Design of a Verifiable Subset for HAL/S

James C. Browne
Donald I. Good
Anand R. Tripathi
William D. Young

FINAL REPORT

December 31, 1979

INSTITUTE FOR COMPUTING SCIENCE AND COMPUTER APPLICATIONS
The University of Texas at Austin
Austin, Texas 78712

TABLE OF CONTENTS

1.0 INTRODUCTION

Program verification has been a topic of intense research interest in recent years. The desire to construct proveably correct software has impacted computer science in various ways, for instance by motivating the use of program design methodologies such as modular programming. Various languages, e.g. Gypsy [1] have been designed with the prinary intention of constructing formally verifiable software. This was accomplished by incorporating into these languages only features amenable to modern verification techniques or, in many cases, by inventing new verification techniques for desirable constructs.

The research described in this report and the accompanying documents was an attempt to evaluate the applicability of such techniques to an existing programming language, HAL/S [2,3]. HAL/S is a general purpose high level language designed to accomodate the software needs of the NASA Space Shuttle project. This goal mandated a diversity of features for scientific computing, concurrent and real-time programming, error handling, etc. Many of these features, however, are not susceptible to existing techniques of program proving. Our concern was to examine the various features of HAL/S and evaluate them according to certain criteria for verifiability. The result, HAL/S/V, is a subset of the language which we believe consists only of those constructs which lend themselves to program verification.

Our task was viewed only as one of subsetting--that is, we have eliminated or restricted existing features and imposed a discipline on

the programmer rather than suggested new language constructs. This approach insures that the verifiable subset is downwards compatible with HAL/S and readily implementable utilizing existing HAL/S compilers and support facilities. Of course, to formally verify HAL/S/V programs it would be necessary to augment the language with specification capabilities and, preferably, design an automated verification system in the mode of the Gypsy verification system [1] designed and implemented at the University of Texas.

This report consists of three sections in addition to this introduction. The following section describes the criteria by which features were evaluated for inclusion into the verifiable subset. Section 3 examines individual features of HAL/S with respect to these criteria and provides justification for the omission of various features from the subset. In the final section, conclusions drawn from this research are presented and recommendations made for the use of HAL/S with respect to the area of program verification.

The documents which should be considered along with this one are the following.

1. HAL/S/V: A Verifiable Version of HAL/S: This document is the HAL/S/V counterpart of the HAL/S Programmer's Guide [2] and is derived by editing that document to reflect the changes which define HAL/S/V.

2. "An Overview of Differences Between HAL/S and HAL/S/V": This document is a list of these differences along with the justification for each. Entries are grouped according to the corresponding chapters in the HAL/S Programmer's Guide.

3. "Evaluation of Verifiability in HAL/S": This paper was a preliminary version of much of the material in this report. It was presented to the AIAA/NASA/ACM/IEEE Computers in Aerospace Conference II and published in the preceedings of that conference [4].

2.0 CRITERIA OF EVALUATION

Before examining individual features for inclusion into the verifiable subset it was necessary to establish criteria of selection. A few obvious criteria were readily available--aliasing was to be avoided, programs should be deterministic, language features should be axiomatizable. Other desirable characteristics from a verifiability standpoint were more subjective. Verifications should be of manageable complexity; programs should be readable; modularity was to be encouraged. The following emerged as areas to be considered in evaluating language features for verifiability: aliasing, axiomatizability, simplicity of verification, non-determinacy and implementation-dependency, and textual clarity.

2.1 Aliasing

In most program proof methods, assertions are made about the state of computation at certain points in the program. These assertions reference program variables. However, if there are various paths to reference a variable--various "aliases"--then an assertion made using one name may be invalidated by changing the value of the

variable using some other name. For instance, if x and y refer to the same storage location, then for the code sequence

```
|
|       x = 2
|       y = 3
|  /* Assert: x=2 and y=3  */
|
```

the given assertion is false. Such aliasing complicates the verification process by requiring the verifier to keep track of all aliases for each storage location.

2.2 Axiomatizability

Axiomatizability of a language feature refers to the ability to write a collection of rules which completely and unambiguously describes the results of applying that feature to any data item. Such descriptions are essential for automatic verification because the verification system manipulates symbolic quantities which represent arbitrary input values. One technique for assuring axiomatizability is to insist that language constructs which have mathematical counterparts conform to them as closely as possible. They will not, in general, conform entirely. Integer addition in HAL/S, for instance, is axiomatizable from a verification point of view with axioms borrowed from the mathematical domain. These can be assumed to hold even though the associative law of addition may be violated near the maximum integer representable on a given machine.

2.3 Simplicity Of Verification

In many cases even if it is possible to axiomatize a certain feature of the language and give a complete mathematical definition of its semantics, it may turn out to be a very tedious job to verify programs involving the construct. Language constructs should be easy to grasp and simple to axiomatize. Simplicity of axiomatization leads to ease in applying proof methods.

2.4 Non-determinacy And Implementation Dependency

It is desirable that the semantics of the language specify completely and unambiguously the results of an computation. This is unfortunately not always possible. For instance, in concurrent programming the order in which operations are performed may be non-deterministic. However, non-determinacy which results from the incomplete specification of the language semantics is unacceptable. The result is lack of clarity, possibility of error, and implementation dependency which requires the verifier to be concerned with implementation details. Escape from such considerations is one of the primary advantages of high level languages, not only from the point of view of the programmer but also from the point of view of the verifier.

2.5 Textual Clarity

Textual clarity has long been recognized as important in the design of programming language constructs; witness the long-standing desire for "self-documenting" languages. Language constructs should in every way encourage textual clarity. Operators should perform a clear and simple function with a minimum of hidden effects. Information should be available locally to decipher any language fragment. A statement which is so obscure as to be understandable only by resort to global declarations and the language manual probably indicts some language feature as too complex. A feature which is so complex that its meaning cannot be easily grasped is probable too complex to be easily verified.

3.0 EVALUATION OF HAL/S

3.1 Data Types And Structures

3.1.1 Data Types -

A primary determinant of the utility of any general purpose programming language is the ease with which different classes of data can be represented and operated upon. HAL/S addresses this issue by incorporating a diverse set of built-in data types with associated operations. Some of these data types (INTEGER, BOOLEAN, CHARACTER, BIT STRING) require only minor modifications to make them accessible to existing verification techniques. Others (EVENTS, NAMES) would be verifiable but only in a particularly restricted and disciplined form. Finally, for still others (SCALAR, MATRIX, VECTOR) no manageable

verification techniques are currently available. The data types which are either eliminated or significantly restricted in HAL/S/V are the SCALAR, MATRIX, VECTOR, EVENT, and NAME data types.

3.1.1.1 SCALAR, VECTOR, And MATRIX Data Types -

Most existing verification techniques require that assertions about the values of variables be inserted at various points in the program body. These values are manipulated according to well-defined axioms to yield results which are invariantly true of the computation whenever control reaches that point. Scalar (floating point) arithmetic does not lend itself well to these techniques.

The values which result from scalar operations are seldom exact. Each instance of a scalar operation potentially introduces an error the magnitude of which depends upon the type of operation, size of the operands, implementation, etc. Sophisticated numerical techniques and error analysis are needed to make valid and useful assertions about the results.

The axioms of real arithmetic do not in general apply to scalar operations. In this regard scalar arithmetic is unlike integer arithmetic for which there is a "ready-made" axiomatization available which yields exact values except at very large positive and negative values. Only a very complex set of axioms would suffice for scalar arithmetic. Such an axiomatization still would be of doubtful value since even a "correct" algorithm may yield vastly inaccurate results due to accumulating errors, catastrophic cancellation, etc. Such data dependent difficulties cannot easily be dealt with in axioms.

These considerations indicate that any scalar operation probably is not verifiable with existing techniques. Therefore, the SCALAR data type has been omitted from the verifiable subset.

The VECTOR and MATRIX data types of HAL/S are composed of scalar components. Consequently, they inherit all of the difficulties of scalars and, like scalars, are not present in the verifiable subset.

3.1.1.2 NAME Data Items -

The HAL/S NAME facility permits the declaration of "pointers" to data items, data structures, tasks, and program units. The ability to maintain pointers is a valuable feature of many programming languages. However, it presents a problem for verification unless the facility is severely restricted and disciplined. PASCAL, for instance, allows pointers only to un-named dynamically created objects. This permits aliasing of a limited sort since there may be several pointers to one data object. However, the verifier is assured that no named data item is also accessible via pointers.

In contrast, HAL/S treats a NAME data item as merely another name for the designated object. Consequently aliasing of the worst kind can result. Access is permitted to an object via any number of names and assertions regarding the object can be written using any of them. Assertions made using any of the aliases may be falsified by accessing the object via any other, a fact not apparent from the program text. Therefore, the verifier is constrained to keep track of all aliases for every data object. To eliminate these difficulties, the HAL/S NAME facility has not been included in the verifiable subset.

3.1.1.3 EVENT Data Items -

A HAL/S event is a Boolean valued data item whose value is visible at any instant to the HAL/S run-time executive. An event with the optional latching property may be set to either true or false. An event without this property is normally false but may become transiently true when so specified.

HAL/S/V permits declaration only of events with the latching property for the following reason. Events are useful primarily for scheduling and synchronization purposes--a process may be scheduled when the value of some event data item is set to true. If e1 and e2 are event data items and T a task, the following is legal.

```
Schedule T on (not e1 and e2);
```

However, for transient events which become true for at most an "infinitesimal" time, such expressions do not make sense. There is no obvious interpretation for "not e1" when e1 is only transiently true. Therefore, the primary utility of events is lost for transient events. Transient events do not seem to serve an additional useful purpose in the language and have been excluded.

An additional restriction is necessary for event data items. Any variables, including events, in HAL/S are permitted to belong to lock groups whose members may be accessed only within update blocks. This is the HAL/S mechanism for implementing mutual exclusion. However, no

WAIT or SCHEDULE statements are allowed within update blocks. Therefore, the obvious use one would make of events is not permitted if they belong to lock groups. In HAL/S/V event data items are not allowed to belong to any lock group.

3.1.2 Data Initialization -

HAL/S allows optional initialization of almost any declared data items including arrays and structures. Most initializations are verifiable, being semantically equivalent to a block of assignment statements inserted just after the declarations. However, two areas present problems for verication: partial initialization and static initialization for reentrant procedures.

3.1.2.1 Partial Initialization -

Partial initialization of arrays and structures violates our criterion of simplicity of verification. Consider the following code segment:

```
|
|  Declare A array (50) integer
|     Initial (1,2,3,45#,4,5);
|  /*  Assert:  A$i = A$i  */
|
```

The given assertion is a tautology and should be uniformly true. However, the elements of A indexed 4 through 48 have undefined values. To verify a program containing such partially initialized data structures the verifier must have rules for dealing with undefined values of every type and keep track of all locations which contain

undefined values. A substantial simplification can be gained if data structures are such that either all component values have been initialized or all are undefined. To gain this simpification, partial initialization is not included in HAL/S/V.

3.1.2.2 Static Initialization For Reentrant Procedures -

HAL/S permits data to be initialized in either of two ways. The keyword AUTOMATIC specifies that initialization is to take place upon each entry to a block. STATIC signifies that initialization occurs only upon the first entry, with each subsequent entry inheriting whatever values were left in the data items by its predecessor. HAL/S/V permits both types of initialization with the single restriction that static initialization is not permitted for any procedure that is declared to be REENTRANT. This stipulation ensures that two activations of a procedure are not updating a common data store concurrently. Such updating introduces all of the difficulties of concurrency into procedures, an area where the simpler verification techniques for sequential programming should be applicable.

3.2 Arithmetic And Computation

Arithmetic in HAL/S/V is substantially simplified by the omission, cited earlier, of the SCALAR, MATRIX, and VECTOR data types and the associated operations--scalar multiplication, matrix multiplication, matrix inversion, vector cross product, etc. However, the remaining computational resources of the language still involve features and permit interactions which violate the criteria of

verifiability outlined in section 2. We discuss these in the areas of expressions, assignments, and array processing.

3.2.1 Expressions -

One of our criteria in evaluating for verifiability is textual clarity. HAL/S violates this criterion in the evaluation of expressions principally in its wide range of implicit type and precision conversions. Mixed operations are a convenience for the expert programmer but present a pitfall for everyone else. Such operations have effects which are textually indicated only in the sense that they are implicit in the data declarations and the definitions of the operations. But the individual instance of an operation is far enough removed from these to be susceptible to error and difficult to decipher if implicit conversions are present. Therefore, HAL/S/V disallows any type of implicit conversions. The implications for expression evaluation are the following.

1. There is no "mixed-mode" arithmetic in HAL/S/V. The presence in the language of a comprehensive set of explicit type conversion functions ensures that this restriction is at worst an inconvenience and not a significant loss in language utility.

2. There is no mixed precision arithmetic. Again explicit conversion functions can effect the result of mixed precision with a substantial gain in clarity and programmer control.

These restrictions are intended to be comprehensive. The absence of the SCALAR data type in HAL/S/V removes the possibility of scalar-integer operations but there are other mixed mode operations in HAL/S. HAL/S/V forbids, for instance, catenation of character and non-character data items and relational comparisons of bit strings of differing lengths.

3.2.2 Assignments -

3.2.2.1 Coercion Across Assignments -

In keeping with the principle espoused above that all conversions should be explicit, HAL/S/V permits no implicit type or precision conversions across assignments. Such conversions constitute changing information under the guise of transferring information. In many cases information is lost and in some cases spurious information is added as, for instance, when a bit string is assigned to a bit string variable whose declared length is greater. Requiring explicit conversions does not alter this process but at least guarantees that the programmer as well as the users and readers of his program are aware of it. As with expressions, a complete set of type and precision operators in HAL/S provides any capability which might otherwise be lost by disallowing implicit conversions.

3.2.2.2 Multiple Assignments -

Multiple assignments present a difficulty for verification only if the results cannot be translated in a straightforward fashion into a sequence of single assignments. This is not always possible since HAL/S does not specify an order in which multiple assignments are performed. Consider the code sequence:

```
|
|     i = 2;
|M    A ,i = 3;
|S     i
|
```

Whether 3 is assigned as the value of A(2) or of A(3) depends upon the order in which the multiple assignment is performed.

The simplest way to eliminate this non-determinacy is to insist upon some set order of assignment. However, this would be an addition to the language semantics and in keeping with the view that HAL/S/V should be strictly a subset of HAL/S we impose the following rule instead: no variable which appears on the left hand side of a multiple assignment may appear on both the main line and the subscript line.

3.2.3 Array Processing -

HAL/S contains extensive array processing capabilities which have been excluded from the verifiable subset. HAL/S allows nearly all operations which can be legally performed on the components to be applied to an entire array. For instance, if A1, A2, and A3 are declared to be arrays of dimensionality 3 x 4, the statement

```
|
|   A3 = A1 + Div (A2, 5);
|
```

has the same effect as the sequence of statements

```
|
|   Do for i = 1, 3;
|   Do for j = 1, 4;
|M     A3     = A1     + Div (A2    , 5);
|S       i,j      i,j            i,j
|   End;
|   End;
|
```

However convenient the first version may be, the second is to be preferred on the basis of textual clarity.

1. The number of times operations are performed is textually apparent and not dependent merely on the dimensionality of the operands.

2. The order in which operations are performed is determined by the programmer rather than by the compiler. For operations such as assignment which may have side effects on the operands, this may affect the result of the computation. It is not enough that information about the order of assignment is available somewhere in the language description.

3. The operations involved need not be treated as generic operations and the proof rules can simplified accordingly.

4. In the above example, either of A1 or A2 could have been single valued data items. In such a case the addends would have been size-mismatched and a form of implicit coercion required to make the operands compatible.

These considerations along with the fact that little additional effort is involved in explicitly programming the operations on a component-wise basis has led to the exclusion of such array processing features from HAL/S/V. The only operation permissible on an entire array in HAL/S/V is assignment to another array of identical dimensionality.

3.3 Concurrency

HAL/S has certain explicit as well as implicit (at least conceptually) provisions for concurrent processing. Multiprocessing can be specified explicitly using the scheduling of concurrent processes, whereas certain other language constructs such as the array processing facility provide implicit concurrency. These array processing features have been discussed in Section 3.2.

3.3.1 Access Control -

HAL/S provides two mechanisms for synchronizing processes, UPDATE blocks and EVENTS. Events have been discussed in Section 3.1.1.3. Update blocks are like critical sections. A shared data object declared to belong to some lock group may only be accessed inside an update block. During the execution of an update block a lock is placed on all data belonging to those lock-groups which are accessed inside the update block. At any instant at most one process can be accessing data belonging to a particular lock-group. In HAL/S the use of such lock-groups on shared data is optional; however, in the verifiable subset we make it mandatory that all shared data (global

and compool data) belong to some lock-group. This insures that all sharing is done within update blocks. There are two reasons for this. It textually isolates the places where the concurrency impacts the proof process. Also, it guarantees that accesses to shared data be within indivisible program blocks. This prohibits many of the problems commonly associated with proving concurrent programs and allows proofs on the level of update blocks rather than on the individual statement level.

Update blocks in HAL/S cannot contain any i/o, wait, or schedule statements or calls to routines declared outside the update block. Additionally, task blocks cannot be defined inside an update block. In HAL/S/V, because direct accessing of non-local data is not permitted to routines this restriction could be relaxed; an update block could call routines defined outside the update block. However, such routines must not contain any i/o, wait, or schedule statements or call any other routines containing such statements. This kind of checking could be done at compile time.

3.4 Flow Of Control

3.4.1 Program Flow-control -

The flow of control facilities in HAL/S present few difficulties for verification. One feature which has been omitted from the verifiable subset is the GOTO statement. GOTOs in HAL/S are actually quite controlled allowing branches only within or out of a block--never into a block. However, with GOTOs present in the language, it is difficult to assert the values of variables after any

labelled statement since control may have branched from various places within the block. Although programs with this feature probably could be proved, because of the complexity involved GOTOs have been omitted from HAL/S/V. As compared to the GOTO, the EXIT statement is much more restricted in changing the flow of execution; this statement always branches to the end of a block. This form of branching has been retained.

All forms of repetitive statements in HAL/S except for one are acceptable from the verification point of view. Statements of the form

```
|
|  DO FOR var = exp1, exp2, ..., expn
|
```

are not retained in HAL/S/V. In this kind of iteration the control variable is successively assigned the values of the given expressions and the following statement group is executed for that value of the control variable. The characteristic method for verifying loops, however, is to use an inductive proof rule which assumes that the control variable is assuming some arithmetic progression of values. If this is not the case, each iteration must be separately verified. The increase in generality does not justify this increase in complexity of verification. Therefore, this feature has been omitted.

3.4.2 Error Recovery -

HAL/S provides mechanisms for user-level specification of error handling routines. Besides the system defined error conditions, there can be user-defined error conditions. Recovery under error conditions

is specified using an ON ERROR statements by which the programmer can state what actions need to be taken if a given error condition is raised. When the error occurs, control is transferred to this error handler after which execution resumes at the following executable statement.

In HAL/S/V we suggest the following discipline on the programmer in order to write verifiable programs. All ON ERROR (or OFF ERROR) statements must be placed in the beginning of the block (or routine) following the declaration of local variables. ON ERROR statements would normally be preceded by the some "back-up" block to store the values of ASSIGN parameters and some critical local variables which might be needed to be restored in case of any attempt at roll-back and recovery. Additionally, inclusion of error-recovery mechanisms in the verifiable subset mandates the need for specification methods to distinguish between normal exits and exits under error conditions.

3.4.3 Real-Time Programming -

A task block is the static counterpart of a process. HAL/S has numerous real-time scheduling constructs. One process is designated as the primal process which schedules other task or program processes. These in turn can schedule other tasks or program processes names of which are visible according to the scoping rules.

The problems associated with the proof of real-time programs have two dimensions, the proof of concurrent programs and the proof of time-dependent constructs. The set of problems associated with the second dimension are hard to approach because of their sensitivity to

the hardware underlying the implementation.

The most promising approach to the proof of concurrent programs in HAL/S seems to be the use of the proof methods of Owicki and Gries. Proofs of this kind necessitate knowledge of all the concurrent processes at any given instant which operate on some shared data. This can become a very complex problem in HAL/S because of the quite general scheduling rules of HAL/S. Any process may schedule another task or program process. In HAL/S/V all scheduling is restricted to the primal process level. This makes it possible to assert at any point in the primal program the possibly active concurrent processes.

Processes can be assigned priorities at the time of scheduling which can be changed dynamically. HAL/S does not restrict which processes have the capability to change the priorities of other processes. In the subset no process other than the primal process can change its own or any other proccess' priority.

At the time of scheduling a process can be either declared INDEPENDENT or DEPENDENT. The existence of a dependent process is contingent on the scheduling process being in the active state. In HAL/S/V there is no need to make this disinction since all processes are dependent on the primal process alone. A process can terminate itself or can be terminated by the primal process only. On abnormal termination a process must execute a "clean-up" block of code which might be specified by some construct such as ON TERMINATION similar to ON ERROR specifications.

3.5 Program Units

3.5.1 Procedures And Functions -

Considerations of modularity and incremental development make it desirable that we should be able to prove procedures and functions independently of the context in which they are called. In other words, we would like to see the abstraction facilities of these units retained during proofs as well as during program development. Starting with the"entry" specification one must be able to prove the "exit" specification only with reference to the code of the routine body. Some features of HAL/S which prohibit this are non-local referencing, the passing of shared data objects as parameters, and the run-time dependency of some HAL/S constructs.

3.5.1.1 Non-local Referencing -

The static lexical structure of HAL/S programs is block-oriented like Pascal and Algol. The name-scope rules are determined by the block-structure rules such as in those languages. Such scope rules permit a routine to access non-local variables other than those passed as parameters. Non-local referencing causes serious difficulties for verification primarily because it permits aliasing. A data item which is passed as a parameter may be referenced via the formal parameter name or may be referenced directly. Also, non-local referencing violates the principle that routines should be understandable and verifiable as much as possible in isolation. If such referencing is allowed, it is impossible to verify routines without a knowledge of the global environment in which they will be called. This inhibits

factoring the proofs of programs into manageable subunits and can make the proof of large systems prohibitively complex.

Non-local referencing also makes HAL/S functions unlike their mathematical counterparts in the sense that the value returned is not dependent solely upon the values of the parameters. This allows functions to have side-effects and complicates the verification substantially.

3.5.1.2 Parameter Passing -

HAL/S has facilities to provide two kinds of parameters to routines, INPUT and ASSIGN parameters. For input parameters the routine has read-only capability and these parameters cannot be altered during the course of routine's execution. In contrast, assign parameters can be assigned new values and thereby can be changed by the routine. In order to avoid side effects, both in HAL/S and in the verifiable subset functions are not allowed assign parameters.

HAL/S does not specify whether input parameters are implemented by passing a copy to the routine or by passing a pointer to the original copy with a read only capability. If the implementation uses the second alternative then side effects due to aliasing may occur if the same variable is passed as both an input and an assign parameter to a routine at one invocation. In order to overcome such implementation dependent non-determinacy in the results, in HAL/S/V the input and assign parameter lists must be disjoint. Again to avoid any aliasing, the actual parameters in the assign parameter list at any call point must all be distinct.

What happens if a procedure, inside its body, passes one of its input parameters as an assign parameter to a procedure called within? Such a situation potentially causes difficulties if input parameters are passed by setting pointers (reference) with read-only capability. No side-effects occur if input parameters are implemented by giving a new copy to the called routine. In HAL/S/V we require that an input parameter to a routine cannot be passed as an assign parameter to any procedure called within the routine's body.

3.5.1.3 Procedures And Shared Data -

In HAL/S the access-control of shared data in a concurrent execution environment is achieved using lock-groups as discussed earlier. A procedure or function can be declared to have formal parameters belonging to certain lock-groups. At the call point the lock-group of the actual parameter must match the lock-group of the formal. In HAL/S lock-group parameters can only be referenced within UPDATE blocks which enforce mutual exclusion on the variables of the lock group during its execution.

Proving procedures with shared data as parameters poses severe problems; such procedures must be proved for each invocation since, in order to prove the correctness of the procedure, one must know which other concurrent processes may operate on that shared object during the procedure call. Such information depends on the point where the procedure is being called and also depends on the shared data object passed as parameters. This implies that in order to prove the procedure one must prove it for each call in which a shared data item

is passed to it as a parameter. This destroys during proof time the abstraction and modularization facilities provided by routines. To alleviate this difficulty we suggest that in HAL/S/V procedures (or functions) with lock-group parameters be executed as critical sections thereby enabling their proofs as stand-alone units. We propose to use the UPDATE (lock-group) facility, as provided in HAL/S, for this purpose.

One implication of this rule is that a procedure or function with locked parameters must adhere to the following five restrictions which are imposed by the language upon update blocks. An UPDATE block may not contain statements of the following kind:

1) input/output
2) wait or schedule statements
3) another update block
4) task blocks, but it may contain definition of new procedures or functions
5) any procedure or function defined outside the update block.

In order to make these restrictions effective an additional restriction must be placed on procedures called within these "update-procedures". One alternative is to insist that a procedure with formal parameters belonging to a lock group cannot call any procedure or function defined outside its block which does not have any lock-group parameters--such a procedure may contain a 'schedule' or 'wait' statement in its body. An alternative is to refuse to permit any procedure or function to have 'wait' or 'schedule' statements in its body. However, in itself that would not be

sufficient and one would have to further disallow i/o statements within procedures. Either alternative is unattractive because of the loss of useful facilities such as procedures for scheduling and synchronization.

A third alternative is enforced in HAL/S/V. Though the procedures and functions with lock-group parameters are treated as UPDATE blocks, they may call other procedures and functions provided the called routines conform to the restrictions on update blocks mentioned earlier. To clarify, a procedure with lock-group parameters can call another procedure with lock-group parameters although both these are treated as update blocks. This is unlike the general case in which update blocks within update blocks may lead to potential deadlock situations because of the scope rules of HAL/S/V. Since the only variables accessible within a routine are those which are explicitly passed as parameters to it, the only procedures or functions which can be called are those which need parameters of exactly the same lock-groups (or a subset of these) as those passed to the calling routine. Also, a compile-time check can be made to ensure that a procedure with lock-group parameters does not call any routine which contains statements of the five categories prohibited in update blocks. This effectively creates two classes of routines one, which contain statements like wait, schedule, i/o, etc., and other routines which don't contain such statements.

3.5.1.4 Run-time Dependency -

Run-time dependent system functions, like RUNTIME and DATE, cause certain unique problems for verification as do non-memoryless functions like RANDOM. Such functions have results which are not dependent solely upon the values of the parameters. Hence two invocations of the function will return different values even with identical parameter lists. If a user-defined function during its execution calls some system-defined, run-time dependent function, then it is possible for the user-defined function also to return different values at different invocations even though the same parameter values are passed to it. Therefore in HAL/S/V one is not allowed to reference run-time dependent functions within routines. This implies that run-time dependent functions can only be called at the outermost level.

3.5.2 Tasks And Programs -

The outermost level of programs may contain task declarations and definitions. The proof of a task block is highly dependent on the scheduling in the program. In order to prove a task one must have knowledge of which other tasks execute concurrently and share data with it. More about these problems in discussed in the sections on real-time scheduling.

In HAL/S (and also in the subset) tasks are the static counterpart of processes. A task block may not be nested within another task block. In HAL/S, a task can schedule another task. Although a very useful feature from a functional abstraction point of

view, this makes the proof of concurrent tasks very complex. In HAL/S/V we propose to restrict all scheduling to the outermost level of programs. In this way, when proving concurrent tasks one does not have to worry about any subtasks which may be scheduled within them.

The most promising way to approach the proof of concurrent tasks in HAL/S seems to be the use of the techniques proposed by Owicki and Gries. In this method, in order to prove concurrent programs, one has to show that the execution of statements in one program does not change the pre or post conditions of statements in the other programs. This is called non-interference between the programs. In HAL/S/V, this kind of non-interference only needs to be shown between programs at the level of update blocks. After showing non-interference, each of the programs can be proved separately as a sequential program.

3.5.3 External Declarations -

In HAL/S external declarations permit users to define external program units, procedures, functions, and global data within separately compiled blocks called compools. In HAL/S/V these have all been retained except that all compool data items must be declared to belong to some lock-group. Also, procedures and functions may not contain templates for compool data since no non-local referencing by procedures or functions is allowed in HAL/S/V.

3.6 Textual Integrity

The behavior and hence the verifiability of a program is entirely dependent upon the program text. Changing one token or even one letter can lead to incorrect results or invalidate a previous verification. Therefore, features are to be avoided which lead to textual alterations. The major offenders in HAL/S are replace statements and replace macros.

These facilities permit almost unrestricted textual modification. The effect of each such statement is to present the verifier with two programs to verify--the original and the modified version. Depending upon the extent of replacement these verifications may be nearly identical or quite different. In the worst case the difficulty of verification may be exponential in the number of replace statements appearing. Such a severe penalty seems to outweigh the benefits of these facilities. Therefore, there are no replace statements or replace macros in HAL/S/V.

4.0 CONCLUSIONS

It is clear that the verifiable subset, HAL/S/V, contains much less than the full HAL/S language. The absence of the scalar, matrix, and vector data types, replace statements, partial initialization, the name facility, replace macros, inline functions, transient events, general array processing features, all non-local referencing, and function side-effects are major omissions. There is, however, much that has been retained in HAL/S/V including the integer, character, Boolean, and bitstring data types, most control features of HAL/S,

arrays and structures, subprogram definitions, real-time and concurrent programming constructs, and all input/output facilities of HAL/S. There are many significant programs that can be expressed entirely in HAL/S/V. Also because HAL/S/V is a strict subset of HAL/S, it is possible to express in HAL/S/V critical segments of a complete HAL/S program.

In order to obtain verified programs, a language that will support verification effectively is required. Is HAL/S/V such a language? In principle, "Yes". By expressing programs strictly within HAL/S/V, certain limited amounts of manual verification could be done. However, any sizeable amount of verification will require verification tools (verification condition generators, algebraic simplifiers, theorem provers, etc.). The building of these tools is a large effort, and it is doubtful that building such tools for HAL/S/V would be cost effective in view of the other alternatives.

One alternative that should be considered is the translation of an existing verifiable language with existing tools. Gypsy, for example, is a fully verifiable language with an extensive set of verification tools. Building verified programs in Gypsy using existing tools and then mechanically translating the verified Gypsy program into HAL/S is an alternative that should be weighed against the sizeable cost of building verification tools for HAL/S/V.

Another alternative that should be considered seriously is the use of the Ada language in place of HAL/S. We have begun a verifiability analysis of Ada similar to the study of HAL/S. Although Ada is by no means fully verifiable, it appears to be significantly

more supportive of verification than HAL/S. Although there are currently no more verification tools for Ada than for HAL/S/V, it is almost certain that some of these tools will be developed. Ada also provides several important dvantages in addition to verifiability. Ada appears to provide all of the capability provided by HAL/S. Ada programs can be interfaced to existing HAL/S programs. Ada is a much more modern language design than HAL/S. It incorporates much of the structured programming and software engineering technology that has been developed in the last 5-10 years. Ada is expected to receive widespread use and support within the DoD. The general use of Ada in favor of HAL/S, therefore, is an alternative that, we believe, should be seriously considered.

REFERENCES

1. Good, D.I., R.M. Cohen, C.G. Hoch, L.W. Hunter, D.F. Hare, "Report on the Language Gypsy, Version 2.0", The University of Texas at Austin, ICSCA-CMP-10 (September, 1978).

