROL-STRUCTURE in the READY_PC queue (both defined as in Section 13 3) after all entries having an equal or higher PRIORITIE and before all entries that are lower Remember to maintain both forward and backward links

13B Write a program which will read in two hexadecimal numbers (of up to six digits) separated by either a plus or minus sign, and print their sum or difference in both decimal and hexadecimal

Appendix A

ARITHMETIC FUNCTIONS

- Arguments may be integer or scalar
- The data type of the result matches the argument type unless otherwise noted
- Arrayed arguments generate multiple invocations of a function, one for each element in the array When two or more arguments are arrayed, their arrayness must match *

Name <Arguments(s)>	Comments
ABS(X)	Absolute value |X|
CEILING(X)	Smallest integer ≥ X CEILING(–3 4) returns –3
DIV(X,Y)	Integer division X/Y, where scalar arguments are rounded to integers This construct is the only way to do integer division in HAL DIV(5,2) returns 2 Note Where X, Y, Z are integers X = 5, Y = 2 The statement Z = X/Y results in two integer to scalar conversions and a scalar divide Finally, the result is converted to an integer type In this case Z = X/Y sets Z to 3
FLOOR(X)	Largest integer ≤ X FLOOR(–3 4) returns –4
MIDVAL(X,Y,Z)	The value of the argument which is algebraically between the other two If two or more arguments have the same value, that value is returned MIDVAL(– 4, – 6, 3 5) returns – 4
MOD(X,Y)	X MOD Y (modulus) The result is scalar unless both arguments are integers MOD(5,3) returns 2 MOD(5,–3) returns 2 MOD(–5,3) returns 1 MOD(–5,–3) returns 1 MOD(–5,2 1) returns 1 3

*For a discussion of arrayness, see Section 6 2

ARITHMETIC FUNCTIONS (CONT'D)	
Name <Argument(s)>	**Comments**
ODD(X)	Result is BOOLEAN True if X is odd, false if X is even IF(ODD(X)) THEN .. Note Scalar arguments are rounded to integer
REMAINDER(X,Y)	Signed remainder of integer division X/Y REMAINDER(–5,3) returns –2 REMAINDER(5,–3) returns 2. REMAINER(–5,–3) returns –2. Note Scalar arguments are rounded to integers
ROUND(X)	Nearest integral value to X, essentially the same as HAL scalar to integer conversion
SIGN(X)	Returns an integer +1 if $X \geq 0$, –1 if $X < 0$
SIGNUM(X)	Returns an integer +1 if $X > 0$, 0 if $X = 0$, –1 if $X < 0$. DO CASE(SIGNUM(X)+2)
TRUNCATE(X)	Strip off fractional part of the scalar (X) TRUNCATE(–3 4) returns –3 TRUNCATE(7 8) returns 7

ALGEBRAIC FUNCTIONS	
• Arguments may be integer or scalar types – conversion to scalar occurs with integer arguments • Result type is always scalar • Arrayed arguments cause multiple invocations of the function, one per each array element • Angular values are supplied or delivered in radians * • Arguments that are outside the domain specified in the comments result in HAL/S runtime errors, (see Chapter 10)	
Name <Argument(s)>	Comments
ARCCOS(X)	$\lvert X \rvert \leqslant 1$
ARCCOSH(X)	$X > 1$
ARCSIN(X)	$\lvert X \rvert \leqslant 1$
ARCSINH(X)	
ARCTAN2(X,Y)	Returns $\theta = \tan^{-1}(X/Y)$ where the proper quadrant for $-\pi < \theta < \pi$ is determined from the signs of X and Y Proper quadrant results if $\left.\begin{array}{l} X = K \sin\theta \\ Y = K \cos\theta \end{array}\right\} K > 0$
ARCTAN(X)	Principle value only, see above
ARCTANH(X)	$\lvert X \rvert < 1$
COS(X)	
COSH(X)	
EXP(X)	e^X
LOG(X)	$\log_e X$, $X > 0$.
SIN(X)	

*One radian equals 57 2957795131 degrees, so that
π radians equals 180 degrees,
$\pi/2$ radians equals 90 degrees

ALGEBRAIC FUNCTIONS (CONT'D)	
Name <Argument(s)>	Comments
SINH(X)	
SQRT(X)	$\sqrt{X}$, $X \geqslant 0$.
TAN(X)	
TANH(X)	

ARRAY FUNCTIONS	
• Arguments may be single or multi-dimensional arrays of scalars or integers • The type of the result matches the type of the argument and is unarrayed	
Name <Argument(s)>	**Comments**
MAX(X)	Maximum of all elements of X
MIN(X)	Minimum of all elements of X
PROD(X)	Product of all elements of X
SUM(X)	Sum of all elements of X

BIT FUNCTIONS

- HAL/S provides AND, OR, and NOT operators for bit operands XOR (exclusive OR) is available as a built-in function

Name <Argument(s)>	Result Type	Comments
XOR(X,Y)	BIT	Exclusive OR, where X and Y are bit strings The length of the result is the length of the longer argument The shorter argument is padded on the left with zeros

CHARACTER FUNCTIONS		
• The first argument in each of the functions below is a character string If a scalar or integer is specified where a character string is expected, a conversion to character type is performed		
Name <Argument(s)>	**Result Type**	**Comments**
INDEX(C1,C2)	Integer	C2 is a character string If string C2 is contained within string C1, an index which is the location of the first character of C2 in C1 is returned, otherwise, zero is returned INDEX('CHARACTER', 'ACTER') returns 5. INDEX('ALPHA', 'BETA') returns 0
LENGTH(C)	Integer	Returns the current length of character string C
LJUST(C1,n)	Character	n is integer type – the string C1 is expanded to length n by padding on the right with blanks If n is less than the current length of C1, an error is signaled and C1 is truncated to length n
RJUST(C1,n)	Character	n is integer type – the string C1 is expanded to length n by padding on the left with blanks If n is less than the current length of C1, an error is signaled and C1 is truncated to length n
TRIM(C1)	Character	Leading and trailing blanks are stripped from C1

MISCELLANEOUS FUNCTIONS

- Arguments are as indicated, if none are indicated the function has no arguments
- Result type is as indicated

Name <Argument(s)>	Result Type	Comments
CLOCKTIME	Scalar	Elapsed time since midnight (format is implementation dependent) See Chapter 12
DATE	Integer	Returns date (implementation dependent format)
ERRGRP	Integer	Returns group number of last error detected, or zero if no error was detected See Chapter 10
ERRNUM	Integer	Returns number of last error detected, or zero if no error was detected See Chapter 10.
NEXTIME (<label>)	Scalar	<label> is the name of a program or task The value returned is determined as follows a) If the specified process was scheduled with the REPEAT EVERY option, and has begun at least one cycle of execution, then the value is the time the next cycle will begin b) If the specified process was scheduled with the IN or AT phrase, and has not yet begun execution, then the value is the time it will begin execution c) Otherwise, the value is equal to the current time (RUNTIME function)
PRIO	Integer	Returns priority of process calling function
RANDOM	Scalar	Returns pseudo-random number from rectangular distribution over range 0-1 *

*Note that for any particular HAL program complex which contains references to random and/or randomg, the same set of "random" numbers will be generated in each execution

MISCELLANEOUS FUNCTIONS (CONT'D)		
Name <Argument(s)>	Result Type	Comments
RANDOMG	Scalar	Returns pseudo random number from Gaussian distribution with a mean of zero, variance of one *
RUNTIME	Scalar	Time since the software began executing (implementation dependent format) See Chapter 12
SHL(X,Y)	Integer	X shifted left Y bit positions X and Y may be scalar or integer, but scalars are converted to integer before shifting This is an arithmetic (signed) shift SHL(–2,2) returns –8
SHR(X,Y)	Integer	X shifts right Y bit positions As above, this is an arithmetic shift SHR(–4,2) returns –1
SIZE(X)	Integer	One of the following must hold – X is an unsubscripted arrayed variable with a one-dimensional array specification – function returns length of array. – X is an unsubscripted major structure with a multiple copy specification – function returns number of copies – X is an unsubscripted structure terminal with a one-dimensional array specification – function returns length of array Result is of integer type

*Note that for any particular HAL program complex which contains references to random and/or randomg, the same set of "random" numbers will be generated in each execution

Appendix B

Although the main body of this manual has avoided references to specific compilers, there is considerable similarity in the compilers now available In this appendix we will describe additional software development support which is typically provided

The HAL/S compiler is not simply a language translator All current implementations include features not usually found in other common compilers, such as PL/1, FORTRAN, etc These include special processing and annotation of the listings, facilities for restricting usage of variables or language features, and additional outputs for post-compilation tools

In addition to annotating identifiers and indenting as described in the text, the compiler adds several types of summary information to the listing At the end of each procedure or function block, that block's interfaces are listed The information presented includes lists of global variables referenced or modified, external procedures called, event variables modified, compool REPLACE macros used, and so forth At the end of the listing a table of identifiers is printed, including the data type and a list of all statements which use the identifier Some compilers produce a listing of annotated assembly language which corresponds to the machine code actually generated This aids in debugging on flight hardware, although more sophisticated debugging supports is also provided

Two facilities provide for the establishment of managerial control over HAL/S usage ACCESS rights allow restrictions to be placed on the modification of selected variables or on the usage of blocks Since this can be done separately for each compilation unit, ACCESS rights provide managers with an important tool for controlling the interfaces between modules Another device is the SUBSETing capability, which provides the ability to restrict the usage of a user selected subset of HAL/S language features or built-in functions This mechanism does not affect the code generated but merely flags by a warning message on the primary listing those statements violating the SUBSET

The efficiency and reliability of program complexes can be improved by use of a special-purpose link editor or binder These programs (e g , HALLINK) can reduce storage requirements by generating the call tree beneath each program or task and allocating a temporary storage area (or stack) just large enough for the longest limb of the tree If a compiler system includes an appropriate link editor, it may also add to software reliability While the various HAL/S modules are being bound together, they can also be checked for consistency The template generation system (chapter 11) passes information to the link step that, for instance, allows verification that every program used the same compool template

Another output of each compilation is a Simulation Data File or SDF This is a random access data base containing attribute and cross reference information for variables and code blocks Data concerning executable statements is also included, as well as global statistics found in the primary listing It is this large database that allows for many post-compilation analysis tools, ranging from execution-time debuggers to HALSTAT, a statistics and analysis package

Programmers have many modes of execution available to them in most implementations of HAL/S Even running stand-alone (on a host computer) one can obtain detailed error diagnostics related directly to the HAL/S source by statement number and block name, and optionally obtain an end of run formatted dump of all variables And if a program terminates abnormally, a full traceback, showing the flow of control from block to block, will be given Another package allows one to request dumps and traces of variables while running in a batch environment This package can also provide a detailed log of real-time transactions, showing the transitions from process to process Moreover, certain implementations provide the capability of "functional simulation," or FSIM, of another target computer In this usage, the amount of memory used is approximated by allocating variables in the same fashion as on the target machine Also, the extent of CPU utilization is estimated for the target machine with a running accumulation of time maintained automatically The FSIM facility is very useful in cases where the target machine is not commonly available or is difficult to use One very valuable feature available under FSIM is the "profile" capability. A listing can be generated which shows the number of times each HAL/S *statement* in the program complex was executed The estimated total execution time for each statement, and other statistics, allow the efficiency programs written in HAL/S to be attacked at the point of greatest leverage

One host computer contains an interactive HAL/S debugger This program uses information from the simulation data files as well as "hooks" inserted in the machine code to allow debugging at the HAL/S level (i e , without knowing *any* details of the underlying computer) Breakpoints can be set by statement number or label For instance, "AT LOOP + 3," sets a breakpoint three *HAL/S statements* after the label "Loop". Variables can be inspected and modified by their symbolic names, all values are entered or presented in the standard external format Data aggregates may be subscripted or printed in entirety Since the SDFs contain full type information, there is no need to debug in hexadecimal or octal, or to continually specify display formats Since HAL/S programs reference variables via scoping rules, this debugger provides a SCOPE command This command has a block name as its argument References to variables in subsequent commands are interpreted as they would be in the named block A SCOPE command is automatically performed when a breakpoint is reached Thus commands at a breakpoint can reference any variable that is visible from the block in which the breakpoint was hit The SDFs contain sufficient information to allow similar capabilities in a "cross-debugger" to test actual flight code

The large amount of data contained in the compiler's outputs, especially the SDF's and the object modules, permits the development of many post-compilation analysis programs Perhaps the best known of these is the HALSTAT program, which is used to accumulate global data about a program complex HALSTAT performs three major functions Verifying the consistency of SDF's, printing statistics for each module, and giving a global dictionary of variables SDF's are consistent if all variables shared by processes are in agreement with respect to such factors as data type, size, location, and so on Variables are also checked on a global basis to insure that none are referenced that have not ever been assigned, if this situation occurs a warning message will be given Multitudinous statistics are printed for each HAL module in the program complex, giving the name of the module and the date of compilation, size statistics, and the modules' pattern both in terms of HAL/S blocks incorporated and location of code sections The global-symbol directory (GSD) portion of HALSTAT is a listing of every variable used in every module of the program complex, including both compool and local variables. It shows not only variable attributes and locations, but also the cross reference data for each variable *across all modules in which it is used.* The cross reference shows both the HAL/S statements, by number, where an item is used, and also the way in which it is used, e.g., REFERENCED, ASSIGNED, SUBSCRIPT, etc.

