8. REAL TIME CONTROL

HAL/S contains a comprehensive facility for creating a multi-processing job structure in a real time programming environment. At run time a Real Time Executive (RTE) controls the execution of processes held in a process queue. HAL/S contains statements which schedule processes (enter them in the process queue), terminate them (remove them from the process queue), and otherwise direct the RTE in its controlling function. HAL/S also contains means whereby the use of data by more than one process at a time is managed in a safe, protected manner at specific, localized points within the processes.

8.1 Real Time Processes and the RTE.

In HAL/S, a program or task may be scheduled as a process and placed in the process queue. Although the process created is given the same name as the program or task, it is important to distinguish the static PROGRAM or TASK block from the dynamic program or task process created. Two processes are actually involved in the creation of a process: the scheduling process, or "father"; and the scheduled process, or "son".[1]

A process is said to be either "dependent" or "independent", as designated when created. A program or task process is "dependent" if it is absolutely dependent for its existence upon the existence of its father. If a program process is "independent" its existence is independent of that of all other processes. If a task process is "independent" its existence is generally independent of that of all other processes with an important exception: the program process in whose static PROGRAM block the static TASK block of the task process is defined.

Each process in the RTE's process queue is at any instant in one of a number of states. For the purposes of this Section, the following states are defined:[2]

- "active" - a process is said to be in the active state if it is actually in execution. Depending on the implementation it may be possible for several processes to be in execution simultaneously.
- "wait" - a process is said to be in the wait state if it is ready for execution but the RTE has decided on a priority basis that its execution should be delayed or suspended.
- "ready" - a process is said to be in the ready state if it is in either the active or the wait states.
- "stall" - a process is said to be in the stall state if some as yet unsatisfied condition prevents it from being in the ready state.

The occurrence of a process being brought into the active state for the first time is called its "initiation".

---

[1] except of course for the first or "primal" process which must be created by the RTE itself.

[2] these states are not necessarily definitive of those actually existing in any particular implementation of the RTE.

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

Execution of a CLOSE or RETURN statement by an active process causes termination of the process and return to the RTE. In the case of a process which has schedule dependent processes, such execution places the father in a stall state until the sons have all terminated before the process is itself terminated.