2. HAL/S Programmer's Guide - Vol. 1 2", Intermetrics Inc., 701 Concord Ave., Cambridge, Massachusetts 02138, December 16, 1974.

3. HAL/S Language Specification", Intermetrics Inc., 701 Concord Ave., Cambridge, Massachusetts 02138, June 16, 1976.

4. Young, W.D., A.R. Tripathi, D.I. Good, J.C. Browne, "Evaluation of Verifiability in HAL/S", Proceedings of Computers in Aerospace Conference II, 1979, pp.359-366.

Overview of Differences

Between HAL/S and HAL/S/V

James C. Browne
Donald I. Good
Anand R. Tripathi
William D. Young

December 31, 1979

INSTITUTE FOR COMPUTING SCIENCE AND COMPUTER APPLICATIONS
The University of Texas at Austin
Austin, Texas 78712

Overview of Differences Between HAL/S and HAL/S/V

This document lists all changes to HAL/S incorporated in the HAL/S/V verifiable subset. Changes are grouped according to the chapters of the HAL/S Programmer's Guide. Reasons for the changes are included though more elaboration and examples can be found with the descriptions of the constructs in the document "HAL/S/V: A Verifiable Version of HAL/S."

1.0 STRUCTURE OF HAL/S

1. A procedure or function block can reference a non-local variable only if it is passed as an actual parameter at the call site. This does not apply to global constants.

   There are several reasons why this restriction is imposed. First, verification of large programs is a difficult process. This difficulty is compounded if the program is not written in such a way that it can be decomposed into smaller modules which can be independently verified. An unrestricted facility to reference global data inhibits such decomposition since the behavior of every module which references global data is affected by the global environment in ways which may be evident only with detailed analysis of the progam text.

   In the case of functions, it is desirable that the behavior be describable in terms of a time invariant functional (in the mathematical sense) relationship between the parameters and the output value. This becomes impossible with non-local referencing.

   Finally, one of the most pernicious problems in the way of verifying any program is aliasing--data items having more than one name by which they can be referenced. Assertions made concerning a variable under one name then may be invalidated by changing the value of that variable under any of the aliases. Non-local referencing allows aliasing since global data may be referenced directly and also through the parameter list.

2.0 HAL/S SYMBOLOGY

1. Arithmetic literals are of the form +ddddd where ddddd

   represents an arbitrary number of decimal digits.

   The INTEGER data type is the only arithmetic data type of HAL/S/V. Therefore, scalar constants are not required.

3.0 A HAL COMPILATION--THE PROGRAM BLOCK

HAL/S/V makes no explicit changes to the material in Chapter 3 of the HAL/S Programmer's Guide.

4.0 DATA DECLARATION

1. HAL/S/v possesses no SCALAR, MATRIX, or VECTOR data types.

   Automatic verification of a program requires that the effect of every language construct upon the program state be symbolically representable in an exact way. Only in that way can meaningful assertions be constructed to describe the progam state. The SCALAR data type (the HAL/S representation of floating point numbers) is not so representable. SCALAR operations usually introduce some error term whose magnitude depends upon the particular object machine, the magnitudes of the operands, the operation involved, etc. Verification techniques cannot handle such contingencies. Therefore, the SCALAR data type is not included in HAL/S/V.

   The MATRIX and VECTOR data types, being composed of SCALAR elements, are perforce also not present in HAL/S/V.

5.0 REPLACE STATEMENTS

1. There is no replace statement in HAL/S/V.

   The HAL/S replace statement permits almost unrestricted textual modification. The effect of each such statement is to present the verification system with two programs to verify--the original and the modified version. Depending upon the extent of replacement these verifications may be nearly identical or quite different. In the worst case the difficulty of verification may be exponential in the number of replace statements appearing. Such a severe penalty seems to outweigh the benefits of this facility.

6.0 DATA REFERENCING AND SUBSCRIPTING

1. Component subscripting applies only to CHARACTER data items since MATRIX and VECTOR data items are not permitted in HAL/S/V.

7.0 EXPRESSIONS

1. The only arithmetic operations recognized in HAL/S/V are exponentiation (with positive exponents), multiplication, addition, and subtraction or negation. Division is integer division using the system function DIV. The following operations do not appear because they operate on data types or return data types not supported in HAL/S/V: exponentiation with negative or fractional exponents,

inversion, transposition, vector cross product, and floating point division.

2. No implicit precision conversion is permitted in HAL/S/V.

   We feel that a reasonable requirement on any programming language be that the effects of each construct be transparent. This leads to programs which are more likely to be correct and which are easier to read and understand.

   Operands of differing precision are essentially of different types. Implicit conversions mask from the programmer and the reader the fact that in the conversion process information may be lost or spurious information added. Requiring that the programmer explicitly perform all precision conversions explicitly using the conversion functions provided by the language forces him to recognize this possibility and perhaps compensate for it.

3. The operation of catenation operates on CHARACTER data items only.

   Catenation of non-character data items is useful primarily for output purposes. However, it violates the principle espoused above that all conversions should be explicit. Operators exist in the language for explicitly converting between data types. Therefore, no power is lost by imposing this requirement while a great deal of clarity is gained.

8.0 ASSIGNMENTS

1. HAL/S/V allows no implicit type conversions across assignment statements. Therefore, the declared type of the receiving data item must be that of the expression on the right hand side of the assignment statement.

   This requirement is in keeping with our view that language constructs can be powerful yet without doing anything behind the programmer's back. Type and precision conversions across assignment statements constitute changing information under the guise of transferring information. The change should be clearly indicated. A complete set of type and precision conversion operators in HAL/S provides any capability which might be lost by disallowing implicit conversions.

2. Arithmetic assignments are only of type INTEGER since MATRIX, VECTOR, and SCALAR data types are not part of HAL/S/V.

3. In accordance with 8.1, the right hand side of CHARACTER assignments must be CHARACTER valued expression.

4. No variable appearing on the left hand side of a multiple assignment may appear on both the main line and the subscript line.

9.0 CONDITIONAL STATEMENTS AND BRANCHES

1. To make Section 9.1 consistent with Section 9.3 the ability to label the "true part" or the "false part" of a conditional has been removed.

2. No comparisions involving operands of type INTEGER of differing precision are allowed.

   Operands of differing precision are essentially of differing types. Therefore, implicit conversion is required for the comparison to take place. All implicit conversions are disallowed in HAL/S/V for the reasons cited in 7.2 and 8.1.

10.0 STATEMENT GROUPS

1. Since SCALAR is not a legal data type in HAL/S/V the initial, and final values of the DO FOR statements must be INTEGER valued expressions and the control variable must be an unsubscripted INTEGER data item.

2. The increment may be +1 or -1 only. This simplifies construction of loop assertions without severely restricting the programmer.

3. At the end of the final cycle, the control variable has the value received after that cycle.

   Although it is bad programming practice to do so, there is no language imposed requirement that the loop control variable not be used after exitting from the loop. For assertion purposes then it is desirable that the value be determinate.

4. A second form of the DO FOR statement in which values of var are listed is disallowed in HAL/S/V.

   This alternate form does not in general permit specification of a loop invariant in any clear way. This is because the value assumed by the control variable may be such that one execution of the loop is related to the next execution in a very tenuous way, if at all. Also, the control variable assumes for each cycle the value of an expression which is evaluated immediately prior to that cycle of execution. Hence, the behavior of any loop execution may be dependent upon prior executions in very unclear ways.

5. The exp of a CASE must be an INTEGER valued expression since the SCALAR data type is not present in HAL/S/V.

11.0 PROCEDURES AND FUNCTIONS

1. In contrast to HAL/S, there is no default type for functions.

   The HAL/S default type is single precision SCALAR. This data type does not exist in HAL/S/V.

2. Since HAL/S/V allows no implicit type or precision conversions, actual parameters to a function or procedure must match the formal parameters exactly in type and precision.

3. There are no SCALAR, MATRIX, or VECTOR parameters since these data types are absent from HAL/S/V.

4. The notion of a function is restricted to make it conform to the mathematical notion of function in the following ways: HAL/S/V allows accessing of non-local data only if it is explicitly passed as parameters. Since functions have only input parameters, this prohibits functions from having side effects. An additional restriction is necessary to enforce this. Namely, procedures called inside function blocks may not have as assign parameters the input parameters of the function. Functions should return values which depend only on the values of the input parameters and are not time dependent. Therefore, functions may not call time dependent system routines such as RUNTIME, DATE, PRIO, etc. Also they may not call user defined procedures which call these routines.

   These restrictions permit the behavior of functions to be completely described in terms of a functional (in the mathematical sense) relationship between input and output values. Thus, HAL/S/V functions become implementations of mathematical functions in a true sense. The advantage of this is that, once proved, the behavior of the function is entirely circumscribed no matter what the circumstances of call. In such a case a function may be considered as a "black box" which produces a unique output value for values in the input domain. Any details of implementation can be hidden from the calling environment.

5. For procedures, the input and assign parameter lists must be strictly disjoint. This prohibits the altering of any input parameter during the execution of the procedure and preserves the distinction between input and assign parameters.

   It is desirable that the effects of a function can be registered solely by the value returned and the effects of a procedure can be described solely in terms of a set of changes to the assign parameters. Acheiving this ideal is the reason for several of the restrictions we have imposed. One way it can be violated is if the input parameters to a routine can be altered during the execution of the routine. If a data item appears as both an assign and an input parameter then it can be changed during execution even though it appears as an input parameter. This possibility is

disallowed.

6. No more than one part of a structured object may appear in the list of assign arguments and no part of a structured object may appear in the input list if any part of the object appears in the assign list.

   For structured objects two parts which are ostensibly distinct (for instance A$i and A$j) may in fact refer to the same storage locations. If these appear one in the input list and one in the assign list, then the problem mentioned above may occur. In any case, dangerous aliasing would result. Hence only one part of a structured object may appear in these situations.

7. No assign argument may be an input argument or any part of an input argument of an enclosing procedure block.

   Without this restriction an input parameter could be changed in value by a called routine.

8. The expression returned by a function must match exactly the declared type and precision of the function.

   No implicit precision conversion is allowed in HAL/S/V.

12.0 INPUT/OUTPUT STATEMENTS

Chapter 12 is concerned with the input/output facilities of HAL/S. HAL/S/V incorporates these entirely. Hence, this chapter has been left out of the HAL/S/V manual, being identical to Chapter 12 of the HAL/S Programmer's Guide.

13.0 REAL TIME PROGRAMMING I

1. Statements involving a task block must always follow the block definition.

2. The priority assigned a process at scheduling must be strictly lower than that of the scheduling process. Assigning a priority of (PRIO - c) where c is some non-negative integer such that (PRIO - c) > 0 is the suggested means of accomplishing this.

3. All of the numeric parameters in SCHEDULE and WAIT statements are INTEGER valued expressions rather than SCALAR expressions.

   HAL/S/V contains no SCALAR data type.

4. No process may be terminated by execution of a TERMINATE statement if it or any of its dependent processes updates

global data.

5. There is no UPDATE PRIORITY statement allowed in HAL/S/V.

6. As noted in Section 11.4.4, the functions RUNTIME and PRIO may not be invoked by any function or any procedure called by any function.

7. The use of a process name as a Boolean variable to indicate the state of a process is not permitted in HAL/S/V.

14.0 SUMMARY OF PART I

Chapter 14 is simply a summary of the contents of Part I, the first 13 chapters.

15.0 COMPOOLS AND COMSUBS

1. No external function or external procedure may have compool templates or access compool data. This is implied by the HAL/S/V scoping rules which do not allow procedures or functions to access non-local data except that passed explicitly as parameters.

2. Identifier names used to declare data in a compool body may not duplicate the names of identifiers used to declare data at the outermost level of programs having a template for that compool.

   To allow duplication leads to ambiguity of reference in the program block. An appearance of the duplicated name in the main program may refer to either the program's global data item or to the compool data item. HAL/S scoping rules apparently are not sufficient to disambiguate this situation.

3. Replace statements are disallowed in external procedures as in other places in the language.

16.0 ADDITIONAL DATA INITIALIZATION FORMS

1. Partial initialization of data is not permitted.

2. STATIC initialization of data is not permitted for reentrant procedures.

   Static initialization for a procedure implies that each entry is saddled with whatever data values the previous entry left behind. This presents a problem for verification generally since the procedure's proof must take into account the

effects preceeding calls may have had upon local data.For reentrant procedures the difficulty is enhanced since the data may be being acessed by several invocations concurrently. Thus the proof requires knowledge not only of preceeding calls which have completed but also of preceeding and suceeding calls which are concurrently active. Thus the difficulties which adhere to the proving of concurrent programs are inherited by procedures--which should be proveable sequentially. Therefore, it is required that every entry to a reentrant procedure have its own copy of the local data. This is accomplished by allowing only AUTOMATIC initialization.

17.0 BIT STRINGS

1. The operations called "conjunction" and "intersection" in the HAL/S Programmer's Guide are called "disjunction" and "conjunction," respectively, in the HAL/S/V document to conform to standard logical usage.

2. Two bit string operators compared by the OR or AND operations must be of equal length.

3. The operands of a bit string assignment must be of equal length.

4. Two bit strings are considered unequal if they are of unequal length.

5. A bit string actual parameter to a function or procedure must be of the same length as the declared length of the formal parameter. This applies both to input and assign parameters.

6. A bit string value specified at the RETURN statement of a function must match in length the declared value of the function.

18.0 MULTI-DIMENSIONAL ARRAYS

HAL/S/V makes no explicit changes to the material in Chapter 18 of the HAL/S Programmer's Guide.

19.0 STRUCTURES

1. A structure template may never possess a node of the same structure type. Whether or not this is possible in HAL/S is unclear from the Programmer's Guide.

   If such a situation were permissible then that node would in

turn have a subnode of the same structure type, etc. Thus there would be an infinite branch on the tree.

20.0 HAL/S/V ARRAY PROCESSING FEATURE

The material contained in Chapter 20 of the HAL/S Programmer's Guide is not available in the HAL/S/V programming language.

Any allowable HAL/S operation one can perform on an array can be done without these features on a component by component basis in a loop. Though somewhat less convenient, this approach has several advantages.

1. Implementation dependencies arising from the ambiguity in the order in which component operations are performed are eliminated since the serial order of the operations is made explicit.

2. Operations such as "+" may be treated by the verification system as non-generic and subject to a set of axioms which are not dependent upon the types of the operands.

3. Verification is greatly simplified. For instance, the number of times a function is invoked is obvious from the text and not dependent upon the dimensionality of its arguments, as may be the case in HAL/S.

For these reasons, the constructs described in Chapter 20 have been removed from HAL/S/V even though several of them mey be theoretically verifiable.

21.0 EXPLICIT CONVERSIONS

Functions for conversion to and from scalar, vector, and matrix types have been ommitted because objects of these types are not allowed in HAL/S/V. Subbit psuedo-conversion does not exist in HAL/S/V because of the complexities which overweigh its usefulness.

22.0 ADDITIONAL INPUT/OUTPUT FEATURES

Chapter 22 is concerned with the additional input/output facilities of HAL/S. HAL/S/V incorporates these entirely. Hence, this chapter has been left out of the HAL/S/V manual entirely, being identical to Chapter 22 of the HAL/S/V Programmer's Guide.

23.0 REAL TIME PROGRAMMING II

1. Two program blocks may not contain templates for each other. This is implicit in the requirement that recursion is not permitted in HAL/S.

2. All or the time and delay conditions in SCHEDULE statements are specified by INTEGER rather than SCALAR expressions.

3. No program process may be terminated if it or any dependent process updates compool data.

4. A process may only cancel itself or dependent processes.

24.0 REAL TIME PROGRAMMING III

In the verifiable subset only latched type of events are permitted. Therefore event expressions can not contain transient events; operation such as "signal" is of no meaning in HAL/S/V.

1.

HAL/S/V does not contain transient events; the reason for not having these in the subset is mainly because the semantics of certain logical operations on this type of events is not well defined. Secondly, they seeem to be redundant as the programs can be written by using the latched events only which are like conditions.
example:

not EV {where EV is a transient event}

does not have any well defined meaning. Similarly "EV1 and EV2" where both are transient events.

2.

All th events are of latched type therefore default type taken in declaration is latched type.

3.

Boolean expressions can not contain items of the type event; if it is permitted then it not possible to assert the expression as true or false because the value of the latched event might be changed by some other concurrent process.

25.0 ERROR RECOVERY AND SIMULATION

This part of HAL/S has been retained in the subset but it needs further thoughts. This could possibly be redesigned on the lines of error recovery in Ada which would lead to substantial simplification.

26.0 DATA STORAGE AND ACCESS

1. All data items which are shared by more two or more tasks or programs must belong to some lock group. This rule does not apply to objects of "event" type.

27.0 HAL/S/V AND REENTRANCY

1. HAL/S/V does not permit multiple copies of reentrant procedures or functions to share local data items. Therefore, the keyword STATIC may not be used in conjunction with the keyword REENTRANT in defining procedures and functions.

   See 16.2 of this document for justifications.

28.0 THE HAL/S NAME FACILITY

1. No NAME data item may be declared in HAL/S/V.

   The ability to maintain "pointers" to specified data items is a valuable feature of many programming languages. However, it presents a problem from the point of view of verification.

   If the pointer value is treated as merely another name for an object, as in HAL/S, aliasinf of the worst kind can result. In such a case, the objecthas a declared name and any number of other aliases. Assertions made using any of the aliases may be falsified by accessing the object via any other--a fact which is not apparent from the program text. Therefore, the verification system is constrained to keep track of all aliases of every data object. For this reason, HAL/S/V does not permit any name data item to be declared.

29.0 REPLACE MACROS AND INLINE FUNCTIONS

1. There are no REPLACE macros or inline functions in HAL/S/V.

   It was stated in 5.1 that the REPLACE statement of HAL/S had been removed from HAL/S/V. This applies as well to the

parameterized version called REPLACE MACROS and for the same reasons.

The behavior of a program and, hence the verifiability, is governed by the program text. The HAL/S Replace facility allows arbitrary changes to be made to the program text. Such changes generate essentially new programs for which a previous verification may be invalid. Thus, complete verification requires that each possible program text be considered individually. The work involved is potentially exponential in the number of replace statements.

"In-line functions" in HAL/S are parameterless functions designed to enhance the versatility of the parametric replacements. However, without REPLACE MACROS, they are of little use. They are executed in-line and cannot be invoked elsewhere in the program. Moreover, according to the HAL/S/V scoping requirements outlined in Chapter 11, a function may not access non-local data unless it is passed as a parameter at the call site. Hence, a parameterless function is essentially a constant. For these reasons in-line functions are disallowed in HAL/S/V.

30.0 MANAGERIAL ACCESS OF CONTROL TO DATA AND CODE

HAL/S/V makes no explicit changes to the material in Chapter 30 of the HAL/S Programmer's Guide.

31.0 INTERFACES WITH NON-HAL/S CODE

HAL/S/V makes no explicit changes to the material in Chapter 31 of the HAL/S Programmer's Guide.

32.0 SUMMARY OF PART II

Chapter 32 is simply a summary of the contents of Part II, Chapters 15 through 31 of the Programmer's Guide.

Manual 1

HAL/S/V:
A Verifiable Version of HAL/S
Volume - 1

James C. Browne
Donald I. Good
Anand R. Tripathi
William D. Young

December 31, 1979

INSTITUTE FOR COMPUTING SCIENCE AND COMPUTER APPLICATIONS
The University of Texas at Austin
Austin, Texas 78712

VERIFIABLE SUBSET OF HAL/S/V

This document defines the preliminary verifiable subset of the HAL/S/V language. Those features in the original HAL/S/V language which make the proof methods cumbersome have been deleted in selecting this subset. This document has been prepared by editing the HAL/S Programmer's Guide. For purposes of comparing HAL/S and HAL/S/V, the same section numbers have been retained in both documents.

TABLE OF CONTENTS

CHAPTER 1

STRUCTURE OF HAL/S/V

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

1.1 STRUCTURING AND HIGHER ORDER LANGUAGES

A common method of problem solving is the so-called "top down" approach. The algorithm for solving the problem is first outlined broadly, then delineated step-by-step in successively deeper level of greater detail. The success of the algorithm in arriving at the solution lies as much in its ability to break the problem down to simple components as in its ability to resolve the problem as a whole.

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

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

Structured programming techniques introduce an order into the writing of programs which not only makes the programs easier to read but also less susceptible to error. Most modern high order languages possess constructs out of which structured programs can be created: the constructs of HAL/S/V have been defined deliberately with structured programs in mind.

1.2 THE BLOCK STRUCTURE OF HAL/S/V

The structure of a HAL/S/V compilation generally consists of a program block with procedure and function blocks nested within it.

These blocks constitute the HAL/S/V interpretation of the "subroutines" mentioned in Section 1.1. The more deeply such a block is nested, the greater the detail of the problem solution it is meant to handle. The blocks at each level contain executable code implementing the appropriate part of the problem solution.

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

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

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

1.2.1 Scoping Of Data

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

The scoping rules in HAL/S/V have been modified so that a procedure or function block can reference a non-local variable only if it is explicitly passed to it as an actual parameter at the call site. Further, to avoid any side effects of the function calls, we retain the restriction imposed on the original HAL/S; by which functions can only have INPUT parameters and cannot possess any ASSIGN parameters.

Restricting the non-local variable referencing only to those variables which are passed explicitly as parameters, helps in avoiding any aliasing of variables during a procedure call. In case of accessing constant identifiers the scope rules of block structured languages, as given for HAL/S, still remain valid.

---

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

Reasons:

In a procedure block no variable can be referenced using more than one name, which ensures that the assertions on the value of an identifier remain valid. This is achieved by restricting the non-local references to the actual parameters passed to a procedure at the call site. If unrestricted accesses are permitted to the non-local variables then it is very hard to make assertions at the call site involving these non-local variables.

Similarly, the type of parameters to a function are restricted to input (value) parameters so that function calls do not modify any non-local variable.

1.2.2 Scoping Of Block Names

The program block, and every procedure of function within it are named: block names have scoping rules identical with the data scoping rules as in a block structured language like PASCAL. The name of any procedure or function block is deemed to have been "declared" in the surrounding block in which the procedure or function is nested. This bounds the region where its name is known, and therefore determines where it may be invoked. Thus, the name of any procedure or function nested at the program level is known anywhere in the program. However, since in HAL/S/V recursion is not allowed, such a procedure or function may be invoked from anywhere in the program except inside itself. Similarly, inner procedures and functions may be invoked from anywhere in the block enclosing them except within themselves. It should be noted that all forms of recursion in HAL/S/V are illegal. The form of recursion not prevented by the rules given above is that in which procedures P and Q are not contained in each other, but P calls Q and Q calls P.

NOTE: HAL/S/V does not support the passing of function or procedure names as parameters. Therefore, the rules governing the "visibility" of block names must be more liberal than those governing variable names. Otherwise, functions and procedures could never call each other. Thus, the original HAL/S scoping rules concerning block names were retained in HAL/S/V whereas the scoping rules concerning variable names were modified.

> It is also possible for a program (or any block within it) to invoke entities outside the compilation unit; i.e. other compilation units. Procedures and functions may be compiled independently for this purpose.
> See: Guide/15.

1.3 STATEMENT GROUPING IN HAL/S

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

* the sequence may be executed once only;

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

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

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

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

1.3.1 Statement Groups And Go To Statements

The design of HAL/S/V minimizes the dangers of "GO TO" statements by limiting the regions which can be branched by them, in a way analogous to the limits imposed on data by the scoping rules described in Section 1.2. While groups can be branched out of, or branched within, they may not be branched into.

1.3.2 Interaction With Block Structure

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

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

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

CHAPTER 2

HAL/S/V SYMBOLOGY

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

2.1 THE CHARACTER SET

The HAL/S/V language uses the following character set:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz

0123456789

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

(blank)
```

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

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

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

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

2.2 RESERVED WORDS, IDENTIFIERS, AND LITERALS

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

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

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

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

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

2.2.1 Reserved Words

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

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

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

2.2.2 Identifiers

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

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

2. The first character must be alphabetic;

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

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

Examples:

```
Elephant_And_Castle   \
A1                     >   legal
P                     /

1B                   \
X_X_                 /    illegal
```

2.2.3 Literals

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

2.2.3.1 ARITHMETIC LITERALS -

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

```
     + ddd...        -
```

1. ddd represents an arbitrary number of decimal digits.
2. The + signs are optional.
3. The maximum number of digits is implementation dependent.*

Arithmetic literals in HAL/S/V all express integer values. Floating point numbers have been discarded in choosing the subset for reasons discussed in Section 4.1.

Examples:

```
     -4
```

---

* Some implementations of HAL/S/V may place extra restrictions upon the names of identifiers. See appropriate User's Manual.

---

* See appropriate user's manual.

2.2.3.2 CHARACTER STRING LITERALS -

These consist of strings of characters chosen from the entire HAL/S/V character set. The generic form is:

'ccccccc'

1. The quote marks delimit the beginning and end of the literal.
2. cccc represents an arbitrary number of characters in any combination.
3. Quote marks within the literal must be represented by a pair of quote marks to avoid confusion with the delimiting quotes.
4. The minimum number of characters is zero (a 'null' string), the maximum is 255. *

Examples:

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

If a literal consists of a single character , or character sequence repeated many times, a condensed form of literal using a repetition factor may be used.
See: Spec./2.3.3.

2.2.3.3 BOOLEAN LITERALS -

These express logical truth or falsehood, and are generally used to set up the values of Boolean data items. Their forms are:

TRUE
> expressing truth, or binary "1"
ON

FALSE
> expressing falsehood or binary "0"
OFF

Literal strings of binary values also exist.

---

* This value may vary between implementations. See appropriate User's Manual.

See: Guide/17.1.

2.3 FORMAT OF SOURCE TEXT

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

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

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

2.3.1 Single-Line Format

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

EXPONENTS are denoted by the operator: **

Example:

```
 t+2
X      is coded as:

|M   X**(t+2)
```

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

Example:

```
a      is coded as:
 i+1

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

2.3.2 Multi-Line Format

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

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

Example:

```
 t+2
X       is coded as:

|E      T+2
|M     X
```

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

Example:

```
a       is coded as:
 i+1

|M     A
|S      I+1
```

When using multi-line format, the user must take care that nothing on the E- and S-lines overlaps anything on the M-line.

> Exponents of exponents and subscripts of subscripts use extra exponent and subscript lines. Special rules apply if exponents are subscripted, or if subscripts possess exponents.
> See: Spec./2.4.

2.4 STATEMENT DELIMITING

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

Examples:

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

2.5 COMMENTS IN HAL/S/V

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

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

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

2.5.1 Imbedded Comments

An imbedded comment takes the form:

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

Such comments may appear between HAL/S/V statements or imbedded in a statement. They may not appear in the middle of a literal, reserved word, or identifier. Nor may they overlap any source text or other comments on other lines of a group written in multi-line format. As far as the sense of the source text is concerned, an imbedded comment is treated as if it were a string of blank characters.

Examples:

```
|  M  X = X + 1;   /* ADD ONE TO X */
|
|  M  X = Y;
|  S   1 /* BAD */
|                  illegal-controverse overlap rule
```

2.5.2 Comment Lines

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

Examples:

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

CHAPTER 3

A HAL/S/V COMPILATION - THE PROGRAM BLOCK

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

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

3.1 OPENING AND CLOSING THE PROGRAM BLOCK

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

3.1.1 Program Opening

The statement opening a program block takes the form:

```
label : PROGRAM;
```

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

3.1.2 Program Closing

The program block is closed with the statement:

```
|
|  CLOSE label ;
|
```

1. The identifier label is optional.

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

3.2 POSITION OF DATA DECLARATIONS

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

3.3 FLOW OF EXECUTION IN THE PROGRAM

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

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

CHAPTER 4

DATA DECLARATION

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

This set has been considerably restricted in defining HAL/S/V. This was done primarily because operations on certain types of data lend themselves to verification methods more readily than others.

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

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

4.1 HAL/S/V DATA TYPES

In the HAL/S/V language, arithmetic data must be expressed as type INTEGER for the representation of integer-valued quantities.

The integer data type may be specified in either single or double precision. The precision determines the maximum absolute value the identifier may take on.

NOTE: Other arithmetic data types which are found in HAL/S have been omitted from HAL/S/V. The SCALAR data type (floating point) has been omitted because of the difficulty of axiomatizing floating point arithmetic. Hence, the VECTOR and MATRIX data types are also omitted since they are composed of scalar elements. In the subsequent sections anything involving these data types has been discarded or

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

modified.

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

* CHARACTER for the representation of strings of text;

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

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

HAL/S/V possesses other data types. The Boolean data type is a degenerate form of the HAL/S/V "bit string" type.
See: Guide/17.

HAL/S/V also possesses hierarchical organizations of data items of any type, known as "structures".
See: Guide/19.

Reasons: Verification is primarily concerned with discrete data; but the representation of real numbers as discrete quantities introduces a degree of complexity which makes verification extremely difficult. Sophisticated numerical technigues and error analyses are often needed to make valid assertions about the results of floating point operations.

4.2 SIMPLE DECLARATION STATEMENTS

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

4.2.1 Integer

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

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

2. Presence of the keyword SINGLE specifies single precision.

3. Presence of the keyword DOUBLE specifies double precision.

4. Absence of either keyword implies default of single precision.

For the integer data type, SINGLE precision usually implies halfword and DOUBLE precision implies fullword, depending on the implementation.*

Examples:

```
DECLARE I1 INTEGER;
DECLARE BIGI INTEGER DOUBLE;
```

4.2.2 Character

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

1. name is any legal identifier.

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

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

Example:

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

4.2.3 Boolean

```
DECLARE name BOOLEAN;
```

---

* See appropriate User's Manual.

---

* This value may vary between implementations. See appropriate User's Manual.

1. name is any legal identifier.

4.2.4 Arrays

The properties of a data item, (its type, precision, and size), as expressed in its declaration are called the "attributes" of the data item. In any of the above declarations, the attributes are specified following the name of the data item.

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

```
|
| DECLARE name ARRAY(n) attributes;
|
```

1. attributes stand for any legal form of attributes for any data type described. It is possible that none appear.

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

Examples:

```
|
| DECLARE AS1 ARRAY(500) INTEGER;
| DECLARE AM1 ARRAY(20) CHARACTER(15);
|
```

HAL/S/V also supports multidimensional arrays of any data type.
See: Guide/18.1.

4.2.5 Compound Declarations

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

Example:

Separate Declarations:

```
|
| DECLARE S;
```

---

* This value may vary between implementations. See appropriate User's Manual.

```
| DECLARE I INTEGER DOUBLE;
| DECLARE B BOOLEAN;
| DECLARE C ARRAY(5) CHARACTER(20);
|
```

Equivalent Compound Declarations:

```
|
| DECLARE S,
|         I INTEGER DOUBLE,
|         B BOOLEAN,
|         C ARRAY(5) CHARACTER(20),
|
```

Note commas separating the declaration of each data item.

If the identifiers in compound declarations have some attributes in common, a third, even more compact form called a FACTORED DECLARATION is possible.

Example:

```
|
| DECLARE  I1 INTEGER,
|          I2 INTEGER DOUBLE,
|          I3 INTEGER DOUBLE;
```

can be rewritten in the factorial form:

```
| DECLARE INTEGER, I1,
|                  I2 DOUBLE,
|                  I3 DOUBLE;
```

Note the comma separating the factored attribute, and the first data item.