Additional programs have been developed to meet the needs of specific installations. One program provides a complete disassembly listing of a HAL/S load module, which shows clearly the relationships between the machine code instructions and the HAL/S source. Since the typical program complex's load module incorporates code from both HAL/S modules and assembly language modules (from the runtime library), a list showing both of these is essential to review the integrated system. Another program provides the above disassembly capability but limits it to user-specified machine instructions, a facility that is very useful in assessing the impact of instructions that are not correctly implemented in a machine's hardware, or in determining the extent and nature of operating system interfaces. There is also a program which produces a list of all locations deemed to be invariant. After executing the load module for a period of time, one can dump the contents of memory and see if these "never-changing" memory locations have indeed changed, which would indicate a problem in the load module. Another program is used to compile, based upon programmer specification of the data items desired, a list of all parameters that will be patched. This list includes detailed information about each variable, such as type, size, and location, to allow it to be modified in the correct fashion.

As more installations use HAL/S on an ever-growing number of target machines, the amount and diversity of the support software is certain to grow. The capabilities described here may and may not be present in a particular system, but like the HAL/S compiler itself, these utilities are written in a high order language, and as machine-independently as possible. The functional simulation and post-compilation analysis tools have proved so valuable in the Space Shuttle program that they may eventually become required components of any HAL/S compiler system.

Appendix C: Answers to Exercises

Solutions

2 1A

a) valid, identifier

b) valid, reserved word

c) invalid

d) valid, literal

e) valid, identifier

f) invalid

g) valid, identifier

h) valid, reserved word

i) invalid

j) valid, reserved word

k) invalid

l) valid, identifier

m) valid, literal

2 2A

a) A X+B Y+C Z

b) (A+B)/(C + D)/(E+F)

c) 2**(N–1)/(2**N–1)

d) X**3–3 X**2+3 X–1

e) (X–1)**3

f) 10**X**Y

g) (10**X)**Y

h) ((V W)/(V V)); V

2 2B

a) '*' is not the multiplication operator in HAL/S

Correct expression M X+B

b) Incorrect operator precedence

Correct expression 2 (X+1)

c) Multiplication is represented by a blank between two operands

Correct expression X**(–2 5 N)

d) Two operators may not occur in succession

Correct expression C**(–5)

e) Order of evaluation is normally from left to right

Correct expression A C/(B D) or (A C)/(B D)

2 3A

```
DECLARE SCALAR INITIAL(1) X_DELTA, Y_DELTA,
DECLARE TIME_DELTA SCALAR CONSTANT(1),
DECLARE DELAY_FACTOR SCALAR CONSTANT( 5),
DECLARE SCALAR, TEMP1, TEMP2, TEMP3,
DECLARE COUNT INTEGER INITIAL(1),
DECLARE POINT_A VECTOR,
DECLARE ORIGIN VECTOR CONSTANT(0,0,0),
DECLARE TRANSFORM MATRIX INITIAL(1,0,0,0,1,0,0,0,1),
```

2A

```
M   ROOTS:
M   PROGRAM,
M      DECLARE SCALAR,
M                 A, B, C, ROOT1, ROOT2,
M      READ(5) A, B, C;
E                     2          0.5
M      ROOT1 = (-B + (B  - 4 A C)    ) / 2 A;
E                     2          0.5
M      ROOT2 = (-B - (B  - 4 A C)    ) / 2 A;
M      WRITE(6) ROOT1, ROOT2;
M   CLOSE ROOTS;
```

2B

```
M   BOUNCE-
M   PROGRAM,
M      DECLARE SCALAR,
M                 HEIGHT,
M                 TIME INITIAL(0),
M      HEIGHT = 110,
E                            1/2
M      TIME = (2 HEIGHT / 32)    ,                                    /* BOUNCE 1 */
M      HEIGHT =  35 HEIGHT,
E                                      1/2
M      TIME = TIME + 2 (2 HEIGHT / 32)    ;                           /* BOUNCE 2 */
M      HEIGHT =  35 HEIGHT,
E                                      1/2
M      TIME = TIME + 2 (2 HEIGHT / 32)    ;                           /* BOUNCE 3*/
M      WRITE(6) TIME,
M      WRITE(6) 4 TIME,
M   CLOSE BOUNCE;
```

2C

```
M   EX2C·
M   PROGRAM,
M      DECLARE MASS_OF_EARTH SCALAR CONSTANT(5 983E27),
M      DECLARE PI SCALAR CONSTANT(3 14159265),
M      DECLARE RADIUS SCALAR INITIAL(4000 160934 4),
M      DECLARE PERIOD SCALAR;
E                     2       3                               0.5
M      PERIOD = ((4 PI  RADIUS ) / (MASS_OF_EARTH 6.67E-8))     ;
M      WRITE(6) PERIOD,
M   CLOSE EX2C,
```

2D

```
M   SOLUTION:
M   PROGRAM,
M      DECLARE SCALAR,
M                    A, B, C, D, E, F, X, Y,
M      READ(5) A, B, C, D, E, F,
M      X = (E D - B F) / (A D - B C),
M      Y = (A F - E C) / (A D - B C),
M      WRITE(6) X, Y;
M   CLOSE SOLUTION,
```

Solutions

3.1A

a) Integer, value is 1.

b) Matrix (3 by 3), value is $\begin{bmatrix} 1 & 2 & 4 \\ 2 & 3 & 5 \\ 2 & 3 & 6 \end{bmatrix}$.

c) 2-vector, value is $\begin{bmatrix} 1 \\ 6 \end{bmatrix}$

3.1B

```
M | TRAN_MUL:
M | PROGRAM,
M |    DECLARE M MATRIX CONSTANT(9, 8, 7, 6, 5, 4, 3, 2, 1),
E |            *         *
M |    WRITE(6) M TRANSPOSE(M),
M | CLOSE TRAN_MUL;
```

3.1C

a) (1+COS (2 X))/2

b) ARCTAN(Y/X)

c) M (R Z_DOT – Z R_DOT) SIN(PHI) – M R Z PHI_DOT COS(PHI)

d) ARCCOS((M/R–M A/N)/SQRT(2 M E+M**2 A**2/N**2))

e) LOG(TAN(X/2+PI/4))

3.2A

a) 1, 7, 0

b) $\begin{bmatrix} 13 \\ 14 \\ 15 \end{bmatrix}$ $\begin{bmatrix} 5 \\ 7 \end{bmatrix}$ $\begin{bmatrix} 3 & 0 & -3 & -6 \\ 2 & -1 & -4 & -7 \end{bmatrix}$

c) DECLARE V1 VECTOR(6) INITIAL(0,1,2,3,4,5),
DECLARE V2 VECTOR(6) INITIAL(10,11,12,13,14,15),
DECLARE M22 MATRIX(2,2) INITIAL(5,6,7,8),
DECLARE M35 MATRIX(3,5) INITIAL(7,4,1,–2,–5,6,3,0,–3,–6,5,2,–1,–4,
–7),

3 2B

This is an example of how over-specifying a program may lead to inefficiency Two answers are given here, the first follows the statement of the problem literally, while the second produces the same result in a different way

```
M | COMP_DOT:
M | PROGRAM,
M |    DECLARE VECTOR,
M |                ORIG_VEC INITIAL(1, 2, 3),
M |                RESULT_X;
M |    DECLARE ORIG_MAT MATRIX INITIAL(1, 2, 3, 4, 5, 6, 7, 8, 9),
E |                    -              -
M |    RESULT_X  = OPIG_VEC . ORIG_MAT   ;
S |            1                    *,1
  |
E |                    -              -
M |    RESULT_X  = ORIG_VEC   ORIG_MAT   ;
S |            2                    *,2
  |
E |                    -              -
M |    RESULT_X  = ORIG_VEC . ORIG_MAT   ;
S |            3                    *,3
  |
E |                       -
M |    WRITE(6) RESULT_X;
M | CLOSE COMP_DOT;
```

```
M | COMP_DOT
M | PROGRAM,
M |    DECLARE VECTOP,
M |                ORIG_VEC INITIAL(1, 2, 3),
M |                RESULT_X,
M |    DECLARE ORIG_MAT MATRIX INITIAL(1, 2, 3, 4, 5, 6, 7, 8, 9);
E |       -          -         *
M |    RESULT_X = ORIG_VEC ORIG_MAT;
E |                  -
M |    WRITE(6) RESULT_X;
M | CLOSE COMP_DOT,
```

3 2C

WRITE(6) V41 will output the vector $\begin{bmatrix} 21 \\ 22 \\ 23 \\ 24 \end{bmatrix}$

The first WRITE(6) M22 will output the matrix $\begin{bmatrix} -5 & -6 \\ -8 & -9 \end{bmatrix}$

WRITE(6) M33 will output the matrix $\begin{bmatrix} 0 & 1 & 2 \\ 8 & 8 & 9 \\ 11 & 12 & 13 \end{bmatrix}$

The second WRITE(6) M22 will output the matrix $\begin{bmatrix} 0 & 1 \\ 2 & 8 \end{bmatrix}$

3 5A

i)	+, −, <>, /, **	results scalar
ii)	+, −, <>, /, **	results scalar
iii)	+, −, <>, /, **	results scalar
iv)	+, −, <> /, **	results integer, results scalar
v)	+, −, * <>	results vector, result matrix, result scalar
vi)	<>	result matrix
vii)	<>, /	results vector.
viii)	<>	result vector
ix)	+, −, <>	results matrix
x)	<>, /, **	results matrix

3A

```
M   ANGLES:
M   PROGRAM,
M      DECLARE VECTOR,
M                V1, V2;
E            -   -
M      READ(5) V1, V2,
E                       -     -            -            -
M      WRITE(6) ARCCOS((V1 . V2) / (ABSVAL(V1) ABSVAL(V2)));
M   CLOSE ANGLES;
```

3B

```
M | TRANS
M | PROGRAM;
M |    DECLARE SCALAR,
M |                ALPHA, X1, X2, Y1, Y2,
M |                PI CONSTANT(3.1415),
M |    READ(5) X1, Y1,
M |    ALPHA = 17 PI / 180,
M |    X2 = (X1 - 54000) COS(ALPHA) + (Y1 - 118000) SIN(ALPHA),
M |    Y2 = -(X1 - 54000) SIN(ALPHA) + (Y1 - 118000) COS(ALPHA);
M |    WRITE(6) X2, Y2;
M | CLOSE TRANS;
```

3C

a) V4 = VECTOR$4(M$(2,2), M$(3,3), M$(4,4), M$(5,5)),

b) M22 = M$(2 TO 3, 8 TO 9),

c) M34 = M$(5 TO 7, 7 TO 10),

d) V10 = M$(9,*),

Solutions

4 1A

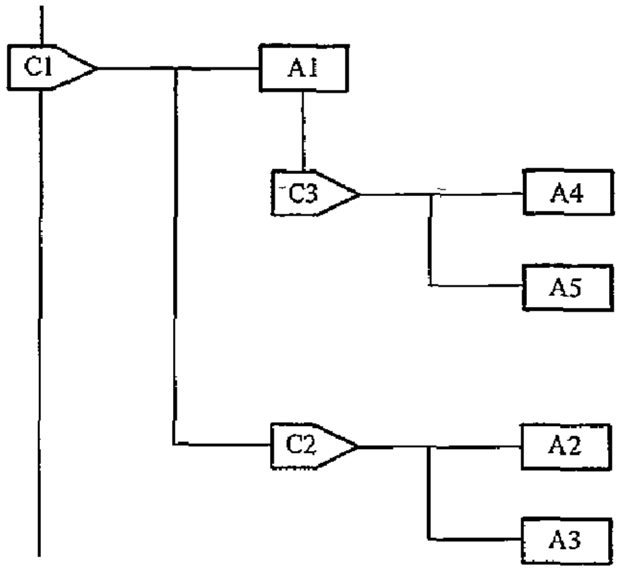
a) Compound conditions like 'A < B < C' are not recognized by HAL/S

b) In HAL/S, two 'ELSE' statements may not follow one another Section 4 2 introduces a way to avoid this problem

c) The expression following the 'NOT' operator must be parenthesized

4 1B

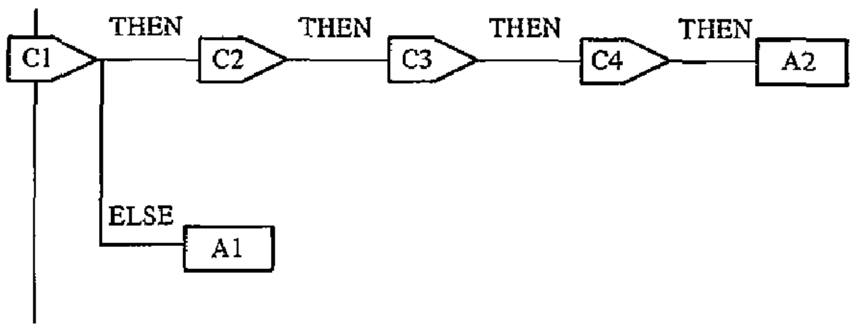
a)



b) Impossible the ELSE clause of C2 branches into the ELSE clause of C3

c) Impossible the THEN clause of C2 loops around, which would require traversing a line upward

d)