124

8.2 Timing Considerations.

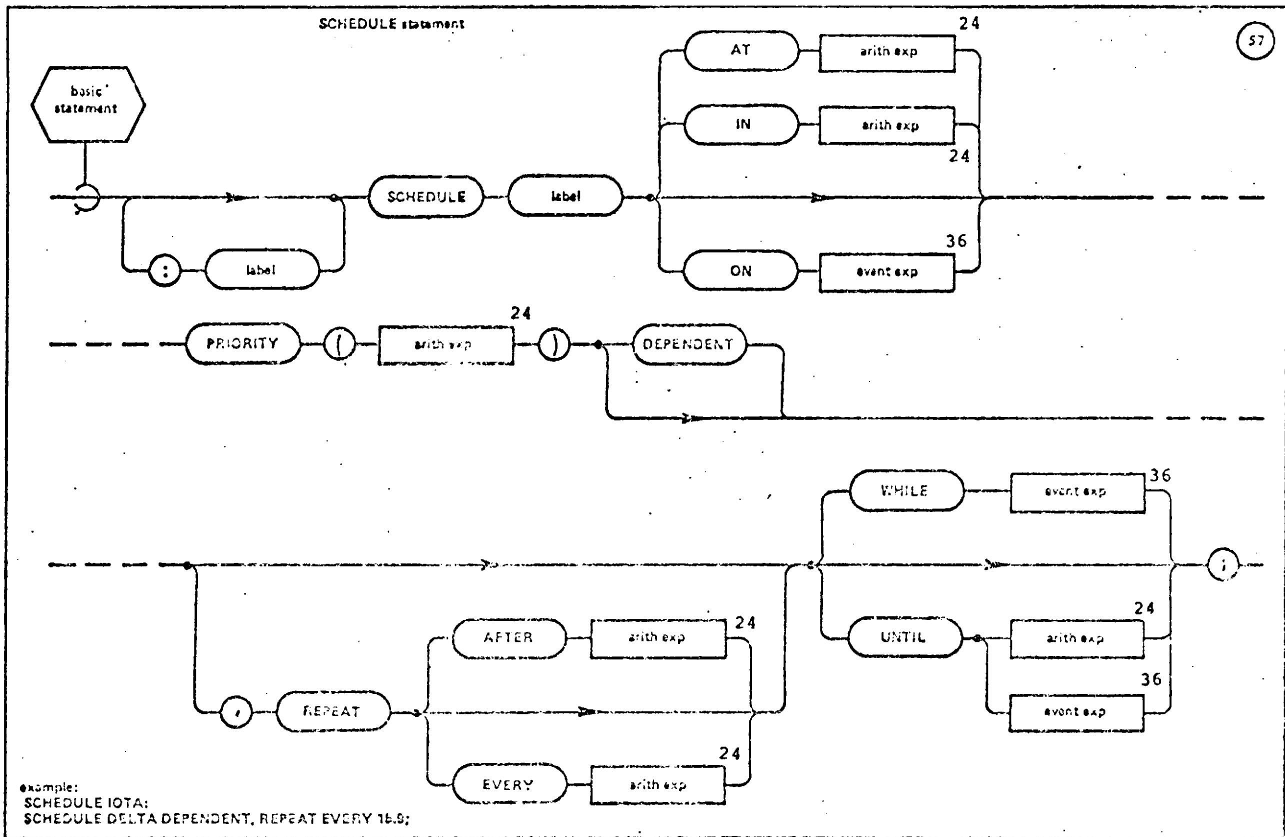
In the HAL/S system, the RTE contains a clock measuring elapsed time ("RTE-clock" time). Time is measured in "machine units" (MU) whose correspondence with physical time is implementation dependent. HAL/S contains several instances of timing expressions which in effect make reference to the RTE-clock.

8.3 The SCHEDULE Statement.

Processes are scheduled (placed in the process queue) by means of the SCHEDULE statement. The statement has many variant forms and offers the following features:

- A process may be scheduled so that the RTE immediately places it in a ready state, or so that the RTE places it in a stall state pending some condition being satisfied.
- A priority of initiation may be specified.
- A process may be designated dependent or independent.
- The cyclic execution of a process may be specified.
- Conditions of future removal of a process from the process queue may be specified.

SYNTAX:



SEMANTIC RULES:

1. SCHEDULE <label> schedules a program or task with the name <label>, placing a new process with name <label> in the process queue. A run time error results if a process of that name already exists in the process queue. Unless otherwise specified the RTE puts the new process in the ready state immediately after execution of the SCHEDULE statement.

2. The phrase IN <arith exp> is used to cause the process to be put in the stall state for a fixed RTE-clock duration. <arith exp> is any unarrayed integer or scalar expression evaluated once at the time of execution of the SCHEDULE statement. If the value is not greater than zero then the process is put immediately in the ready state.

3. The phrase AT <arith exp> is used to cause the process to be put in the stall state until a fixed RTE-clock time. <arith exp> is any unarrayed integer or scalar expression evaluated once at the time of execution of the SCHEDULE statement. If the value is not greater than the current RTE-clock time, and the REPEAT EVERY option is not specified, then the process is put immediately in the ready state. If the value is less than the current RTE time and the REPEAT EVERY option was specified, then phased scheduling takes place. The process is put in a stall state until a future time computed by the expression CT + RE - ((CT-AT) MOD RE), where CT = current time, RE = REPEAT EVERY cycle time, and AT = originally specified AT time.