4.3 INITIALIZATION OF DATA

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

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

* MULTI-VALUED data items are those having more than one element: arrayed data items of any type.

---

* See Guide/16 for certain restrictions and additional forms of initialization.

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

4.3.1 Uni-Valued Data Items

The two variations of the construct for initializing uni-valued data items are:

```
INITIAL ( value )
CONSTANT ( value )
```

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

2. The form using the keyword CONSTANT is required only if the user wishes never to change the initial value during execution.*

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

data type	literal value
CHARACTER	character string
BOOLEAN	boolean
INTEGER	arithmetic

Examples:

```
DECLARE A INTEGER INITIAL(3),
        C CHARACTER(80) INITIAL('YES'),
        D BOOLEAN INITIAL(TRUE);
```

Note: initial working length of C becomes 3.

4.3.2 Multi-Valued Data Items

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

```
INITIAL( value1 value2 , .....)
CONSTANT( value1 , value2 , .....)
```

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

---

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

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

data type	literal value
CHARACTER	character string
BOOLEAN	boolean
INTEGER	arithmetic

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

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

Examples:

```
|
|DECLARE S ARRAY(2) CONSTANT(1,0),
|DECLARE S ARRAY(199) INTEGER INITIAL(256);
|                                  ^
                                   |
                        (all elements of these data items
                           are identically initialized)
```

4.3.3 Order Of Initialization

To complete the specification of initialization, the order of initialization of the elements of multi-valued data items needs to be defined.

The following ordering rule, applied here to the initialization of multi-valued data items, holds true whenever the ordering of elements is called into question.

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

> Literal values appearing in initial lists may be expressions computable at compile time rather than literals.
> See: Guide/Appendix D.
>
> Additional, more compact initialization forms are available if only partial initialization is required, or if subsets of the initial values are identical.

See: Guide/16.

CHAPTER 5

REPLACE STATEMENTS

HAL/S/V does not contain any form of replace statements.

Reasons: Replace statements in HAL/S/V modify the text of the program, and verification of a program containing a replace statement is valid only in the context of that particular replace statement. Any change in the replace statement amounts to generating a totally new program.

CHAPTER 6

DATA REFERENCING AND SUBSCRIPTING

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

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

1. COMPONENT SUBSCRIPTING allows the user to select substrings from character data items.

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

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

6.1 SUBSCRIPTS OF UNARRAYED DATA TYPES

Unarrayed data types, i.e. those whose declarations contain no array specification, may at most possess only component subscripting. Unarrayed data items of integer, and Boolean types may not possess any subscripting. Allowable subscripts of the remaining type - CHARACTER - are described.

---

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

6.1.1 Character

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

* To select the a-th character from STRING:

STRINGa

1. a is an integer expression in the range 1 < a < L.

* To select a characters from STRING, starting from the b-th:

STRINGa AT b

1. a and b are integer expressions.
2. b is in the range 1 < b < L.
3. a is in the range 0 < a < L - b + 1.

* To select a substring starting with the a-th character of STRING, and ending with the b-th:

STRINGa TO b

1. a and b are integer expressions in the range 1 <(a, b)< L.
2. b >/ a.

Examples:

```
if C = 'ABCDEF' then:
   C5 = 'E'
   C2 AT 2 = 'BC'
   C4 TO 6 = 'DEF'
```

---

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

6.2 SUBSCRIPTS OF ARRAYED DATA TYPES

Arrayed data types, i.e. those whose declarations contain an array specification, may possess array subscripting. If the data type is character, then it may, in addition, possess component subscripting.

6.2.1 Array Subscripting Only

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

* To select the a-th array element from TABLE:

TABLE(a)

1. a is an integer expression in the range 1<=a<= L.
2. The colon is optional if the data type of TABLE is integer.

* To select a sub-array of length a starting from the b-th array element of TABLE:

TABLE(a at b)

1. a is an integer literal value in the range 1<= a <= L.
2. b is an integer expression in the range 1 <= b <= L = a + 1.
3. The colon is optional if the data type of TABLE is integer.

* To select a sub-array starting from the a-th array element of TABLE and ending with the b-th:

TABLE(a to b)

1. a and b are integer literal values in the range 1<=(a, b)<= L.
2. b > a.
3. The colon is optional if the data type of TABLE is integer.

Examples:

if T is a 4-array of booleans with

```
T2: = .FALSE                (unarrayed)
T3 TO 4: = (TRUE,TRUE)      (still arrayed)
```

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

```
T2 = 2                  (unarrayed)        } optional colon
T3 TO 4 = (3,4)         (still unarrayed)  } omitted
```

if C is a 3-array of characters, with

```
C = 'YES'                       (selects first array element)
C2 TO 3: = ('NO','MAYBE')       (still arrayed)
```

6.2.2 Array And Component Subscripting

The following rule shows how array and component subscripting are juxtaposed if TABLE represents an array of character data type.

TABLE array:component

1. array represents array subscripting of any of the forms previously described.

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

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

Examples:

if C is a 3-array of characters, with

C = ('YES' 'NO' 'MAYBE') then:

C = 'Y' (selects 3rd character from third array element)

> Apparently, the colon should be optional on Boolean data types also. It is not because the Boolean data type is a degenerate case of a bit string data type which may possess component subscripting.
> See: Guide/17.3.

6.2.3 Component Subscripting Only

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

TABLE*:component

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

Examples:

if C is a 3-array of characters with

C = ('YES', 'NO', 'MAYBE') then:

C = ('Y', 'N', 'M') (makes 3-array from first character of each item)

HAL/S/V allows more general forms of subscript expressions than just those stated above.
See Spec./5.3

In particular, a symbolic form of reference to the last array or other element of a data type is allowed.
See Spec./5.3.2

More complex subscripting forms apply to multi-dimensional arrays, (See Guide/18.3); and to the organization of data called "structures".
See Guide/19.6

Subscript forms stated to be literals may in fact be expressions computable at compile time.
See Guide/Appendix D

CHAPTER 7

EXPRESSIONS

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

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

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

7.1 ARITHMETIC OPERATIONS

Arithmetic operations are the most numerous of all operations in the HAL/S/V language. They comprise operations on integer data type. HAL/S/V recognizes the following operations:

Symbol	Purpose
**	exponentiation, positive exponent
(blank)	multiplication
+	addition
-	subtraction, negative

NOTE: Since all arithmetic operands will be composed of integer valued data items, no real division is needed. The function DIV is the appropriate integer division function.

7.1.1 Negation

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

Symbolic form: - R

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

R-type
Integer

Examples:

```
if I is an integer and I = 5
     then -I = -5
```

7.1.2 Addition And Subtraction

Addition and subtraction can only take place between integer data types:

Symbolic form: L + R

7.1.3 Multiplication

The HAL/S/V language has no explicit symbol for multiplication: the adjacency of two operands signifies this operation. Multiplication can take place with arithmetic operands of integer type.

The symbolic form for multiplication is shown as:

Symbolic form: L R

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

7.1.4 Exponentiation

This operation takes the general symbolic form:

Symbolic form: L ** R

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

2. The operand types are:

L -type	R -type
INTEGER	INTEGER

3. The L operand is integer and the R operand is a non-negative integral-valued literal.

Examples:

If I is an integer with I = 5

then ** 2 = 25 (integer result)

7.1.5 Note On Precision Conversion

No implicit precision conversion is allowed in HAL/S/V. Operations between operands of differing precision requires the explicit conversion of all similar precision before the operation is performed.

Reasons: Operands of differing precision are essentially of different type. Implicit precision conversion masks from the programmer the fact that in the conversion process information that may be lost or spurious information may be added. Having the programmer perform the conversion explicitly forces him to recognize this change and possibly compensate for it.

7.2 CHARACTER OPERATIONS

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

Symbol	Purpose
| | CAT	catenation

7.2.1 Concatenation

The utility of catenating character strings is obvious in the generation of output listings. Concatenation is guided as follows:

Symbolic form: L \| \| R
CAT

The L and R operands are restricted to character types, with the following types being legal:

L -type	R -type
CHARACTER	CHARACTER

Examples:

```
If C is a character item with C = 'UNITS'
     then 'TEN' | | C = 'TEN UNITS'
```

7.3 BOOLEAN OPERATIONS

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

Symbol	Purpose
& AND	logical intersection
| OR	logical disjunction
-- NOT	logical complement

7.3.1 Complement

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

Symbolic form: -- R

NOT

1. The R operand is of Boolean type.

Example:

```
If B is Boolean with B = TRUE
     then -B = FALSE
```

7.3.2 Conjunction

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

Symbolic form: L | R

---

* The term conjunction as used here, means disjunction in the terminology of logic.

OR

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

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

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

Examples:

```
If B is Boolean with B = FALSE
     then B|B = FALSE
          B|TRUE = TRUE
```

7.3.3 Intersection

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

```
Symbolic form:  L & R
                AND
```

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

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

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

Examples:

```
If B is Boolean with B = FALSE
     then B & TRUE = FALSE
          B & B    = FALSE
```

---

* The term intersection, as used here, actually means conjunction in the language of logic.

7.4 COMBINING OPERATION AND PRECEDENCE

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

1. The ORDER in which operations are executed (the order of "precedence");

2. The WAY in which the precedence order can be overridden.

7.4.1 Arithmetic And Character Precedence

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

Symbol	Precedence	Purpose
	FIRST	
**	1	exponentiation, etc.
(blank)	2	multiplication
+	6	addition
-	6	subtraction, negation
||, CAT	7	catenation
	LAST	

Two rules clarify and modify this information:

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

2. EXCEPT for ** which is evaluated right to left.

7.4.2 Boolean Precedence

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

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

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

Examples:

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

```
-B1|B2 & -B3
  ^  ^   ^ ^
  |  |   | |
  1  4   3 2
```

7.4.3 Overriding Precedence Order

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

Examples:

In the following Boolean expression,

```
B1|B2 & B3|B4 & B5
  ^   ^   ^   ^
  |   |   |   |
  2   1   4   3
```

parentheses may change the precedence order as shown:

```
(B1|B2) & ((B3|B4) & B5)
   ^     ^      ^     ^
   |     |      |     |
   1     4      2     3
```

In the following arthmetic expression,

```
S1 + S2(2) + S3/2
   ^     ^  ^   ^
   |     |  |   |
   2     1  4   3
```

parentheses may change the precedence order as shown:

```
((S1 + S2)2 + S3)/2
     ^   ^  ^    ^
     |   |  |    |
     1   2  3    4
```

HAL/S/V allows the operands in an expression to be arrayed, causing parallel evaluation on an element-by-element basis.
See: Guide/20.1.

7.5 SOME EXPLICIT CONVERSIONS

As evidenced in Section 7, there are no implicit type conversions in the HAL/S/V language. However, there is a comprehensive range of explicit conversions, some of which are now described.

7.5.1 Precision Conversion

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

```
(expression)
        @ DOUBLE

(expression)
        @ SINGLE
```

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

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

   See Appendix A.

Examples:

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

```
(A + B)
    @ DOUBLE
```

is double precision, the type remaining unchanged.

> The explicit conversions described are those most commonly required for numerical analysis. However, HAL/S/V contains many other explicit conversion function forms corresponding to conversions between most data types.
> See: Guide/21.

CHAPTER 8

ASSIGNMENTS

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

For convenience, an assignment is classified according to the type of the receiving data item; that is, the data item being assigned into.

* ARITHMETIC ASSIGNMENTS are assignments to integer data items.

* CHARACTER ASSIGNMENTS are assignments to character data items.

* BOOLEAN ASSIGNMENTS are assignments to Boolean data items.

NOTE: HAL/S/V allows no implicit type conversions. Therefore, this type is the same as that of the expression whose value is used in the operation. See Section 8.2.2.

8.1 GENERAL FORM OF ASSIGNMENT

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

Symbolic Form: L = R;

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

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

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

8.2 ARITHMETIC ASSIGNMENTS

Arithmetic assignments are those in which the receiving data type is an integer.

NOTE: Matrix, vector, and scalar data types which are part of the HAL/S language have been removed from HAL/S/V. Arithmetic assignment, therefore, describes a more limited class of operations in HAL/S/V.

8.2.1 Integer

The operand type is:

L -type	R -type
Integer	Integer

8.2.2 Note On Precision Conversion

One may wish to assign an expression to a data item of differing precision. HAL/S/V prohibits any such assignment without explicit conversion according to the rules of Section 7.5.

Example:

If I1 and I2 are double precision data items and we wish to add them in double precision and store the result in a single precision data item I3, then:

```
|
| I3 = (I2 + I1)
|                @ single
```

is the appropriate command.

Reasons: One desirable characteristic for any programming language is consistency. The stipulation that precision conversion in expressions be explicit suggests a similar stipulation for assignments.

Apart from that, implicit conversion in an assignment statement sometimes disguises the loss of significant digits when assigning a double precision value to a single precision data item. Explicit conversion forces the programmer to recognize this possibility and perhaps compensate for it.

8.3 CHARACTER ASSIGNMENTS

The receiving data item is character type.

The operand types are:

L -type	R -type
CHARACTER	CHARACTER

NOTE: In contrast to HAL/S, HAL/S/V requires that the right hand side be of type character. Integer values must be explicitly converted to character values before they can be assigned to a character data item. This is in accord with Section 8.2.2.

Examples:

If C is a character with C = 'ABCDE' and C2 is a character,

then

```
|
|C2 = C ;          results in C2 = 'C'
|      3
```

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

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

Examples:

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

```
|
| C1 = 'ABCDE';
|
```

results in C1 = 'ABCDE' of working length 5 but

```
|
| C2 = 'ABCDE';
|
```

results in C2 = 'A' of working length 1

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

Let
STRING
a
denote a receiving data item of character type:

N is declared maximum length
and
n is working length before assignment

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

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

3. If the range of a lies inside the range 1-n, then simple substitution of the character positions implied takes place.

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

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

Examples:

Let C1 be character of declared maximum length 10 with value C1 = 'ABCD'

Then by rules 2 and 3:

```
        |
        | C1       = 'QQ';
        |   2 TO 3
     results in C1 = 'AQQD'
        |
        | C1       = '1234';
        |   2 TO 3
     results in C1 = 'A12D
        |
        | C1       = 'X';
        |   2 TO 3
     results in C1 = 'AX D'
```

By Rules 2 and 4:

```
        |
        | C1       = 'QQ';
        |   4 TO 5
     results in C1 = 'ABCQQ'
                         (working length increased by 1)
        |
        | C1       = 'X';
        |   4 TO 5
     results in C1 = 'ABCX'
                         (working length increased by 1)
```

By Rules 2 and 5:

```
        |
        | C1       = 'QQ';
        |   5 TO 6
     results in C1 = 'ABCDQQ'
```

```
    |
    | C1       = 'FGH';
    |   7 TO 9
results in C1 = 'ABCD  FGH'
    |
    | C1  = 'FGH';
    |   6
results in C1 = 'ABCD F'
```

8.4 BOOLEAN ASSIGNMENTS

The receiving data item is of a Boolean type.

1. The operand types are:

L -type	R -type
BOOLEAN	BOOLEAN

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

Example:

If B is Boolean, then

```
    |
    | B = FALSE;
```

results in B = FALSE

8.5 MULTIPLE ASSIGNMENTS

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

Symbolic form: L1, L2, ... Ln = R;

1. The value of the R-expression is assigned to all L1 ... Ln in turn.

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

3. No particular order of assignment is guaranteed.

4. No variable appearing on the left-hand side of a multiple assignment may appear on both the main line and the subscript line. A variable may, however, appear an arbitrary number of times on any one line.

In HAL/S/V, the receiving data item or items may be arrayed. This can produce varying effects depending on whether or not the R-expression also is arrayed (i.e. has arrayed operands).
See: Guide/20.3.

CHAPTER 9

CONDITIONAL STATEMENTS AND BRANCHES

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

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

9.1 THE CONDITIONAL STATEMENT

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

9.1.1 Simple IF Statement

The form of the simple version is:

```
IF exp THEN statement;
```

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

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

3. statement may not be another conditional statement.

4. statement may not possess a label.

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

   NOTE: In contrast to 4., the HAL/S Programmer's Guide states that "statement may possess a label but cannot be branched to from outside the IF statement". However, this contradicts section 9.3 of that document.

Examples:

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

X is set to 0 if either B or C or both is true:

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

The true part is a compound statement containing two assignments.

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

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

9.1.2 Augmented If Statement

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

```
| IF exp THEN statement ;
| ELSE else statement ;
```

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

2. ELSE statement constitutes the FALSE part of the conditional statement. It may be any executable statement either simple or compound.

3. ELSE statement may not possess a label.

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

NOTE: As with the simple IF statement the HAL/S Programmer's Guide is inconsistent on whether the else statement may possess a label. Number three has been made consistent with Section 9.3 which states that the "false part" of a conditional may not possess a label.

Examples:

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

X is set to 0 if B or C or both is true, otherwise X is set to 1.

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

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

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

This is legal: the false part of a conditional statement may itself be another conditional statement. ---

9.2 RELATIONAL EXPRESSIONS

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

The simplest form of a relational expression is merely a comparison between two like quantities. The result is either TRUE or FALSE. More complex forms of relational expressions result from

combining comparisons with the BOOLEAN operators &, |, and ¬ .

9.2.1 Comparative Operations

HAL/S/V recognizes the following comparative operators:

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

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

CLASS II ONLY

$$\text{Symbolic form: } L \begin{matrix} \text{NOT} \\ \neg \end{matrix} \begin{matrix} = \\ = \end{matrix} R$$

1. The L-type and R-type are both Boolean.

$$\text{Symbolic form: } L \begin{matrix} > \\ < \\ >= \\ <= \\ \text{NOT} > \\ \neg > \\ \text{NOT} < \\ \neg < \\ = \\ \text{NOT} = \\ \neg = \end{matrix} R$$

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

L -type	R -type
CHARACTER	CHARACTER

3. For character comparisons, a shorter string is "less" than a longer one. For strings of equal length, a string earlier in

the collating sequence is "less" than one later in the sequence.*

Examples:

```
If I is an integer with I = 5
     then I = 5 is TRUE
          I < 4 is FALSE
          I >= 5 is TRUE

If C is a character data item with C = 'ABC'
     then C = 'ABC' is TRUE
          C = 'BCD' is FALSE
          C > 'AB' is TRUE
```

9.2.2 Note On Precision Conversion

Comparisons of operands of different precision is prohibited in HAL/S/V. Therefore, explicit precision conversion may be required when both operands are integer.

NOTE: HAL/S/V allows no implicit precision conversions in expressions or assignment statements. To preserve consistency, that prohibition is extended to comparisons. See Section 7.5 for the explicit conversions.

Example:

Let X be a single precision data item and y be double precision.

```
IF Y < X THEN Z = 1;                    is illegal

IF Y < (X) @ DOUBLE   THEN Z = 1;       is legal.
```

9.2.3 Combining Comparative Operations

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

* BOOLEAN EXPRESSIONS may contain no comparative operations.

* RELATIONAL EXPRESSIONS may contain no Boolean operands.

---

* The collating sequence is implementation dependent. See appropriate User's Manual.

Examples:

If I1 and I2 are integer data items with
I1 = 17 and I2 = 8

and C is character with C = 'ABC'
then
I1 = I2 | C1 = 'A' is TRUE
I1 = I2 & C1 = 'A' is FALSE

If B is Boolean then
B | I1 = I2 is illegal

9.2.4 Precedence

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

Symbol	Precedence	Purpose
	FIRST	
	1	Operations involving operands of comparisons
>, <, <=, NOT >, ~ >, >=, NOT <, ~<, =, NOT =, ~=	2	Comparative operations
-, NOT	3 *	
&, AND	4	Logical operations on comparisons
|, OR	5	

* Any operand of this operator must always be parenthesized.

Example:

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

```
IF  S1 + S2 = 0 | ~  (S3 > 0) & ~  (S4 < 0 | S5 > 0) THEN
      ^    ^   ^ ^      ^    ^ ^      ^   ^    ^
      |    |   | |      |    | |      |   |    |
      1    2  10 4      3    9 8      5   7    6
```

Relational expressions may be arrayed, additional rules being required to determine if the result is TRUE or FALSE.
See: Guide/20.4.

9.3 LABELS AND BRANCHES

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

9.3.1 Labels

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

```
| label : statement ;
|
```

1. statement is any executable statement or statement group (see Section 10), with two exceptions.
2. statement may not be the "true part" or "false part" of a conditional statement.
3. label is a user-defined identifier name (see Section 2.2).

Examples:

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

The following are illegal since they violate Rule 2:

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

However, the conditional statement itself may be labelled:

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

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

Example:

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

9.3.2 Go To Statement

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

```
|
| GO TO   label ;
|
```

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

Examples:

```
|
| GO TO ONE;
|
```

The GO TO statement itself may be labelled:

```
|
| TWO:  GO TO THREE;
```

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

9.3.3 Eliminating Go To Statements

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

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

Examples:

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

This example is programmed in HAL/S/V in the simplest way (possibly having been translated from Fortran or an assembly language).

The same algorithm is more clearly programmed as follows:

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

CHAPTER 10

STATEMENT GROUPS

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

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

* The sequence may be repetitively executed until some condition is satisfied;

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

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

10.1 DELIMITING STATEMENT GROUPS

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

10.1.1 The Do Statement

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

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

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

2. Control is optional. If it is absent, the sequence of statements within the group is executed in order once only.

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

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

10.1.2 The End Statement

Every statement group is closed with an END statement:

```
|
|  END   label ;
|
```

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

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

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

Examples:

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

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

|    FIX:        DO FOR I = 1 TO 10;
|                  A = -A ;           } one statement in group
| S                I      I           }
|                END FIX;             } label specification in
                                        END matches label of DO
```

The following examples show the importance of being able to group statements together for use in conjunction with a conditional

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

It is required to conditionally execute both assignments: one solution might be -

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

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

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

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

10.2 REPETITIVE EXECUTION OF STATEMENT GROUPS

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

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

* The DO FOR statement, in which the sequence is

executed once for each of a set of assigned values of a "control variable".

10.2.1 The Do While Statement

The form of the DO WHILE statement is:

```
|
|  DO WHILE condition;
|
```

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

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

3. If the value of condition is FALSE, the stopping condition is satisfied. Execution proceeds to the statement following the END statement of the group.

Examples:

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

Here the group is executed five times, after which the value of I is -1.

It is possible for a group never to be executed:

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

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

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

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

```
|
| I = 9;
| DO WHILE TRUE;
```

```
    I = I - 2;
    IF I < 0 THEN GO TO ALL_DONE;
END;
.
.
.
```

If the keyword UNTIL is substituted for the keyword WHILE, then the group is always executed at least once. After the first cycle, the relational or Boolean expression is evaluated at the beginning of each cycle as in the DO WHILE, except that the logic of the test is inverted: cycles of execution continue until the result of the expression becomes TRUE.

Example:

```
I = 0;
DO UNTIL I <= 0;
  I = I - 1;
END;
.
.
.
```

The group is executed once, and the final value of I is -1.

10.2.2 The Do For Statement

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

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

1. var is an unarrayed and unsubscripted INTEGER data item. It is called the "control variable" of the DO FOR statement.

2. initial and final are integer expressions: initial is the initial value asigned to var; final is the value against which var is tested at the start of every cycle to determine if the stopping condition is satisfied.

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

   All three expressions are evaluated once prior to the first cycle of execution.

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

4. increment may be either +1 or -1. The phrase:
   BY increment
   is optional. If omitted, the implied increment is +1.

5. At the end of the final cycle var has the value it received after that cycle.

NOTE: Since HAL/S/V has no scalar data type, initial and final must be integer expressions and var is INTEGER variable.

The increment may be only +1 or -1. This simplifies loop assertions without restricting the programmer significantly. Finally, the addition of 5 simply makes explicit the value of the control variable upon leaving the loop. Though it is bad programming practice to do so, no language requirement prohibits the programmer from subsequently using the control variable for other purposes. In such a case it is desirable for program verification that its value be determinate.

Examples:

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

Here the group is executed 10 times. I is initially 1, and increments each time until 10 is reached. At the end of execution of the group, the value of I is 11.

```
|
| I = 7;
| DO FOR I = I + 5 TO I - 3 BY -1;
|    X = X + I;
| END;
|
```

This example demonstrates some of the subtleties of the DO FOR statement. The initial and final values are precomputed as 12 and 4 respectively. Then I is reused as the control variable: the group is executed 9 times, and after the last cycle of execution, I retains the value 3.

This DO FOR statement may

possess a WHILE or UNTIL

clause which furnishes a

supplementary stopping
condition.
See: Spec./7.6.5.

NOTE: A second form of DO FOR statement is allowed in HAL/S which is not permitted in HAL/S/V: that is, one in which the values of the control variable do not form a regular progression and are listed explicitly.

REASONS: This second form of the DO FOR statement does not in general permit specification of a loop invariant in any clear way. This is because the values assumed by var may be such that one execution of the loop is related to the next execution in a very tenuous way if at all. Also, since var assumes the immediately prior to the cycle of execution in which they are used. The behavior of any loop execution may be dependent upon prior executions of the loop in very nuclear ways.

10.3 SELECTIVE EXECUTION OF STATEMENT GROUPS

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

```
|
|  DO CASE exp ;
|
```

1. exp is an integer expression.

2. If its value is k then the kth statement of the group is selected for execution.

3. A run time error results if k < 0

   or k is greater than the number of

   statements in the group.

The flexibility of a DO CASE statement lies in that the selected statement may be a compound statement (i.e. it may

itself be a statement group).

Example:

```
I = 3;
DO CASE I;
   X = 4;                    case 1
   X = 3;                    case 2
   DO;
      X = 7;              }  case 3
      Y = 3;              }
   END;
```

```
      X = 1;                    case 4

      X = 0;                    case 5
END;
```

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

X = 7, Y = 3

An ELSE clause may be added to the DO CASE statement which is executed instead of an error being signalled, if the value of the case variable is outside the legal range for the statement group.
See: Spec./7.6.2.

10.4 BRANCHING IN STATEMENT GROUPS

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

The EXIT statement is, in effect, a controlled branch out of a statement group.

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

10.4.1 The Exit Statement

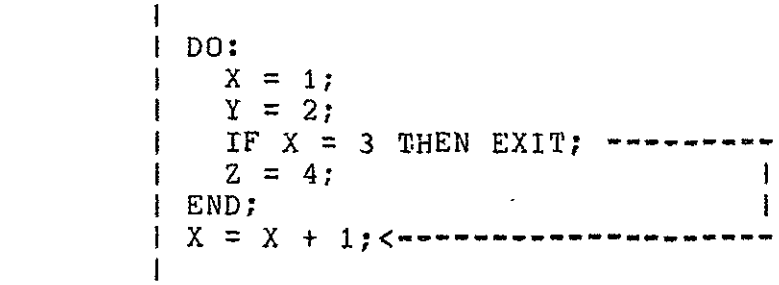
The simplest form of the EXIT statement is:

```
EXIT;
```

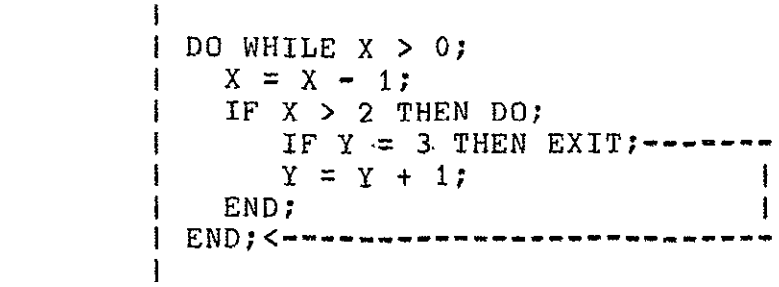
1. Its execution causes an immediate branch out of the innermost statement group in which it is enclosed.

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

Examples:

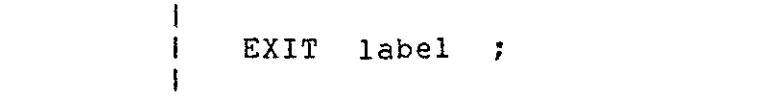


Arrow shows branch in execution if Z ^&=\& 3



Arrow shows branch in execution if Y = 3: execution branches to the end, but not out of DO WHILE group.

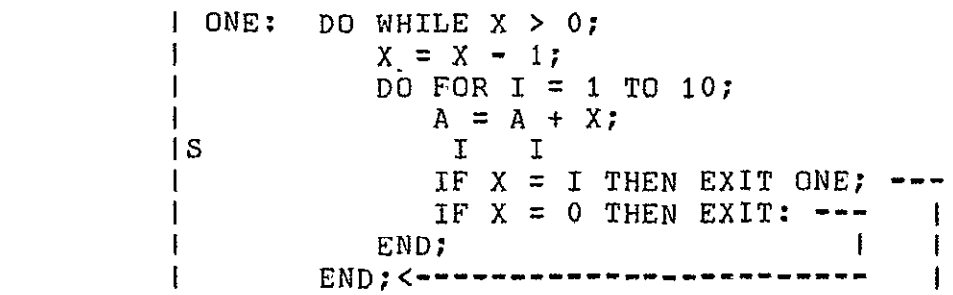
There exists a second form of the EXIT statement to allow branches out of other than the innermost statement group:



1. Its execution causes a branch out of the enclosing statement group whose DO statement possesses the label label.

2. Execution is directed to the first statement after the END of the group branched out of.

Example:



```
|       X = 0;<--------------------------
```

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

10.4.2 The Repeat Statement

The simplest form of the REPEAT statement is:

```
|
| REPEAT;
|
```

1. It must be enclosed in a DO FOR or DO WHILE group.
2. Its execution causes an immediate branch to the beginning of the innermost enclosing DO FOR or DO WHILE group.
3. The next cycle of execution of the group then starts (unless of course the stopping condition is satisfied).

Examples:

```
|
| DO WHILE X > 0;
|    X = X - 1;
|    IF X = 4 THEN DO;
|       Y = Y + X;
|       IF Y = 1 THEN REPEAT;
|    END;
| END;
```

If Y = 1 then a branch back to the beginning of the DO WHILE is made. Note that although the DO WHILE is not the innermost group, it is the innermost repetitive group.