4 1C

```
IF W < L THEN SQ = 0,
ELSE IF W > L THEN SQ = 0,
   ELSE SQ = 1,
AREA = WL,
IF SQ = 0 THEN WRITE(6) 'NO SQUARE',
ELSE IF AREA < 4 THEN WRITE(6) 'SMALL SQUARE',
   ELSE WRITE(6) 'LARGE SQUARE',
```

4 1D

a) Not satisfied

b) Illegal

c) Satisfied

d) Satisfied

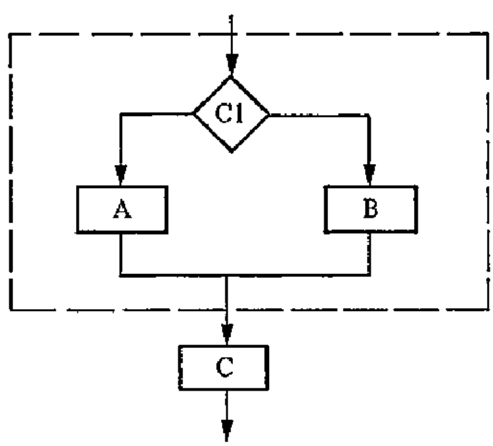
e) Illegal.

f) Not satisfied

g) (A > B) & (A < C)

h) (V ¬ = S) & ((C > = D)/(D = 4))

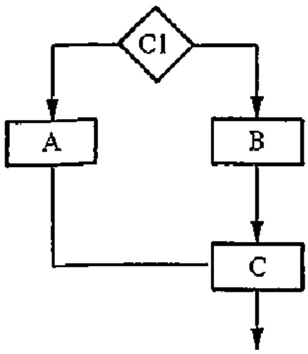
4 2A



The original code was over 300 statements, while the new code is about 160 statements

This change *can* be made in a valid HAL/S program group C is removed entirely from the IF statement, which now consists only of the section of the flow chart lying within the dotted rectangle

Note that this flowchart



does represent a shorter program than the original, though it cannot be translated into a valid HAL/S program, as this would require branching into the ELSE clause of the condition, which is not legal in HAL/S

4 2B

```
M  SOLUTION:
M  PROGRAM;
M     DECLARE SCALAR,
M                 A, B, C, D, E, F, X, Y;
M     READ(5) A, B, C, D, E, F;
M     IF (A D - B C) = 0 THEN
M        WRITE(6) 'NO SOLUTION EXISTS',
M     ELSE
M        DO;
M           X = (E D - B F) / (A D - B C);
M           Y = (A F - E C) / (A D - B C),
M           WRITE(6) X, Y,
M        END,
M  CLOSE SOLUTION,
```

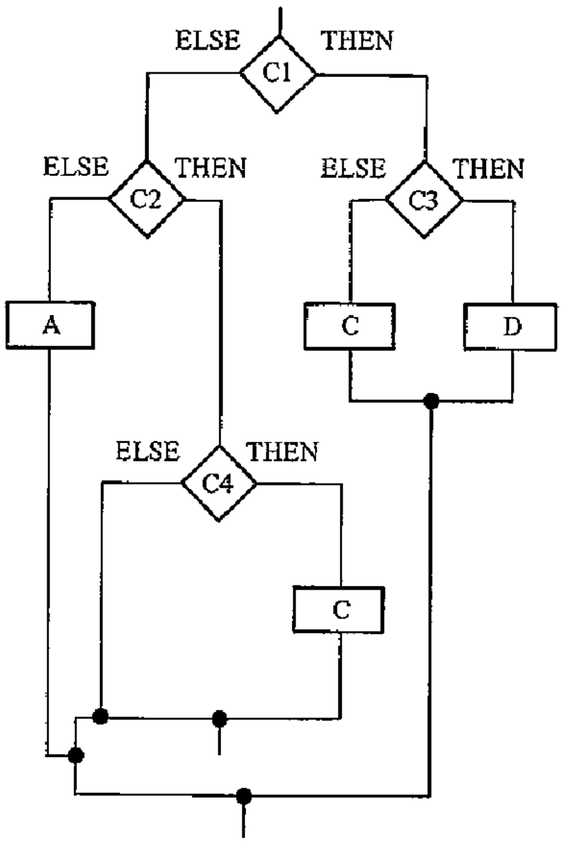
4 2C

```
IF Y < X THEN DO,
   IF Y < X - 1 THEN Y = Y + 1,
                ELSE Y = Y - 1,
END,
ELSE IF Y > X + 1 THEN X = X - 1,
ELSE X = X + 1,
```

4 2D

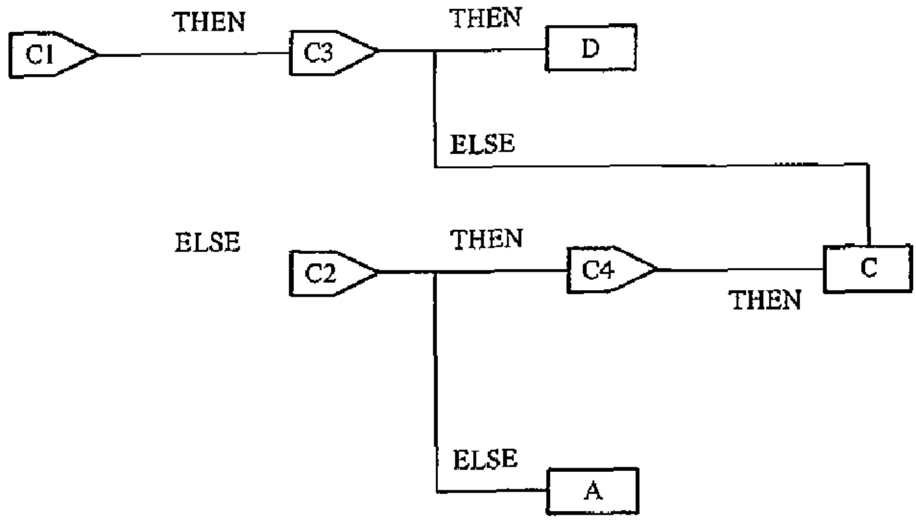
a) The line from C4 to C represents a branch into the ELSE clause of C3, which is illegal in HAL/S

b) The following flowchart removes the difficulty without making any change in the order of execution of any statements

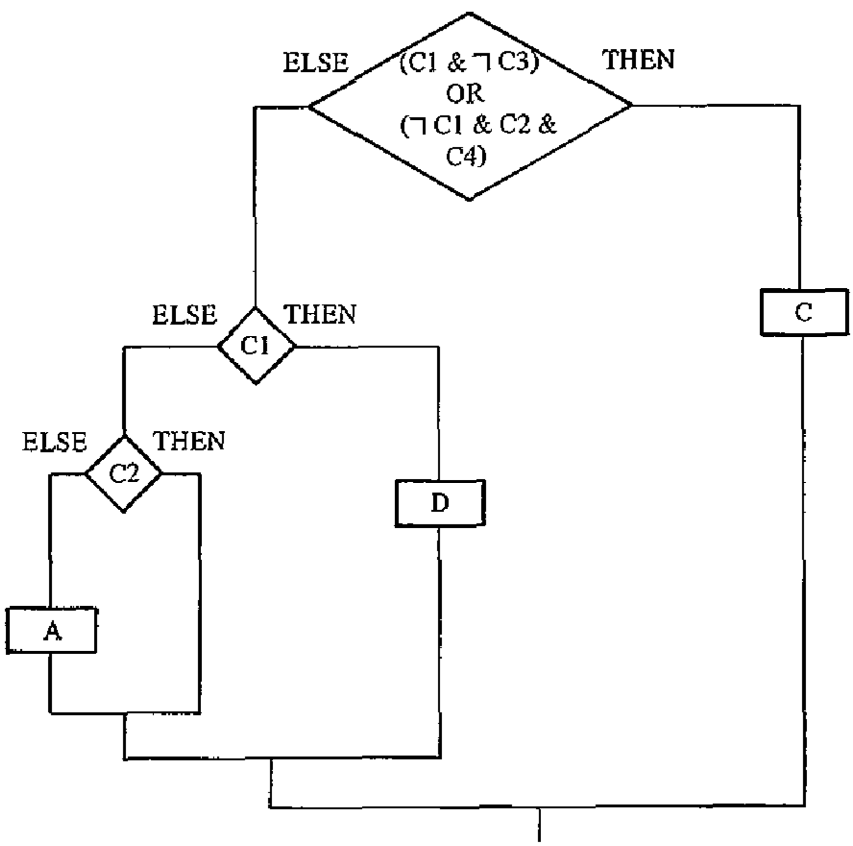



```
IF C1 THEN DO,
   IF C3 THEN D,
   ELSE C,
END,
ELSE IF C2 THEN C4 THEN C,
ELSE A,
```

c) If the flowchart had been structured, it would have been awkward even to draw lines from both C3 and C4 to C, and the fact that there was an illegal construct in the flowchart would have been obvious To illustrate



4 2E There are several possible solutions, one of which is given here



HAL/S code to implement the revised flowchart would be

```
IF (C1 AND (NOT C3)) OR (NOT C1 AND C2 AND C4) THEN C,
ELSE IF C1 THEN D,
ELSE IF ¬C2 THEN A,
```

4 3A

a) Relational expression

b) Boolean expression

c) Relational expression

d) Illegal

e) Illegal

f) Relational expression

g) Boolean expression

4 4A

```
DO CASE I + 1,
   ELSE SCRAMBLE = 3,
   SCRAMBLE = 4,
   SCRAMBLE = 0,
   SCRAMBLE = 5,
   SCRAMBLE = 3,
   SCRAMBLE = 1,
   SCRAMBLE = 2,
END,
```

Solutions

5 1A

Since the loop control variable is an integer, while the increment is the scalar value 1, on each iteration 1 will be added to 1, the resulting 1 1 will be rounded to 1, and the control variable will never change That is to say, the loop will never terminate, so the question is unanswerable

5 1B

```
DECLARE V VECTOR(5),
DECLARE NEG_PART INTEGER,
DO FOR NEG_PART = 5 TO 1 BY -1,
   IF V$NEG_PART < 0 THEN EXIT,
END,
```

Note that if no component of V is negative, NEG_PART will equal zero upon exit from the loop

5 1C

N is equal to 14 on exit from the loop, because in DO FOR I = 1 TO N BY 2, N is evaluated only once, upon entry to the loop, when its value is 9 The loop will therefore be executed five times, leaving N equal to 14

5 1D

a) The code assigns the value 2 to all the elements of A

b)

```
DO FOR X = 1 TO 5,
    DO FOR Y = 1 TO 5,
        A$(X,Y) = 2,
    END,
END,
```

5 2A

a) The program will write the values

```
 2 INITIAL_VALUE
 4 INITIAL_VALUE
 8 INITIAL_VALUE
16 INITIAL_VALUE
```

b)
```
DO FOR X = 1 TO 4,
   N = 2 N,
   WRITE(6) N,
END,
```

is one possibility,

```
DO FOR X = 1 TO 4,
   WRITE(6) 2**N,
END,
```

is another, and clearly there are many others

5 3A

```
DECLARE V VECTOR(5),
DECLARE NEG_PART INTEGER,
DO FOR NEG_PART = 1 TO 5 WHILE V$NEG_PART >= 0,
END,
IF NEG_PART 5 THEN NEG_PART = 0,
```

5 4A

If V$1 = 0, the code shown will not exit with NEG_PART = 1, as it should This occurs because the UNTIL clause will not be evaluated for the first time until 2 has been assigned to NEG_PART in the DO FOR loop

5A

For this solution, we take the original DELTA to be $\frac{\text{FINAL-INITIAL}}{5}$, and assume that INITIAL < FINAL

```
M | SIMPSON:
M | PROGRAM;
M |    DECLARE SCALAR,
M |                INITIAL_VALUE, FINAL_VALUE, OLD_APPROX, NEW_APPROX, POINT;
M |    DECLARE SCALAR,
M |                DELTA, EPSILON;
M |    OLD_APPROX, NEW_APPROX = 0;
M |    READ(5) INITIAL_VALUE, FINAL_VALUE, EPSILON;
M |    DELTA = (FINAL_VALUE - INITIAL_VALUE) / 5;
M |    DO UNTIL (NEW_APPROX - OLD_APPROX) < EPSILON;
M |       OLD_APPROX = NEW_APPROX;
M |       NEW_APPROX = SQRT(INITIAL_VALUE) + SQRT(FINAL_VALUE);
M |       DO FOR POINT = INITIAL_VALUE + DELTA TO FINAL_VALUE - (DELTA / 2) BY DELTA;
M |          NEW_APPROX = NEW_APPROX + 2 SQRT(POINT);
M |       END;
M |       NEW_APPROX = NEW_APPROX DELTA / 2;
M |       DELTA = DELTA / 2;
M |    END;
M |    WRITE(6) NEW_APPROX;
M | CLOSE SIMPSON;
```