125

141

4. The phrase ON <event exp> is used to cause the process to be put in the stall state until some event condition is satisfied. Starting from the time of execution of the SCHEDULE statement, the <event exp> is evaluated at each "event change point"[3] until its value becomes TRUE. At that time the process is placed in the ready state. If the value of <event exp> is TRUE upon execution of the SCHEDULE statement, then the process is immediately put in the ready state.

---

[3] the meaning of an "event change point" is defined in Section 8.8.

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

5. The initiation priority is set by means of the phrase PRIORITY (<arith exp>) where <arith exp> is an unarrayed integer or scalar expression which is evaluated once on execution of the SCHEDULE statement. Scalar values are rounded to the nearest integral value. Its value must be consistent with the priority numbering scheme set up for any implementation, otherwise a run time error results.

6. When the keyword DEPENDENT is specified, the process created by the SCHEDULE statement is dependent upon the continued existence of the scheduling process. Note, however, that a TASK process is always ultimately dependent upon the enclosing PROGRAM process. Thus when scheduling a TASK from the PROGRAM level of nesting, the keyword DEPENDENT is redundant and need not be specified.

7. The REPEAT phrase of the SCHEDULE statement is used to specify a process which is to be executed cyclically by the RTE until some cancellation criterion is met. If the REPEAT phrase is not qualified, then cycles of execution follow each other with no intervening time delay. To cause execution of consecutive cycles to be separated by a fixed intervening RTE-clock time delay, the qualifier AFTER <arith exp> is used. <arith exp> is an unarrayed integer or scalar expression evaluated once at the time of execution of the SCHEDULE statement. If the value is not greater than zero then no time delay results. To cause the beginning of successive cycles of execution to be separated by a fixed RTE-clock time delay, the qualifier EVERY <arith exp> is used. <arith exp> is an unarrayed integer or scalar expression evaluated once at the time of execution of the SCHEDULE statement. If the value is such as to cause a cycle to try to start execution before the previous cycle has finished execution, then a run time error results.

8. Between the successive cycles of execution of a cyclic process, the process is put in a stall state and retains the machine resources the RTE reserved for it. It is not temporarily removed from the process queue.

9. The WHILE and UNTIL phrases provide a cancellation criterion for a cyclic process. Before the cyclic process is initiated, they also provide a means of removal of the process from the process queue. In this latter capacity, they also apply to non-cyclic processes. If a cyclic process has no dependent sons, then cancellation merely implies termination of the process between cycles (and thus removal from the process queue). If dependent sons exist, then cancellation implies that the cyclic process is put in a stall state until the sons have all terminated before the process is itself terminated.

10. The UNTIL <arith exp> phrase specifies a cancellation criterion based on RTE-clock time. <arith exp> is an unarrayed integer or scalar expression evaluated once at the time of execution of the SCHEDULE statement. For any process, cyclic or non-cyclic, the following is true. If the value of <arith exp> is not greater than the current RTE-clock time, then the process is never entered in the process queue. Otherwise if the value of <arith exp> becomes equal to the RTE-clock time before the process is initiated, then the process is removed from the process queue. If neither of the above is true, then the following ensues. If the process is non-cyclic, the phrase has no further effect. If the process is cyclic, then cancellation may occur in one of two ways. If the value of <arith exp> becomes equal to the RTE-clock time while the process is in an inter-cycle stall state, the process is cancelled immediately. If it happens during a cycle, the process is cancelled immediately on completion of the current cycle. The next cycle proceeds if cancellation does not occur.

11. The WHILE <event exp> phrase specifies a cancellation criterion based on an event condition. For any process, cyclic or non-cyclic, the following is true. If the value of <event exp> is FALSE at the time of execution of the SCHEDULE statement, then the process is never placed in the process queue. If not, then <event exp> is evaluated at every "event change point" until its value becomes FALSE. If it becomes FALSE before the process is initiated, then the process is removed from the process queue. If neither of the above is true, then the following ensues. If the process is non-cyclic, the phrase has no further effect. If the process is cyclic, then cancellation may occur in one of two ways. If <event exp> becomes FALSE at any "event change point" occurring while the process is in an inter-cycle stall state, the process is cancelled immediately. If it occurs during the cycle, the process is cancelled immediately on completion of the current cycle. If cancellation does not occur, the next cycle proceeds.