```
|
| X = 4;
| DO WHILE X > 1;
|    X = X - 1;
|    IF X = 1 THEN REPEAT;
|    Y = X;
|S    X
| END;
|
```

When X = 2 the REPEAT branch is executed: a new cycle of execution does not begin however, because the initial test shows that the stopping

As with the EXIT statement, there exists a second form of the repeat statement allowing branches back to the beginning of other than the innermost DO WHILE or DO FOR group:

```
REPEAT label ;
```

1. Its execution causes an immediate branch to the beginning of the enclosing DO FOR

   or DO WHILE group whose DO statement possesses the label label.

2. The next cycle of execution of the group then starts (unless the stopping condition is satisfied).

Example:

```
     ONE:  DO FOR I = 1 TO 10;
              J = I;
              DO WHILE J > 0;
                 J = J - 1;
                 X = X + J;
S                 J   J
                 IF X = 25 THEN REPEAT;
S                     J
                 IF X = 0 THEN REPEAT ONE;
S                     J
              END;
           END;
           Z = 0;
```

The second REPEAT statement restarts the outer DO FOR group rather than the inner DO WHILE by virtue of its label.

CHAPTER 11

PROCEDURES AND FUNCTIONS

Section 1.2 of the Guide introduced the block structure of HAL/S/V programs on the abstract level. To summarize, any program can contain nested procedure and function blocks, which are two levels of "subroutines" characterized by the sequence:

invocation --> execution --> return to caller

The invocation of procedures and functions is governed by well-defined name scoping rules.

This section explains how, in practice, procedure and function blocks are defined in HAL/S/V, and describes how they are invoked returned from.

11.1 INTRODUCTION

A procedure is a subroutine block invoked by a CALL statement. It may have two kinds of parameters:

* INPUT PARAMETERS - by which values may be passed into a procedure only.

* ASSIGN PARAMETERS - by which values may be passed into and out of a procedure.

A function is a subroutine block invoked by the appearance of its name in an expression. It returns a value and therefore has a defined HAL/S/V data type. It may possess input parameters only.

11.1.1 Relative Position Of Block Definitions

Section 1.2 described the scoping rules which determine the regions of a program where any given procedure or function block may be invoked.

An important consequence of these rules is that a procedure invocation may either follow or precede its block definition.

always follow its block definition.

> A number of rules restrict the kind of function which may be invoked preceding its block definition.
> See: Spec./9

11.2 BLOCK DEFINITIONS

Procedure and function block definitions have forms very similar to the form of a program block, which was described in Section 3. The first statement is one defining the name and type of block, and listing its parameters. The last statement is a statement closing the block.

11.2.1 Procedure Opening

The statement opening block takes the form:

```
|
|label:  PROCEDURE (i1,i2,...) ASSIGN (a1,a2,...);
|
```

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

2. i1, i2,... are legal identifier names defining input parameters. If the entire parenthesized list is omitted, then the procedure has no input parameters.

3. a1,a2,... are legal identifier names defining assign parameters. If the entire parenthesized list and the keyword ASSIGN are omitted, then the procedure has no assign parameters.

11.2.2 Function Opening

The statement opening a function block takes the form:

```
|
|
| label:  FUNCTION (i1,i2,...) attributes;
```

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

2. i1,i2,... are legal identifier names defining input

parameters. If the entire parenthesized list is omitted, then the function has no input parameters.

3. attributes defines the type of attributes and, where applicable, precision and size. The form specification is the same as used in data declarations (see Section 4.2).

NOTE: In contrast to HAL/S, there is no default type for Functions. This is because the HAL/S default type, scalar, is not legal in HAL/S/V.

Also, it is good programming practice to make explicit the type of value being returned, particularly in light of the HAL/S/V prohibition of implicit conversion between data types, which requires a function RETURN value to be of the same type as the function.

11.2.3 Block Closing

Both procedure and function blocks are closed with the statement:

```
CLOSE label;
```

1. The identifier label is optional;

2. If supplied, it must be the name of the procedure or function block.

11.3 DECLARATION OF PARAMETERS AND LOCAL DATA

Procedures and functions commonly require the use of locally-defined data. As with program-level data, all data names must be declared before use by means of declaration statements. In addition, all input and assign parameters must appear in local declaration statements.

Data and parameter declarations must be placed after the procedure or function opening statement, and before the first executable statement. It is good practice, and mandatory in some inplementations*, to place parameter declarations before local data declarations. The forms of local data and parameter declarations are identical, and are as described in Section 4.

---

* See the User's Manual for any given implementation.

Examples:

General positioning -

```
|  ONE: PROCEDURE (ARG1) ASSIGN (ARG2)
{PARAMETER DECLARATIONS}
{LOCAL DATA DECLARATIONS}
{EXECUTIVE STATEMENTS}
```

Particular instance -

```
|   ONE: PROCEDURE (ARG1) ASSIGN (ARG2);
|      DECLARE ARG1 INTEGER;
}--parameters
|      DECLARE ARG2 ARRAY(100) INTEGER DOUBLE;}
|      DECLARE TEMP INTEGER;                  }--local
data
|      .
|      .
|      .
|   CLOSE ONE;
|
```

11.3.1 Character Parameter Declarations

Parameters of character type may be declared to possess an indefinite maximum length which is bounded by 255.* By this means problems of truncation of character data during argument passage can be avoided.

The basic form of declaration is:

```
DECLARE name CHARACTER (*);
```

1. The asterisk denotes an indefinite maximum length.

Example:

```
|
|  ONE:  PROCEDURE(A);
|     DECLARE A CHARACTER(*);
|     .
|     .
|     .
```

---

* This value may be implementation dependent. Consult user's manual for any given implementation.

11.4 FUNCTION INVOCATIONS

A function is invoked by the appearance of its name as an operand in an expression. If the function is defined with input parameters, a list of arguments to be passed must follow the appearance of the name. The precise form of invocation is:

```
label(i1,i2,...)
```

1. label is the defined name of the function.

2. i1,i2,... is a list of arguments, which must correspond in number with the parameters of the function invoked. Each argument is a HAL/S/V expression.

3. If the function has no parameters, then the entire parenthesized argument list must be absent.

NOTE: Since no implicit type or precision conversions are allowed in HAL/S/V the actual parameters must match the formal parameters exactly in type and precision.

The transmission of the argument list during function invocation may be viewed as the assignment of the value of each expression in turn to its corresponding input parameter (although in any given implementation this may not actually be the mechanism of transmittal).

11.4.1 Integer Parameter

1. The corresponding argument must be of integer type.

2. Explicit precision conversion is necessary if the precision of the formal parameter varies from that of the actual parameter.

11.4.2 Character Parameter

1. The corresponding argument must be character type.

Generally, the working length of the parameter becomes equal to the length of the expression (after conversion, where applicable). However, if this would cause the declared maximum length of the parameter to be exceeded, truncation of the excess from the right takes place.

11.4.3 Boolean Parameter

1. The corresponding argument must be of Boolean type.

11.4.4 Note On Function Restrictions In HAL/S/V

The mathematical notion of a function specifies that every invocation of a given function with a given set of arguments should return the same value. For verification purposes it is desirable that the programming language concept of function have a similar attribute. One feature of HAL/S/V which helps ensure this is the requirement that no function can access non-local data except that which is explicitly passed in the input argument list.

Since any function has only input parameters and not assign parameters this implies that functions have no side effects, a desirable result for verification purposes. To enforce this, additional restrictions must be placed on procedures that can be called inside a function block. These additional restrictions are noted in Section 11.5.1.

In particular, problems can arise if a function calls some procedure or function which returns time dependant results. For example, if a call is made within the function body to CLOCK, DATE, or RANDOM functions, then the function could return different values for the same input parameters.

Example:

```
|
| F: FUNCTION (X) INTEGER;
|    DECLARE X INTEGER;
|        .
|        .
|        .
|    RETURN  CLOCK;
| CLOSE F;
|
```

Then F(X1,) ne F(X2) even if X1 = X2.

In order to ensure the consistency of user defined functions, HAL/S/V restricts the use of system functions* such as RUNTIME, DATE, PRIO, to the outermost block of the program or the task. Therefore, a function or procedure in its body, will never use a system function returning time dependent results. In case an integer valued random number generator is defined, a similar restriction would apply.

---

* The use of "function" in this context is misleading. Rather these can be thought of as system co-routines accessing variables which are continually updated.

Arguments may possess arrayness. The effects of this depend on whether or not the corresponding parameter is declared to be an array.
See: Guide/20.5

11.5 PROCEDURE INVOCATIONS

A procedure is invoked by the use of a CALL statement, which may in the case of a procedure with parameters, also specify the arguments to be passed. The precise form of invocation is:

```
|
| CALL label (i1,i2,...) ASSIGN(a1,a2,...);
|
```

1. label is the defined name of the procedure.

2. i1,i2,... is a list of input arguments which must correspond in number with the input parameters of the procedure invoked. Each input argument is a HAL/S/V expression.

3. If the procedure has no input parameters, then the entire parenthesized argument list must be absent.

4. a1,a2,... is a list of assign arguments which must correspond in number with the assign parameters of the procedure invoked. Each argument must be a HAL/S/V data item.*

5. If the procedure has no assign parameters, then the entire parenthesized list of assign arguments, and the ASSIGN keyword, must be absent.

6. The input and assign parameter lists must be disjoint, also, no more than one part of any structured object may appear in the list of assign arguments and no part of a structured object may appear in the input list if any part appears in the assign list.

The transmission of the input argument list during procedure invocation is identical in nature to function argument list transmission. The related rules are given in Section 11.4.

The transmission of the assign argument list follows stricter rules since values are passed both into and out of a procedure by this mechanism.

---

* Or an assign parameter, if the invocation is nested within a procedure block.

NOTE: By the HAL/S/V scoping requirements outlined in Section 1.2, no changes can be made in any variable in any enclosing block unless that variable is explicitly passed as an assign parameter. This prevents a problem of aliasing which could occur in PASCAL, for instance, when an argument to a procedure is identical to a global variable referenced by the procedure.

11.5.1 Assign Arguments

1. An assign argument must be a declared HAL/S/V data item.*
2. An assign argument must match the corresponding assign parameter in type and precision.
3. No structure may have more than one component appear in the assign argument list.
4. No assign argument may be an input argument of any enclosing function or procedure block or any part of a structured object which is an input argument to an enclosing function or procedure block.

Reasons: One should be able to assert of any procedure that the input parameters are unchanged after the procedure execution. This is not the case if arguments are allowed to appear both as input parameters and as assign parameters. Moreover, whether the value of an input parameter remains constant throughout the procedure execution becomes dependent upon whether input parameter passing is implemented by a read only reference to a variable or by copying into local storage hence, the truth value of certain assertions becomes implementation dependant.

The requirement that several components of structured objects not appear prevents the possibility of aliasing. For instance, let A be an array of integers and EXAMPLE be a procedure of one input and two assign parameters. Then the procedure call

```
|
|  CALL EXAMPLE (A ) ASSIGN (A ,B)
|S                 I           J
|
```

strictly violates the rule that the parameter lists be disjoint only if I = J. However, which assertions one can make about the procedure become dependant upon the equality or non-equality at call time of variables external to the program in a way that is not readily apparent. Hence such a call is disallowed in the verifiable subset. Consider the following call:

```
|
|   CALL EXAMPLE (B) ASSIGN (A ,A )
|S                           I  J
```

This is also disallowed. If I = J at call time aliasing occurs. Apart from the general undesirability of aliasing, any assertions made about EXAMPLE would require that the case where I = J be explicitly treated. But no feature in the procedure definition would indicate this since the formal parameters are all distinct.

Finally, an assign argument should not be an input argument of any enclosing block because this violates our earlier requirement that input parameters not be changed by procedure execution. Moreover, it allows functions to have side effects by calling procedures whose side effects are not confined to variables local to the function block.

> Both input and assign arguments may possess arrayness, in which case the corresponding parameters must have an array declaration.
> See: Guide/20.5.

11.6 RETURNS FROM PROCEDURES AND FUNCTIONS

When execution reaches the CLOSE statement of a procedure block, an automatic return to caller takes place. However, if execution reaches the CLOSE statement of a function block, a run time error results since the function has no value to return to the caller. Hence a function block needs an explicit RETURN statement to cause the return to take place. In addition, if returns are required from parts of the code in a procedure block other than at the CLOSE, an explicit RETURN statement is required.

11.6.1 Procedure Return

The RETURN statement of a procedure takes the form:

```
|
| RETURN;
|
```

Example:

```
|
|  CHOICE:  PROCEDURE (FLAG) ASSIGN (DIR);
|     DECLARE FLAG BOOLEAN;
|     DECLARE DIR INTEGER;
|     IF FLAG THEN RETURN;
|     DIR = 1;
|  CLOSE;
|
```

> If FLAG = TRUE then procedure merely returns execution at RETURN. If FLAG = FALSE then DIR is set equal to 1, and

procedure returns execution at CLOSE.

11.6.2 Function Return

The RETURN statement of a function takes the form:

```
|
| RETURN exp ;
|
```

1. The resultant value of the expression exp is returned when the function returns to its caller.

NOTE: The type and precision of exp must match exactly the declared type and precision of the function. This is in accord with the HAL/S/V prohibition of implicit conversions.

The return of an expression by a function is similar in nature to the transmission of an input argument of a function to the corresponding parameter, the function itself playing the role of parameter.

Note that since a function block may not be defined with an array specification, no function may return an array result.

CHAPTER 12

INPUT/OUTPUT STATEMENTS

HAL/S/V incorporates entirely the Input/Output mechanisms of HAL/S. Therefore, Chapter 12 of this document is omitted, being identical to Chapter 12 of the HAL/S Programmer's Guide.

CHAPTER 13

REAL TIME PROGRAMMING - I

So far the Guide has made no reference to the dynamic properties of HAL/S/V programs. Clearly, any program will take a finite time to execute but none of the constructs hitherto described depend on any sense of time for their operation.

However, the HAL/S/V language does contain constructs which depend on a sense of time for their operation. This is what is meant by the statement that HAL/S/V programs is a "real time programming language". In other words, HAL/S/V programs can be written which, when executed, cause operations to be carried out at desired points or during desired intervals in "real time".

In some implementations of HAL/S/V, "real time" may be just what the phrase implies, real clock time. In others, the "real time" may be simulated in some way by the operating environment of a HAL/S/V program: in this case, it can be referred to as "pseudo-real time".

This section of the Guide explains the basic HAL/S/V concepts of real time programming, and describes some of the more elementary real time programming language forms.

13.1 HAL/S/V REAL TIME CONCEPTS

The true HAL/S/V concept of a program at run time is an entity executing over some interval in "real time", directed and controlled by a Real Time Executive (RTE). At the outset, the RTE begins execution of the program. When program execution is completed, control is returned to the RTE. In HAL/S/V terminology, the dynamic counterpart of the static program block which is executing under RTE control, is called a "real time process".

13.1.1 Multi-Processing In HAL/S/V

Multi-processing is the simultaneous handling of more than one "real time process". With most present-day machines, "simultaneous" really means interleaved, because most machines can at one time only support the execution of a single machine instruction sequence. However, this distinction has no significance at the higher level of

the HAL/S/V language.

NOTE: This is not strictly true. A priority driven scheme can have very different results on an actual multi-processing system if processes update shared data items.

The RTE of HAL/S/V can simultaneously handle an arbitrary* number of processes created by the user. A number is attached by the user to each process, called its "priority". The RTE maintains processes in a "process queue" ordered by priority, and always endeavors to execute the processes in order of priority, highest first.

The HAL/S/V program itself, beginning execution under the RTE, constitutes the first or "primal process". All other processes are brought into existence by the execution of SCHEDULE statements coded into the program. Just as the primal process has a static counterpart, which is the program block coded by the user, so must the other processes have their static counterparts. These are so-called task blocks, which are coded inside the program block in a very similar way to procedure blocks. Each time a task block is invoked by execution of a SCHEDULE statement, a new process is created and queued by the RTE.

13.1.2 States Of A Process

It is now possible to represent the behavior of the RTE by a more formal description of the possible states** in which a process can exist. This in turn will introduce other HAL/S/V constructs for controlling the activities of the RTE. A process can be in either of the following two major states at a given time:

* ACTIVE STATE: a process is in an active state when it exists in the RTE's process queue. The state acutally comprises three substates or minor states in any one of which an active process may be at a given time.

* INACTIVE STATE: a process is defined for completeness as being in the inactive state if it does not exist in the

---

* See the User's Manual for the maximum number supported in any given implementation.

** The states to be defined do not correspond one-to-one with the RTE states described in the Language Specification document. The latter are defined for the convenience of the formal description of language constructs. The former are defined with user convenience in mind.

process queue.

The minor states of an active process are as follows:

EXECUTING: an active process is "executing" when it has actually been put into execution by the RTE, operating on the priority principle already described. The number of processes which can be in this state simultaneously is implementation dependent.***

READY: an active process is "ready" if it is available for execution, but higher priority processes in execution are currently barring it. The occurrence of a process first entering the ready state will be called its "initiation".

WAITING: an active process is "waiting" if it is neither ready nor executing. Some condition set up by the user prevents it being available for execution by the RTE.

When a process is created by invoking a task block by a SCHEDULE statement it makes a transition from the inactive state to an active state. It is entered into the process queue in either the ready or the waiting state, depending on the form of the SCHEDULE statement. If it is entered in the ready state, then depending on its priority, it may immediately be elevated to the executing state.

A process is caused to make a transition from an active state to the inactive state (or removed from the process queue) by a TERMINATE statement. The process is said to have been "terminated".

A process may be forced into the waiting state by execution of a WAIT statement.

NOTE: In HAL/S the priority of an active process can be changed by an UPDATE PRIORITY statement. This facility has been removed from HAL/S/V.

The statements outlined above are among the real time programming language forms to be described later in this section.

*** In most implementations it is likely to be 1, but see the User's Manual for a given implementation.

13.1.3 Process Swapping & Breakpoints

A process swap is a pair of state transitions in which one process leaves the executing state, and second enters it from the ready state. The process swap may occur because the first process has been forced into the inactive state or the waiting state, or because the second process has a higher priority than the first.

The HAL/S/V language itself makes no assumptions on where process swapping can occur. However, most implementations, depending on the object machine characteristics, limit process swapping to given places in the HAL/S/V code sequences under execution by the RTE. These places are called "breakpoints". The determination of breakpoints is a function of the HAL/S/V compiler for a given implementation, and no language construct exists to modify their existence*.

The effect of breakpoints is to superimpose a kind of time granularity on the operation of the RTE.

13.1.4 Priority Scales

The number specifying the priority P of a process is an integer in the range:

$$0 < P < 255^{**}$$

The primal process is assigned a priority of 50** by the RTE on beginning execution.

NOTE: Because of a restriction to be discussed subsequently (Section 13.4.1) the priorities which can be assigned to a process in HAL/S/V are less than or equal to 50.

13.1.5 Process Dependency

Suppose that there are two processes, A and B, and that A creates process B during the course of its execution. At the time of creation, B may be specified to be either "dependent" on or "independent"of A. If B is dependent, it means that it depends for its existence on the existence of A. If B is independent, then A may cease to exist without affecting B's existence.

---

* As an example, in the HAL/S/V-360 implementation, breakpoints occur at the end of every executable statement.

** These values are, however, implementation dependent. See appropriate User's Manual.

However, an overriding rule is that all other processes are always dependent on the primal process for their existence.

The consequences of dependency will be seen when the flow of execution through program and task blocks is described in Section 13.3, and again when the TERMINATE statement in introduced in Section 13.5.

13.2 TASK BLOCK DEFINITIONS

A task block is a static block of code interior to a program, from whence processes can be created by means of the SCHEDULE statement. Task blocks may only be defined at the program level, and not nested inside procedure or function blocks or other task blocks defined in a program.

Task block definitions are similar to program block definitions as described in Section 3, and have similar opening and closing statements.

13.2.1 Relative Position Of Task Definitions

Statements invoking a task block should always follow its block definition.

NOTE: In a language such as PASCAL which allows two processes to call each other, this is not always possible and necessitates some mechanism for forward referencing. But HAL/S/V specifically disallows this and all other forms of recursion. Hence, no forward referencing of functions and procedures is necessary or allowed.

13.2.2 Task Opening

The statement opening a task block takes the form:

```
|
| label:TASK;
|
```

1. label is any legal identifier name, and constitutes the name of the task block.

13.2.3 Task Closing

The statement closing a task block takes the form:

```
|
| CLOSE label;
```

1. The identifier label is optional.

2. If supplied, it must be the name of the task block.

Example:

```
DISPLAY:TASK;
-----------
|         |
|         |  task body
|         |
|         |
-----------
CLOSE DISPLAY;
```

13.2.4 Local Data Declarations

Local data can be declared in a task block in exactly the same way as it is declared in a procedure or function block. The declarations appear after the task opening statement, and before the first executable statement of the block. The forms of the declarations have been described in Section 4.

Examples:

General Positioning -

```
DISPLAY: TASK;
------------
|          |
|          |    local data declarations
|          |
------------

------------
|          |
|          |    executable statements
|          |
------------
CLOSE DISPLAY;
```

Particular Instance -

```
|
|  DISPLAY: TASK;
|    DECLARE S CHARACTER(10),}------local data
|            I INTEGER;
|    .
|    .
|    .
|    .
|  CLOSE DISPLAY;
|
```

13.3 FLOW OF EXECUTION IN PROGRAM AND TASK BLOCKS

The flow of execution through program and task blocks is subject to a new interpretation, based on the concepts of real time programming introduced in this section. Programs and tasks are treated together since their representations at run time are in both cases real time processes.

Execution of a process begins with the first executable statement in the corresponding static program or task block. It continues, and if not terminated by some other process, ends in one of the following ways:

1. by execution of a TERMINATE statement terminating itself;

2. by reaching the CLOSE statement of the block;

3. by execution of a RETURN statement in the block;

If execution ends by self-termination, the process goes into the inactive state and is removed from the process queue. All dependents of the process are treated likewise. This is subject to certain restrictions described in Section 13.5.

If execution ends on a CLOSE or RETURN statement, the process goes into the inactive state directly only if it has no dependents. Otherwise, it goes into a waiting state until the dependents have in their turn terminated.

13.3.1 Form Of Return Statement

The form of RETURN statement for programs and tasks is the same as for procedures:

```
|
| RETURN;
|
```

13.4 THE SCHEDULE STATEMENT

The SCHEDULE statement is an executable statement causing a new process to be placed in the process queue, or "initiated". The SCHEDULE statement specifies a task block from which the process is to be created, and the priority which it is to be given. A condition for the initiation of the process can be supplied.

Only one process derived from a given task block may be active at any given time.

13.4.1 Immediate Initiation

The following variant of the SCHEDULE statement is the simplest. It causes the creation of a process which is placed in the process queue in the ready state. The process is thus available for execution immediately.

```
|
| SCHEDULE label PRIORITY(a) DEPENDENT;
|
```

1. A process is created from the task block label and placed in the process queue in the ready state. The process created is also known by the name label.

2. a is an integer expression specifying the priority of the newly-created process. It must lie in the legal range for a given implementation. Moreover it must be lower than the priority of the scheduling process.

3. The keyword DEPENDENT is optional. Its presence denotes the dependency of the process created on the process executing the SCHEDULE statement. In its absence, the processes are independent.

NOTE: From a verification standpoint it is desirable that the effects of a process be "localized" affecting the global state only in definite and apparent ways. The ability of a low priority process to schedule other processes with arbitrary priority is counter to this goal. It renders even a very low priority process the ability in effect to usurp the CPU from any other process whatsoever by scheduling another task with high priority. Thus the restriction is placed on the SCHEDULE statement in HAL/S/V that the priority assigned to a scheduled process be lower than that of the scheduling process. This effect can always be achieved by assigning a priority of (PRIO - C) where PRIO is a system function returning the priority of the calling process and C is a positive integer such that PRIO - C > 0.

From a methodological viewpoint this restriction has the additional benefit that process priority is determined relative only to the scheduling process and not in relation to possibly unrelated processes in the global environment. Thus the generation of a priority driven scheme is discouraged though not entirely prevented.

Another effect of this restriction is that no process ever has priority more than 50* since the primal process is assigned priority 50 by the RTE.

13.4.2 Delayed Initiation

The following form of the SCHEDULE statement causes a process to be placed in the process queue in the waiting state. The process is transferred to the ready state on a specified time criterion being met. There are two variants, each with a different time criterion.

INITIATION after some duration:

```
|
| SCHEDULE label IN interval PRIORITY(a) DEPENDENT;
|
```

1. A process called label is created from the corresponding task block and placed in the process queue in the waiting state.

2. PRIORITY(a) and DEPENDENT have the same meanings as described in the previous form of SCHEDULE statement.

3. The phrase IN interval indicates that the process is to be put in the ready state after a specified interval in the waiting state. interval is an integer expression whose value specifies the duration in seconds.

4. If the value is negative or zero, the process is put in the ready state immediately.

INITIATION at a given time:

```
|
|SCHEDULE label AT time PRIORITY(a) DEPENDENT;
|
```

1. A process called label is created from the corresponding task block and placed in the process queue in the waiting state.

2. PRIORITY(a) and DEPENDENT have the same meanings as described in the previous forms of SCHEDULE statement.

3. The phrase AT time indicates that the process is to be put in the ready state at a specified real time. time is an integer expression whose value specifies the time in seconds.*

4. If the indicated time is in the past, the process is placed in the ready state immediately.

---

* This value is implementation dependent.

NOTE: The introduction of time into programming present a difficulty for verification. Programs run at different rates on different machines and often at varying rates on the same machine. The programmer in general should minimize the dependence of his results on time.

SCHEDULE statements can also specify the cyclic execution of a process until a stopping criterion is met. An explicit specification of the interval between cycles can also be given.
See: Guide/23.4 & 23.5.

13.5 OTHER REAL TIME FEATURES OF HAL/S/V

Three other real time programming statements which have already been mentioned are now described. These are the TERMINATE, WAIT, and UPDATE PRIORITY statements. Certain other useful constructs are also introduced.

13.5.1 Terminate Statement

A process is forced to the inactive state (removed from the process queue) by means of the TERMINATE statement. Its form is shown below:

```
|
| TERMINATE label ;
|
```

1. The appearance of label is optional. If present, the statement terminates an active process called label.

2. If label is absent, then the process executing the TERMINATE statement is terminating itself.

3. No process may be terminated by execution of a TERMINATE statement if it or any of its dependent processes updates global data.

In order to make independent processes truly independent, HAL/S/V places an added restriction on the operation of the TERMINATE statement. A process is only allowed to use it to terminate itself or its dependents.

* The real time origin is not speicified by the language. The origin is normally coincident with the initiation of the primal process. Some implementations allow its value to be preset at run time. See appropriate User's Manual.

Note that when a process is terminated by execution of a TERMINATE statement, all its dependents are automatically terminated at the same time.

NOTE: The ability of a process to terminate itself and its dependents presents a problem for verification if any global data items are updated by those processes. In general the synchronization of a process with its dependents occurs only by the SCHEDULE statement. Beyond scheduling, the actual execution of the processes occurs asynchronously, influenced by such implementation dependent features as the number of processors available. Hence, when a process terminates itself with concomitant termination of dependents, there is in general no way to make valid assertions concerning the state of data items updated by those dependents. This is significant only where global data is involved. Therefore, HAL/S/V includes the restriction that no process may be terminated if it updates global data. This implies also that no process may terminate itself if any dependent (or dependent of a dependent, etc.) updates global data.

Examples:

```
|
| TERMINATE;----------------self termination
| TERMINATE BETA;-----------termination of dependent
|
```

If a number of processes are to be terminated simultaneously, the TERMINATE statement can specify a list of process names:

```
|
| TERMINATE ALPHA, BETA, GAMMA;
|
```

13.5.2 Wait Statement

The WAIT statement is used to force the process executing it into a waiting state until some condition is met, whereupon it returns to the ready state. Three forms, each with a different condition, are described below.

WAIT for a duration:

```
|
| WAIT interval;
|
```

1. The statement indicates that the process is to be placed in the waiting state for a specified duration.

2. interval is an integer expression specifying the duration in seconds.

3. A negative or zero value results in the process not leaving the ready state.

WAIT until some time:

```
|
| WAIT UNTIL time;
|
```

1. The statement indicates that the process is to be placed in the waiting state until some given time.

2. time is an integer expression specifying the time of return to the ready state, in seconds.*

3. Specification of a time in the past results in the process not leaving the ready state.

WAIT for dependents

```
|
| WAIT FOR DEPENDENT;
|
```

1. The statement indicates that the process is to be placed in the waiting state until all its dependent processes have terminated.

2. If there are no dependents, the statement has no effect.

Examples:

```
|
|  WAIT UNTIL DELTA + 10;
|  WAIT 15;
|  WAIT FOR DEPENDENT;
|
```

13.5.3 Update Priority Statement

No UPDATE PRIORITY statement is allowed in HAL/S/V.

NOTE: For the reasons cited in Section 13.4 it is undesirable that processes be able to tamper with the priorities of other processes except in very controlled ways. To be consistent with the restrictions imposed there, we could require that any modifications of the priority of a process only lower it by some constant. However, this solution is unsatisfactory for the following reasons. If a process is inactive updating its priority in the manner specified could only lower its priority relative to that of other active processes and possibly result in delaying its use of the CPU or changing its state from executing to waiting. But unless the process

---

* See the discussion on the SCHEDULE statement in Section 13.4 for a footnote remarking on the real time origin.

shares data with other processes these have no real effect. If it does share data, the effects of lowering priority can be generated in a more controlled manner either by scheduling the process with a lower priority initially or by use of the WAIT statement. Hence UPDATE PRIORITY is disallowed.

13.5.4 Real Time Built-In Functions

Two built-in or library functions are of utility in constructing real time programs:

Function	Comments
RUNTIME	returns the current value of real time as a scalar, in seconds
PRIO	returns the priority of the process invoking the function as an integer

NOTE: As noted earlier, certain system functions may not be invoked by procedures and functions since they return time dependent values. RUNTIME and PRIO are two such system functions and are subject to the restrictions described in Section 11.4.4.

13.5.5 Major State Indication

HAL/S/V allows the use of a process name as a Boolean variable which returns true if and only if the process is in an active state. This is not allowed in HAL/S/V.

NOTE: Any attempt to synchronize processes by such a mechanism presents grave difficulties for verification since it makes the behavior of a program depend upon other, possibly independent processes whose rate of execution may differ under differing circumstances and on different machines.

> The constructs described above enable real time processes to be manipulated according to time criteria. Other constructs enable their manipulation according to "event" criteria. HAL/S/V "events can signal conditions to the RTE. Their values can be manipulated by the user thus indirectly controlling the real time process states. See: Guide/24.
>
> The problem of controlling the

sharing of data by two or more processes is also important.
See: Guide/26.4.

CHAPTER 14

SUMMARY OF PART I

Part I of the Programmer's Guide has presented a wide variety of the simpler constructs of the HAL/S/V language. It has laid sufficient ground work for the understanding of more complex language forms which are to be presented in Part II.

Manual 2

HAL/S/V:
A Verifiable Version of HAL/S
Volume - 2

James C. Browne
Donald I. Good
Anand R. Tripathi
William D. Young

December 31, 1979

INSTITUTE FOR COMPUTING SCIENCE AND COMPUTER APPLICATIONS
The University of Texas at Austin
Austin, Texas 78712

TABLE OF CONTENTS

CHAPTER 15

COMPOOLS AND COMSUBS

15.1 RELATIONS BETWEEN PROGRAMS, COMPOOLS AND COMSUBS

The compools and comsubs referenced by a program are themselves separately compilable entities. For example, when a program invokes an external procedure, which shares with it the use of data in a single compool, then a total of three separate compilation units is involved.*

Section 3 of the Guide described one kind of compilation unit - the program block - but there are four kinds of compilation units in the HAL/S/V language:

1. PROGRAM, the only independently executable compilation unit;

2. EXTERNAL PROCEDURE, callable from a program or any other comsub;

3. EXTERNAL FUNCTION, also callable from a program or any other comsub;

4. COMPOOL, defining data shared by programs and comsubs, but containing no executable code.

The HAL/S/V language insists upon a full declaration of all data, and invariably checks the compatibility of function and procedure definitions with their invocations. These precautionary measures are specifically extended to compool data and comsubs through the use of so-called "block templates".

Every program or comsub which references compools or other comsubs must be provided with block templates of the compilation units referenced.

* COMPOOL TEMPLATE - contains data declarations

---

* The object modules resulting from their compilation have to be "link-edited" to produce a single executable load module.

identical with those of the compool itself, so that the referencing compilation unit possesses a complete description of the data.

* EXTERNAL FUNCTION TEMPLATE - contains an input parameter list identical with that of the external function itself, so that the compatibility of its invocations by the referencing compilation unit can be verified.

* EXTERNAL PROCEDURE TEMPLATE - contains input and assign parameter lists identical with those of the external procedure itself, so that the compatibility of its invocations by the referencing compilation unit can be verified.

The required block templates are included in the compilation units which reference the corresponding compools and comsubs. No external procedure or function unit may use a compool block directly, because no global referencing is allowed in HAL/S/V. This implies that compool templates do not appear with the external procedures and functions.

To summarize, when the term "compilation unit" was introduced in Section 3 of the Guide, its meaning was the same as "program block" because the existence of compools and comsubs had not been considered. Now it is apparent that a compilation unit does not necessarily contain executable code (it may be a compool), and neither is it necessarily just a single block of executable code (one or more templates may be included in it).

In HAL/S/V block templates are designed to eliminate incompatibility between separately compiled modules as a source of software unreliability. It may be objected however that no language construct can force the properties of a compool or comsub to be reflected correctly in the corresponding block template.* The use of correct templates is generally insured by an implementation dependent software management scheme. Part of such a scheme would be the automatic generation of block templates during compilation of the corresponding compools and comsubs.

15.2 THE COMPOOL BLOCK

The compool block has been introduced as an external block of data accessible to programs and comsubs with which the appropriate block template is included. It consists of opening and closing statements delimiting a sequence of data declarations.

---

* Neither can it ensure that the object modules "link-edited" together are the correct versions.

15.2.1 Compool Opening

The statement opening a compool block takes the form:

```
label:  COMPOOL;
```

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

15.2.2 Compool Closing

The compool block is closed with the statement:

```
CLOSE label;
```

1. The identifier label is optional.

2. If label is supplied, it must be the label supplied on the opening statement of the block.

15.2.3 Compool Data Declarations

Declaration of data in a compool differs in no respect from data declarations in a program, as described in Section 4. In particular, there is no objection to the initialization of data in a compool.

The identifier names used to declare data in a compool body should not be used to declare any other data item in outer most levels of the programs using that compool block.

15.3 EXTERNAL PROCEDURE AND FUNCTION BLOCKS

Comsubs have been introduced as external function and procedure blocks which may be called from programs or other comsubs.

The forms of external function and procedure blocks are identical with ordinary function and procedure blocks, whose definitions were described in Section 11. Likewise, they are invoked in a manner identical with that described in Section 11. However, the external procedures and functions do not contain any compool templates because no direct accessing of global data is allowed in HAL/S/V.

15.4 BLOCK TEMPLATES

Block templates indicate the properties of compools and comsubs to the program or comsub referencing them. Their form is similar to the corresponding compool or comsub.

15.4.1 Compool Templates

A compool template is identical with its corresponding compool block except that the opening statement is modified by the keyword EXTERNAL:

```
|
|label:  EXTERNAL COMPOOL;
|
```

1. label is the name of the corresponding compool block.

Example:

Compool Block:

```
|  POOL:  COMPOOL;
|     DECLARE INTEGER DOUBLE, I, J, K;
|     DECLARE CC CHARACTER(10);
|  CLOSE POOL;
```

Corresponding Template:

```
|
|  POOL:  EXTERNAL COMPOOL;
|     DECLARE INTEGER DOUBLE, I, J. K;
|     DECLARE CC CHARACTER(10);
|  CLOSE POOL;
```

15.4.2 External Procedure Templates

An external procedure template differs from its corresponding procedure block in the following respects:

1. The body of the block is empty except for declarations describing the attributes of input and assign parameters;

2. The opening statement is modified as shown below by the keyword EXTERNAL.