5B

a) This program, admitted an inefficient one, will print all prime numbers from 3 through 499

b) A solution that does not change the computations performed is

```
M | BETTER:
M | PROGRAM;
M |    DECLARE INTEGER,
M |                NUMBER, DIVIDER;
M |    DO FOR NUMBER = 3 TO 499;
M |       DO FOR DIVIDER = 2 TO NUMBER - 1;
M |          IF MOD(NUMBER, DIVIDER) = 0 THEN
M |             EXIT;
M |       END;
M |       IF DIVIDER = NUMBER THEN
M |          WRITE(6) NUMBER;
M |    END;
M | CLOSE BETTER;
```

Solutions

6 1A

a) Illegal X is set to 3, but a variable with the INITIAL attribute is not considered to be computable at compile time, so the declaration of LIST_ONE is erroneous

b) Legal LIST_ONE is an array of 4 scalars, value (2, 2, 2, 2) LIST_TWO is an array of 4 integers, values unknown

c) Legal LIST_THREE is an array of 18 scalars, value
(1, 1, 1, 1, 1, 1, 1, 1, 1, 1,?,?,?,?,?,?,?,?,)

d) Legal LIST_FOUR is a 9 by 3 array of 27 scalars, value

$$\begin{pmatrix} .1 & 1 & 1 & .2 & 2 & 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & ? & ? & ? & ? & ? & ? \\ ? & ? & ? & ? & ? & ? & ? & ? & ? \end{pmatrix}$$

e) Illegal The ARRAY specification must *precede* the type specification

6 1B

a)

```
M | EXERCISE_2.
M | PROGRAM;
M |    DECLARE M MATRIX(5, 5);
M |    DECLARE TIME APRAY(100) SCALAR INITIAL(0);
M |    DECLARE SCALAR INITIAL(0),
M |                TMIN, THAX, TMEAN, SUM_OF_SQUARES, STAN_DEV;
M |    DECLARE INTEGER,
M |                I, J, K,
M |    DO FOR I = 1 TO 100,
M |       DO FOR J = 1 TO 5,
M |          DO FOR K = 1 TO 5,
M |             M    = RANDOM;
S |              J,K
  |
M |          END,
M |       END,
M |       TIME  = RUNTIME,
S |           I
  |
E |       *   *-1
M |       M = M  ;
M |       TIME  = RUNTIME - TIME ;
S |           I                  I
  |
M |    END;
  |
C |  NOW PROCESS THE HUNDRED-SAMPLES IN THE ARRAY [TIME]
  |
M |    TMAX, TMEAN, TMIN = TIME ;
S |                         1
```

```
                                    :
M     DO FOR I = 2 TO 100,
M        TMEAN = TMEAN + TIME ,
S                            I

M        IF TIME  > TMAX THEN
S               I

M           TMAX = TIME ,
S                     I

M        IF TIME  < TMIN THEN
S               I

M           TMIN = TIME ,
S                     I

M     END,
M     TMEAN = TMEAN / 100,
C
C    COMPUTE STANDARD DEVIATION
C

M     SUM_OF_SQUARES = 0,
M     DO FOR I = 1 TO 100,
E                                                    2
M        SUM_OF_SQUARES = SUM_OF_SQUARES + (TIME  - TMEAN) ,
S                                             I

M     END,
M     STAN_DEV = SQRT(SUM_OF_SQUARES / 100),
M     WRITE(6)  MIN = ', TMIN, ' MEAN =  , TMEAN, ' MAX = ', TMAX, ' STANDARD DEVIATION = ', STAN_DEV,
M  CLOSE EXERCISE_2,
```

b)

```
M  EXERCISE_2
M  PROGRAM,
M     DECLARE M MATRIX(5, 5)
M     DECLARE TIME SCALAR INITIAL(0),
M     DECLARE SCALAR INITIAL(0),
M                 TMIN, TMAX, TMEAN, SUM_OF_SQUARES, STAN_DEV,
M     DECLARE INTEGER,
M                 I, J, K,
M     TMEAN, SUM_OF_SQUARES = 0,
M     TMAX = -1,                                         /* LESS THAN ANY POSSIBLE TIME VALUE *
M     TMIN = 1000,                                    /* GREATER THAN ANY FEASIBLE TIME VALUE *
M     DO FOR I = 1 TO 100,
M        DO FOR J = 1 TO 5,
M           DO FOR K = 1 TO 5,
M              M    = RANDOM,
S               J,K

M           END
M        END,
M        TIME = RUNTIME,
E        *   *-1
M        M = M  ,
M        TIME = RUNTIME - TIME,
M        TMEAN = TMEAN + TIME,
E                                              2
M        SUM_OF_SQUARES = SUM_OF_SQUARES + (TIME) ,
M        IF TIME > TMAX THEN
M           TMAX = TIME,
M        IF TIME < TMIN THEN
M           TMIN = TIME,
M     END,
M     TMEAN = TMEAN / 100,
E                                                2
M     STAN_DEV = SQRT((SUM_OF_SQUARES / 100) - TMEAN ),
M     WRITE(6)  MIN = ', TMIN,   MEAN = ', TMEAN,   MAX = ', TMAX,   STANDARD DEVIATION = ', STAN_DEV,
M  CLOSE EXERCISE_2,
```

6 1C

```
M   EXAMPLE_2-
M   PROGRAM,
M      DECLARE GYRO_INPUT ARRAY(12) INTEGER INITIAL(0);
M      DECLARE ATT_RATE ARRAY(12) SCALAR;
M      DECLARE SCALE ARPAY(3) CONSTANT( 013,  026,  013),
M      DECLARE BIAS SCALAR INITIAL(57 296),
M      DO FOR TEMPORARY I = 0 TO 9 BY 3,
M         DO FOR TEMPORARY J = 1 TO 3,
M            ATT_RATE    = GYRO_INPUT    SCALE  + BIAS,
S                    I+J             I+J      J

M         END,
M      END,
M   CLOSE EXAMPLE_2;
```

6 1D

```
M   EXAMPLE_4A:
M   PROGRAM;
M      DECLARE A ARRAY(5) SCALAR;
M      DECLARE TEMP SCALAR;
M      TEMP = A ,
S              5

M      DO FOR TEMPORARY T = 4 TO 1 BY -1,
M         A    = A ,
S          T+1    T

M      END,
M      A  = TEMP;
S       1

M   CLOSE EXAMPLE_4A,
```

6 2A

a) T	k) T
b) F	l) F
c) T	m) F
d) T	n) T
e) T	o) F
f) F	p) F
g) T	q) T
h) T	r) T
i) F	s) T
j) T	t) T

6 2B

A single arrayed statement takes the place of one or more loops and a statement to perform the same operation on each array element that the arrayed statement performs on the entire array If the programmer writes these loops, loop variables must be declared, correct loop limits must be coded, and such loops must be nested if the array is of two or more dimensions This means extra work for the programmer, and more complicated and potentially incorrect or unreadable code If an arrayed statement is coded, the compiler does the bookkeeping, and may even be able to produce more efficient code, since loop variables will not need to be saved for later reference

6 3A

a) $$X = \left(\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} ? & ? & ? \\ ? & ? & ? \\ ? & ? & ? \end{bmatrix} \begin{bmatrix} ? & ? & ? \\ ? & ? & ? \\ ? & ? & ? \end{bmatrix} \right)$$

b) $$Y = \left(\begin{matrix} \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} & \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} & \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \\ \begin{bmatrix} ? \\ ? \\ ? \end{bmatrix} & \begin{bmatrix} ? \\ ? \\ ? \end{bmatrix} & \begin{bmatrix} ? \\ ? \\ ? \end{bmatrix} \\ \begin{bmatrix} ? \\ ? \\ ? \end{bmatrix} & \begin{bmatrix} ? \\ ? \\ ? \end{bmatrix} & \begin{bmatrix} ? \\ ? \\ ? \end{bmatrix} \end{matrix} \right)$$

c) $$Z = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \begin{bmatrix} ? \\ ? \\ ? \end{bmatrix} \begin{bmatrix} ? \\ ? \\ ? \end{bmatrix} \begin{bmatrix} ? \\ ? \\ ? \end{bmatrix} \begin{bmatrix} ? \\ ? \\ ? \end{bmatrix} \begin{bmatrix} ? \\ ? \\ ? \end{bmatrix} \begin{bmatrix} ? \\ ? \\ ? \end{bmatrix}$$

d) $A = (1, 1, 1, 1, 1, 1, 1, 1, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)$

6 3B

```
S  =  X$(3 1,3),
S  =  Y$(3,1 3),
S  =  Z$(7 3),
S  =  A$21,
```

6 3C

```
M$(1,2 TO 4)  =  X$(2 3,*),   }
M$(1,5 TO 7)  =  X$(3 1,*),   }   from X
M$(1,8)       =  X$(3 2,1),   }

M$(1,2 TO 4)  =  Y$(2,3 *),   }
M$(1,5 TO 7)  =  Y$(3,1 *),   }   from Y
M$(1,8)       =  Y$(3,2 1),   }

M$(1,2 TO 4)  =  Z$(6 *),     }
M$(1,5 TO 7)  =  Z$(7 *),     }   from Z
M$(1,8)       =  Z$(8 1),     }

M$(1,2 TO 8)  =  A$(16 TO 22),
```

6 3 1A

```
M | PRIMES:
M | PROGRAM,
M |    REPLACE LIMIT BY "100";
M |    DECLARE PRIME ARRAY(LIMIT) BOOLEAN INITIAL(TRUE);
M |    DO FOR TEMPORARY I = 2 TO LIMIT,
E |
M |       IF PRIME   THEN
S |            I-
  |
M |          DO;
M |             DO FOR TEMPORARY J = 2 I TO LIMIT BY I,
E |
M |                PRIME   = FALSE;
S |                     J:
  |
M |             END,
M |             WRITE(6) I;
M |          END;
M |    END,
M | CLOSE PRIMES;
```

6 4.1A

DECLARE TEMP VECTOR(27),

TEMP = VECTOR$27(X),
M$(1,2 TO 8) = TEMP$(16 TO 22),

TEMP = VECTOR$27(Y),
M$(1,2 TO 8) = TEMP$(16 TO 22),

TEMP = VECTOR$27(Z),
M$(1,2 TO 8) = TEMP$(16 TO 22),

The assignment from A is already quite simple

6 4 1B

a) ARRAY(2,3) INTEGER $\begin{pmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \end{pmatrix}$

b) ARRAY(12) INTEGER (1 2 3 1 2 3 1 2 3 1 2 3)

c) ARRAY(3) SCALAR (1 1 1)

d) ARRAY(2,6) INTEGER $\begin{pmatrix} 1 & 2 & 3 & 1 & 2 & 3 \\ 1 & 2 & 3 & 1 & 2 & 3 \end{pmatrix}$

e) MATRIX(3,3) $\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$ ✓

f) VECTOR(6) $\begin{bmatrix} 1 \\ 2 \\ 3 \\ 1 \\ 2 \\ 3 \end{bmatrix}$

6A

```
M | MEDIAN
M | PROGRAM,
M |     DECLARE INTEGER,
M |                X, TEMP, SMALLEST,
M |     DECLARE VALUE_LIST ARRAY(25) INTEGER INITIAL(76, 87, 65, 54, 43, 32, 21, 12, 23, 34, 45, 56, 67,
M |     78, 123, 234, 345, 456, 567, 678, 789, 890, 987, 876  765),
M |     DO FOR X = 1 TO 13
M |        SMALLEST = X,
M |        DO FOR TEMPORARY J = X + 1 TO 25,
M |           IF VALUE_LIST  < VALUE_LIST           THEN
S |                        J             SMALLEST
  |
M |              SMALLEST = J,
M |        END,
M |        IF SMALLEST ¬= X THEN
M |           DO,
M |              TEMP = VALUE_LIST         ,
S |                               SMALLEST
  |
M |              VALUE_LIST         = VALUE_LIST ;
S |                        SMALLEST            X
  |
M |              VALUE_LIST  = TEMP,
S |                        X
  |
M |           END,
M |     END,
M |     WRITE(6) 'MEDIAN =  , VALUE_LIST  ,
S |                                     13
  |
M | CLOSE MEDIAN,
```