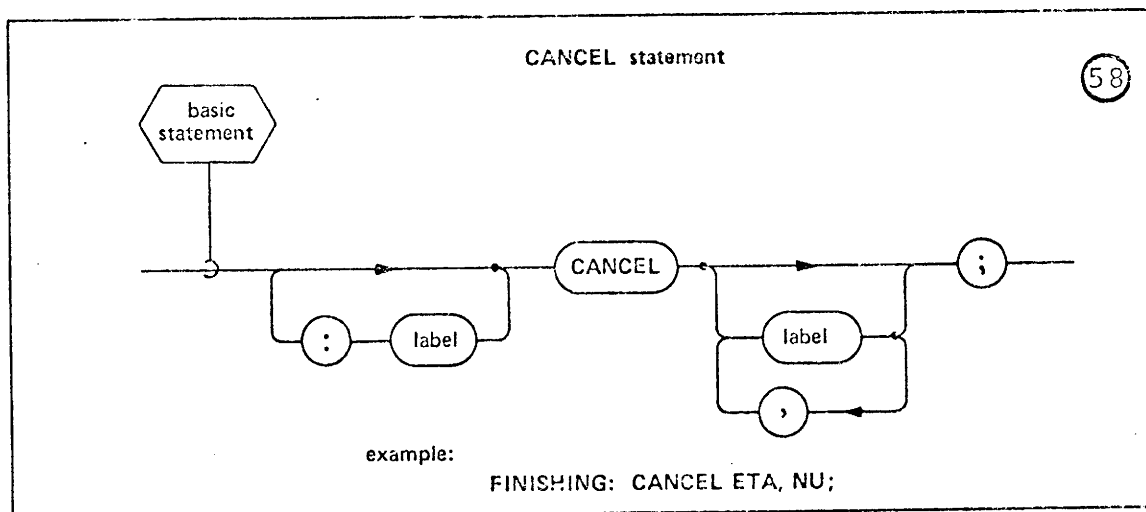
INTERMETRICS INCORPORATED • 701 CONCORD AVENUE • CAMBRIDGE, MASSACHUSETTS 02138 • (617) 661-1840

12. The UNTIL <event exp> phrase also specifies a cancellation criterion based on an event condition. However, it differs fundamentally from the WHILE <event exp> phrase in that it always allows at least one cycle of a cyclic process to be executed. Consistent with this, the phrase has no meaning and therefore no effect in the case of a non-cyclic process. For a cyclic process, the value of the <event exp> is evaluated at every "event change point" from the time of execution of the SCHEDULE statement. Beginning with the end of the first cycle of execution, cancellation may occur in one of two ways. If the <event exp> becomes TRUE at any "event change point" occurring while the process is in an inter-cycle stall state, the process is cancelled immediately. If it occurs during a cycle, the process is cancelled immediately on completion of the current cycle. If cancellation does not occur, the next cycle proceeds.

8.4 The CANCEL Statement.

Cancellation of a process may be the result of the enforcement of a cancellation criterion in the SCHEDULE statement which created the process, or alternatively may be the result of executing a CANCEL statement.

SYNTAX:



SEMANTIC RULES:

1. CANCEL <label> causes cancellation of the process <label> and its dependent processes. A run time error results if the process queue contains no process with that name.[4] The CANCEL statement can be used to cancel any number of processes simultaneously. 106

2. If the CANCEL statement has no <label>, cancellation of the process executing the CANCEL statement and its dependents is implied. 106

---

[4] the default action taken by the Error Recovery Executive for this and other similar errors may be to ignore the error.