```
|
|label:      EXTERNAL      PROCEDURE(I1,      I2,...)
ASSIGN(A1,A2,...);
|
```

1. label is the name of the corresponding procedure block.

2. i1,i2,... and a1,a2,... are lists of input and assign parameters respectively, identical with those in the corresponding procedure block.

Example:

External Procedure:

```
| FIXIT:  PROCEDURE (INCR) ASSIGN (RESULT);
|      DECLARE RESULT INTEGER,
|              INCR INTEGER;
|      DECLARE DELTA CONSTANT INTEGER (14);
|      RESULT = RESULT + DELTA INCR;
|  CLOSE FIXIT;
```

Corresponding Procedure Template:

```
|  FIXIT:  EXTERNAL PROCEDURE (INCR) ASSIGN (RESULT);
|       DECLARE RESULT INTEGER,
|               INCR INTEGER;
|  CLOSE FIXIT;<---------
                        |
                  no local data or executable code.
```

15.4.3 External Function Templates

An external function template differs from its corresponding function block in the following respects:

1. the body of the block is empty except for declarations describing the attributes of input parameters;

2. the opening statement is modified as shown below by the keyword EXTERNAL.

```
|
| label:  EXTERNAL FUNCTION(i1,i2,...) attributes;
|
```

1. label is the name of the corresponding function block.

2. i1,i2,... is a list of input parameters identical with those of the corresponding function block.

3. attributes defines type, precision and size attributes, of the corresponding function block.

Example:

External Function:

```
|
|  SWITCH:  FUNCTION(ARG) BOOLEAN;
|       DECLARE ARG INTEGER DOUBLE;
|       IF ARG<0 THEN RETURN FALSE;
|       RETURN TRUE;
|  CLOSE SWITCH;
```

Corresponding Function Template:

```
|
|  SWITCH:  EXTERNAL FUNCTION(ARG) BOOLEAN;
|       DECLARE ARG INTEGER DOUBLE;
|  CLOSE SWITCH;
|
```

Function templates, like procedure templates, may also contain structure template definitions.

CHAPTER 16

ADDITIONAL DATA INITIALIZATION FORMS

This Section supplements the discussion in Section 4.3 on initialization by introducing the following topics:

* the implied repeated use of initial lists;
* other ways of reducing the length of an initial list;
* partial initialization of a data item;
* control of the actual occurrence of initialization.

16.1 IMPLIED INITIAL LIST REPETITION

Section 4.3 stated that for single-valued data items, only one literal value can be supplied in an INITIAL/CONSTANT specification. It stated that for multi-valued data items, two alternatives are possible:

1. The number of literal values specified in the INITIAL/CONSTANT specification matches the total number of elements implied by the data declaration;
2. Only one literal value is supplied, in which case that same initial value is given to all elements implied by the data declaration.

16.2 USE OF REPETITION FACTORS

If a number of consecutive values in an INITIAL/CONSTANT specification are identical, they may be replaced by one value and a repetition factor:

```
          r  r+1  r+2     r+n
...i ,i   ,i   ,...i   ,....
```

```
          r    r+1
...i ,n#i   ,....
```

1. In both forms, i represents a literal value in an INITIAL/CONSTANT specification.

2. In the first form i(r+1),...,i(r+n) are literal values.

3. The second form shows the replacement of i(r+1),...,i(r+n) by n#i(r+1), where n is a positive nonzero integer.

If a sequence of values is repeated over and over, they may be treated in a similar way. The sequence is written once, enclosed in parenthesis, and prefaced with a repetition factor.

Example:

```
|
|  DECLARE S ARRAY(10) INTEGER
|         INITIAL(1,2,3,4,5,6,3,4,5,6);
```

may be replaced by:

```
|  DECLARE S ARRAY(10) INTEGER
|         INITIAL(1,2,2#(3,4,5,6));
```

The factored form may be nested if necessary, and can be especially convenient in the initialization of multi-dimensional arrays.

Example:

```
|
| DECLARE V ARRAY(3,2,2)
|       INITIAL(1,2,3,2,3,1,2,3,2,3,1,2);
```

may be replaced by:

```
| DECLARE V ARRAY(3,2,2)
|       INITIAL(2#(1,2,3,2,3),1,2);
```

which may in turn be replaced by:

```
|
| DECLARE V ARRAY (3,2,2)
|       INITIAL(2#(1,2#(2,3)),1,2);
```

16.3 PARTIAL INITIALIZATION

In HAL/S/V, partial initialization of data is not permitted. The partial initialization makes the proof methods more cumbersome because of the introduction of the type "undefined".

16.4 STATIC AND AUTOMATIC INITIALIZATION

Although initialization has been discussed at length, the circumstances under which it actually is effective have not been considered. In particular, it has not been stated whether initialization is effective only on the first entry of execution into a block, or on every such entry.

STATIC initialization is initialization effective only on first entry into a block. It is called static because generally it results in the generation of initialized data areas by a compiler, rather than executable code.

AUTOMATIC initialization is intialization on every entry into a block. It generally results in executable code being generated by a compiler.

In HAL/S/V both "STATIC" and "AUTOMATIC" initialization features of HAL/S have been retained. The reason for retaining STATIC initialization feature is to enable construction of abstract data types in the language. However, it does not permit reentrent procedures to have STATIC initialization thereby not allowing sharing of a single copy of data by multiple invocations of a procedure. In case of AUTOMATIC initialization at each invocation of the procedure a new copy of data is created and initialized, "automatically" can be asserted at every entry into a procedural block.

The keywords STATIC or AUTOMATIC attached to the declaration of an initialized data item serve to distinguish between two forms.

16.4.1 Legal Use Of Specification

No STATIC/AUTOMATIC specification may be used in the declaration of initialized data items in a compool (see Section 15.2). A COMPOOL block is not executable, so the question of entry does not arise. Initialization is viewed as taking place before execution of a program begins.

No data item initialized by the CONSTANT specification may possess a STATIC/AUTOMATIC specification. Such data items are viewed as being similar to literals, so that the question of entry again does not arise.

STATIC/AUTOMATIC specifications can appear, then, in data declarations in any kind of block except for COMPOOL blocks.

16.4.2 Form Of Static Specification

In the absence of any explicit indication, static initialization is assumed. Alternatively, the keyword STATIC may be used, placed either before or after the INITIAL specification.

Examples:

```
|
|   DECLARE I INTEGER STATIC INITIAL(5),
|           J INTEGER INITIAL (0) STATIC,
|           K INTEGER INITIAL (1);
|
```

16.4.3 Form Of Automatic Specification

The keyword AUTOMATIC is used, placed either before or after the INITIAL specification.

```
|
|   DECLARE I INTEGER AUTOMATIC INITIAL(5),
|           J INTEGER INITIAL(0) AUTOMATIC;
```

CHAPTER 17

BIT STRINGS

The form and use of Boolean data was discussed at various points in Part I of the Guide. Their stated purpose was the manipulation of binary valued (logical) quantities. The ability to handle strings of binary values is often useful. In HAL/S/V, this ability is characteristic of the "bit string" data type, which is essentially a generalization of the Boolean data type already described.

17.1 BIT STRING LITERALS

Boolean literals were described in Section 2. These are the corresponding literal forms for bit string quantities:

```
BIN'bbbbbb'
OCT'oooooo'
HEX'hhhh'
DEC'dddd'
```

1. In the above forms,

   b ~ binary digit
   o ~ octal digit
   h ~ hexadecimal digit
   d ~ decimal digit

2. The number of binary digits represented must not exceed 32.*

Examples:

```
BIN'10110'
HEX'FAC2'
OCT'777'
```

---

* This number may vary between implementations. See appropriate User's Manual.

Note that BIN'0' = FALSE = OFF and BIN'1' = TRUE = ON

17.2 DECLARATION OF BIT STRING DATA ITEMS

The basic declaration statement for bit string data items is shown below:

```
DECLARE name BIT(n);
```

1. name is any legal identifier.

2. n specifies the length of the bit string (i.e. the number of binary digits in it). It must be in the range 1 <= n <= 32.*

Examples:

```
|
| DECLARE B1 BIT(16);
|
```

Note that the following two forms are equivalent:

```
|
| DECLARE B2 BIT(1);
| DECLARE B2 BOOLEAN;
|
```

Declarations of bit string data items can be integrated into compound declarations as described for other data types in Section 4.2.

17.2.0.1 Initialization -

Initialization of bit string data items follows the rules given in Section 4.3, using bit string literals in the list of initial values.

Examples:

```
|
| DECLARE B16 BIT(16) INITIAL(HEX'FFFF');
| DECLARE B1 BIT(1) CONSTANT(TRUE);
| DECLARE B ARRAY(2) BIT(3) INITIAL(OCT'7',OCT'5');
|
```

---

* This value may vary between implementations. See appropriate User's Manual.

Literals are padded or truncated as required to fit the data item initialized:

```
|
| DECLARE B8 BIT(8) INITIAL(OCT'770');
| DECLARE B11 BIT(11) INITIAL(HEX'FF');
|
```

results in

B8 = 11111000, B11 = 00011111111

17.3 BIT STRING SUBSCRIPTING

Subscripting forms for bit string data items are similar to those for character data items, as described in Section 6.

17.3.1 Unarrayed Bit Strings

In bit strings, bit positions are indexed left to right starting from 1. In the subscript forms given below, STRING represents an unarrayed bit string data item of length L.

To select the ath bit from STRING:

```
STRING
       a
```

1. a is an integer expression in the range 1 <= a <= L.

To select a bits from STRING, starting from the Bth:

```
STRING
      a at B
```

1. a is an integer literal value in the range 1 <= a <= L.
2. B is an integer expression in the range 1 <= B <= L - a + 1.

To select a substring starting with the ath bit of STRING, and ending with the Bth:

```
STRING
      a to B
```

1. a and B are integer literal values in the range 1 <= (a,B) <= L.

2. B => a.

If a data item is declared to be Boolean, it is really defined as a 1-bit string. It may therefore possess component subscripting consistent with the above rules, even though in this case it performs no useful purpose.

17.3.2 Arrayed Bit Strings

The subscripting forms for arrayed bit string data items are as described in Section 6.2. The colon following an array subscript is mandatory.

17.4 BIT STRING OPERATIONS

Section 7.3 of the Guide outlined the logical operations which could be performed on Boolean data. Operations on bit strings are an extension of these. HAL/S/V recognizes the following operations:

Symbol	Purpose
& and	conjunction
| or	disjunction
- not	complement
|| cat	catenation

17.4.1 Complement

The complement operation complements the logical value of every bit in the bit string.

Symbolic form: - R

1. The operand R is a bit string.

Example:

If B is an 8-bit string with B = 11000101
then -B = 00111010

17.4.2 Disjunction

The disjunction operation causes the logical values of corresponding bit positions in the operands to be OR'ed together.

Symbolic form: L | R

1. The L and R operands are bit strings.

2. The two operands must be of equal length.

Example:

If B is a 5-bit string with B = 00100
and BB is a 5-bit string with BB = 10110
then B&BB = 00100

Note that a 5-bit result is obtained.

17.4.3 Catenation

The two operands are catenated to form one longer bit string.

Symbolic form: L || R

1. The L and R operands are bit strings.

2. The L operand is catenated to the left of the R operand.

3. If the sum of the lengths exceeds 32 * the L operand is left truncated as required.

Example:

If B is a 12-bit string with B = 7E0
and BB is a 24-bit string with BB = 42F50B
then B||BB = E042F50B ,
the left-most 4 bits of B being truncated.

17.4.4 Precedence

The following table summarizes the precedence rules for bit string operations, and is an extension of the table for Boolean operations given in Section 7.4.

---

* This value may vary between implementations. See appropriate User's Manual.

Symbol	Precedence	Purpose
	FIRST	
-, NOT	1	complement
||, CAT	2	catenation
&, AND	3	conjunction
|, OR	4	disjunction
	LAST	

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

17.5 BIT STRING ASSIGNMENT

Bit string assignment is an extension of Boolean assignment as described in Section 8.4.

1. The operand types are both bit string:

L-type	R-type
BIT STRING	BIT STRING

2. The logical value of each bit position of the R-operand is transferred to the receiving data item.

3. Both operands must be of equal length.

Examples:

If B is a 6-bit string,
and BB is a 6-bit string with BB = 101101,
then

```
B = BIN '110110';
```

results in B = 110110,
and

```
B = BB;
```

results in B = 101101

17.6 BIT STRINGS IN CONDITIONAL CONSTRUCTS

Execution of the HAL/S/V IF statement described in Section 9.1, and of the DO WHILE statement described in Section 10.2, are controlled by the logical value of an expression which was stated to be either Boolean or relational in type. Bit string expressions may be used directly in place of Boolean expressions or as parts of relational expressions in such statements.

17.6.1 Direct Use Of Bit Strings

If a bit string expression is used instead of a Boolean expression in an IF statement or DO WHILE statement, the following rule applies:

> The bit string expression is considered to be evaluated as FALSE if the right-most bit is 0 and TRUE if it is 1.

Examples:

```
Let B be a 4-bit string with B = 1101
and BB be a 4-bit string with BB = 0010

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

The condition B|BB = 1111 : the right-most bit is 1, so that the statement

```
X = 0;
```

is executed.

The statement group

```
|   DO WHILE B;
|       .
|       .
|       .
|   END;
```

is executed repetitively until the right-most bit of B becomes 0. The values of other bits in B do not affect this process.

17.6.2 Bit Strings In Relational Expressions

Section 9.2 showed how data items of each type, including Boolean, could be combined into relational expressions which evaluated to either TRUE or FALSE. Using the same nomenclature as that section, bit strings can be used in Class II comparative operations only:

Symbol	Purpose	Class
=	equals	
not = - =	not equals	II

The rules for bit string comparisons are given below:

Symbolic form: L NOT = R

1. The only legal type combination for the L and R operands is:

L-type	R-type
BIT STRING	BIT STRING

2. The result is FALSE if the L and R operands are of unequal lengths.

Examples:

If B is a 4-bit string with B = 1101,
and BB is a 3-bit string with BB = 101,
then
B = BIN'1101' is TRUE
and
B = BB is FALSE

The above comparative operations can be combined as described in Section 9.2, using the given precedence rules. Note that the important rule that Boolean and relational expressions cannot be mixed extends to bit string expressions as well. However, this may impose serious limitations on writing assertions.

The following are some examples clarifying the use of bit string relations.

Examples:

Let B be a 3-bit string with B = 110,
and I be an integer with I = 5

```
IF (B=BIN'110')&(I>4) THEN I = 0;
```

In the above IF statement, both comparative operations evaluate to TRUE so that the condition is itself TRUE so that the and the assignment

```
I = 0;
```

is executed. The statement

```
IF (B-=BIN'01')&BIN'11' THEN I = 0;
```

is illegal because a relational expression is being mixed with a bit string literal to form the condition of the IF statement.

Note that the statement

```
IF B-=BIN'101' & BIN'11' THEN I = 0;
```

is also illegal because the syntax is ambiguous. Parentheses must be used to specify its only legal interpretation:

```
IF B-=(BIN'01' & BIN'11') THEN I = 0;
```

17.7 BIT STRING ARGUMENTS AND PARAMETERS

Section 11 described procedure and function blocks and how they were invoked. Procedures and functions may be defined with bit string parameters, and be passed bit string arguments.

17.7.1 Form Of Bit String Parameters

Any input parameter of a function or any input or assign parameter of a procedure may be declared to be of bit string type, using the forms of declaration described in Section 17.2.

Example:

```
| FLAGS:  PROCEDURE(B1) ASSIGN(B2);
|    DECLARE B1 BIT(16),
|            B2 BIT(8);
|
|                      } procedure body
|
| CLOSE FLAGS;
```

17.7.2 Argument Passage

An argument of a function or procedure invocation corresponding to a bit string parameter must conform to the following rules:

INPUT PARAMETER. The transmission of the argument can be viewed as its assignment to the input parameter. The following rules apply to both procedures and functions:

1. The corresponding argument must be of bit string type.

2. The input parameter must be of the same length as the argument.

ASSIGN PARAMETER. The following rules apply for the matching of arguments to bit string assign parameters:

1. The assign argument must be a declared HAL/S/V bit string data item.

2. The length of the argument must be the same as that of the parameter.

3. The argument may not possess subscripting.

These rules are only relevant to procedures.

17.8 BIT STRING FUNCTIONS

In Section 11.2 it was stated that functions of any legal HAL/S/V type could be created. Accordingly, it is legal to define functions of bit string type.

17.8.1 Block Definition

The opening statement of the function block takes the form:

```
label:  FUNCTION(i, i,...) BIT(n);
```

1. label is the name of the function.

2. i, i,... is the list of input parameters.

3. n indicates the number of bits, and lies in the range 1 <= n <= 32.*

The closing statement is as described in Section 11.2.

Example:

```
|   F1:  FUNCTION(B) BIT(5);
|
|                    } function body
```

---

* This value may vary between implementations. See appropriate User's Manual.

```
|
|   CLOSE F1;
```

17.8.2 Return Of Bit String Quantities

The RETURN statement should contain a bit string of the same length specified in the function declaration. No implicit length conversion by truncating or padding with zeroes is permitted.

17.9 BIT STRINGS IN INPUT/OUTPUT

Bit strings may participate in input/output in the same way as other data types, as described in Section 12. The format of bit string data fields for input and output are described in Appendix F.

CHAPTER 18

MULTI-DIMENSIONAL ARRAYS

Section 4.1 stated that it was possible to declare an array or table of any given data type. Section 4.2 showed the form of declaration for 1-dimensional arrays. HAL/S/V actually supports arrays of multiple dimensions.

First, the general form of declaration is presented. Then, some remarks on the order of initialization precedes a discussion of the subscripting of multi-dimensional arrays.

18.1 DECLARATION

To declare an array of any data type and of any legal dimension, the following form of declaration is used:

```
DECLARE name ARRAY(n, n,...) attributes;
```

1. name is the name of the data item declared.

2. attributes are the attributes appropriate to the data type being declared.

3. n, i = 1, 2... are the sizes corresponding to each array dimension. The upper limit on i is 3.* The number of elements in any dimension must lie in the range 1 <= n <= 32768.**

Examples:

---

* The limiting number of dimensions may vary between implementations; See appropriate User's Manual.

** This value may vary between implementations. See the appropriate User's Manual. In some implementations, there may also be restrictions upon the contexts in which very large arrays may be used.

```
DECLARE S ARRAY(5,5) INTEGER,
          W ARRAY(2,2,1000) INTEGER;
```

18.2 ORDER OF INITIALIZATION

Section 4.3 stated the order of initialization of elements of 1-dimensional arrays of any data type. The order for multi-dimensional arrays is generated by the rules given in Appendix C.

The following examples illustrate the effect of these rules in initialization of 2- and 3-dimensional arrays.

Example:

```
DECLARE I ARRAY(2,3) INTEGER INITIAL(1,2,3,4,5,6);
```

18.3 SUBSCRIPTING

Section 6.2 gave the forms of array subscripting for 1-dimensional arrays. To summarize, the following kinds of subscript could be used:

1. simple indexing, to select one array element;
2. AT-partitioning, to select a sub-array of a given size starting from a given index value;
3. TO-partitioning, to select a sub-array starting from one given index value and ending on a second.

In multi-dimensional arrays, such subscripting can be applied to each dimension of the array.

18.3.1 Array Subscripting Only

Let TABLE be an n-dimensional array. The general subscripting form is then:

```
TABLE
     array 1,...,array n:
```

1. array stands for any array subscript of the form given in Section 6.2.

2. The colon is optional for integer data types only.

3. Any array may be replaced by an asterisk to denote specification of every element in that dimension.

18.3.2 Array And Component Subscripting

If TABLE represents an n-dimensional array of vector, matrix, character or bit string type, then the general form when component and array subscripting is present is:

```
TABLE
     array 1,...,array n:component
```

1. array stands for any array subscript of the form given in Section 6.2.

2. component represents any form of component subscripting legal for the data type of TABLE, as described in Section 6.1 and 17.3.

3. Any array may be replaced by an asterisk to denote specification of every element in that dimension.

18.3.3 Component Subscripting Only

When only component subscripting is required, array subscripting cannot be totally omitted, but must rather be replaced with asterisks. If, as before, TABLE represents an n-dimensional array of vector, matrix, character or bit string type, then the general form is:

```
TABLE
     *,...,*:component
```

1. n asterisks correspond to n dimensions of absent array subscripting.

2. component represents any form of component subscripting legal for the data type of TABLE.

Literal subscripts may alternatively be expressions computable at compile time.
See: Guide/Appendix D.

For a complete description of all subscript forms see Spec./5.3.

CHAPTER 19

STRUCTURES

Section 4.1 of the guide introduced some of the types of data definable in the HAL/S/V language. It further made reference to the fact that "hierarchical organizations of data items" exist in the language. It is the purpose of this Section to describe the form and use of these so-called "structures" data.

The HAL/S/V array feature is a useful construct for forming aggregates of data items, if they are homogeneous in attributes. Frequently, however, it is of great convenience to be able to form aggregates of data items with heterogeneous attributes. In addition, requirements may exist to reference not only the aggregate as an entity, but also subsets of it, or subsets of subsets of it. The HAL/S/V STRUCTURE data type fulfills both of these requirements.

19.1 HAL/S/V STRUCTURE CONCEPTS

HAL/S/V data structures have two characteristic properties:

1. Data items or arrays of almost any type can be combined to form a structure.

2. The data items can be organized into a tree-like hierarchy (similar in concept to a geneological tree, for example.)

The tree consists of nodes connected by "branches". Every "leaf" node of the tree corresponds to one of the actual data items making up the aggregate. The whole tree can be referenced by using the name of the "root" node. Subsets of the tree can be referenced by using the name of the appropriate "fork" node.

The conversion consists of recording the name of each node (root, fork or leaf) and its level when the tree walk passes it in a simple pre-order traversal.

The reverse conversion consists of the following steps. First draw the "root" node appearing at the top of the list. Then, treat each of the remaining nodes in order as follows:

1. Draw the node to the right of previous node with the same level number (if any), and under nodes with smaller level numbers.

2. Connect it by a "branch" to the last-connected node with a level number one smaller.

In HAL/S/V language, the specification of a structure tree organization is separated from the declaration of the structure or structures possessing that organization.

> STRUCTURE TEMPLATES are used to specify structure tree organizations in a linear list representation. A structure template specifies all nodes in a tree from level one downwards.
>
> STRUCTURE DECLARATIONS are used to declare structures possessing pre-defined templates. For reasons which will become apparent when the referencing of structure is considered, the declared name of the strucutre is assigned as the "root" node name of the tree organization.

In the remainder of the section, structures will be referred to as data items, since even though they are aggregates of data items, they can be manipulated as entities in themselves.

19.2 STRUCTURE TEMPLATES

The structure template is the HAL/S/V construct which defines the structure tree organization in the form of a linear list. It defines by name and level all "fork" and "leaf" nodes in a tree from level one downwards.

In the HAL/S/V implementation of structure trees, the following nomenclature is used.

> TEMPLATE NAMES are names identifying structure templates. The appear as part of the template specification, and also in structure declarations.
>
> MINOR STRUCTURE NODES are the "fork" nodes of a structure template.
>
> STRUCTURE TERMINALS are the "leaf" nodes of a structure template. Every structure terminal is one of the data items comprising the structure aggregate.

19.2.1 General Form Of A Template

The form of a structure template consists of its name followed by a specification of all its minor structure and structure terminal nodes.

19.2.1.1 OVERALL FORM -

The overall form is as follows:

```
|
|  STRUCTURE name :
|    node, node, ....
|  ...node ;
|
```

1. name is the structure template name, and is any legal HAL/S/V identifier name.
2. node , node ,...node is a list of nodes forming the tree organization.
   1 2 n

19.2.1.2 MINOR STRUCTURE NODES -

The form of a minor structure node of a template is as follows:

```
n name
```

1. n is the level number of the node.
2. name is the name of the minor structure node, and may be any legal identifier name.

19.2.1.3 STRUCTURE TERMINAL NODES -

The form of a structure terminal node of a template is as follows:

```
n name attributes
```

1. n is the level number of the node.
2. name is the name of the structure terminal node, and may be any legal identifier name.
3. attributes consists of array, type, size and other attributes applicable to data items.

4. The following data types are legal as structure terminals:

INTEGER	BOOLEAN
	BIT STRING
	CHARACTER
	STRUCTURE

There is never any confusion as to whether a node is a structure terminal or a minor structure since the level number sequence is sufficient to distinguish the two cases. Structure terminals of structure type are a special case which is discussed later.

19.2.2 Restrictions

The attributes attached to the specification of a structure terminal node are written in the same form and order as in a declaration statement (described in Section 4 and expanded in Sections 16, 17.2, and 18.1). However, the following restrictions are made:

1. No INITIAL/CONSTANT specification can be applied to a structure terminal.

2. No STATIC/AUTOMATIC specification can be applied to a structure terminal.

Example:

```
|
|   STRUCTURE Q:
|     1 QT CHARACTER(80),
|     1 QN1,
|       2 QI INTEGER
|       2 QS ARRAY(100)
|     1 QN2,
|       2 QM ARRAY(3,3) OF CHARACTER;
|       2 QB BOOLEAN;
|
```

19.2.3 Location Of Structure Templates

Structure templates are essentially parts of data declarations and therefore must appear before the first executable statement of the program or other block in which they are coded.

19.3 STRUCTURE DECLARATIONS

Structure declarations are used to declare structure data with a tree organization defined by a pre-existing structure template. Structure declarations are in the same general form as declarations of other kinds of data items, as described in Section 4.

19.3.1 Basic Form Of Declaration

The basic form of structure declaration is shown below:

```
DECLARE name a-STRUCTURE;
```

1. name is the name of the structure data item, and may be any legal identifier name.

2. a is the name given to a pre-existing structure template which specifies the tree organization of the structure being declared.

Note that the structure template referenced by a structure declaration must have been defined previously in the same block, or have been declared in a block enclosing the block containing the declaration.

Examples:

form of declaration -

```
|
|   STRUCTURE Q:
|     1 QA INTEGER,
|     1 QB CHARACTER(80),
|     1 QC BOOLEAN;
|   .
|   .
|   .
|   DECLARE ZZ1 Q-STRUCTURE;
|   DECLARE ZZ2 Q STRUCTURE;
```

Structure declarations can be integrated into compound declarations of the kind described in Section 4.2.

Example:

```
|
|   DECLARE A INTEGER,
|           B Q-STRUCTURE,
|           C CHARACTER(80);
|
```

19.3.2 Multiple Copy Structures

Structures can be declared to have multiple copies of the data specified by the tree organization. Although the form of specification is different from HAL/S/V arrays, they can in some contexts be viewed as arrays of structures.

The data declarations for a multiple-copy structure takes the following modified form:

```
DECLARE name a-STRUCTURE(n);
```

1. name is the name of the structure.
2. a is the name of the predefined structure template.
3. n is the number of copies of the data required. It must lie in the range 1 <= n <= 32768.*

19.3.3 Initialization Of Structures

Structures are initialized by supplying an INITIAL/CONSTANT specification with the structure declaration, rather than with the template. The specification is added to the declaration as described in Section 4.3.

Example:

```
|
|   STRUCTURE Q:
|      1 QI INTEGER
|      1 QS CHARACTER(2);
|   .
|   .
|   .
|   DECLARE Z Q-STRUCTURE INITIAL(5,"ME");
|
```

The order of initialization for structures is as follows:

> SINGLE-COPY STRUCTURES. The number of literal values in the list (or implied by the use of repetition factors) must equal the total number of elements summed over all the structure terminal nodes. Each structure terminal is initialized in the order it appears in the structure template, according to the rules given in Section 4.3 and further expanded in Sections 16 and 18.2.

---

* This value may vary between implementations. See appropriate User's Manual.

MULTIPLE-COPY STRUCTURES. The number of literal values in the initial list may either match the total number of elements summed over all copies, or match the number in one copy, in which case all copies are identically initialized. Each copy is initialized in turn in order of increasing index, according to the rules for single-copy structures.