6B

```
DECLARE TIMING_DATA ARRAY(4,26) INTEGER INITIAL(0),
DECLARE I INTEGER,


DO FOR I = 1 TO 25,
   TIMING_DATA$(1 TO 3,I) = TIME_VALUES$(*,I),
   TIMING_DATA$(4,I) = SUM(TIM_VALUES$(*,I)),
END,
DO FOR I = 1 TO 3,
   TIMING_DATA$(I,26) = SUM(TIM_VALUES$(I,*)),
END,
```

Solutions

7 1A

2
4

7 1B

line 4	the outer block may not call procedure PROC2, which is nested with PROC1
line 5	the variable Y is known only within the scope of procedure PROC1
line 8	Procedure PROC1 cannot call itself

7 1C

Block	May be invoked from block(s)
2	1,3,4,5,6
3	1,2
4	3,5,6
5	3,4
6	5

7 2A

Move the code block defining ALMOST_EQUAL from the end of the program to a point before ALMOST_EQUAL is invoked, i e , immediately before or after the block MASS

7 2B

a The function RANDOM returns a scalar X with uniform distribution in the range $0 \leq X < 1$ The function ROLL uses the implicit scalar-to-integer conversion supplied by HAL/S, with implied rounding Its results may be described by a table

a random value in the range	yields an amount of
$0 \leq X < 1$	1
$1 \leq X < 3$	2
$3 \leq X < 5$	3
$5 \leq X < 7$	4
$7 \leq X < 9$	5
$9 \leq X < 1$	6

Thus, it is clear that the probabilities that ROLL will return 1 and 6 are 1/10, while the probabilities of 2,3,4, and 5 are 1/5 b

```
M   FIX_ROLL.
M   PROGRAM,
M      DECLARE COUNT INTEGER INITIAL(0);
M      DECLARE I INTEGER,
M   ROLL:
M   FUNCTION INTEGER,
M      RETURN TRUNCATE(6 RANDOM + 1),
M   CLOSE,
M      DO FOR I = 1 TO 5;
M         DO UNTIL ROLL + ROLL = 7,
M            COUNT = COUNT + 1,
M         END;
M      END,
M      WRITE(6) COUNT,
M   CLOSE FIX_ROLL,
```

7 2C

```
M   FIND_GCDS:
M   PROGRAM;
M      DECLARE ARRAY(5) INTEGER,
M                 X, Y;
M      DECLARE I INTEGER;
M   GCD
M   FUNCTION(I1, I2) INTEGER,
M      DECLARE INTEGER,
M                 I1, I2, X, Y, R;
M      X = I2;
M      Y = I1;
M      DO WHILE X ¬= 0;
M         R = REMAINDER(Y, X);
M         Y = X;
M         X = R;
M      END;
M      RETURN ABS(Y);
M   CLOSE GCD,
M      READ(5) [X], [Y],
M      DO FOR I = 1 TO 5;
M         IF GCD(X , Y ) ¬= 1 THEN
S                 I   I

M            WRITE(6) X , Y , GCD(X , Y );
S                     I   I        I   I

M      END;
M   CLOSE FIND_GCDS,
```

7 3A

```
M | FIX_ROLL-
M | PROGRAM,
M |    DECLARE COUNT INTEGER INITIAL(0);
M |    DECLARE INTEGER,
M |                I, ROLL1, ROLL2,
M | ROLL
M | PROCEDURE ASSIGN(A);
M |    DECLARE A INTEGER;
M |    A = TRUNCATE(6 RANDOM + 1);
M | CLOSE ROLL;
M |    DO FOR I = 1 TO 5;
M |       DO UNTIL ROLL1 + ROLL2 = 7;
M |          COUNT = COUNT + 1;
M |          CALL ROLL ASSIGN(ROLL1);
M |          CALL ROLL ASSIGN(ROLL2),
M |       END,
M |    END;
M |    WRITE(6) COUNT;
M | CLOSE FIX_ROLL;
```

The solution in which ROLL is a function is clearly preferable, because the code to invoke ROLL is much simpler in that case

In general, when a block is to produce as output a single value of any HAL/S type, the FUNCTION form will tend to produce more comprehensible code than the PROCEDURE form This is because the calling sequence for a function mirrors closely the mathematical notation for a function, and because often (as in this example) use of the functional form avoids the introduction of "dummy" variables with no intrinsic meaning to the algorithm being implemented In the procedure form, these dummy variables must be used as ASSIGN parameters

7A

```
M | DROP:
M | PROGRAM;
M |    DECLARE SCALAR,
M |               DROP_TIME, BOUNCE_TIME;
M |    DECLARE SCALAR INITIAL(0),
M |               TIME, HORIZ_DIST,
M |    DECLARE HEIGHT SCALAR INITIAL(110);
M |    DECLARE HORIZ_SPEED CONSTANT(4);
M |    DECLARE G CONSTANT(32);
M |    DECLARE I INTEGER,
M |    REPLACE NUMBER_OF_BOUNCES BY "10";
M | TIME_TO_DROP
M | FUNCTION(H),
M |    DECLARE H SCALAR,
M |    RETURN SQRT(2 H / G);
M | CLOSE TIME_TO_DROP,
M | HORIZ_MOTION
M | PROCEDURE(T) ASSIGN(H),
M |    DECLARE SCALAR,
M |               T, H,
M |    H = H + HORIZ_SPEED T;
M | CLOSE HORIZ_MOTION;
M | BOUNCE
M | PROCEDURE ASSIGN(H, T);
M |    DECLARE SCALAR,
M |               H, T;
M |    H =  75 H;
M |    T = SQRT(2 H / G);
M | CLOSE BOUNCE;
M |    DO FOR I = 1 TO NUMBER_OF_BOUNCES;
M |       DROP_TIME = TIME_TO_DROP(HEIGHT);
M |       CALL HORIZ_MOTION(DROP_TIME) ASSIGN(HORIZ_DIST),
M |       TIME = TIME + DROP_TIME,
M |       WRITE(6) 'BOUNCE', I, 'TIME', TIME, 'HORIZONTAL DISPLACEMENT', HORIZ_DIST;
M |       CALL BOUNCE ASSIGN(HEIGHT, BOUNCE_TIME),
M |       CALL HORIZ_MOTION(BOUNCE_TIME) ASSIGN(HORIZ_DIST),
M |       TIME = TIME + BOUNCE_TIME;
M |    END,
M | CLOSE DROP;
```

7B

```
M   SIMPSON:
M   PROGRAM;
M      DECLARE SCALAR,
M                 INITIAL_VALUE, FINAL_VALUE, OLD_APPROX, NEW_APPROX, POINT;
M      DECLARE SCALAR,
M                 DELTA, EPSILON, A, B, C, D;
M   POLY
M   FUNCTION(X) SCALAR;
M      DECLARE X SCALAR,
E                3      2
M      RETURN A X  + B X  + C X + D,
M   CLOSE POLY;
M      OLD_APPROX, NEW_APPROX = 0,
M      READ(5) A, B, C, D, INITIAL_VALUE, FINAL_VALUE, EPSILON;
M      DELTA = (FINAL_VALUE - INITIAL_VALUE) / 5;
M      DO UNTIL (NEW_APPROX - OLD_APPROX) < EPSILON;
M         OLD_APPROX = NEW_APPROX;
M         NEW_APPROX = POLY(INITIAL_VALUE) + POLY(FINAL_VALUE),
M         DO FOR POINT = INITIAL_VALUE + DELTA TO FINAL_VALUE - (DELTA / 2) BY DELTA;
M            NEW_APPROX = NEW_APPROX + 2 POLY(POINT),
M         END,
M         NEW_APPROX = NEW_APPROX DELTA / 2;
M         DELTA = DELTA / 2,
M      END,
M      WRITE(6) NEW_APPROX;
M   CLOSE SIMPSON;
```

7C

```
M  DROP
M  PROGRAM;
M     DECLARE SCALAR,
M                 ,DROP_TIME, BOUNCE_TIME;
M     DECLARE SCALAR INITIAL(0),
M                  TIME, HORIZ_DIST;
M     DECLARE HEIGHT SCALAR INITIAL(110);
M     DECLARE HORIZ_SPEED CONSTANT(4),
M     DECLARE G CONSTANT(32),
M     DECLARE I INTEGER,
M     REPLACE NUMBER_OF_BOUNCES BY "10";
M  TIME_TO_DROP
M  FUNCTION(H);
M     DECLARE H SCALAR;
M     RETURN SQRT(2 H / G);
M  CLOSE TIME_TO_DROP;
M  BOUNCE
M  PROCEDURE ASSIGN(H, T),
M     DECLARE SCALAR,
M                  H, T;
M     H = .75 H,
M     T = SQRT(2 H / G),
M  CLOSE BOUNCE;
M     DO FOR I = 1 TO NUMBER_OF_BOUNCES - 1,
M        DROP_TIME = TIME_TO_DROP(HEIGHT);
M        HORIZ_DIST = HORIZ_DIST + HORIZ_SPEED DROP_TIME;
M        TIME = TIME + DROP_TIME,
M        WRITE(6) 'BOUNCE', I, 'TIME', TIME, 'HORIZONTAL DISPLACEMENT', HORIZ_DIST,
M        CALL BOUNCE ASSIGN(HEIGHT, BOUNCE_TIME),
M        HORIZ_DIST = HORIZ_DIST + HORIZ_SPEED BOUNCE_TIME,
M        TIME = TIME + BOUNCE_TIME,
M     END,
M     DROP_TIME = TIME_TO_DROP(HEIGHT);
M     HORIZ_DIST = HORIZ_DIST + HORIZ_SPEED DROP_TIME,
M     TIME = TIME + DROP_TIME,
M     WRITE(6) 'BOUNCE', I, 'TIME', TIME, 'HORIZONTAL DISPLACEMENT', HORIZ_DIST;
M  CLOSE DROP,
```

Solutions

8 1A

There are several advantages to naming I/O channels

1) If several channels are in use, giving them descriptive names makes it clearer what any particular I/O statement is doing

2) References-to REPLACE macros are collected in the cross-reference table, allowing all I/O statements to be found quickly and easily

3) If it becomes necessary to reassign a channel, the channel number need only be changed once, in the REPLACE statement, and all I/O statements, referencing that channel will automatically be changed

8 1B

The expressions in the list are evaluated one by one, and data items converted to character string standard external format These strings are then assembled into lines and transmitted in an implementation dependent fashion to the output device associated with the channel number specified in the WRITE statement

Any legal HAL/S expression may appear in a WRITE statement There are no restrictions whatsoever on output

8 1C

a) 1 and 5

b) 1, 3, 4, and 5.

8 2A

a) First, the three matrices in MAT_ARR1 will be printed, then the three matrices in MAT_ARR2

b) The easiest way to do this is with loops

```
DO FOR TEMPORARY I = 1 TO 3,
   DO FOR TEMPORARY J = 1 TO 3,
      WRITE(6) MAT_ARR1$(I 1,*),TAB(20),MAT_ARR2$(I J,*),
   END,
   WRITE(6) SKIP(2),
END,
```

It could also be done with a single WRITE statement

```
WRITE(6) MAT_ARR1$(1 1,*),TAB(20),MAT_ARR2$(1 1,*),SKIP(1),
COLUMN(1),MAT_ARR1$(1 2,*),TAB(20),MAT_ARR2$(1 2,*),SKIP(1),
COLUMN(1),MAT_ARR1$(1 3,*),TAB(20),MAT_ARR2$(1 3,*),SKIP(3),
COLUMN(1),MAT_ARR1$(2 1,*),TAB(20),MAT_ARR2$(2 1,*),SKIP(1),
COLUMN(1),MAT_ARR1$(2 2,*),TAB(20),MAT_ARR2$(2 2,*),SKIP(1),
COLUMN(1),MAT_ARR1$(2 3,*),TAB(20),MAT_ARR2$(2 3,*),SKIP(3),
COLUMN(1),MAT_ARR1$(3 1,*),TAB(20),MAT_ARR2$(3 1,*),SKIP(1),
```

```
COLUMN(1),MAT_ARR1$(3 2,*),TAB(20),MAT_ARR2$(3 2,*),SKIP(1),
COLUMN(1),MAT_ARR1$(3 3,*),TAB(20),MAT_ARR2$(3 3,*),
```

8 2B

1) b
2) a, c
3) d
4) c (paged files only)
5) a, e
6) none of a-e, overrides the default SKIP(1)
7) c

8 3A

a) INTS = (8,7,7), SCALS = (−1,225,4)

b) INTS = (0,1,1), SCALS = (7 2,0,0)

c) INTS = (2,1,3), SCALS = (2 49,0,2 51)

8 3B

Change the READ statement to

```
READ(5) COLUMN(8),INTS,SKIP(1),COLUMN(8),SCALS,
```

8 4A

All are legal character subscripts Only a, b, c, and e are legal vector subscripts, all the others have partition sizes not computable at compile time

8 4B

The output will be similar to this

```
ABC    ABCABC
123AB    BC456
1223ABC456
ABCABC    ABC
```