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

3. If at the time of execution of the CANCEL statement, a process to be cancelled has not yet been initiated, then the process is merely removed from the process queue. This applies to both cyclic and non-cyclic processes.

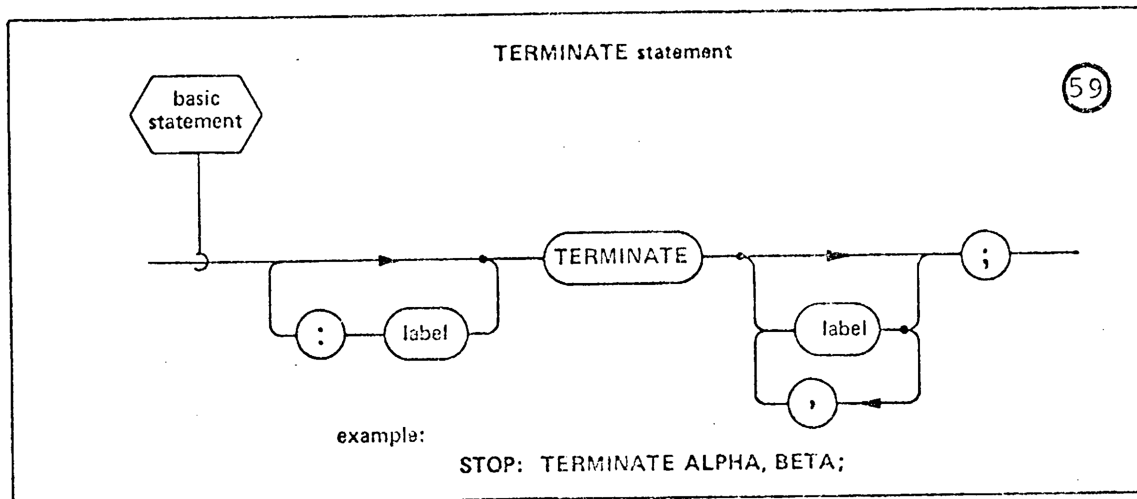
4. If at the time of execution of the CANCEL statement a process to be cancelled has already been initiated, then the following ensues. If the process is non-cyclic and it has already been initiated, the CANCEL statement has no effect. If the process is cyclic, then the process is cancelled at the end of the current cycle of execution.

5. A cancelled process with dependent offspring is put in the stall state until its dependent offspring are terminated. At that time, the parent process is terminated.

8.5 The TERMINATE Statement.

The termination of a process implies the immediate[5] cessation of execution of the process and all its dependent sons, and their removal from the process queue. The TERMINATE statement is used to direct the RTE to terminate specified processes.

SYNTAX:



SEMANTIC RULES:

1. TERMINATE <label> causes termination of the process <label>. A run time error results if a process of that name is not in the process queue, or if it is not a dependent son of the process currently executing the TERMINATE statement. The TERMINATE statement can be used to terminate any number of processes simultaneously.

2. If the TERMINATE statement has no <label>, termination of the process currently executing the TERMINATE statement is implied.