These ordering rules are a restatement of those given in Appendix C. The supplementary initialization forms described in Section 16 are fully applicable to structure data types.

19.4 NESTED STRUCTURES

Section 19.2 stated that structure terminal nodes could themselves be of structure type. The effect of this is to nest a second template into the first, thus expanding the tree organization of the former.

Example:

```
|
|   STRUCTURE A:
|     1 AI INTEGER,
|     1 A1,
|       2 AC CHARACTER(80),
|       2 AB BOOLEAN;
|
|   STRUCTURE B:
|     1 BS INTEGER,
|     1 B1,
|       2 BV ARRAY(3) OF BOOLEAN,
|       2 BA A-STRUCTURE;
|
```

The structure template B is in many aspects like a template C given by:

```
|
|   STRUCTURE C:
|     1 BS INTEGER,
|     1 B1,
|       2 BV ARRAY(3) OF BOOLEAN,
|       2 BA,
|         3 AI INTEGER,
|         3 A1,
|           4 AC CHARACTER(80),
|           4 AB BOOLEAN;
|
```

which has superficially the same tree organization.

19.4.1 Restrictions

A structure terminal of structure type may not possess multiple copies.

Example:

The following template is illegal:

```
|
|   STRUCTURE Q:
|      1 QI INTEGER,
|      1 QS T-STRUCTURE(20);
|
```

NOTE: Recursion definitions are explicitly prohibited in HAL/S/V. Therefore, a structure template may never possess a node of that same structure type.

Example:

The following is illegal:

```
|
|   STRUCTURE Q:
|      1 QI INTEGER;
|      1 QQ Q-STRUCTURE;
|
```

As is the following:

```
|
|   STRUCTURE Q:
|      1 QT T-STRUCTURE;
|   STRUCTURE T:
|      1 TQ Q STRUCTURE;
|
```

19.5 QUALIFICATION AND STRUCTURE REFERENCING

The basic types of data item introduced in Section 4 are referenced merely by stating their names in the desired context. A structure in its entirety can be referred to in the same way. Referring to part of a structure is more complex, however, because in general more than one structure may possess the tree organization expressed by a particular template.

19.5.1 The Qualified Reference Concept

Any node of a structure other than the "root" node is referred to by a composite or "qualified" name which is generated conceptually in the following way. A tree walk is started at the "root" node, and continued down to the node to be referenced. The names of all the nodes traversed, including the "root" and final nodes, are listed. The resulting composite or "qualified" name is an unambiguous reference to the desired "leaf" node (given certain restrictions on duplicate naming which are to be described).

19.5.2 Referencing Structure Terminals

The qualified name of a structure terminal is generated by catenating the names of all nodes between the "root" node and the desired "leaf" node of the tree organization.

```
          1     2           n
      name .name . .....name
```

1. $name_1$ is the name of the structure as declared.

2. $name_n$ is the name of the structure terminal to be referenced.

3. $name_2,....name_{n-1}$ are the names of intervening minor structure nodes if any.

.end list
.b;Examples:
.tp10.lm6.b.lit

```
|
|   STRUCTURE Q:
|     1 QI INTEGER,
|     1 Q1,
|       2 QS BOOLEAN,
|       2 QC CHARACTER(80);
|   .
|   .
|   .
|   DECLARE ZQ Q-STRUCTURE;
|
```

To reference QI and QC in ZQ requires the following names, respectively:

```
ZQ.QI
ZQ.Q1.QC
```

19.5.3 Referencing Minor Structure Nodes

If it is required to perform an operation on a sub-tree of a structure (i.e. all parts of the tree beneath a certain "fork" node), the occasion arises to refer to a minor structure node name. The qualified name is generated by catenating the names of nodes between the "root" node and the desired "fork" node.

```
name_1 .name_2 .  ...name_n
```

1. name_1 is the name of the structure as declared.

2. name_n is the name of the minor structure node to be referenced.

3. name_2,...name_n-1 are the names of intervening minor structure nodes, if any.

Example:

```
|
|   STRUCTURE Q:
|     1 QI INTEGER,
|     1 Q1
|       2 QS BOOLEAN,
|       2 QC CHARACTER(80);
|
|   .
|   .
|   .
|   DECLARE ZQ Q-STRUCTURE;
|
```

To reference Q1 in ZQ, requires the name ZQ.Q1.

19.5.4 Naming Uniqueness

The node names used in a structure tree specification need only be unique in so far as all tree walks used to generate qualified names must be distinguishable. This means that some node names may actually duplicate others without error.

Examples:

```
|
|   STRUCTURE Q:
|     1 Q1,
|       2 QS INTEGER,\
|     1 Q2,           >-----legal duplication names
|       2 QS INTEGER;/
|   .
|   .
|   .
|   DECLARE ZQ Q-STRUCTURE;
|
```

The above duplicate names are legal because qualified references to each are distinguishable:

```
ZQ.Q1.QS
ZQ.Q2.QS
```

```
|
|   STRUCTURE R:
|     1 R1,  <--------------
|       2 RS BIT(4);        >--illegal duplicate names
|     1 R1 CHARACTER(80);<--
|   .
|   .
|   .
|   DECLARE ZR R-STRUCTURE;
|
```

The above duplicate names are illegal. ZR.R1 might be referring to a minor structure node or a structure terminal of character type.

The following situations are also permitted:

* The name of minor structure or terminal node may duplicate the name of any minor structure or terminal node in a different structure template.

* The name of a minor structure or terminal node may duplicate the name of any ordinary data item.

19.5.5 Unqualified References

Qualified referencing of parts of structures can become laborious if the node names assigned are long, or there are many levels in the structure. By accepting certain restrictions, unqualified, or direct naming of minor structure or terminal nodes is permissible.

To be able to refer to a structure in an unqualified manner the following must apply:

Unqualified reference may only be made to a structure whose name is the same as the template defining its tree organization.

It follows that only one unqualified structure may be declared for any template.

Examples:

```
|
|   STRUCTURE Q:
|     1 QI INTEGER,
|     1 Q1,
|       2 QS BOOLEAN,
|       2 QC CHARACTER(80);
|   .
|   .
|   .
|   DECLARE ZQ Q-STRUCTURE;
|   DECLARE Q Q-STRUCTURE;
```

QC in ZQ must be referred to as:

ZQ.Q1.QC

QC in Q may be referred to simply as:

QC

More restrictive rules apply to the construction of a structure template used to declare an unqualified structure.

The name of each node in the template must be unique to the block in which the template is defined.

The template must be defined in the same block as the unqualified structure is itself declared.

The template may contain no structure terminals of structure type (i.e. nested structures).

19.6 SUBSCRIPTING IN STRUCTURES

A structure terminal may possess "terminal" subscripts as a result of its type (character, bit string) or its array property. In addition, any reference to the whole or part of a structure with multiple copies can introduce a level of "structure" subscripting.

The discussion on subscripting is divided into two parts:

subscripting on references to the entire structure or to minor structure nodes;

subscripting on references to terminal data items.

19.6.1 Subscripting Of Structure Data Items

A reference to an entire structure or to one of its minor structure nodes may only possess subscripting if the structure is declared to possess multiple copies.

In the subscripting forms below, TREE represents any data item of structure type (i.e. either a "root" or "fork" node of the structure tree), the reference being unqualified or qualified. It is assumed that the entire structure is declared to possess L copies.

To select the ath copy from TREE:

```
TREE
    a;
```

1. a is an integer expression in the range 1 <= a <= L.

2. The semicolon is optional.

To select a subset of a copies starting from the Bth copy of TREE:

```
TREE
    a AT B;
```

1. a is an integer literal value in the range 1 <= a <= L.

2. B is an integer expression in the range 1 <= B <= L - a + 1.

3. The semicolon is optional.

To select a subset of copies starting from the a'th copy and ending with the Bth copy of TREE:

```
TREE
    a TO B;
```

1. a, B are integer literal values in the range 1 <= (a, B) <= L.

2. B >/ a.

3. The semicolon is optional.

Examples:

Given

```
|
|   STRUCTURE Q:
|     1 QI INTEGER,
|     1 Q1,
|       2 QS BOOLEAN,
|       2 QC CHARACTER(80);
|   .
|   .
|   .
|   DECLARE ZQ Q-STRUCTURE(3);
```

then ZQ ; selects copy 2.

ZQ.Q1 selects copies 1 and 2 of the subtree under Q1.

QZ.Q1 selects copy 1 of the subtree under Q1.

19.6.2 Subscripting Of Structure Terminals

If a structure terminal is part of a single copy structure, then it can only possess subscripting by virtue of its type or array property. Such subscripting is the same as for ordinary data items, and has been described in Sections 6, 17.3, and 18.3.

If, on the other hand, a structure terminal is part of a multiple copy structure then it may possess subscripting by virtue of its type or array property, and by virtue of the multiple copy property. Three cases of subscripting thus arise:

> STRUCTURE SUBSCRIPTING ONLY. The form of subscripting is the same as for structure data items, as described above. The only difference is that the terminating semicolon is optional only if the structure terminal is of integer type, and unarrayed.
>
> STRUCTURE AND TERMINAL SUBSCRIPTING. The structure subscripting takes the same form as before. Terminal subscripting (consequent on type or arrayness) follows the mandatory semicolon, and takes the forms described in Sections 6, 17.3 and 18.3.
>
> TERMINAL SUBSCRIPTING ONLY. The subscript forms are the same as in the previous case except that the structure subscript is replaced by an

asterisk.

Literal subscripts may alternatively be expressions computable at compile time.
See: Guide/Appendix D.

19.7 TREE EQUIVALENCE OF STRUCTURES

Most operations involving more than one operand of structure type require their operands to possess tree organizations which are in most respects identical. Two structures which are compatible in this sense are said to be "tree-equivalent". Two basic requirements have to be satisfied to establish tree-equivalence:

1. The actual shape of the trees must be equivalent;

2. The attributes of corresponding structure terminal nodes must be the same.

19.7.1 Equivalence Of Tree Shape

The equivalence of tree shape can be achieved in a number of different ways:

USE OF SAME TEMPLATE - If two structures are declared using the same template, they cannot avoid meeting both requirements for tree equivalence.

Example:

```
|   STRUCTURE Q:
|     1 QI INTEGER,
|     1 Q1,
|       2 QB BOOLEAN,
|       2 QC CHARACTER(80);
|
|   DECLARE ZQ1 Q-STRUCTURE,
|           ZQ2 Q-STRUCTURE(20);
```

ZQ1 and ZQ2 are tree-equivalent, (notwithstanding the mismatch in number of copies).

USE OF TEMPLATE OF SAME SHAPE - If two structures are declared using distinct templates which do, however, have the same shape, then the first requirement of tree-equivalence is met.

Example:

```
|
|   STRUCTURE Q:
|     1 QI INTEGER,
|     1 Q1,
|       2 QB BOOLEAN,
|       2 QC CHARACTER(80);
|
|   DECLARE ZQ Q-STRUCTURE;
|   .
|   .
|   .
|   STRUCTURE R:
|     1 RI INTEGER,
|     1 R1,
|       2 RB BOOLEAN,
|       2 RC CHARACTER(80);
|   DECLARE ZR R-STRUCTURE;
```

The tree shapes of ZR and ZQ are the same.

MATCHING OF SUB-TREES - If the tree shape of a sub-tree of one structure matches the same of another structure, or sub-tree thereof, then the first requirement of tree-equivalence is met.

Examples:

```
|
|   STRUCTURE Q:
|     1 QI INTEGER,
|     1 Q1,
|       2 QB BOOLEAN,
|       2 QC CHARACTER(80);
|   DECLARE ZQ Q-STRUCTURE;
|   .
|   .
|   STRUCTURE R:
|     1 RB BOOLEAN,
|     1 RC CHARACTER(80);
|   DECLARE ZR R-STRUCTURE;
```

The tree shapes of ZQ and ZR clearly are not the same. However, the tree shapes of ZQ.Q1 and ZR are the same.

19.7.2 Matching Of Terminal Node Attributes

Once matching of tree shape has been established, to obtain tree-equivalence, corresponding structure terminal nodes of each tree must be verified as having identical attributes. Generally, terminal nodes must match exactly in their type and array property (if any). Additionally, for each type the following matching requirements must be met:

TYPE	MATCHING REQUIREMENTS
BIT STRING	number of bits (BOOLEAN is equivalent to BIT(1))
CHARACTER	maximum declared length
INTEGER	precision
STRUCTURE	specified structure template

Examples:

```
STRUCTURE Q:
  1 QI INTEGER,
  1 Q1,
    2 QC CHARACTER(80);
DECLARE ZQ  Q-STRUCTURE;
.
.
.
STRUCTURE R:
  1 RI INTEGER DOUBLE,
  1 R1,
    2 RC CHARACTER(80);
DECLARE ZR R-STRUCTURE;
```

ZQ fails to be tree-equivalent to ZR solely due to one precision mismatch: ZQ.QI is single precision, while ZR.RI is double precision.

However, ZQ.Q1 is completely tree-equivalent to ZR.R1 since the offending terminal node is not present.

Note that the matching requirement for terminal nodes of structure type preclude tree-equivalence in cases typified by the following example:

```
STRUCTURE Q:
  1 QB BOOLEAN,
  1 QC CHARACTER(80);
STRUCTURE R:
  1 RI INTEGER,
  1 RQ Q-STRUCTURE;
DECLARE ZR R-STRUCTURE;
STRUCTURE S:
  1 SI INTEGER,
  1 S1,
    2 SB BOOLEAN,
    2 SC CHARACTER(80);
DECLARE ZS S-STRUCTURE;
```

ZS is not tree-equivalent to ZR although their tree organizations are superficially alike (see Section 19.4). ZS would be

tree-equivalent to ZR only if the template S had been specified as:

```
STRUCTURE S:
  1 SI INTEGER,
  1 SQ Q-STRUCTURE;
```

Where structure templates are declared with additional attributes such as RIGID, DENSE, LOCK, etc., matching extends to these also.
See Spec./4.3 and 4.5.

19.7.3 STRUCTURE ASSIGNMENTS

Values of one structure data item* may be transferred to another in a body using a structure assignment. Structure assignments have the same general form as other assignments: this form has been described in Section 8.1.

19.8 BASIC FORM

As applied to structures, the rules become:

Symbolic form: L := R;

1. L is the receiving structure data item. It may possess structure subscripting.

2. R is either a second structure data item, subscripted or not, or alternatively a structure function (see Section 19.11).

3. L, R must be tree-equivalent in the sense described in Section 19.7.

---

* Unless specifically stated in Sections 19.8 through 19.12, a structure data item may either be a declared structure, or a minor structure node.

19.9 MULTIPLE ASSIGNMENTS

Several structure data items may be assigned values at one assignment by the following construction first presented in Section 8.5:

Symbolic form:
L1, L2, L3, ... Ln = R;

1. L1,... Ln are receiving structure data items.
2. Any L must be tree-equivalent to the R structure operand.
3. No particular order of assignment is assumed.
4. No variable may appear on the subscript line if it also appear on the main line.

Examples:

Given:

```
|
|  STRUCTURE Q:
|    1 QI INTEGER,
|    1 QB BOOLEAN;
|  DECLARE Q-STRUCTURE, ZQ1, ZQ2, ZQ3 (20);
|  DECLARE I: INTEGER;
```

then

```
|
|  ZQ1, ZQ2 = ZQ3
|                5
```

assigns the values of the 5th copy of ZQ3 to ZQ1 and ZQ2.

But

```
|
|  I, ZQ3.QI  = 10;
|      .     5
```

is illegal.

19.9.1 Structures In Conditional Constructs

Relational expressions appear in the IF statement described in Section 9.1 and the DO WHILE statement described in Section 10.2. Such expressions may contain comparative operations with structure operands.

Using the same nomenclature as in Section 9.2, structures can be used in Class II comparative operations only:

C-3

Symbol	Purpose	Class
=	equals	II
NOT =	not equals	

The rules for structure comparisons are:

Symbolic form: L NOT =
=
= R

1. The L and R operands are either structure data items or structure functions (see Section 19.11).
2. The operands must be tree-equivalent.
3. Two structures are equal if, and only if, all corresponding terminals have equal values.

19.9.2 Structure Arguments And Parameters

HAL/S/V procedures and functions may be defined with structure parameters, and be passed structure arguments.

19.10 FORM OF STRUCTURE PARAMETERS

Any parameter of a function, or any input or assign parameter of a procedure, may be declared to be a structure using the forms of declaration described in Section 19.3.

Example:

```
ANALYZE: PROCEDURE(S1) ASSIGN(S2);
   STRUCTURE S:
     1 SI INTEGER,
     1 SN,
       2 SB BOOLEAN,
       2 SC CHARACTER(80);
   DECLARE S1 S-STRUCTURE;
           S2 S-STRUCTURE;

           .
           .          executable code
           .
```

```
|   CLOSE ANALYZE;
```

Observe the position of the structure template.

19.11 ARGUMENT PASSAGE

Any argument of a function or procedure invocation corresponding to a structure parameter must conform to the following rules:

INPUT PARAMETER. The transmission of the argument can be viewed as its assignment to the input parameter. The following rules apply:

1. The corresponding argument must be a structure data item or a structure function.
2. The argument and parameter must be tree equivalent.
3. Additionally, the input arguments must conform to those for parameters of other types as specified in Chapter 11. The input and assign lists must be strictly disjoint - it is not permissible for one part of a structure to be in the input list and another in the assign list.

ASSIGN PARAMETER. The following rules apply for matching of arguments to structure assign parameters:

1. The assign argument must be a structure data item.
2. The argument and parameter must be tree-equivalent.
3. The argument may only be subscripted if it is a declared structure as opposed to a minor structure, and only then if the subscript reduces the number of copies to one.
4. The additional requirements on assign parameters enumerated in Chapter 11 apply to structure arguments as well. For instance, no assign parameter may be any part of an input parameter of an enclosing function or procedure block.

These rules are only relevant to procedures.

Examples:

Let the following be declared:

```
|
|     STRUCTURE Q:
|        1 Q1 INTEGER,
```

```
|        1 Q1,
|          2 QB BOOLEAN,
|          2 QC CHARACTER(80) ;
|     STRUCTURE R:
|        1 RB BOOLEAN,
|        1 RC CHARACTER(80) ;
|     DECLARE ZQ Q-STRUCTURE,
|             ZR R-STRUCTURE,
|             YQ Q-STRUCTURE(10) ;
```

and let the following procedure be defined

```
|     TREE:  PROCEDURE(D1) ASSIGN(D2);
|        DECLARE D1 R-STRUCTURE,
|                D2 Q-STRUCTURE,
|
|                .
|                .          procedure body
|                .
|
|
|     CLOSE TREE;
```

Both legal and illegal invocations of this procedure are shown below:

```
|    CALL TREE(ZR) ASSIGN(ZQ);
|    CALL TREE(ZR) ASSIGN(YQ);
|S                          4
|    CALL TREE(ZQ.Q1) ASSIGN (ZQ);    illegal, input
and
                                  and assign lists must
                                  be disjoint
|    CALL TREE(ZR) ASSIGN (ZR);      illegal - no tree-
                                     equivalence, and
                                     input and assign
                                     lists not disjoint.
```

19.11.1 Structure Functions

In HAL/S/V user functions may return a structure result type. Such functions can be used instead of structure data items in many of the structure operations described above.

Structure functions follow similar patterns for their block definitions and invocations as given in Section 11 for other data types.

19.12 BLOCK DEFINITION

As usual, the block is opened with a characteristic opening statement of the form:

```
label:  FUNCTION (i1,i2, ...) a-STRUCTURE;
```

1. label is the name of the function.

2. i1,i2, ... is the list of input parameters. The entire parenthesized list may of course be omitted.

3. a is the name of the template describing tree oganization of the function. The template must be defined in a block visible (according to usual HAL/S/V scoping rules) to the opening statement. Note in particular that the template cannot be defined in a group of declaration statements inside the function.

NOTE: For external procedure and function blocks this is the only time when they may need compool templates since they may not access variables except those which are passed as parameters, as noted in Chapter 15. Thus, compools containing only structure templates may have templates in external procedure and function blocks.

Example:

```
    STRUCTURE Q:
       1 Q1 INTEGER,
       1 Q1,
         2 QB BOOLEAN,
         2 QC CHARACTER(80);


    TREE: FUNCTION(I,J) Q-STRUCTURE;

             .....

             .....
          function body
             .....

             .....

    CLOSE TREE;
```

19.13 RETURN OF STRUCTURE QUANTITIES

The RETURN statement of a structure function follows the general form described in Section 11.6. The return is similar to the transmission of structure input arguments, the function itself playing the role of parameter. The relevant rules are the same as those described for the passage of input arguments, as given in Section 19.10.

Examples:

```
|
|    STRUCTURE S:
|      1 SB BOOLEAN,
|      1 SC CHARACTER(80);
|    STRUCTURE Q:
|     1 Q1 INTEGER,
|     1 Q1 S-STRUCTURE;
|
|    .
|    .
|    .
|    TREE: FUNCTION(D1) S-STRUCTURE;
|      DECLARE D1 Q-STRUCTURE;
|      .
|      .
|      .
|      RETURN D1.Q1;
|      .
|      .
|      .
|      RETURN D1;
|      .          ---------- illegal, lack of
|      .                     tree equivalence
|      .
|    CLOSE TREE;
```

19.14 INVOCATION OF STRUCTURE FUNCTIONS

A structure function is invoked in the same way as a function of any other data type, as described in Section 11.4. It should be noted, however, that the function may only be referenced as a whole No reference, qualified or unqualified, may be made to minor structure terminal nodes of its tree.

Example:

```
STRUCTURE Q:
  1 Q1 INTEGER,
  1 Q1,
    2 QS BOOLEAN,
    2 QC CHARACTER(80);
DECLARE ZQ Q-STRUCTURE;
TREE: FUNCTION Q-STRUCTURE;
 ------------

          function body

 ------------
CLOSE TREE;
.
.
.
ZQ = TREE;          legal invocation
ZQ.Q1 = TREE.Q1;    illegal invocation
```

19.14.1 Structures In Input/output

Input/output is not being considered in this document.

CHAPTER 20

HAL/S/V ARRAY PROCESSING FEATURE

The constraints described in Chapter 20 of the HAL/S programmer's Guide are primarily of the nature of programming conveniences. That is, any allowable HAL/S operation one can perform on an array can be done without them on a component by component basis in a loop. Though somewhat less convenient, this approach has several advantages.

1. Implementation dependencies arising from the ambiguity in the order in which component operations are performed is eliminated since the serial order of the operations is made explicitly.

2. Operations such as "+" may be treated by the verification system as non-generic and subject to a set of axioms which is not dependent upon the types of the operands.

3. Verification is greatly simplified. For instance, the number of times a function is invoked is obvious from the text and not dependent upon the dimensionality of its arguments as it may be in HAL/S.

For these reasons, the constructs described in Chapter 20 have been removed from HAL/S/V even though several or all of them may be theoretically verifiable. Future work may reinstate some or all of these constructs

CHAPTER 21

EXPLICIT CONVERSIONS

Section 7.5 in Part 1 of the Guide introduced some of the more common explicit conversions of HAL/S/V and explicit precision conversion was described. were described. The language contains many more kinds of explicit conversions, however, which provide a controlled and highly visible interface between the various data types.

This section deals with conversion functions, classifying them according to the date type of their results.

21.1 INTEGER CONVERSIONS

The INTEGER conversion function converts to integer type. The behavior of this functions varies, depending on whether it possesses a single expression as argument, or a list of expressions.

SIMPLE FORM

The simple form of the INTEGER conversion function is:

```
INTEGER(exp)
```

1. exp is an expression of any of the following types:
   BIT STRING (and BOOLEAN) INTEGER
   CHARACTER

2. exp may possess arrayness, in which case the arrayness must match that of the expression of which the conversion forms a part. The result is to cause an elemental conversion for every elemental evaluation of the outer expression (See Section 20.2).

3. Conversions to integer type proceed according to the rules given in Appendix A.

LIST FORM

The list form of the integer conversion function creates an array result, in addition to type converting the list of expressions constituting its arguments. Its form is as follows:

```
INTEGER          (exp , exp ,...)
       1  2
      n ,n ...
```

1. The subscripts n_i for i = 1,2 ... are positive integers specifying the number and size of dimensions of the resulting array. The total number of values summed over all the expressions in the list must be consistent with the number of array elements implied. The upper limit on i is 3.*

2. The subscripts may be omitted entirely, in which case a linear 1-dimensional array is created, whose length is equal to that total number of values summed over all the expressions.

3. Each exp is an expression of any of the following types:

   INTEGER
   BIT STRING (and BOOLEAN)
   CHARACTER

   and may optionally possess arrayness.

4. Conversions to integer type proceeds according to the rules given in Appendix A.

Note that the list form can only have one expression in the list without reverting to the simple form if explicit subscripting of the function is present.

The ordering of values of the expression list in the resulting array is specified by the following:

1. The values of each expression in turn are converted to a linear list by applying the rules of Appendix C.

2. The lists are catenated from left to right forming a single linear list of values.

3. The linear list is regenerated to an array of the given dimensions by applying the rules of Appendix C.

Note that even though the function appears in an

---

* This number may vary between implementations. See the appropriate User's Manual.

arrayed expression, in this and all other cases involving the list form, the implementation is generally to precompute the entire array result, and then evaluate the expression containing the conversion on an element-by-element basis.

EXPRESSION REPETITION

The expressions in the list of an INTEGER conversion may be repeated using the form:

```
          i
... n# exp ,....
```

1. n is a positive non-zero integer literal specifying the number of times the value or values of the expression to repeated.

SIMULTANEOUS PRECISION SPECIFICATION

In the absence of any explicit indication, the result of an INTEGER conversion is always single precision.

If no subscripting is present, the forms are:

```
INTEGER      (....
      @SINGLE

INTEGER      (....
      @DOUBLE
```

1. The first form forces a single precision result; the second double precision.
2. Precision conversion is carried out for each expression in turn before assembly of the result.

If subscripting is present, the corresponding forms are:

```
INTEGER                (....
                  1   2
      @SINGLE,n   ,n ...

INTEGER                    (....
                  1    2
      @DOUBLE,n   , n ...
```

Examples:

```
INTEGER        (X)                          simple form
      @DOUBLE

INTEGER           ('15',BIB'1011')          list form
      @DOUBLE,2,2
```

21.2 BIT CONVERSION

Conversions to bit string type are carried out by the BIT conversion function. There are two forms: the simple form converts other data types to bit string type using the standard conversion rules; the radix form can only convert character data type to bit string type, and uses different conversion rules.

Both forms are similar to the simple form of INTEGER functions, in that they have one expression only.

SIMPLE FORM

The simple form of BIT conversion is as follows:

```
BIT          (exp)
   subscript
```

1. exp is an expression of any of the following types:

   INTEGER
   BIT STRING (and BOOLEAN)
   CHARACTER

2. exp may possess arrayness in which case the arrayness must match that of the expression of which the conversion forms a part. The result is to cause an elemental conversion for every elemental evaluation of the outer expression. (see Section 20.2)

3. Conversion to bit string type proceeds according to the rules given in Appendix A. The result is always a 32-bit string.*

4. subscript represents component subscripting on the result of the conversion. It possesses the same forms as component subscripting on bit string data items as described in Section 17.3.

5. If subscript is absent, the result of the function is the entire bit string generated by the conversion.

---

* This value may vary between implementations. See appropriate User's Manual.

Examples:

If I is a halfword integer with I= 5

then BIT (I) = 000005
16

If C is a character data item with C = '10110011101'
then BIT(C) = (00000000000000000000010110001110 1)
2

BIT (C) = (0000010110111 01)
16 to 32 2

and BIT (C) = (11101)
28 to 32 2

RADIX FORM

The radix form of BIT conversion is used when a character value is to be converted by an explicit rule to a bit string. A radix specifying the conversion rule is supplied in place of a subscript. The possible forms are as follows:

```
BIT    (exp)
   @BIN
BIT    (exp)
   @OCT

BIT    (exp)
   @DEC

BIT    (exp)
   @HEX
```

1. exp is an expression of character type whose value must consist entirely of a string of digits legal for the specified radix.

2. The radices have the following meanings:

radix	digit string
@BIN	binary
@OCT	octal
@DEC	decimal
@HEX	hexadecimal

3. exp may possess arrayness with the same implications as in the simple
form of BIT conversion.

4. The conversion generates the binary representation of the input digit string. The binary representation is truncated or padded with binary zeroes on the left to create a 32-bit string.*

Examples:

```
BIT    ('FA0') = 00000FA0
   @HEX                  16

BIT    ('1024') = 00000400
   @DEC                  16


BIT    ('177777') = 0000FFF
   @OCT                   16


BIT    ('F0F1F2F3F4') = F1F2F3F4
   @HEX                        16
```

21.3 CHARACTER CONVERSION

Conversions to character type are carried out by the CHARACTER conversion function. As with the BIT conversion, there are two forms: the simple form converts other data types to character form using the standard conversion rules; the radix form can only convert bit string data to character type, and uses different conversion rules.

SIMPLE FORM

The simple form of CHARACTER conversion is as follows:

```
CHARACTER    (exp)
           subscript
```

1. exp is an expression of any of the following types:

   INTEGER
   BIT STRING (and BOOLEAN)
   CHARACTER

2. exp may possess arrayness, with the same implications as in the BIT conversion function. (See Section 21.3).

3. Conversion to character type proceeds according to the rules given in Appendix A. The length of the result of conversion

---

* This value may vary between implementations. See appropriate User's Manual.

depends on the type of the input data.

4. subscript represents component subscripting on the result of the conversion. It possesses the same forms as component subscripting on character data items as described in Section 6.1.

5. If subscript is absent, then the result of the function is the entire string of characters generated by the conversion.

Examples:

```
If I is a halfword integer with I = 173
then CHARACTER(I) = '173'

     CHARACTER    (I) = '17'
            1 to 2
     CHARACTER    (I) = '173'
            1 TO 5

If B is a bit string of length 4 with
     B = (0101)
           2
then
     CHARACTER(B) = '101
                (note removal of leading zeroes)
```

RADIX FORM

The radix form of CHARACTER conversion is used when a bit string value is to be converted by an explicit rule to a character string. Analogous to the radix form of BIT function, a radix specifying the conversion rule is supplied in place of a subscript. The possible forms are as follows:

```
CHARACTER    (exp)
          @BIN

CHARACTER    (EXP)
          @OCT

CHARACTER    (exp)
          @DEC

CHARACTER    (exp)
          @HEX
```

1. exp is an expression of bit string type, and possibly possessing arrayness with the same implications as in the BIT conversion function.

2. The value of the bit string is coverted to a string of digits as specified by the radix, removing leading zeroes.

3. The radices have the following meanings:

radix	digit string
@BIN	binary
@OCT	octal
@DEC	decimal
@HEX	hexadecimal

4. The length of the resulting string varies depending on the value of exp.

Examples:

```
CHARACTER (BIN'001010') = '1010
          @BIN

CHARACTER     (BIN'001010') = '12'
          @OCT

CHARACTER     (BIN'001010') = '10'
          @DEC

CHARACTER     (BIN'001010') = 'A'
          @HEX
```

CHAPTER 22

INPUT AND OUTPUT

HAL/S/V incorporates entirely the input/output mechanisms of HAL/S. Therefore, Chapter 22 of this document is omitted, being identical to Chapter 22 of the HAL/S Programmer's Guide.