8.4C

All the expressions listed are true

8 5A

Only character strings may be read using the READALL statement

8 5B

All characters on the input file are retrieved by the READALL statement, no matter what they are Character strings to be input using the READ statement must be surrounded by single quotes, which are *not* placed into the target variable Furthermore, single quotes represent themselves in READALL input, while they must be represented by a pair of quotes in succession in READ input

8A

```
M   REVERSE:
M   PROGRAM;
M      DECLAPE ARPAY(5) CHARACTER(5),
M                 CHAR_ARR1, CHAR_ARR2,
M      DECLARE X INTEGER;
M   REV:
M   FUNCTION(C) CHARACTER(5);
M      DECLARE C CHARACTER(*);
M      DECLARE CHAPACTER(8),
M                 CTEMP, CHAR_REV,
M      DECLARE INTEGER,
M                 I, L,
E          ,         ,     ,
M      CHAR_REV, CTEMP = C,
E          ,
M      IF CTEMP = '' THEN
M         RETURN '';
E                       ,
M      L = LENGTH(CTEMP),
M      DO FOR I = 1 TO L,
E              ,           ,
M         CHAR_REV  = CTEMP      ,
S                 I        L+1-I

M      END;
E                   ,
M      RETURN CHAR_REV,
M   CLOSE REV,
E                      ,         ,
M      READ(5) [CHAR_ARR1], [CHAR_ARR2];
M      DO FOR X = 1 TO 5,
E              ,              ,
M         CHAR_ARR1   = TRIM(CHAR_ARR1  );
S                 X                    X·

E              ,              ,
M         CHAR_ARR2   = TRIM(CHAR_ARR2  ),
S                 X·                   X

E                           ,       ,                          ,        ,
M         WRITE(6) COLUMN(5), REV(CHAR_ARR1  ), COLUMN(15), REV(CHAR_ARR2  ),
S                                          X:                             X:

M      END,
M   CLOSE REVERSE,
```

8B

```
M  DECODE_NAMES:
M  PROGRAM,
M     DECLARE NAMES ARRAY(50, 2) CHARACTER(15),
M     DECLARE INLINE CHARACTER(80);
M     DECLARE I INTEGER;
M     REPLACE NO_OF_NAMES BY "50";
E      ,
M     INLINE = '',
M     DO FOR I = 1 TO NO_OF_NAMES;
M        CALL GET_NAME(I),
E           ,
M        IF NAMES       = 'S' THEN
S                I,2*1

E                      ,                      ,
M           WRITE(6) NAMES      || ' ' || NAMES    ;
S                        I,1:                   I,2

M     END,
M  GET_NAME'
M  PROCEDURE(N),
M     DECLARE INTEGER,
M                N, K;
E         ,
M     IF INLINE = '' THEN
M        CALL GET_LINE,
E               ,
M     K = INDEX(INLINE, ','),
M     IF K = 0 THEN
E                                   ,
M        CALL FIRST_AND_LAST(INLINE, N),
M     ELSE
M        DO,
E                                  ,
M           CALL FIRST_AND_LAST(INLINE          , N),
S                                 1 TO K-1

E                ,          ,
M           INLINE = TRIM(INLINE          ),
S                             K+1 TO #

M        END;
M  FIRST_AND_LAST
M  PROCEDURE(C, N),
M     DECLARE C CHARACTER(*),
M             N INTEGER,
M             I INTEGER,
E                  ,
M     I = INDEX(C, ' '),
E       ,      ,
M     NAMES      = C           ,
S          N,1.    1 TO I-1

E       ,      ,
M     NAMES      = C           ;
S          N,2'    I+1 TO #

M  CLOSE FIRST_AND_LAST;
M  GET_LINE:
M  PROCEDURE,
E                    ,
M     READALL(5) INLINE,
E       ,         ,
M     INLINE = TRIM(INLINE),
M  CLOSE GET_LINE,
M  CLOSE GET_NAME,
M  CLOSE DECODE_NAMES,
```

8C

```
M   NUMBER_TO_ENGLISH
M   PROGRAM,
M      DECLARE INTEGER,
M                 N, H, T, U,
M      DECLARE CHAPACTER(30),
M                 LEFT_PART, RIGHT_PART,
M      DECLARE TENS ARRAY(9) CHARACTER(7) INITIAL('TEN', 'TWENTY', 'THIRTY', 'FORTY',  FIFTY , 'SIXTY',
M       SEVENTY',  EIGHTY', 'NINETY'),
M      DECLARE TEENS AFRAY(9) CHARACTER(9) INITIAL('ELEVEN , 'TWELVE', 'THIRTEEN', 'FOURTEEN', 'FIFTEEN
M      ,  SIYTEEN , 'SEVENTEEN', 'EICHTEEN',  NINETEEN ),
M      DECLARE UNITS ARRAY(9) CHARACTER(5) INITIAL('ONE',  TWO , 'THREE',  FOUP',  FIVE', 'SIX', 'SEVEN
M      , 'EICHT , 'NINE ),
M      READ(5) N,
M      IF N = 0 THEN
M         DO,
E                ,
M            LEFT_PART = '';
E                ,
M            RIGHT_PART = 'ZERO ,
M         END,
M      ELSE
M         DO,
M            H = DIV(N, 100),
M            T = DIV(REMAINDER(N, 100), 10),
M            U = PEMAINDER(N, 10),
M            IF H > 0 THEN
E                   ,        ,
M               LEFT_PART = UNITS   || ' HUNDRED ',
S                                H

M            ELSE
E                   ,
M               LEFT_PART = ' ,
M            IF U = 0 THEN
E                   ,        ,
M               RIGHT_PART = TENS  ,
S                                T

M            ELSE
M               DO,
M                  IF T > 1 THEN
E                        ,        ,              ,
M                     RIGHT_PART = TENS  ||  -' || UNITS  ,
S                                     T             U

M                  ELSE IF T = 1 THEN
E                        ,        ,
M                     RIGHT_PART = TEENS  ,
S                                      U

M                  ELSE
E                        ,        ,
M                     RIGHT_PART = UNITS  ,
S                                      U

M               END,
M         END,
E                 ,                  ,
M      WRITE(6) LEFT_PART || RIGHT_PART,
M   CLOSE,
```

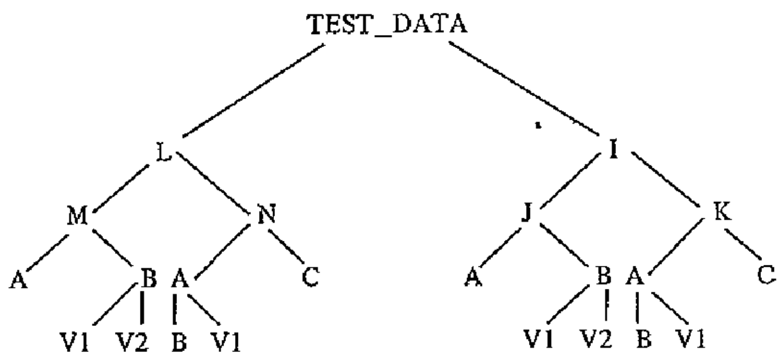
Solutions

9 2A

```
STRUCTURE X
    1 A1,
       2 C1 VECTOR,
       2 D1 MATRIX,
    1 B1,
       2 E1 VECTOR,
       2 F1 MATRIX,

STRUCTURE Y
    1 A2 SCALAR,
    1 B2,
       2 D2 ARRAY(5) VECTOR,
       2 E2 ARRAY(5) VECTOR,
    1 C2 SCALAR,
```

9 2B

a)



b) TEST_DATA L M A
TEST_DATA L M B V1
TEST_DATA.L M B.V2
TEST_DATA L N A B
TEST_DATA L N A V1
TEST_DATA L N C
TEST_DATA I J A
TEST_DATA.I J B V1
TEST_DATA I J B V2
TEST_DATA I K A B
TEST_DATA.I K A V1
TEST_DATA I K C

c)
```
STRUCTURE DATA
    1 L,
       2 M,
          3 A INTEGER,
          3 B,
             4 V1 VECTOR,
             4 V2 VECTOR,
       2 N,
          3 A,
             4 B INTEGER,
             4 V1 VECTOR,
          3 C SCALAR,
    1 I,
       2 J,
          3 A INTEGER,
          3 B,
             4 V1 VECTOR,
             4 V2 VECTOR,
       2 K,
          3 A,
             4 B INTEGER,
             4 V1 VECTOR,
          3 C SCALAR,
```

d) All of the assignments shown are legal

9.2C

```
STRUCTURE MINOR
   1 V VECTOR,
   1 T SCALAR,
STRUCTURE MAJOR
   1 X1 MINOR-STRUCTURE,
   1 X2 MINOR-STRUCTURE,
   1 X3 MINOR-STRUCTURE,
   1 X4 MINOR-STRUCTURE,
   1 X5 MINOR-STRUCTURE,
DECLARE DATA MAJOR-STRUCTURE,



READ(5) DATA,
CALL PROCESS(DATA),
```

The procedure PROCESS must be modified to accept a MAJOR-structure as input instead of the ARRAY(2) it originally took

9 3A

```
STRUCTURE MINOR
    1 V VECTOR,
    1 T SCALAR,
DECLARE DATA MINOR-STRUCTURE(5),


READ(5) DATA,
CALL PROCESS(DATA),
```

Now PROCESS must be changed to accept a 5-copy MINOR-structure as its argument The data is still read in the same order as before

9 3B

a) A$(25,) or A$25	type	A1-STRUCTURE
b) A B$(*,3)	type	ARRAY(100) INTEGER
c) A C$(10 TO 20,)	type	ARRAY(11) SCALAR
d) A D$(75 TO 85,)	type	ARRAY(11) VECTOR(6)
e) A D$(1,1)	type	SCALAR

9 3C

```
M | MEAN:
M | PROGRAM,
M |    STRUCTURE PERSON:
M |       1 SS INTEGER DOUBLE,
M |       1 SALARY SCALAR,
M |       1 JOB_CODE INTEGER,
M |       1 PNAME CHARACTER(32);
M |    DECLARE COMPANY PERSON-STRUCTURE(100),
E |                +
M |    READ(5) {COMPANY},
M |    WRITE(6) SUM({COMPANY SALARY}) / 100;
M | CLOSE MEAN;
```

9 4A

a) No X E F has the RIGID attribute, Y does not.

b) Yes

c) Yes.

d) Yes

e) Yes

9 4B

a)	The 20th copy of A	type	A-STRUCTURE
b)	The 10th and 11th copies of A	type	A-STRUCTURE(2)
c)	C from the first copy of A	type	INTEGER
d)	D from the 4th-6th copies of A	type	ARRAY(3) VECTOR(6)
e)	The 4th-6th components of D from all copies of A	type	ARRAY(20) VECTOR(3)

9A

Structures allow the programmer to organize data of mixed types into one logical unit that may be input, output, assigned, and passed as a parameter When a structure is passed as a parameter, overhead is saved, as all the components of the structure became available to the called procedure or function without being passed individually as separate parameters

The use of structures also allows the transfer of an aggregate of assorted data in a single FILE I/O statement In I/O contexts, multiple-copy structures are particularly convenient for reading or writing large blocks for the sake of efficiency

9B

```
M  BEST_ONE
M  PROGRAM;
M     STRUCTURE ITEM_DATA
M        1 VEC VECTOR,
M        1 TIMETAG SCALAR;
M     STRUCTURE UNIT_DATA
M        1 ACCEL ITEM_DATA-STRUCTURE,
M        1 VEL ITEM_DATA-STRUCTURE,
M        1 PITCH ITEM_DATA-STRUCTURE;
M     STRUCTURE BEST
M        1 BEST_ACCEL ITEM_DATA-STRUCTURE,
M        1 BEST_VEL ITEM_DATA-STRUCTURE,
M        1 BEST_PITCH ITEM_DATA-STRUCTURE;
M     DECLARE BEST_DATA BEST-STRUCTURE;
M     DECLARE SYSTEM_DATA UNIT_DATA-STRUCTURE(3);
M  MIDDLE
M  FUNCTION(DFU) ITEM_DATA-STRUCTURE;
M     DECLARE DFU ITEM_DATA-STRUCTURE(3);                          /* DATA FROM UNIT */
M     IF DFU TIMETAG  = MIDVAL(DFU TIMETAG  , DFU TIMETAG  , DFU TIMETAG  ) THEN
S               1;                     1,             2,              3,

E                 +
M        RETURN DFU  ;
S                  1;

M     IF DFU TIMETAG  = MIDVAL(DFU TIMETAG  , DFU TIMETAG  , DFU TIMETAG  ) THEN
S               2,                     1,             2,              3,

E                 +
M        RETURN DFU  ;
S                  2;

E              +
M     RETURN DFU  ;
S               3;

M  CLOSE MIDDLE;
E                 +
M     READ(5) (SYSTEM_DATA);
E                  +          +                    +
M     BEST_DATA BEST_ACCEL = MIDDLE({SYSTEM_DATA ACCEL}  );
S                                        -             *;

E                 +        +                +
M     BEST_DATA BEST_VEL = MIDDLE({SYSTEM_DATA VEL}  );
S                                                 *;

E                  +          +                    +
M     BEST_DATA BEST_PITCH = MIDDLE({SYSTEM_DATA PITCH}  );

S                                                      *;

M  CLOSE BEST_ONE;
```