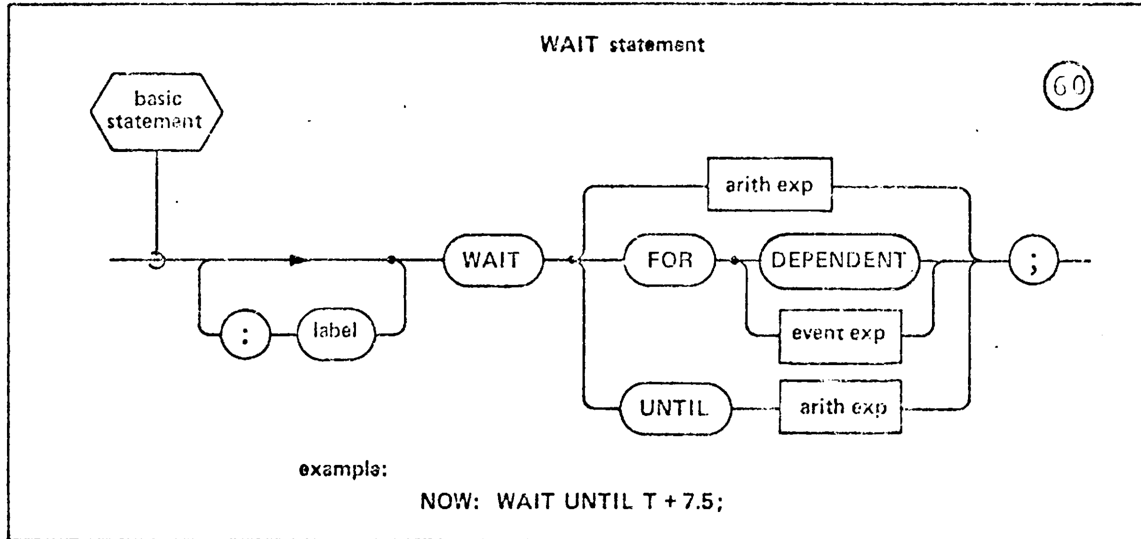
---

[5] subject of course to implementation dependent safety constraints.

8.6 The WAIT Statement

The WAIT statement allows the user to cause the RTE to place a process in the stall state until some condition is satisfied.

SYNTAX:



SEMANTIC RULES:

1. The WAIT <arith exp> version specifies that the process executing the WAIT statement is to be placed in the stall state for an RTE-clock duration fixed by the value of the expression. <arith exp> is an unarrayed integer or scalar expression evaluated once at the time of execution of the WAIT statement. If the value is not greater than zero, the WAIT statement has no effect.

2. The WAIT UNTIL <arith exp> version specifies that the process executing the WAIT statement is to be placed in the stall state until an RTE-clock time fixed by the value of the expression. <arith exp> is an unarrayed integer or scalar expression evaluated once at the time of execution of the WAIT statement. If the value is not greater than the current RTE-clock time, the WAIT statement has no effect.

3. The WAIT FOR DEPENDENT version specifies that the process executing the WAIT statement is to be placed in the stall state until all its dependent sons have terminated. If there are no such processes, the WAIT statement has no effect.

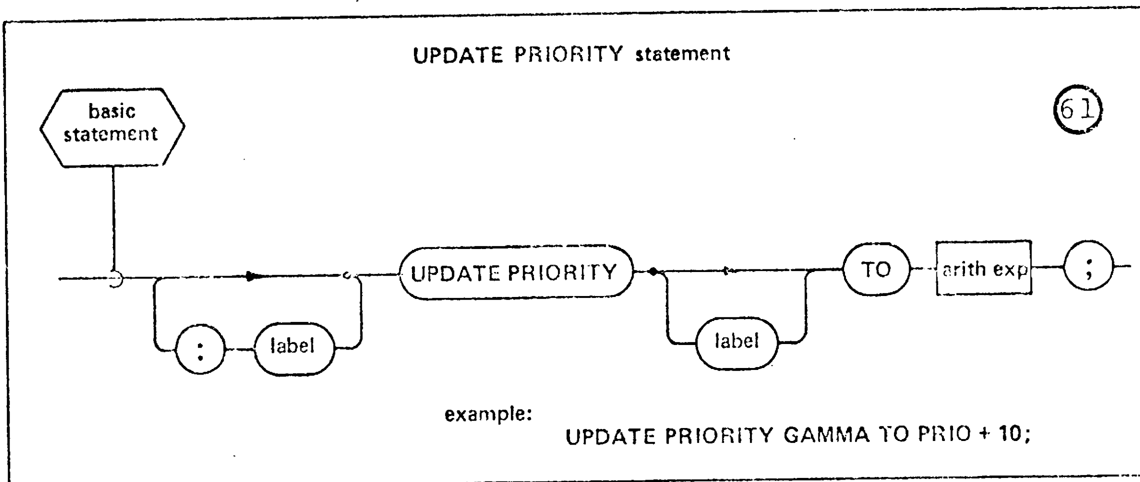
4. The WAIT FOR <event exp> version specifies that the process executing the WAIT statement is to be placed in the stall state until an event condition is satisfied. Starting from the time of execution of the WAIT statement, the <event exp> is evaluated at every "event change point" until its value becomes TRUE, whereupon the process is returned to the READY state. If the value of <event exp> is TRUE upon execution of the WAIT statement, then the statement has no effect.