CHAPTER 23

REAL-TIME PROGRAMMING II

23.1 PROGRAM PROCESSES

Section 13.1 explained that at run time, the dynamic counterpart of HAL/S/V program is a real time process executing under control of a Real Time Executive (RTE). It stated that this "primal process" could create other processes whose static counterparts are task blocks embedded in the program block. However, it is also possible to create processes whose static counterparts, rather than being task blocks, are other program blocks. In order to avoid confusion, in the remainder of this Section the program block corresponding to the primal process will be called the "primal program".

The program blocks are the same in every respect as the primal program block: they are separately compiled blocks of code. The scheduling of program processes therefore, requires the bringing together of a number of compilation units at run time.*

This situation is analogous to the invocation of external procedures and functions as described in Section 15.

A program may invoike any other program in the same assemblage of compilation units, or invoke any task block within itself, in order to create a new process. The programs will probably need to share data in one or more compools, and may also share the use of comsubs.*

Any program which creates a program process, otherwise controls its execution, perforce contains references to the program block which is the process' static counterpart. The first program must, under these circumstances, be provided with a block template of the program block referenced. The program template is included in the compilation unit of the first program in the same way as if it were a compool or

---

* The object modules resulting from their compilation have to be "link-edited" to produce a single executable load module. The way in which the primal program is distinguished from the others in such a load module is extra-lingual and implementation dependent.

---

* Interfaces with compools and comsubs have been described in Section 15.

comsub template, and is discussed subsequently. Note however, that since no recursion is permitted in HAL/S/V it would be useless for two program blocks each to contain templates for the other and this may not be done.

Program template should appear just before compool templates.

External procedure and function blocks, as well as program blocks, may contain SCHEDULE statements for creating processes. However, because external procedure and function blocks may not contain task block definitions, only program processes may be created thereby.

To ensure correctness of version, program templates would be subject to the same implementation dependent software management scheme as for compool and comsub templates (see Section 15.1).

23.2 PROGRAM TEMPLATES

If a program template is included with a compilation unit, then that compilation unit may invoke the corresponding program to create a new real time process.

A program template differs in the following respects from its corresponding program:

the body of the block is empty;

the opening statement is modified as shown by the keyword EXTERNAL.

```
|
|   label:  EXTERNAL PROGRAM;
|
```

1. label is the name of the corresponding program.

Example:

program block

```
|
|  ONE:  PROGRAM;
|     DECLARE I INTEGER;
|     .
|     .
|     .
|     .
|     I = I  + 1;
|     .
|     .
|     .
|     .
```

```
|     CLOSE ONE;
```

corresponding program template:

```
ONE:  EXTERNAL PROGRAM;
   CLOSE ONE;
```

23.3 CREATING AND CONTROLLING PROGRAM PROCESSES

Processes created by invocation of a program differ very little from processes created by invocation of a task block. Only the notion of process dependency need by updated to allow for the existence of program processes.

23.3.1 Program Processes And Process Dependency

Section 13.1 introduced the concepts of the dependency of one process upon another. The basic notion of dependency still stands:

When a process A creates process B, the latter may be specified as "dependent" on the former, or "independent" of it. If B is dependent on A, then it depends for its existence on the existence of A. If B is independent of A, then A may cease to exist without affecting the existence of B.

If B is a program process, these rules are always unequivocally true. However, if B is a task process, as stated in Section 13.1, there exists an overriding rule. Reinterpreted, this rule states that a task process C, however created, is always dependent on the program process whose static counterpart contains the task block whose invocation caused C to be created.

By the use of program processes one can override somewhat the restriction that tasks may not be nested. That is, if tasks are actually written as external programs, one can produce the effect of nesting to an arbitrary degree. To do so, however, is usually indicative of poor program structure.

23.4 CYCLIC PROCESSES

Hitherto, a real time process has been characterized as being in the active state for some duration, wherein it is either ready, executing, or waiting. As described in Section 13.3, such a process finally returns to the inactive state when one of two conditions are met:

> the process is terminated by a TERMINATE statement.

execution reaches a RETURN or CLOSE of the related static program or task block.

In either circumstance, the process makes only one pass through the HAL/S/V code contained in the related program or task block. Subsequent passes through the same code would thus require the scheduling of a new process for each pass. Because of the uniqueness requirement stated in Section 13.4, each new process could only be created when the previous one returned to the inactive state.

To avoid the burden of continual intervention otherwise required to maintain cyclic execution of a program or task real time processes are created by an extension of the SCHEDULE statement described in Section 13.4. Without further intervention, the process will, during execution, make an arbitrary number of passes through the code in the related program or task block until some predetermined condition is met.

23.4.1 States Of A Cyclic Process

The possible states of a cyclic process are the same as those of a non-cyclic process, as described in Section 13.1.

When a cyclic process is created by invoking a program or task block from a SCHEDULE statement, the process makes a transition from the inactive state to the active state. It is entered on the process queue in the ready or waiting state, according to the same criterion as for a non-cyclic process.

When the cyclic process is first elevated to the executing state by the RTE, it begins the first pass through the code of the related program or task block. Unless otherwise prevented, execution will eventually reach a RETURN or CLOSE statement in the block, whereupon the process will go into a waiting state until predetermined conditions for the beginning of the next cycle are met. At the expiration of this waiting period, the process is returned to the ready state. The relative priority of the cyclic process then determines when the next cycle of execution begins.

A cycle process can return to the inactive state in one of two ways:

by being terminated through execution of a TERMINATE statement;

by being "cancelled" at the end of the current cycle of execution, either because some prespecified condition is met, or through the execution of a CANCEL statement.

The implications of "cancellation" as opposed to termination will be examined in Section 23.6.

23.5 SCHEDULE STATEMENT FOR CYCLIC PROCESSES

The form of a SCHEDULE statement for creating cyclic processes is an extension of that for creating non-cyclic processes. The cyclic SCHEDULE statement conveys two additional items of information:

a condition for starting each new cycle of execution;

a cancellation condition.

There are several versions, depending on the way in which the above conditions are specified.

23.5.1 Immediate Recycling

The simplest version of cyclic SCHEDULE statement is one in which a new cycle of execution of the process is specified to start immediately after the end of the previous cycle. This form is shown below:

```
|
|  SCHEDULE label initiation, REPEAT UNTIL time;
|
```

1. A process called label is created from the corresponding program or task block.

2. initiation specifies a priority, and optionally an initiation condition and dependency of the new process, as described in Section 13.4.

3. The keyword REPEAT signifies that the process is to be cyclic. By default one cycle is to follow another with no interval in the waiting state.

4. UNTIL time specifies a cancellation condition. time is an integer expression which when evaluated at the time of scheduling gives the time in seconds* at which the process is to be cancelled.

5. If the UNTIL phrase is absent, execution cycles indefinitely until inhibited by other means.

Cancellation actually takes place at the end of the first cycle which finishes later than the specified time.

Example:
---

* After the real time origin.

```
|
|   SCHEDULE A AT 1600 PRIORITY (40);
```

a non-cyclic schedule statement creating a process A to be initiated 1600 seconds after the real time origin.

```
|
|   SCHEDULE B at 1600 PRIORITY (40), REPEAT UNTIL 3200;
```

A cyclic schedule statement creating a cyclic process B to be initiated 1600 seconds after the real time origin, and to cease cycling at the end of the first cycle completed after 3200 seconds.

Note that the following case causes a run time error:

```
|
|  SCHEDULE C AT 1600 PRIORITY(40), REPEAT UNTIL 1000;
```

because the initiation time is later than the time at which cycling is to cease.

23.5.2 Constant Intercycle Delay

The second version of cyclic SCHEDULE statement specifies a constant delay between cycles of execution. This form is shown below:

```
|  SCHEDULE label initiation, REPEAT AFTER delay UNTIL time;
```

1. A process called label is created from the corresponding program or task block.

2. The meaning of initiation and time are the same as for the previous version of cyclic SCHEDULE statement.

3. AFTER delay specifies a constant delay between the end of one cycle of execution and the start of the next. delay is an integer expression whose value at the time of scheduling specifies the delay in seconds.

Cancellation takes place in the same way as before, with the provision that if the cancellation condition is met in the interval between cycles, cancellation takes place immediately.

Example:

```
|  SCHEDULE A AT 1600 PRIORITY(49), REPEAT AFTER 100 UNTIL 3200;
```

A cyclic process A is scheduled, specifying a delay of 100 seconds between cycles of execution.

23.5.3 Recycling At Specified Intervals

The third and last version of cyclic SCHEDULE statement specifies that each new cycle is to start a fixed interval of time after the start of the previous cycle. This form is shown below:

```
| SCHEDULE label initiation, REPEAT EVERY interval UNTIL time;
```

1. A process called label is created from the corresponding program or task block.

2. The meaning of initiation and time are the same as for the previous two versions of the cyclic SCHEDULE statement.

3. EVERY interval specifies that each cycle is to start a given interval after the start of the previous cycle. interval is an integer expression whose value at the time of scheduling specifies the interval in seconds.

Cancellation takes place in exactly the same manner as with the previous version of the SCHEDULE statement.

Example:

```
| SCHEDULE A AT 1600 PRIORITY(40), REPEAT EVERY 200 UNTIL 3200;
|
```

A cyclic process A is schedule, specifying that cycles are to succeed each other at intervals of 200 seconds. Note that if a cycle takes longer than 200 seconds to execute, the next cycle cannot start on time and a run time error occurs.

> An UNTIL phrase can also be used in a non-cyclic SCHEDULE statement. See: Spec./8.3.

23.6 TERMINATING AND CANCELLING CYCLIC PROCESSES

When a cyclic statement is terminated by execution of the TERMINATE statement described in Section 13.5, both the process and its dependents are terminated, possibly in mid-cycle, subject to the same restrictions described there. In the case of cyclic program processes, an additional restriction is described in Section 23.3.

Cancellation is a more graceful way of termination. It cannot occur when a process is in mid-cycle. Further, when a process is cancelled, its dependents are not terminated immediately: the following happens instead:

> non-cyclic dependents are allowed to execute until their normal termination;
>
> cyclic dependents are allowed to finish their own

current cycle of execution.

The process being cancelled is put in a waiting state until all its dependents have become inactive; it then becomes inactive itself. Cancellation conditions in SCHEDULE statements cannot be dynamically modified. To cancel a cyclic process arbitrarily, the CANCEL statement must therefore be used.

23.6.1 Cancel Statement

A CANCEL statement specifies the cancellation of a process. Its form is as shown below:

```
|
|  CANCEL label;
|
```

1. The appearance of label is optional. If present, the statement causes cancellation of the active process called label.

2. If label is absent, the process executing the CANCEL statement is itself cancelled.

3. A process may cancel only itself or its dependent processes.

The effect of a CANCEL statement is as follows:

If the process has not yet been initiated, it is terminated and removed from the process queue.

If the process is in a cycle of execution, it is cancelled at the end of the cycle.

If the process is waiting between cycles, it is cancelled immediately.

NOTE: The effects of cancellation are somewhat implementation dependent. Namely, depending upon how many processors are available on a particular machine and how fast processes run, dependent processes may have cycled a variable number of times. The programmer should be aware of this fact when accessing shared data after cancellation.

CANCEL statements can actually be applied to non-cyclic processes, but unless the process has not yet initiated they have no effect. If the process has not been initiated, the process is removed from the process queue, just as if it were cyclic.

Examples:

```
|   CANCEL;                          self cancellation
|   CANCEL BETA;
```

If a number of processes are to be cancelled simultaneously, the CANCEL statement can specify a list of process names:

```
|   CANCEL ALPHA, BETA, GAMMA;
|
```

CHAPTER 24

REAL TIME PROGRAMMING - III

This section concludes the description of HAL/S constructs for real time programming, which was begun in Section 13 and continued in Section 23. The remaining topic of discussion is a HAL/S/V construct called the "event", and its use in real time programming.

The original idea behind the HAL/S/V "event" was that it should serve as an interface between HAL/S/V software and hardware interrupts; that is, the medium through which the arrival of interrupts would be signalled to the HAL/S/V program. Hence, the HAL/S/V "event" was conceived as a Boolean-valued data item, normally FALSE in value, but becoming TRUE, on the arrival of the interrupt.* The assumption was that the values of "events" at any given time could control the execution of real time processes by the RTE.

An extension of this idea was the definition of the ability to simulate the arrival of interrupts by changing the values of "events" within the HAL/S software itself.

However, the underlying operating systems of most machines do not allow for interfaces with interrupts of the above nature. Hence, the simulation property of "events" has become their major role: the ability to signal a software condition in one real time process asynchronously to other processes by use of HAL/S "events" has become a real time programming tool of considerable importance.

24.0.1 Hal/s Events

A HAL/S event is a Boolean-valued data item whose value is visible at any instant to the RTE. Except for this latter qualification, whose importance will be appreciated later, an event differs little from the Boolean data item first introduced in Section 4 of Part I.

* Clearly, there would need to be some extra-lingual, implementation dependent way of relating particular "events" to particular hardware interrupts.

A HAL/S event possesses a "latching" property and may be set in value to either TRUE or FALSE. The values of events can only be changed by special HAL/S statements, not by simple assignment.

Event expressions consisting of logical operations on event data items can be synthesized: the instantaneous values of such event expressions can be used to modify the activity of the RTE in controlling real time processes. Event expressions can be used in the following circumstances:

in a SCHEDULE statement, to specify a condition for initiating a process;

in a cyclic SCHEDULE statement, to specify a cancellation condition;

in a WAIT statement, to specify a condition for ending the period a process is to remain in the waiting state.

24.0.2 Declaration Of Event Data Items

The declaration of event data items is similar to the declaration of Boolean data items as described in Section 4.2 of the Guide. The basic forms are as follows:

```
DECLARE name EVENT LATCHED;
```

name is any legal HAL/S identifier.

Note: the word LATCHED is retained to make the HAL/S/V subset compatible with HAL/S.

Examples:

```
DECLARE EV2 EVENT LATCHED;
```

COMPOUND DECLARATIONS

Declaration of events may be mixed with declarations of other data types in compound declarations:

```
DECLARE I INTEGER DOUBLE,
        E EVENT LATCHED;
```

The keyword LATCHED is an attribute which may be factored.

Example:

```
|       DECLARE E1 EVENT LATCHED,
|               E2 EVENT LATCHED,
|               E3 EVENT LATCHED,
```

may be rewritten more compactly as

```
|
|       DECLARE EVENT LATCHED, E1, E2, E3;
|
```

INITIALIZATION

All declared event data items are implicitly initialized to a FALSE value*. An event data item may possess explicit initialization. It is initialized as if it were a Boolean data item, as described in Section 4.3.

Examples:

```
|       DECLARE EV1 EVENT LATCHED INITIAL (TRUE);
|       DECLARE EV2 EVENT LATCHED CONSTANT (OFF);
|
|       (Note:  a constant event is of little use
                even though legal in HAL/S.
```

ARRAYS OF EVENTS

An event data item may be arrayed, its array property being specified in the same way as described in Sections 4.2 and 18.1. Event arrays with the latching property may be initialized as described in Sections 4.3 and 18.2.

Example:

```
|
|       DECLARE E2 ARRAY(2,2) EVENT LATCHED INITIAL (4#TRUE);
|
```

EVENTS IN STRUCTURES

A terminal node in a structure may not be an event. (See Section 19.)

* This is the only HAL/S data type which is implicitly initialized.

24.0.3 Event Expressions

An event expression is an expression composed in general of a series of logical operations upon event operands in the context of a SCHEDULE or WAIT statement. The simplest case of an event expression is a lone event operand.

An event expression has the curious property that its evaluation is under control of the RTE and may take place more than once at times other than that of execution of the SCHEDULE or WAIT statement it appears in.

OPERATIONS AND OPERANDS

The operations legal in an event expression are the Boolean operations described in Section 7.3.

Symbol	Purpose
& AND	logical intersection
| OR	logical conjunction
- NOT	logical complement

The behavior of the operations is exactly as if the operands were of Boolean data type rather than event.

The operands in an event expression are solely event data items. Operands which are event arrays must possess array subscripting which selects one, and only one, array element. Such array subscripting is the same as used for the selection of array elements from Boolean arrays, and has been described in Section 6.2 and 18.3, with the exception that the ending colon is optional rather than mandatory.

EXECUTION OF EVENT EXPRESSIONS

It was stated earlier that event expressions are evaluated under direct control of the RTE, and not necessarily only at the time of execution of the SCHEDULE or WAIT statement in which they appear. The reason for this can now be explained.

Event expressions are placed in SCHEDULE and WAIT statements to provide dynamic conditions for controlling the execution of processes. On a basic level the conditions control the transition of processes from state to state, and thus the activity of the RTE in swapping processes.

Hence, it is appropriate to evaluate an event expression, not only at the time of execution of the SCHEDULE or WAIT statement it appears in, but subsequently whenever the value of any of its event operands is modified. This is why the values of events are visible to the RTE. Not only each event operand, but the entire event expression has to be accessible to the RTE so that it can perform re-evaluations when required.

If an event expression contains subscripting which has to be evaluated at run time, then the subscript calculation takes place only once, when the event expression itself is first evaluated upon the execution of the SCHEDULE or WAIT statement it appears in.

Example:

```
|   DECLARE EV ARRAY(5) EVENT LATCHED;
|   DECLARE I INTEGER INITIAL(1);
|   .
|   .
|   .
|   .
|   .
|   .
|   WAIT FOR EV ;
|S             I
|   I=I+1;
|
```

The RTE first evaluates EV(I) when the WAIT statement is executed, and thus is interested in the value of EV(1) since I=1. Whenever the expression is re-evaluated, it is the value of EV(1) which is examined, even though the value of I may since have changed.

24.0.4 CHANGING VALUES OF EVENTS

HAL/S uses a special terminology for the operation of changing event values.

> An event with the latching property is said to be "set" when its value is forced TRUE, and "reset" when its value is forced FALSE.

These operations are carried out by the HAL/S SET and RESET statements respectively. Changes in value of an event data item as a result of one of these statements is visible to the RTE for the reason outlined in Section 24.3.

SET AND RESET

The forms of the SET and RESET statements are shown below.

	SET var; RESET var;
1.	In either form, var is a latched event data item. If it is arrayed, it must possess array subscripting causing the selection of one and only one array element. (See Sections 6.2 and 18.3).
2.	SET causes the value of var to be forced TRUE; RESET causes it to be forced FALSE.

Note that the SET statement does not cause an event which is already FALSE to change in value. Hence, the RTE does not necessarily always sense an event change when such a statement is executed.

24.0.5 EVENT EXPRESSIONS IN SCHEDULE STATEMENT

Event expressions may appear in a SCHEDULE statement for two reasons:

to specify a condition for initiating a process;

to specify a condition for ceasing to cycle a process.

INITIATION OF AN EVENT CONDITION

Section 13.3 described two time conditions under which the initialization of a process created by the SCHEDULE statement could be delayed. A third means of delaying initiation is to delay it pending the value of some event expression becoming TRUE. The basic form of SCHEDULE statement for this is shown below.

SCHEDULE label ON exp PRIORITY(a) DEPENDENT;

1. A process label is created from the corresponding program or task block and placed on the process queue.

2. PRIORITY(a) and DEPENDENT have the same meanings as described in Section 13.3 for other forms of SCHEDULE statement.

3. exp is any event expression. If its value is TRUE, when the SCHEDULE statement is executed, the process is placed in the ready state.

4. If its value is FALSE, the process is placed in a waiting state until its value becomes TRUE, whereupon it is transferred to the ready state.

CANCELLATION ON AN EVENT CONDITION

Section 23.5 described three versions of cyclic SCHEDULE statement, in each of which the cancellation could be specified at a certain time. There are two ways of causing cancellation on an event condition:

Cycling may be allowed to proceed while an event expression remains TRUE.

Cycling may be allowed to proceed until an event expression becomes TRUE.

* CYCLING while TRUE

The following form of cyclic SCHEDULE statement causes cycling of execution to proceed while an event expression remains TRUE.

!SCHEDULE label initiation, REPEAT cycle WHILE exp;

1. A process called label is created from the corresponding program or task block.

2. initiation specifies a priority, and optionally an initiation condition, and the dependency of the new process, as described in Section 13.4.

3. cycle optionally specifies a criterion for recycling execution as described in Section 23.5.

4. WHILE exp specifies that cycling is to continue while the value of exp remains TRUE. exp is any event expression.

5. If the value of exp becomes FALSE before the process is initiated, it is merely removed again from the process queue, and becomes inactive.

Cancellation of the process actually occurs at the end of the first cycle in which the event expression becomes FALSE*. If the event expression becomes FALSE in the interval between cycles, cancellation takes place immediately.

* CYCLING until TRUE

A modification of the above form allows cycling of execution to proceed until an event expression becomes TRUE. This is not merely a simple inversion of logic since the value of the event expression is not allowed to take effect until after the first cycle of execution of the process has started. In contrast to the above form, the following modification always allows at least one cycle of execution to be completed.

* Even if it subsequently becomes TRUE again during the same cycle.

| |SCHEDULE label initiation, REPEAT cycle UNTIL exp;

1. A process called label initiation is created from the corresponding program or task block.

2. The meanings of initiation and cycle are as for the previous form of SCHEDULE statement.

3. UNTIL exp specifies that cycling is to continue until the value of exp becomes TRUE, with the provision that at least one cycle shall be executed. exp is any event expression.

Cancellation of the process occurs at the end of the first cycle in which the event expression becomes TRUE*. If it becomes TRUE in the interval between cycles, cancellation takes place immediately.

24.0.6 EVENT EXPRESSIONS IN WAIT STATEMENT

Section 13.5 explained how the WAIT statement could be used to force a process into a waiting state until some timing condition is satisfied. The WAIT statement can alternatively specify an event condition. This causes a process to remain in a waiting state until some event expression becomes TRUE, whereupon the process returns to the ready state.

The form of this version of the WAIT statement is as follows.

| WAIT FOR exp;

1. exp is any event expression.

2. The process executing the WAIT statement is placed in the waiting state until the value of exp becomes TRUE.

3. If exp is already TRUE when the WAIT statement is executed, the statement has no effect.

* Even if it subsequently becomes FALSE again during the same cycle.

24.0.7 PROCESS EVENTS

Section 13.5 stated that the name of a process could be used as if it were a Boolean data item in order to determine the major state of the process. The names of processes can also be used in event expressions as if they were event data items. In this context they are called "process events."

The truth table shows again the correspondence between logical value and major state.

State	Value
ACTIVE	TRUE
INACTIVE	FALSE

CHAPTER 25

ERROR RECOVERY AND SIMULATION

HAL/S compilations can be created which, although seen as legal at compile time, violate the rules of the language during execution. Such violations give rise to "run time errors". Run time errors are also produced when abnormal hardware conditions are encountered during execution.

HAL/S has a comprehensive and flexible mechanism for detecting and recovering from run time errors. It also has the capability of simulating run time errors, which can be extremely useful for checkout purposes. Another feature of the language is the ability to specify and signal user-defined run time errors.

This section explains how run time errors are handled as part of the activity of the Real Time Executive (RTE) and describes statements by which HAL/S programmers can extend or modify this activity.

25.1 HAL/S RUN-TIME ERROR CONCEPTS

Each HAL/S implementation possesses a defined set of run time errors which are detectable during execution. These errors are called "system-defined" errors. The HAL/S user may, at will, create a certain limited number of supplementary "user-defined" errors for his own purposes. Each run time error, whether system-defined or user-defined, possesses a unique numerical "error code" by which it may be referenced in a HAL/S compilation. This error code consists of two parts:

* an error number;
* an error member number *

---

* The classification into groups and the assignment of error codes is implementation dependent. See appropriate User's Manual.

25.2 ERROR DETECTION AND RECOVERY

The activity of detecting and recovering from run time errors is handled by an Error Recovery Executive (ERE) which in practice is part of the Real Time Executive (RTE). For every error group, and

implementation-dependent, standard, system recover action is defined*. On detecting an error belonging to a certain group, the ERE takes the appropriate system recover action for the group, unless otherwise directed by the user.

Depending upon the kind of error, the system recovery action may be any one of the following:

- to execute a fix-up routine and continue;
- to terminate execution abnormally;
- to ignore the error.

25.3 ERROR ENVIRONMENT OF A PROCESS

The behavior of the ERE in detecting and recovering from run time errors must be viewed from the standpoint of HAL/S as a real time programming language.

Every active real time process possesses its own so-called "error environment", which is essentially a description of the recovery actions in force for all possible run time errors the process could be subject to. On initiation of the process, the system recovery action is in force for all run time errors. During the life of a process, its error environment may be modified by the specification of a "user recovery action" for some error or error group. The user recovery action is enforced by the execution of specific HAL/S error control statements which will be described later.

A process may only modify its own error environment, never that of another process.

25.4 DYNAMIC SCOPING OF ERROR ENVIRONMENTS

During its execution, a process may invoke procedures and functions,which may in turn invoke further procedures and functions, and so on to an arbitrary depth of nesting. made to the error environment during execution of a procedure or function remain in force only until return from it. Thus, execution of HAL/S error control statements has an inherent dynamic scoping property.

Consider an example where process A is invoking procedure B during execution, which in turn invokes procedure C.

- Modifications to the error environment made in A remain in force for the remainder of A's execution unless countermanded by removal or further modification.
- Modifications made in B remain in force until

* See appropriate User's Manual.

return from B unless countermanded by removal or further modification B.

- Modifications made in C remain in force until return from C unless countermanded by removal or further modification in C.

It is stressed that this is a dynamic scoping property, that is not related to whether or not, for example, procedure block C is physically nested inside procedure block B.

Further clarification is required in cases where more than one process can invoke the same procedure or function. If two processes A1 and A2 both execute the same procedure B, then error control statements executed in B affect the error environment of whichever process is executing B.

The error environment in force for each process on invocation of B is reinstated on return from B. There is no cross-coupling effect between the two error environments.

25.4.1 Error Environment Modification

HAL/S possesses two statements which can alter the error environment of the process which executes them.

- The ON ERROR statement modifies the error recovery action for a particular error or error group.
- The OFF ERROR statement causes the removal of a previously-applied modification for a particular error or error group.

Both statements have an identical construct for representing the error group and member numbers involved.

25.5 ERROR GROUP AND MEMBER NUMBER SPECIFICATION

Error group and member numbers appearing in the HAL/S ON ERROR and OFF ERROR statements are specified by appropriately subscripting the keyword "ERROR". Three basic froms exist. Each form is dealt with in order of decreasing generality.

- SPECIFICATION OF ALL ERRORS

To specify all errors, the keyword ERROR, without subscript, is used:

```
ERROR
```

1. Lack of subscript implies all members of all error groups.

SPECIFICATION OF ALL ERRORS IN A GIVEN GROUP

To specify all members in a given error group the following form is used:

```
ERROR
    m:
```

1. m is an unsigned integer literal.
2. All members in group m are specified.
3. The colon is optional.

SPECIFICATION OF A GIVEN ERROR

To specify a given error member of an error group, the following form is used:

```
ERROR
   m:n
```

1. m, n are unsigned integer literals.
2. Error member n in group m is specified.

25.6 ON ERROR STATEMENT

The ON ERROR statement is used to modify the error environment with respect to the error or errors specified. The statement can modify the error environment in the following ways:

- by causing the error or errors to be ignored; ... CASE 1
- by causing the standard system recovery action to be taken; ... CASE 2
- by causing execution to branch to specified HAL/S code on occurrence of the error. ... CASE 3

In addition, in the first two forms, the value of an event data item can be changed on occurrence of the error or errors.

An ON ERROR statement may specify system-defined or user-defined errors*.

CASES 1 2 : SYSTEM AND IGNORE ACTIONS

The basic form of the ON ERROR statement is as shown below:

```
| ON specification SYSTEM;
| ON specification IGNORE;
```

1. specification is an error specification of the form previously described.

2. The keyword SYSTEM states that stan-standard system recover action is to take place.

3. The keyword IGNORE implies that errors specified in the specification are to be ignored.

Examples:

```
|   ON ERROR SYSTEM;          revert to standard system
|                             recovery action for all errors.
|
|   ON ERROR    IGNORE;       ignore error member 4 in
|S          1:4               group 1.
|
|   ON ERROR  SYSTEM;         revert to standard system
|S          3                 recovery action for all
|                             errors in group 3.
```

If the value of an event is to be changed in addition to the actions specified above, one of the following clauses is added after the keyword SYSTEM or IGNORE.

```
... AND SET var ...
... AND RESET var ...
... AND SIGNAL var ...
```

1. SET, RESET, and SIGNAL have the same actions as described in Section 24.4 of the Guide.

---

* For reasons of software security, some implementations may prohibit the modification of the error environment with respect to certain errors. See appropriate User's Manual.

2. If var contains run time subscript evaluations, they are carried out at the time of execution of the ON ERROR statement rather than on the occurrence of the specified error or errors.

On the occurrence of an error covered by the error specification, the value of the specified event date item is modified before the SYSTEM or IGNORE is taken by the ERE.

Examples:

```
|   ON ERROR IGNORE AND SET EV1;
|   ON ERROR    SYSTEM AND SIGNAL EV2 ;
|S          1:1                        5
|   ON ERROR  SYSTEM AND SIGNAL EV3 ;
|S           5                        I
|
```

I is evaluated on execution of the ON ERROR statement, not on the occurrence of an error in group 5.

* CASE 3 : USER-SUPPLIED ACTION

The user can supply the action to be performed on an error occurrence by means of the following form of ON ERROR statement.

```
|  ON specification statement;
|
```

1. specification is an error specification in the form previously described.

2. statement is an executable HAL/S statement with which execution is resumed after occurrence of the specified error condition.

3. statement may possess a statement label but cannot be branched to from outside the ON ERROR statement

4. This kind of ON ERROR statement may not form by itself the "true part" of an IF statement. (See Section 9.1).

It is important to understand the flow of execution implied by the above form, both when the ON ERROR is executed, and on the occurrence of an indicated error. The following example shows this in detail.

Example:

```
|   On ERROR    DO;
|S          5:1                       user-supplied error
|                                       recovery action is
|                                       this entire DO...END
|                                       group.
|
|               END;
|   I = I+1;
|   .
|   .
|   .
|   .
|   .
|   .
|   *          error 5:1 occurs.
|
```

The ON ERROR 5:1 DO...END; statement modified the ERE's action for ERROR 5:1. During the program execution, wen ERROR 5:1 is encountered, ERE directs the flow of execution to the statement ON ERROR5:1 DO...END; in the program block.

In HAL/S/V, we make it mandatory that the first executable statement in the block be:

```
ON ERROR RETURN; (for procedure)
ON ERROR RETURN expression;  (for functions)
```

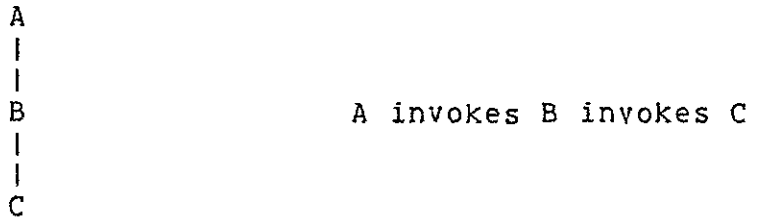
This will prevent the execution flow from branching to any error handling block outside the current procedure or function block. In HAL/S, depending on the dynamic structure of the error-specification environment, a process can jump outside the block in which the error is encountered.