Solutions

10 1A

Control falls through to the statement following the ON ERROR statement, unless the ON ERROR statement has

1) caused a GO TO or RETURN statement to be executed, or

2) specified SYSTEM or IGNORE, in which case either control returns to the program at the point where execution was interrupted, or the program terminates, depending on the particular error

10 1B

If the error should occur after control has left the loop, an unexpected transfer of control into the loop will occur, potentially causing disastrous results since loop variables may have unusual values, and TEMPORARY variables may even have been redefined since leaving the loop

The compiler normally enforces a ban on branching into DO END groups In this case where the compiler is unable to do so, the programmer should follow the same course

10 1C

1) SYSTEM If no ON ERROR statement is active for the current error, or if the active one is ON ERROR SYSTEM, the standard action, if any, is taken and an error message is sent

2) IGNORE If an ON ERROR IGNORE statement is in effect for the error in question, the standard fix-up is taken and no error message is sent

3) If an ON ERROR statement defining a user action is in effect for the specified error, then the user code receives control without possibility of returning to the point where the error occurred No error message is sent

10 1D

Error Specification	Precedence	
ERROR$(m n)	1	first
ERROR$(m)	2	↓
or		
ERROR$(m)	2	↓
ERROR	3	last

10 2A

An error handler may be deactivated

1) when flow of control leaves the block containing the handler,
2) when it is superseded by another error handler, and
3) when an OFF ERROR statement of the same form is executed

10 2B

a) All three error handlers are still active both OFF ERROR statements were ignored

b) ON ERROR$(1 1) IGNORE, and ON ERROR$(2) IGNORE, are still active The first OFF error statement cancelled the first ON ERROR statement, and the second had no effect

10 3A

The SEND ERROR statement is used

1) to simulate the occurrence of system-defined errors for testing, and
2) to allow the user to define errors and write error handlers for them

10 3B

When an applicable error handler is found in the local block, higher level blocks need not be searched, as handlers in the calling blocks are overridden by the local handler

10A

a) No message
b) Message
c) No message
d) No message
e) Message
f) No message
g) No message
h) Message
i) No message
j) Message
k) Message
l) No message

Solutions

11.1A

1 If several programmers are working on a single large project, it will probably be convenient to assign them separately-compilable sections of the program complex

2 In a multiprogramming environment where several PROGRAMs are to run concurrently, there is no way to compile them all in a single compilation step, so a program complex must be created

3 If the overall structure of a program is fixed, but small sections are under-going revision, separating those sections out as COMSUBs may allow those parts to be revised and recompiled without requiring recompilation of the entire program

11 1B

Just as if the COMSUB were an internal procedure, the error environment of the caller is searched for an applicable error handler, then the environment of the caller's caller, and so on

11 1C

a) Compiling a COMPOOL reserves space for the variables declared therein Also, in most implementations, a template is produced when the COMPOOL is compiled

b) The COMPOOL template, when included in the compilation of another compilation unit, makes the variables declared in the COMPOOL known to that compilation unit, without causing any space to be reserved for those variables

11 2A

The SCALARs A and B can only be referenced inside the program P but outside the FUNCTION block F Inside of F, scoping rules will cause A and B to refer to the local INTEGER variables

11 2B

FILTER does not require any of the data in GNC_POOL, so there is no need to include the template for GNC_POOL in the compilation of FILTER

11 2C

If several compool templates are included in a single compilation, names of variables *must* be unique, because there is only one scoping level outside the main block of a compilation Hence, it is in general desirable to give compool variables unique names, so that it is possible to refer to any compool from any other compilation unit if necessary

11 2D

a) A template for FILTER is needed in order to compile NAVIGATION, and with this order of compilation, it would need to be hand coded

b) In this case, CONTROL needs the template for FILTER

c) No template need be hand coded, as all will be available when they are needed

d) This order of compilation is particularly inconvenient, all templates will need to be hand coded.

11 3A

a) It is possible that the savings account for one ID might be updated, then the procedure interrupt and another account updated When control returned to the first task, the updating of the checking account would then be done incorrectly, transferring funds from one customer to another

b) If SAVINGS and CHECKING are declared with the LOCK attribute, and the transfer is enclosed in an UPDATE block, there is no possibility of an incorrect transfer of funds as described above

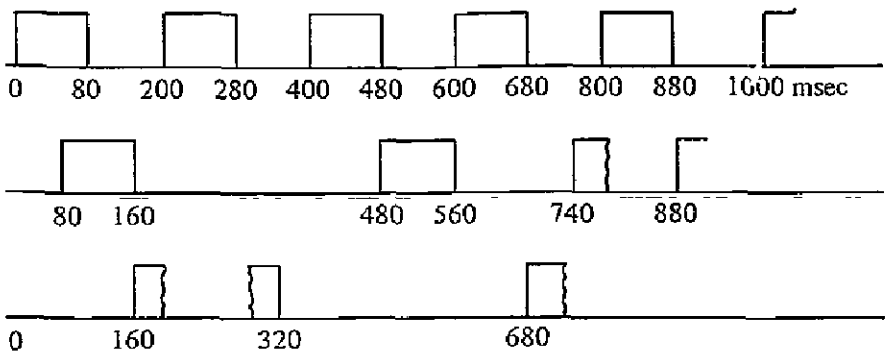
11 3B

a) In this case, any interruption of an execution of AWARD_INTEREST by another process that calls AWARD_INTEREST may cause either an error in updating the account, or in logging the interest

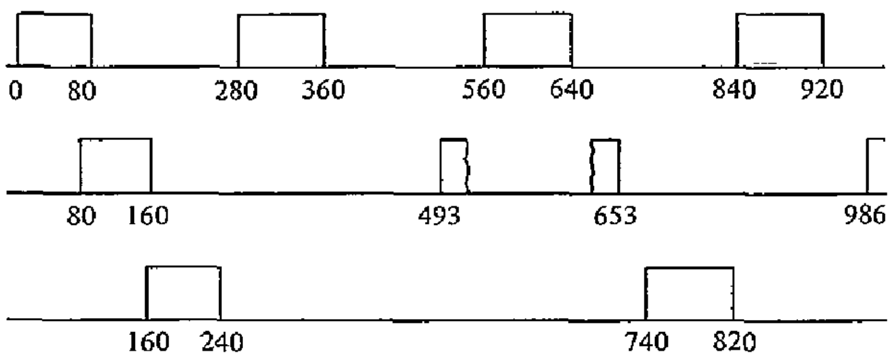
b) Make the procedure AWARD_INTEREST EXCLUSIVE Then there is no possibility that two processes will attempt to run AWARD_INTEREST concurrently.

Solutions

12.1A



12.1B



12.1C

SCHEDULE X PRIORITY(1), REPEAT UNTIL 3.5;
SCHEDULE Y IN 2.5 PRIORITY(2), REPEAT EVERY 1 UNTIL 6;

12.2A

The AT clause allows a process to be scheduled at a definite, predetermined time. The ON clause, on the other hand, allows a process to be scheduled depending on occurrences of an unpredictable nature. Either one can be appropriate, depending on the desired effect.

12.2B

Q is active only at B.

12.2C

SIGNAL X; will cause X to become TRUE just long enough for all active event expressions referencing X to be evaluated. In particular, no code testing X as a BOOLEAN variable will ever find it TRUE as a result of SIGNAL X; The sequence SET X; RESET X; will also cause X to become TRUE, then return to FALSE, but if in the meantime the process executing the SET and RESET statements relinquishes control, X will remain SET during execution of some HAL/S code, and may be found to be TRUE if tested.

12 2D

SCHEDULE X PRIORITY(1), REPEAT UNTIL TRAN2,
SCHEDULE Y ON TRAN1 PRIORITY(2), REPEAT EVERY 1 UNTIL 6,

12 2E

a) Unlatched, there is no need to specify LATCHED, so take the default

b) Latched, it is not possible to signal several events simultaneously

c) Latched, an unlatched event will always test FALSE

d) Latched, RESET is illegal for an unlatched event

e) Unlatched, presumably the loop is to execute once for each event transition, which would probably not happen if the event were SET and remained on

12 2F

```
SCHEDULE T ON MASTER PRIO(999) REPEAT,


T  TASK,
   RESET COMPL,
   WAIT FOR ¬MASTER,
   SET COMPL,
   WAIT FOR MASTER,
CLOSE T,
```

12 3A

```
M | P:
M | PROGRAM,
M |    DECLARE DENOM INTEGER INITIAL(10);
M |    SCHEDULE T PRIORITY(999), REPEAT UNTIL 1;
M | T
M | TASK,
M |    WAIT UNTIL 1 / DENOM,
M |    WRITE(6) RUNTIME;
M |    DENOM = DENOM - 2,
M |    IF DENOM < 1 THEN
M |       DENOM = 1;
M | CLOSE T,
M | CLOSE P,
```

12 3B

Unless something causes P to exit from the DO WHILE TRUE loop, CANCEL P will have no effect

If X is necessary to keep P as it is, it can be stopped with

TERMINATE P,

However, it is safer simply to remove the DO WHILE TRUE, and END, statements from P, and derive the same effect from writing

12A

```
M | FSW
M | PROGRAM,
M |    DECLARE VECTOR,
M |               POSITION, ATTITUDE, VELOCITY,
M |    DECLARE SCALAR,
M |               PITCH_COMMAND, ROLL_COMMAND,
M |   .DECLARE DESTINATION VECTOR,
M |    DECLARE ARRAY(4),
M |               SENSED_ATTITUDE VECTOR,
M |               SENSED_VELOCITY VECTOR,
M | INPUT_PROC
M | PROCEDURE,                                         /* SCALE AND FORMAT DATA FROM SENSORS *
M | CLOSE INPUT_PROC,
M | ELEVON_CMDS
M | PROCEDURE,                                                  /* COMMAND AEROSURFACES */
M | CLOSE ELEVON_CMDS,
M | TELEMETRY
M | PROCEDURE,                                                 /* DOWNLINK STATUS VARIABLES *
M | CLOSE TELEMETRY,
M | RUDDER_CMDS
M | PROCEDURE,                                                  /* CONTROL YAW AXIS */
M | CLOSE RUDDER_CMDS,
M | GUIDANCE
M | PROCEDURE,                                               /* COMPUTE DESIRED FLIGHT PATH *
M | CLOSE GUIDANCE,
M | FC_GAINS
M | PROCEDURE,                                                 /* COMPUTE CONTROL LAW GAINS *
M | CLOSE FC_GAINS,
M | NAVIGATION
M | PROCEDURE,                                         /* COMPUT REAL POSITION AND VELOCITY *
M | CLOSE NAVIGATION,
M | DISPLAY_UPDT
M | PROCEDURE,                                                  /* REFRESH CRT */
M | CLOSE DISPLAY_UPDT,
M |    SCHEDULE T1 PRIORITY(4), REPEAT EVERY  1,
M |    SCHEDULE T2 PRIORITY(3), REPEAT EVERY  2,
M |    SCHEDULE T3 PRIORITY(2), REPEAT EVERY  4,
M |    SCHEDULE T4 PRIORITY(1), REPEAT EVERY  8,
M | T1
M | TASK,
M |    CALL INPUT_PROC,
M |    CALL ELEVON_CMDS,
M |    CALL TELEMETRY,
M | CLOSE T1,
M | T2
M | TASK;
M |    CALL RUDDER_CMDS,
M |    CALL GUIDANCE,
M | CLOSE T2,
M | T3
M | TASK;
M |    CALL FC_GAINS,
M | CLOSE T3,
M | T4
M | TASK,
M |    CALL NAVIGATION,
M |    CALL DISPLAY_UPDT,
M | CLOSE T4,
M | CLOSE FSW,
```

The priorities here serve to fix the order of execution to be identical with that in the chapter seven example