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

8.7 The UPDATE PRIORITY Statement.

The SCHEDULE statement which creates a process can also specify the priority of its initiation. At any time between the scheduling and the termination of the process, that priority may be changed by means of the UPDATE PRIORITY statement.

SYNTAX:



SEMANTIC RULES:

1. UPDATE PRIORITY <label> is used to change the priority of the process with name <label>. The new priority is given by the value of <arith exp>. <arith exp> is an unarrayed integer or scalar expression whose value must be consistent with the priority numbering scheme set up for any implementation, otherwise a run time error results. Scalar values are rounded to the nearest integral value. A run time error results if there is no process with name <label> in the process queue.

2. UPDATE PRIORITY with no <label> specification is used to change the priority of the process executing the UPDATE PRIORITY statement. <arith exp> has the same meaning as before.

8.8 Event Control

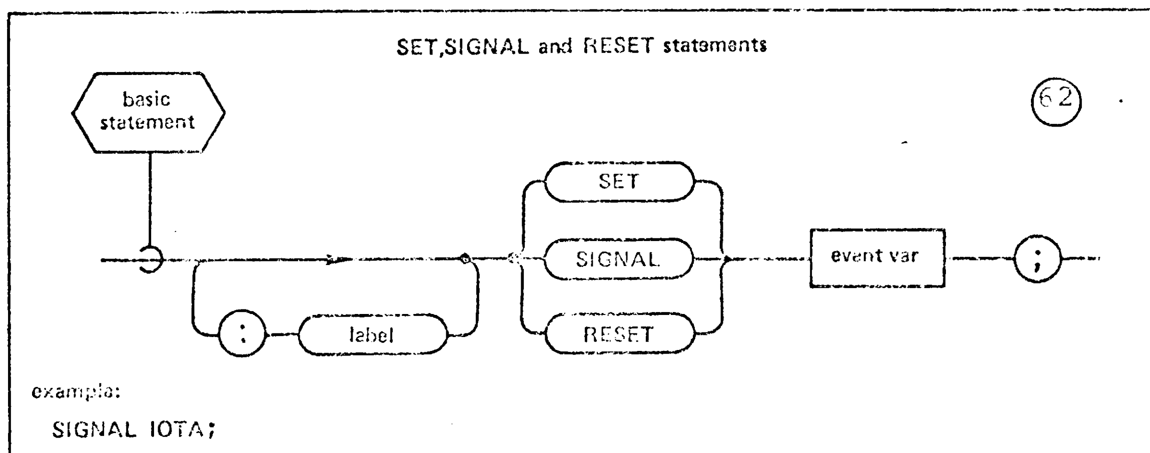
Although a formal specification of events and event expressions has already been given in Sections 4 and 6.3, the Specification has not yet made their purpose clear in the context of real time programming. Superficially event variables are closely akin to boolean variables in that they are binary-valued. Conceptually the two forms of HAL/S events (latched and unlatched) may be thought of as the software counterparts of hardware discretes and timing lines, respectively.

- a latched event may be thought of as a boolean system state which may be SET or RESET by appropriate actions, or momentarily changed for signalling purposes.

- an unlatched event may be thought of as the software counterpart of a timing line which is used purely for signalling - it is normally FALSE but becomes TRUE momentarily when a signal action is executed.

This analogy is no accident, since event variables can actually form the interface between HAL/S software and such hardware control signals. The design and operation of this interface is implementation dependent.