The exit assertions for a block can be written by taking into account all the possible points, inside the block, where the errors can occur. In a similar fashion the entry assertions, for the error-handling blocks in ON ERROR statements, can be specified.

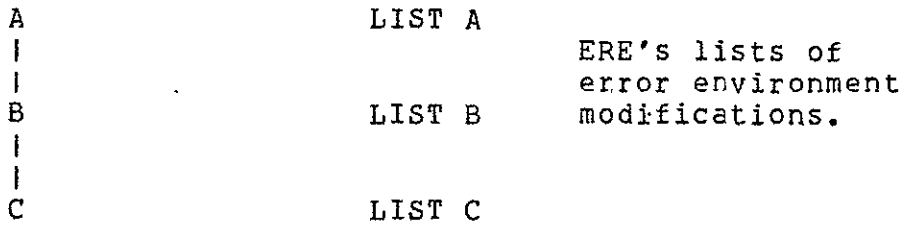
25.7 PRECEDENCE OF ON AND OFF ERROR STATEMENTS

Some additional information needs to be supplied in order to understand in detail how successive execution of several ON and OFF ERROR statements modifies the error environment of a process.

In general, an executing process A is executing code in some block several nesting levels of invocation depth, as illustrated below:

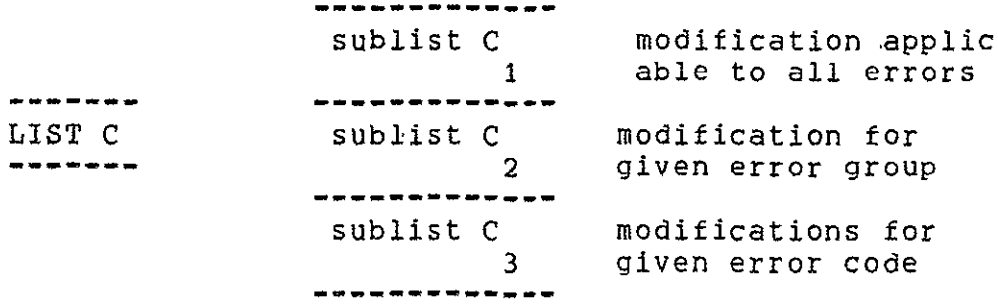


The ERE keeps continuously updated lists of all error environments modifications in force at any instant of time.* When execution of the process A described above is in the body of block C, the ERE possesses three linked lists of ON ERROR modifications, each corresponding to a block not yet returned from:



When block C is returned from, LIST C is deleted, leaving LIST A and LIST B in force. When block B is returned from, LIST B is deleted leaving only LIST A in force.

Each lit is diveded into three sublists as illustrated below for LIST C:



- Sublist C3 contains modifications generated by an ON ERROR statements of the form:

```
ON ERROR .....
         m:n
```

- Sublist C2 contains modifications generated by ON ERROR statements of the form:

---

* This description of the ERE's behavior is representational only: an actual implementation of the ERE may employ different algorithms producing the same result.

ON ERROR .....
m

- Sublist C1 can contain at most one entry, the modification generated by an ON ERROR statement of the form:

ON ERROR.....

If a new ON ERROr statement in block C is executed, then one of the following happens:

- if an entry in the appropriate sublist exists for the given error specification, the entry is replaced with the new information gained, thus erasing memory of the previous recover action specified;
- otherwise a new entry is added at the end of the sublist.

With this background, the behavior of the ERE in recovering from a run time error can now be des ribed in more detail. Suppose that a run time error occurs while execution is in block C. On detecting the error, the ERE gains control and scans backwards through the lists until it finds an entry applicable to the error which occurred. The ERE may find such an entry in any of the lists A, B, or C, in which case it takes the indicated recovery action; or it may find no such entry, in which case it takes the standard system recovery action.

Bearing in mind how entries are made into the sublists of error environment modifications, up to three entries may be applicable to a given run time error:

- an entry applicable only to the given error;
- an entry applicable to the whole group of which the given error is a member;
- an entry applicable to all errors.

Given the sublist scanning order described, it is clear that there is an inherent precedence order of ON ERROR statements.

Error Specification		Precedence
ERROR m:n	error code specification	FIRST 1

ERROR m	error group specification	2
ERROR	specification of all errors	3

Example:

If the following statements have been executed in a block:

```
| ON ERROR    GO TO ALPHA;
|S          5:1
| ON ERROR   GO TO BETA;
|S          5:
| ON ERROR IGNORE;
|
```

Then if error 5:1 occurs, execution branches to ALPHA. If error 5:3 occurs, execution branches to BETA. If error 6:1 occurs, the error is ignored.

The above are true no matter in what order the ON ERROR statements have been executed.

The behavior of an OFF ERROR statement now also becomes clearer. On execution of an OFF ERROR statement in, say, block C, the ERE looks through the whole of LIST C and on finding an entry with the same error specification, removes it from its sublist. This may expose to the scanning process another modification in another sublist of LIST C or a modification in LIST A or LIST B.

Example:

If the following statements have been executed in a block:

```
| ON ERROR    GO TO ALPHA;
|S          5:1
| ON ERROR   GO TO BETA;
|S          5:
|
```

then if error 5:1 occurs, execution will branch to ALPHA. If now the following statement is executed:

```
|   OFF ERROR    ;
|S           5:1
|
```

and afterwards error 5:1 occurs, execution branches to BETA.

25.7.1

ERROR SIMULATION

At the beginning of Section 25 it was stated that run time errors could be simulated. In fact, the same HAL/S construct is used both to simulate "system-defined" errors and to signal "user-defined" errors. This construct is the SEND ERROR statement, whose form is shown below:

```
|     SEND ERROR   ;
|                m:n
|
```

1. m and no are unsigned integers representing an error group number, and an error member number respectively.

2. If the error code m:n represents a system-defined error, that error is being simulated*.

3. If the error code m:n represets a user-defined error, that error is being signalled.

The recover action taking place on execution of a SEND ERROR statement is as if the corresponding run time error had really occurred.

Example:

```
| ON ERROR   GO TO ALPHA;
|S          5:
|   .
|   .
|   .
|   .
|   SEND ERROR   ;
|S            5:2
|
```

Error 5:2 is simulated or signalled: a previous ON ERROR statement has modified the recovery action for error group 5, so that the result is a branch to ALPHA.

In this example, it is immaterial whether error

---

* For reasons of software security, some implementations may prohibit certain system-defined errors from being simulated. See appropriate User's Manual.

5:2 is system-defined or user-defined.

CHAPTER 30

MANAGERIAL CONTROL OF ACCESS TO DATA AND CODE

The constructs for access control described in Chapter 30 of the HAL/S Programmer's Guide are not relevant to verifiability. Therefore, HAL/S/V makes no changes to this material.

However, there is another variety of access control for which no constructs are provided by HAL/S. This involves the protection of data types, procedures, program and task blocks, etc., on a more selective basis than that provided by the HAL/S ACCESS construct. For instance, one may wish to permit the use of a module without disclosing its internal structure. Use of the HAL/S keyword ACCESS is suited to "all or nothing" protection of sensitive data or code, but not to selective control.

There exist well-understood techniques for implementing such access control. These include capabilties, access lists, and data abstractions. Future enhancements to HAL/S/V will undoubtedly contain some such facility.

The benefit for verifiability derives from the fact that verified modules can be treated as "black boxes" whose behavior is completely specified as a function of the input parameters. The internal workings of such modules are of no concern to the user and may be made completely inaccessible to him. Moreover, the ways in which modules can be utilized can be carefully controlled.

CHAPTER 31

INTERFACES WITH NON-HAL/S/V CODE

HAL/S/V makes no changes to the material in Chapter 34 of the HAL/S Programmer's Guide. The ability to call subroutines written in non-HAL/S/V code is a valuable feature of HAL/S/V.

The programmer is cautioned, however, that the verifiability of a program is highly dependent upon the programming language in which it is written. Assembly language, for instance, makes no restrictions on the accessing of storage locations, aliasing, etc. Hence, in assembly language programming, the structured approach so critical to writing verifiable programs is left entirely to the discipline of the individual programmer rather than enforced by the scope rules, block structuring, etc., present in HAL/S/V.

The HAL/S/V programmer concerned with the verifiability of his program must consider the possibilty that non-HAL/S/V code segments may be extremely difficult to verify. If such code blocks could be regarded as "black boxes" computing a known function of the input parameters, they would not adversely affect the verifiability of the calling routine. However, the difficulty involved in verifying these code segments is precisely the inability to confidently assert that such a function is computed. Modular programming becomes crucial. The programmer must take care that the interfaces with non-HAL/S/V code segments be well-defined and as narrow as is feasible.

The Gypsy programming language enhances the ability to write verifiable programs since it includes specification capabilities and restricts troublesome constructs. The ability to define interfaces with Gypsy code would permit formal verification of crucial code segments written in Gypsy and called by a HAL/S/V main routine. Alternatively, designing a facility for automatic translation from HAL/S/V to Gypsy and vice-versa, would permit the verificaion facilities designed for Gypsy to serve for HAL/S/V as well. Future enhancements to HAL/S/V might consider such features.

CHAPTER 26

DATA STORAGE AND ACCESS

Given the purposes for which HAL/S/V is intended, the way in which declared data is physically located in the core of the object machine will often be an important concern. In particular, in the design of HAL/S software, the following questions must often be addressed:

1. Does the declared data occupy as small an area of core as is practical?

2. Is the data physically ordered as it was declared?

3. Can some non-critical data be relegated to segments of core addressable by slower methods to make more room for critical data?

4. Can use be made of registers or temporary storage areas for some data?

HAL/S/V contains constructs by whose means some degree of control over each one of these factors can be achieved. Necessarily, the degree of control is implementation dependent.

26.1 PACKING DENSITY OF STORED DATA

The efforts that any HAL/S/V compiler makes to optimize the density of storage of data items are implementation dependent. Generally speaking, however, the default assumption is that optimization is relatively unimportant compared with speed of access.

The attribute DENSE, when applied in the declaration of data items, causes more emphasis to be placed on storage density optimization at the expense of rapid access. Potentially, the attribute DENSE may be applied to data of any type, although it is a matter of implementation as to when it causes packing density to increase.

26.2 DENSE STRUCTURES

Packing density optimization is most commonly applied to HAL/S/V structures. If the packing density of a structure data item is to be optimized, the keyword DENSE must appear in the specification of the structure template defining its tree organization. The form of such a template is as follows:

```
| STRUCTURE name DENSE:
|
|          1        2           n
|     node   , node   ,...,node
```

1. name is the structure template name.

2. The nodes are a list of nodes forming the tree organization, as described in Section 19.2.

3. The keyword DENSE indicates that the storage packing density of all the structure terminals is to be optimized.*

Note that such optimization may cause the physical ordering of structure terminals to differ from that given in the template specification.

Example:

```
| STRUCTURE A DENSE:
|
|     1 A1,
|       2 A11 BIT(16),
|       2 A12 INTEGER,
|       2 A13 ARRAY(10)  BOOLEAN,
|     1 A2 CHARACTER(80);
| DECLARE ZA A-STRUCTURE;
```

All the structure terminals in ZA have their storage packing density optimized.

When the keyword DENSE is used as described above, storage packing density is optimized for the whole structure. If the DENSE keyword is used of a fork or leaf node of a structure template, such optimization can be restricted to a part of the structure.

Nodes connected below a "fork" node on which the keyword DENSE appears inherit the property from it.

The keyword ALIGNED can be used to prevent inheritance of the property. The following example shows how the keywords are actually specified in a structure template.

---

* See appropriate User's Manual for packing algorithms.

Example:

```
STRUCTURE A :
  1 A1 DENSE,
    2 A11 BIT (16),
    2 A12 INTEGER,
    2 A13 ARRAY (10) BOOLEAN ALIGNED,
  1 A2 CHARACTER (80) ;
DECLARE ZA A-STRUCTURE;
```

The ALIGNED keyword on A13 prevents the inheritance of the DENSE property from A1.

Detailed rules for the appearance of DENSE and ALIGNED on fork and leaf nodes of structure templates, and on data items of other types are given in Spec./4.5.

26.3 ORDERING OF STORED DATA

The HAL/S/V language does not guarantee that the physical order in which data is stored is the same as the order of appearance of data items in a compilation, either globally or locally. Nor does HAL/S/V guarantee that the physical order of structure terminals in a structure data item is the same as the order of their definition in its structure template. Indeed, some implementations will deliberately reorder data so that access to it can be optimized.

In most cases such reordering is not of importance to the HAL/S/V programmer. However since there are exceptions, HAL/S/V has a capability for specifying the non-reordering of data in storage.

Reordering may be inhibited in the following constructs:

- an entire compool;
- a structure template.

26.4 NON-REORDERING OF COMPOOLS

To prevent the reordering of data items in a compool, the keyword RIGID is placed in the opening statement of the compool block as shown below.

```
label: COMPOOL RIGID;
```

1. label is the name of the compool.
2. The keyword RIGID denotes that the physical order of storage of data items is the same as the order of their appearance in the compool.

The corresponding compool template must possess the keyword RIGID also.

26.5 NON-REORDERING OF STRUCTURE TERMINALS

The potential reordering of structure terminals may be inhibited by use of the keyword RIGID on the structure template, as shown below:

```
|   STRUCTURE name RIGID:
|     node 1 , node 2, ......
|     .........node n;
```

1. name is the structure template name.

2. node 1, node 2, ... node n is a list of nodes forming the tree organization, as described in Section 19.2.

3. The keyword RIGID denotes that the physical order of structure terminals is guaranteed to be the same as the order of appearance of the terminals in the template.

Both the keywords RIGID and DENSE may appear on a structure template (in any order). The effect of RIGID takes precedence over storage packing density optimization.

```
-----------------------------------
The keyword RIGID may appear on
form and leaf nodes of a template.
See: Spec./4.5
-----------------------------------
```

26.6 TEMPORARY AND REMOTE STORAGE

The data accessing characteristics of some object machines are such that most efficient use of core is made by dividing data into two categories:

1. Data which needs to be accessed quickly and often;

2. Data which needs to be accessed seldom, and where speed is not critical.

Normally all declared data in a HAL/S/V compilation is treated alike, as falling into the first of these categories. However, by appropriate specification, a HAL/S/V data item can be relegated to the second category; such data items are termed "remote".

Sometimes in HAL/S/V code, data items are used only as temporary storage in an extremely localized sequence of statements, and have no significance as far as the algorithm implemented is concerned. If

such data items were declared normally, then the core area they occupy would remain unused for a substantial part of the duration of execution of the code. This waste can be avoided by declaring them as "temporary" data items, whereupon the HAL/S/V compiler can be allowed to locate them in some reusable "scratch pad" area*.

Control variables in repetitive DO groups are a particular instance of data items used for temporary storage purposes. However, in this instance a consideration is the speed with which the value of the control variable can be accessed, since it may be required for many subscript evaluations within the DO group. Here it is more appropriate to set aside a register than to locate the data item in a scratch pad area. Declaration of such variables as "temporary" can allow a HAL/S/V compiler to perform this kind of allocation also.

26.7 SPECIFICATION OF REMOTE DATA

A data item is declared to be remote by use of the keyword REMOTE in its declaration. Data items of any type except event may be designated REMOTE. The position of the keyword in a declaration is illustrated by the following examples.

Example:

```
| DECLARE I INTEGER REMOTE;
| DECLARE B BOOLEAN INITIAL (TRUE) REMOTE AUTOMATIC;
| DECLARE ARRAY (4) INTEGER REMOTE, I, K, L;
| STRUCTURE Q:
|    1 Q1 INTEGER H
|    1 QB BIT (16);
| DECLARE ZQ Q-STRUCTURE REMOTE;
|
```

If remote data items appear in a RIGID compool, then the remote data items appear in the remote storage area in the same order as they were declared; the other data items appear in the regular storage area in the same order as they were declared.

For more precise rules on positioning the keyword REMOTE, see Spec./4.5.

---

* The nature and usage of such areas is implementation specific.

26.8 DECLARING AND USING TEMPORARY DATA

The HAL/S/V language enforces localized use of temporary data items by requiring them to be declared and used within D...END statement groups (See Section 10.). The END statement of a group signals to the HAL/S/V compiler that "scratch pad" storage allocated to temporary data defined in the group is available for other use.

Temporary data items are declared by TEMPORARY statements which are declaration statements in which the keyword DECLARE has been replaced by the keyword TEMPORARY. The basic form is thus:

```
| TEMPORARY name attributes;
```

1. name is a legal HAL/S/V identifier name.

2. attributes describe the type, array property, precision and other properties of the data item as in a declaration statement.

All TEMPORARY statements must appear immediately after the DO statement and before the first statement inside the group.

Examples:

```
| DO;
| TEMPORARY I INTEGER DOUBLE;
| TEMPORARY B BIT
|            ZQ Q-STRUCTURE
|
|
| END;
```

The structure template Q cannot be defined in the DO...END group. Its definition must appear at the beginning of the code block in which the DO...END group is embedded.

The control variable in a DO FOR statement can also be designated a temporary data item by preceding its appearance in the DO FOR statement by the keyword TEMPORARY. In this context, the control variable is taken implicitly to be a single precision (halfword) integer.

Example:

```
| DO FOR TEMPORARY I = 1 TO 18 BY 2 ;
| .
| .
| END;
|
```

The declaration of temporary data items is subject to the following restrictions:

* they may not be initialized;

* they may not be declared remote;

* they may not be of event type;

* the name of a temporary data item may not duplicate the name of another temporary data item in the same DO...END group;

* the name of a temporary data item may not duplicate the name of an ordinary data item known by the scoping rules of Section 1.2 to the body of the DO...END group.

26.9 ACCESS TO SHARED DATA

Generally at run time, an arbitrary number of real time processes are able to share global data and data defined in compools. Thus, it is entirely possible that one process may be in the act of modifying such data while another process is referencing it. It may be crucial to the integrity of the algorithm implemented in the second process that this be guaranteed not to take place.

To handle this situation, it is mandatory to designate such data items as protected, or "locked". Such data items can only be accessed from within areas of code called "update blocks". The boundaries of update blocks are visible to the Real Time Executive (RTE) which can therefore control entry into them and exit from them on a process-by-process basis.

26.10 LOCK GROUPS

The protection of data could be carried out on an individual basis data item by data item. Consider two processes A and B, each requiring to use protected data item Z. Process A accesses Z in an update block UA, and B accesses it in update block UB.

If process A began executing update block UA first, and thus began using Z, then process B would be prevented from beginning execution of update block UB until A had finished executing UA.

Protection of data on an individual basis would impose an arbitrarily large burden on the RTE depending on the number of data items to be protected, and the number of processes required to share them.

In order to limit this overhead of effort, HAL/S/V applies protection on a group basis rather than an individual one. Every compool or global data item is designated as belonging to one of a limited number of "lock groups". The above illustration can be restated for HAL/S/V as follows.

Consider two processes A and B, each requiring to use protected

all protected data in lock group N become unusable by process B which therefore cannot begin executing UB until A finishes executing UA.

For more global protection, some protected data items can be designated as belonging to all lock groups simultaneously.

If in the above illustration, for example, process A required to use a protected data item belonging to all groups, and execution reached UA first, then process B could not enter UB to use protected data from any lock group until A had finished executing UA.

26.11 LOCK GROUP SPECIFICATION

A data item in a compool is designated as protected at the time of its declaration. The following construct is inserted in its declaration:

```
....LOCK(n)....
....LOCK(*)....
```

1. In either form, the keyword LOCK indicates that the data item is to be protected.

2. n is a positive integer denoting that the data item is to belong to lock group n, where 1< n< 15*.

3. * denotes that the data item is to be considered as belonging to all lock groups simultaneously.

The following examples illustrate the positioning of the construct within declarations:

Examples:

```
| DECLARE I INTEGER DOUBLE LOCK (3);
| DECLARE B ARRAY (1000) BOOLEAN LOCK (*) ;
| STRUCTURE Q DENSE:
|     1 QI INTEGER,
|     1 QB BIT (16) ;
| DECLARE ZQ Q-STRUCTURE (20) LOCK (3);
```

For more precise rules concerning the location of the locking attribute see Spec./4.5.

26.12 UPDATE BLOCK DEFINITIONS

An update block is an explicitly delimited body of code wherein locked data may be referenced or modified. Superficially, an update block looks similar to any other kind of code block in the HAL/S/V language. Its delimiting statements are of the form shown below:

```
label : UPDATE;
        .
        .
        .
CLOSE label;
```

1. On the opening statement label is any HAL/S/V identifier, and represents the name of the update block.

2. The update block may be unlabelled, in which case label: is omitted.

3. If the update block is labelled, the closing statement may optionally possess a matching label.

An update block is unique in that it is never invoked as are other kinds of code blocks: rather it is executed when it is encountered in the path of execution. Consistent with this, the label on the opening statement of the block may be treated as a statement label.

The following rules govern the contents of any update block:

1. The opening statement may be immediately followed by the declaration of local data, as if it were a program block (see Section 3.2).

2. Input/output statements of any kind are illegal.

3. SCHEDULE, WAIT, CANCEL, TERMINATE and UPDATE PRIORITY statements are illegal. This rule ensures that a process does not remain in an update block indefinitely, thereby holding certain resources.

4. Procedure and function blocks, but neither task nor other update blocks may be nested within it. An update block is not allowed to have any other update block in its body to eliminate possibility of deadlock. It may call a function or procedure inside the block only if it is declared within the block.

5. The only procedure or function invocations which are legal are those referencing procedure or function blocks defined within it.

26.13 EXECUTION OF UPDATE BLOCKS

The behavior of processes on encountering update blocks has already been described in this section, but only superficially by example. This behavior is now re-examined in more detail.

The simplest case is that of two processes wishing to use data items from the same lock group. Each process has to execute an update block to use the protected data items. The following activity takes place:

> If both of the processes require data items from the same lock group to be modified then the first process to enter its update block must complete execution of it before the second process can enter its own update block. The RTE places the second process in a waiting state for this period of time.
>
> If one or both of the processes only require to reference the data then in some implementations of HAL/S/V, the behavior of the RTE will be the same as before. Alternatively, in other implementations, to reduce the second process' waiting time, the RTE may allow partial overlap in execution of the update blocks, consistent with exclusive use of data by the process modifying it*.

If the two processes wish to use data from more than one lock group, the RTE tracks the use of each lock group in the above way. If one or both processes use data protected by LOCK (*), then the situation is equivalent to one in which the process or processes wish to use date in every lock group.

If data is shared by more than two processes, then all processes except one are put in a waiting state by the RTE. The eventual order in which the processes complete execution of their update blocks will depend on the contents of the process queue and the relative priority of the processes.

26.14 LOCKED ASSIGN ARGUMENTS

The rule that locked data items can only appear in update blocks has one sole exception: it is possible for locked data items to appear as assign arguments in procedure invocations. This provides the ability to "parameterize" update blocks, as will be shown in an ensuing example.

---

* This alternative entails more work by the RTE, thus "stealing" time from the processes' productive work. The behavior of any implementation is therefore the result of a trade-off to achieve an acceptable RTE performance.

The following rules govern the passage of locked assign arguments:

1. If the argument is a data item belonging to lock group n, then the corresponding parameter must be declared LOCK (n) or LOCK (*).

2. If the argument is a data item belonging to all lock groups, the corresponding parameter must be declared LOCK (*).

3. Argument and parameter must also match in the senses described in Sections 11.5, 17.7, or 19.10 as applicable.

4. If any assign argument is locked, then the entire procedure body should be treated as an update block.

CHAPTER 27

HAL/S/V AND REENTRANCY

This section deals with another indirect implication of multi-processing in real time: reentrancy.* In HAL/S/V, reentrancy arises because more than one real time process at a time can use a procedure or function. The HAL/S/V language possesses constructs by which reentrancy can be allowed or inhibited in procedures and functions.

27.1 DETERMINING REENTRANCY REQUIREMENTS

A HAL/S/V user intending to code a procedure or function (either internal or external) to be invoked in a real time context, should first determine which of the following two categories it falls into:

1. The places where it is invoked are such that it can never be in use by more than one process at a time.

2. The places where it is invoked are such that it can potentially be in use by more than one process at a time.

If the user determines that the procedure or function falls into the first category, then the procedure or function block is coded following the rules given in Section 11.

If, on the other hand, it falls into the second category, the user must make a choice between the following courses of action:

1. to insure that during execution, the Real Time Executive (RTE) allows only one process at a time to use it;

2. to insure that during execution, more than one process can use it at a time.

A procedure or function in whose respect the first course of action is taken, is called "exclusive". One in whose respect the second courseof action is taken is called "reentrant". The opening

---

* The term "reentrancy" denotes the property of being reentrant.

statements of such procedures and functions must contain specific indication of their exclusive or reentrant property.

27.1.1 Exclusive Procedures And Functions

An exclusive procedure or function is one in which the RTE allows only one process to use at any given time. A procedure or function is designated exclusive by the presence of the keyword EXCLUSIVE in the opening statement of its block definition.

27.1.2 Defining An Exclusive Procedure

The form of the opening statement of an exclusive procedure is as shown below:

```
| label: PROCEDURE(i1,i2,...) ASSIGN(a1,a2,...)
|        EXCLUSIVE;
```

1. label is a legal HAL/S/V identifier constituting the procedure name.

2. i1, i2,... and a1, a2... are lists of input and assign parameters as described in Section 11.2.

3. The keyword EXCLUSIVE designates an exclusive procedure.

Example:

```
| P: PROCEDURE(A) EXCLUSIVE;
|  DECLARE A INTEGER;
|   .
|   .
|   .
| CLOSE P;
|
```

The template corresponding to an exclusive external procedure must also bear the keyword EXCLUSIVE.

Example:

The template corresponding to P would be:

```
| P:  EXTERNAL PROCEDURE(A) EXCLUSIVE;
|   DECLARE A INTEGER;
| CLOSE P;
|
```

27.2 DEFINING AN EXCLUSIVE FUNCTION

The form of the opening statement of an exclusive function is as shown below:

```
label : FUNCTION (i1,i2,...) attributes EXCLUSIVE;;
```

1. label is a legal HAL/S/V identifier constituting the function name.
2. i1,i2 is a list of input parameters as described in Section 11.2.
3. attributes defines the type and, where applicable, precision of the function, as described in Section 11.2.
4. The keyword EXCLUSIVE designates an exclusive function.

The template corresponding to an exclusive external function must also bear the keyword EXCLUSIVE.

Example:

The template corresponding to:

```
F:  FUNCTION BOOLEAN EXCLUSIVE;
      .
      .
      .
CLOSE F;
```

would be:

```
F: EXTERNAL FUNCTION BOOLEAN EXCLUSIVE;
CLOSE F;
```

27.3 BEHAVIOR OF EXCLUSIVE PROCEDURES AND FUNCTIONS

If an exclusive procedure or function is in use by process A, and a process B tries to invoke it, then the RTE places process B in the waiting state until process A returns from its use.

27.3.1 Reentrant Procedures And Functions

A reentrant procedure or function is one in which deliberate steps are taken by the programmer to ensure correct execution when the RTE allows more than one process to use it simultaneously. A procedure or function which is intended to be reentrant must possess

the keyword REENTRANT in its opening statement.

This is a necessary, but not sufficient condition to ensure reentrancy. The programmer must observe certain additional guidelines unenforceable by a HAL/S/V compiler to ensure that a procedure or function is truly reentrant in all relevant respects.

27.4 DEFINING A REENTRANT PROCEDURE

The form of the opening statement of a reentrant procedure is shown below:

```
|
| label :  PROCEDURE(i1,i2,...) ASSIGN(a1,a2,...)
|          REENTRANT;
|
```

1. label is a legal HAL/S/V identifier constituting the procedure name.
2. i1,i2, and a1,a2,... are lists of input and assign parameters as described in Section 11.2.
3. The keyword REENTRANT indicates that the procedure is to be considered reentrant.

27.5 DEFINING A REENTRANT FUNCTION

The form of any opening statement of a reentrant function is shown below:

```
|
| label : FUNCTION(i1,i2,...) attributes REENTRANT;
|
```

1. label is a legal HAL/S/V identifier constituting the function name.
2. i1,i2,... is a list of input parameters as described in Section 11.2.
3. attributes defines the type and, where applicable, precision of the function as described in Section 11.2.
4. The keyword REENTRANT indicates that the function is to be considered reentrant.

The template corresponding to an external reentrant function must also possess the keyword REENTRANT.

27.6 BEHAVIOR OF REENTRANT PROCEDURES AND FUNCTIONS

If a reentrant procedure or function is in use by a process A, and a process B tries to invoke it, the RTE allows the invocation to proceed without restriction.

27.7 LOCAL DATA IN REENTRANT BLOCKS

The most important consideration in writing reentrant procedures and functions is that of declaring local data. The issue that confronts the programmer is whether for each local data item he merely wants one "copy" of it, to be shared by all processes concurrently executing the block; or whether a separate "copy" for each process is wanted. Normal reentrant procedures require that execution by one process be completely decoupled from execution by another. Unlike HAL/S, in HAL/S/V we don't have the first alternative.

Separate copies of a local data item for each process concurrently executing a reentrant block are generated by the RTE.

Because in HAL/S/V we don't permit sharing of one single copy of data by concurrently executing muliple instances of a procedure (or function) use of words AUTOMATIC or STATIC is not required.

27.8 OTHER CONSIDERATIONS IN REENTRANT BLOCKS

To preserve complete reentrancy of the code inside a reentrant procedure or function, this guideline must be adhered to:

Any procedure or function invoked by the reentrant block should also be reentrant.

It should be noted that no update block in a reentrant procedure or function can itself be reentrant because of the inherent properties of an update block (see Section 26.4). However, the processes executing the reentrant procedure or function can only pass through the update block serially. Hence, it appears as if process swaps were inhibited pending passage through the update block by each process, and cross-coupling of computational results in different processes still cannot occur. Hence, complete reentrancy is still effectively being preserved.

CHAPTER 28

THE HAL/S/V NAME FACILITY

The ability to maintain "pointers" to specified data items is a valuable feature of many programming languages. However, it presents a problem for verification.

If the pointer value is treated as merely another name for an object, as in HAL/S, aliasing of the worst kind can result. In such a case, the object has a declared name and any number of other aliases. Moreover, access is permitted via any of these names and assertions regarding the object may be written using any one of the aliases. Assertions made using any of the aliases may be falsified by accessing the object via any other - a fact which is not apparent from the program text. Therefore, the verification system is constrained to keep track of all aliases for every data object. For this reason, HAL/S/V permits no name data item to be declared.

CHAPTER 29

REPLACE MACROS AND IN-LINE FUNCTIONS

It was stated in Chapter 5 of Part I that the REPLACE statement of HAL/S had been removed from HAL/S/V. This also applies to the parameterized version called REPLACE MACROS, and for the same reasons.

The behavior of a program and, hence the verifiability, is governed by the program text. The HAL/S REPLACE facility allows arbitrary changes to be made to the program text. Such changes generate essentially new programs for which a previous verification may be invalid. Thus, complete verification requires that each possible program text be considered individually. The work involved is potentially exponential in the number of replace statements.

"In-line functions" in HAL/S are parameterless functions designed to enhance the versatility of the parametric replacements. However, without REPLACE MACROS, they are of little use. They are executed in-line and cannot be invoked elsewhere in the program. Moreover, according to the HAL/S/V scoping requirements outlined in Chapter 11, a function may access non-local data only if it is passed as a parameter at the call site. Hence, a parameterless function is essentially a constant. For these reasons, in-line functions are disallowed in HAL/S/V.