12B

```
M   FSW
M   PROGRAM,
M      DECLARE VECTOR,
M                  POSITION, ATTITUDE, VELOCITY,
M      DECLARE SCALAR,
M                  PITCH_COMMAND, ROLL_COMMAND,
M      DECLARE DESTINATION VECTOR,
M      DECLARE ARRAY(4),
M                  SENSED_ATTITUDE VECTOR,
M                  SENSED_VELOCITY VECTOR,
M      DECLARE T1_DONE EVENT,
M   INPUT_PROC
M   PROCEDURE,                                    /* SCALE AND FORMAT DATA FROM SENSORS */
M   CLOSE INPUT_PROC,
M   ELEVON_CMDS
M   PROCEDURE,                                           /* COMMAND AEROSURFACES */
M   CLOSE ELEVON_CMDS,
M   TELEMETRY
M   PROCEDURE,                                          /* DOWNLINK STATUS VARIABLES */
M   CLOSE TELEMETRY,
M   RUDDER_CMDS
M   PROCEDURE,                                           /* CONTROL YAW AXIS */
M   CLOSE RUDDER_CMDS,
M   GUIDANCE
M   PROCEDURE,                                        /* COMPUTE DESIRED FLIGHT PATH */
M   CLOSE GUIDANCE,
M   FC_GAINS
M   PROCEDURE,                                          /* COMPUTE CONTROL LAW GAINS */
M   CLOSE FC_GAINS,
M   NAVIGATION
M   PROCEDURE,                                    /* COMPUTE REAL POSITION AND VELOCITY */
M   CLOSE NAVIGATION,
M   DISPLAY_UPDT
M   PROCEDURE,                                           /* REFRESH CRT */
M   CLOSE DISPLAY_UPDT,
M      SCHEDULE T1 PRIORITY(1), REPEAT,
M      SCHEDULE T2 PRIORITY(2), REPEAT,
M      SCHEDULE T3 PRIORITY(3), REPEAT,
M      SCHEDULE T4 PRIORITY(4), REPEAT,
M   T1
M   TASK,
M      CALL INPUT_PROC;
M      CALL ELEVON_CMDS,
M      CALL TELEMETRY,
M      SIGNAL T1_DONE,
M   CLOSE T1,
M   T2
M   TASK,
M      WAIT FOR T1_DONE,
M      WAIT FOR T1_DONE,
M      CALL RUDDER_CMDS,
M      CALL GUIDANCE,
M   CLOSE T2,
M   T3
M   TASK,
M      DO FOR TEMPORARY I = 1 TO 4,
M         WAIT FOR T1_DONE,
M      END,
M      CALL FC_GAINS,
M   CLOSE T3,
M   T4
M   TASK,
M      DO FOR TEMPORARY I = 1 TO 8,
M         WAIT FOR T1_DONE,
M      END,
M      CALL NAVIGATION,
M      CALL DISPLAY_UPDT,
M   CLOSE T4,
M   CLOSE FSW,
```

This solution guarantees that the various tasks will never be executing any of their procedures simultaneously Thus avoiding the need for UPDATE block protection of any shared variable

Solutions

13.1A

a) If FLAGS AND BIN'110000000000' = BIN'110000000000'

b) If FLAGS AND BIN'010101010101' = BIN'000000000000'
   or
   FLAGS AND BIN'000000111111' = BIN'000000111111'

c) If FLAGS = BIN'101010000010'

d) If FLAGS AND BIN'111010000011' = BIN'101010000010'

13 1B

```
M  FLIP-
M  FUNCTION(B) BIT(12);
M     DECLARE B BIT(12);
M     DECLARE FLIPPED BIT(12),
M     DO FOR TEMPORARY X = 1 TO 12,
E           .         .
M        FLIPPED  = B     ;
S                X    13-X
M     END,
E
M     RETURN FLIPPED;
M  CLOSE FLIP,
```

13 1C

```
M  EXERCISE_C:
M  PROGRAM,
M     DECLARE TABLE ARRAY(50) BIT(24);
M  SET_BITS:
M  PROCEDURE(ENTRY, VALUE);
M     DECLARE INTEGER,
M                ENTRY, VALUE;
E      .
M     TABLE                                       = BIT          (VALUE);
S          DIV(ENTRY,4)-6 AT (6 MOD(ENTRY,4)+1)      6 AT #-5
M  CLOSE SET_BITS;
M  GET_BITS:
M  FUNCTION(ENTRY) INTEGER,
M     DECLARE ENTRY INTEGER;
E                          .
M     RETURN INTEGER(TABLE                                   );
S                         DIV(ENTRY,4)-6 AT 6 MOD(ENTRY,4)+1
M  CLOSE GET_BITS;
M  CLOSE EXERCISE_C;
```

13 1D

```
M  NORMAL
M  FUNCTION(UNNORM) BIT(32),
M     DECLARE UNNORM BIT(32),
M     DECLARE B BIT(32),
M     DECLARE COUNT INTEGER,
E            .
M     IF UNNORM         = HEX'000000' THEN
S              9 TO 32

M        RETURN HEX'00000000';
E            .
M     B = UNNORM,
E
M     DO FOR COUNT = 1 TO 6 WHILE B        = HEX'0';
S                                  4 AT 9

E
M        B         = BIT(SHL(INTEGER(B        ), 4));
S         24 AT 9                 24 AT 9

E        .                              .
M        B       = BIT         (INTEGER(B       ) - 1);
S         7 AT 2      7 AT #-6          7 AT 2

M     END;
E
M     RETURN B,
M  CLOSE NORMAL,
```

13 1E

A OUTPUT = 1E5 INTEGER(INPUTS(4 AT 1)) + 1E4 INTEGER(INPUTS (4 AT 5)) +
1E3 INTEGER(INPUTS(4 AT 9)) + 1E2 INTEGER(INPUTS (4 AT 13)) +
1E1 INTEGER(INPUTS(4 AT 17)) + INTEGER(INPUTS (4 AT 21)),

13 1F

A OUTPUT = INTEGER(BIT(CHARACTERS(@HEX) (INPUT))),

13 2A

1) Partitions of bit strings
2) Columns of a matrix
3) A structure node with copiness

13.2B

a) Yes, if a name variable points to some variable in an outer code block and a variable is checked in an inner code block with the same identifier as that the name variable points to, the outer variables can be referenced

- b) No, need more information than the address which is all the name variable allows

c) Yes, name variables allow sharing Several name variables can point to the same data item

d) No, it is possible to go up and down name pointers but not reference an absolute address

e) No, name variables can only point to data of the same type they were declared

13 3A

```
STRUCTURE LOOP
    1 VALUE INTEGER,
    1 NEXT NAME LOOP-STRUCTURE,
DECLARE CIRCLE LOOP-STRUCTURE,

NAME(CIRCLE NEXT) = NAME (CIRCLE),
```

13 3B

```
STRUCTURE TQE
    1 TIME SCALAR,
    1 ACTION NAME ACTIONS-STRUCTURE,
    1 NEXT NAME TQE-STRUCTURE,
STRUCTURE ACTIONS.
    1 ACTION INTEGER,
    1 AFFECTED-PROCESS NAME PROCESS_CONTROL-STRUCTURE,
    1 NEX NAME ACTION-STRUCTURE,
```

line 28

```
DECLARE NAME TQE-STRUCTURE, NEWTQE, ENT,
DECLARE NAME ACTIONS-STRUCTURE, NEWACT, ENTACT,

NEW TQE TIME = WHEN,
NEWACT ACTION = WHAT,
NAME(NEWACT AFFECTED_PROCESS) = NAME(PROCNAME),
```

after
line 37

```
NAME(ACTV_Q ACTION) = NAME(NEWACT),
```

after
line 40

```
IF ENT NEXT TIME = NEWTQE TIME THEN DO,
   IF NAME(ENT ACTION) = NAME(NULL) THEN DO,
      NAME(ENT ACTION) = NAME(NEWACT),
      RETURN,
DO UNTIL NAME(ENTACT NEXT) = NAME(NULL)
   NAME(ENTACT) = NAME(ENTACT NEXT),
END,
NAME(ENTACT NEXT) = NAME(NEWACT),
RETURN,
```

after 44

```
NAME(ENT ACTION) = NAME(NEWACT),
```

after 50

```
NAME(NEWTQE ACTION) = NAME(NEWACT),
```

13 3C

If PCB is first or last in the ready queue, the code to remove PCB from the ready queue will not work To avoid the difficulty, rewrite STALL as follows

```
      STALL  PROCEDURE  ASSIGN(PCB),
         DECLARE  PCB  PROCESS_CONTROL-STRUCTURE,
C
C     Remove from ready queue
C
         IF NAME(PCB LAST)=NULL THEN NAME(PCREADY)=NAME(PCB NEXT),
         ELSE  NAME(PCB LAST NEXT)=NAME(PCB NEXT),
         IF  NAME(PCB NEXT)¬=NULL  THEN  NAME(PCB NEXT LAST)=NAME
         (PCB LAST),
C
C     Add to stalled queue  same as in the text
C
         NAME(PCB NEXT) = NAME(STALLED),
         NAME(STALLED) = NAME(PCB),
      CLOSE  STALL,
```

```
13A     PC_ENQUEUE  PROCEDURE  ASSIGN(PCB),
           DELCARE  PCB  PROCESS_CONTROL-STRUCTURE,
           DECLARE  PCPTR  NAME  PROCESS_CONTROL-STRUCTURE,

           IF  NAME(READYPC) = NULL  THEN  DO/        /*empty queue*/
              NAME(READYPC) = NAME(PCB),
              NAME(PCB LAST), NAME(PCB NEXT) = NULL,
           RETURN,

        END,

        NAME(PCPTR) = NAME(READYPC),
        DO  WHILE  NAME(PCPTR NEXT) ¬= NULL,
           IF  PCPTR PRIORITIE<PCB PRIORITIE  THEN  DO,
              NAME(PCB LAST) = NAME(PCPTR LAST),
              NAME(PCB NEXT) = NAME(PCPTR),
              IF  NAME(PCB LAST) ¬= NULL  THEN
                 NAME(PCB LAST NEXT) = NAME(PCB),
              RETURN,
           END,
           NAME(PCPTR) = NAME(PCPTR NEXT),
        END,
C
C      PCB  IS  LOWEST  PRIORITY  TAG  ON  END  OF  LIST
C
        NAME(PCPTR NEXT) = NAME(PCB),
        NAME(PCB NEXT) = NULL,
        NAME(PCB LAST) = NAME(PCPTR),
     CLOSE  PC_ENQUEUE,
```

13B

```
M   HEXCALC:
M   PROGRAM;
M      DECLARE INTEGER DOUBLE,
M                 INT1, INT2,
M      DECLARE INLINE CHARACTER(80),
M      DECLARE PLUS BOOLEAN,
M      DECLARE K INTEGER,
E                   ,
M      READALL(5) INLINE;
E        ,         ,
M      INLINE = TRIM(INLINE),
E                    ,
M      K = INDEX(INLINE, '+'),
M      IF K > 0 THEN
E           .
M         PLUS = TRUE,
M      ELSE
M         DO,
E                .
M              PLUS = FALSE,
E                          ,
M              K = INDEX(INLINE, '-'),
M         END,
E                                           ,
M      INT1 = INTEGER        (BIT     (INLINE        ));
S                    @DOUBLE     @HEX         1 TO K-1

E                                           ,
M      INT2 = INTEGER        (BIT     (INLINE        )),
S                    @DOUBLE     @HEX        K+1 TO #

E           .
M      IF PLUS THEN
M         INT1 = INT1 + INT2,
M      ELSE
M         INT1 = INT1 - INT2,
M      WRITE(6) INT1, CHARACTER    (BIT(INT1));
S                                 @HEX

M   CLOSE HEXCALC;
```

Appendix D

HAL/S Reserved Words

ABS
ABVAL
ACCESS
AFTER
ALIGNED
AND
ARCCOS
ARCCOSH
ARCSIN
ARCSINH
ARCTAN
ARCTANH
ARCTAN2
ARRAY
ASSIGN
AT
AUTOMATIC

BIN
BIT
BOOLEAN
BY

CALL
CANCEL
CASE
CAT
CEILING
CHAR
CHARACTER
CLOCKTIME
CLOSE
COLUMN
COMPOOL
CONSTANT
COS
COSH

DATE
DEC
DECLARE
DENSE
DEPENDENT
DET
DIV
DO
DOUBLE

ELSE
END
EQUATE
ERRGRP
ERRNUM
ERROR
EVENT
EVERY
EXCLUSIVE
EXIT
EXP
EXTERNAL

FALSE
FILE
FLOOR
FOR
FUNCTION

GO

HEX

IF
IGNORE
IN
INDEX
INITIAL
INTEGER
INVERSE

LATCHED
LENGTH
LINE
LJUST
LOCK
LOG

MATRIX
MAX
MIDVAL
MIN
MOD

NAME
NEXTIME
NONHAL
NOT
NULL

OCT
ODD
OFF
ON
OR

PAGE
PRIO
PRIORITY
PROCEDURE
PROD
PROGRAM

RANDOM
RANDOMG
READ
READALL
REENTRANT
REMAINDER
REPEAT
REPLACE
RESET
RETURN
REMOTE
RIGID
RJUST
ROUND
RUNTIME

SCALAR
SCHEDULE
SEND
SET
SHL
SHR
SIGN
SIGNAL
SIGNUM
SIN
SINH
SINGLE
SIZE
SKIP
SQRT
STATIC
STRUCTURE
SUBBIT
SUM
SYSTEM

TAB
TAN
TANH
TASK
TEMPORARY
TERMINATE
THEN
TO
TRACE
TRANSPOSE
TRIM
TRUE
TRUNCATE

UNIT
UNTIL
UPDATE

VECTOR

WAIT
WHILE
WRITE

XOR

INDEX

NASA

National Aeronautics
and
Space Administration

HAL/S