At any instant of time the RTE may be viewed as having a knowledge of all existing events. Whenever the value of an event changes, the RTE senses this so-called "event change point", and may in response perform the evaluation of certain <event exp>s. Depending on the results of the evaluations, the states of one or more processes may be changed. This response of the RTE to changes in event variables is termed an "event action". The value of an event variable can change in response to the environment external to the HAL/S software; depending upon the type of event (see SEMANTIC RULES), a SET, RESET, or SIGNAL statement may also be used to alter the state of an event variable. The only event change actions possible are to ready or cancel one or more process.



GENERAL SEMANTIC RULE:

1. <event var> denotes any unarrayed event variable, subscripted or unsubscripted.

SEMANTIC RULES (latched <event var>s):

1. SET changes the value of the <event var> to TRUE and initiates all event actions depending upon the TRUE state of this event. No action is taken if the <event var> is already TRUE.

2. RESET changes the value of the <event var> to FALSE and initiates all event actions depending upon the FALSE state of this event. No action is taken if the <event var> is already FALSE.

3. SIGNAL does not change the state of a latched event.

4. If a latched event is TRUE, SIGNAL initiates all event actions depending upon the FALSE state of this event.

5. If a latched event is FALSE, SIGNAL initiates all event actions, depending upon the TRUE state of this event.

SEMANTIC RULES (unlatched <event var>s):

1. SET and RESET are illegal for unlatched <event var>s.

2. When used in a <bit expression>, an unlatched event variable is equivalent to a literal "FALSE".

3. SIGNAL initiates all event actions depending upon the TRUE state of this event. Note that when an event expression depends upon a logical product of multiple <event var>s, at most one such <event var> can be unlatched if the event action is ever to be taken.

SUMMARY:

		SET	RESET	SIGNAL
unlatched event		illegal	illegal	Take all event actions depending on TRUE state of <event var>
latched event	old value is FALSE	1. Set event state to TRUE 2. Take all event actions depending on TRUE state of <event var>	no action	Take all event actions depending on TRUE state of <event var>
latched event	old value is TRUE	no action	1. Set event state to FALSE 2. Take all event actions depending on FALSE state of <event var>	Take all event actions depending on FALSE state of <event var>

8.9 Process-events.

Every program or task block has associated with it a "process-event" of the same name. This process-event behaves in every way like a latched event except that it may not appear in SET, RESET or SIGNAL statements. Its purpose is to indicate the existence of its associated program or task process. If a process of the same name as the process-event exists in the process queue, the value of the process-event is TRUE, otherwise it is FALSE.

8.10 Data Sharing and the UPDATE Block.

The UPDATE block provides a controlled environment for the use of data variables which are shared by two or more processes. If controlled sharing of certain variables is desired, they must possess the LOCK(N) attribute, where N indicates the "lock group" of the variable (see Section 4.5). LOCKed variables may only be used inside UPDATE blocks. A LOCKed variable appearing inside an UPDATE block is said to be "changed" within the block if it appears in one or more statements which may change its value (the left-hand side of an assignment for example). It is said to be "accessed" if it only appears in contexts other than the above.

A formal specification of the UPDATE block appears in Section 3.4. The manner of operation of an UPDATE block is implementation dependent, but is such as to provide certain safety measures.

OPERATIONAL RULES:

1. If two processes both require variables from the same lock group to be changed, then the first process entering its UPDATE block must complete execution of the block before the other process can enter its own UPDATE block. The second process is placed in a stall state for the duration.

2. If one process entering an UPDATE block requires a variable(s) with the attribute LOCK(*) to be changed, then the situation is equivalent to one in which the process requires use of a variable from every lock group. [124]

3. If only one of the processes requires a variable of a lock group to be changed, the other merely requiring it to be accessed, then depending on the implementation, either Rule 1 or 2 holds, or some overlap in execution of the two processes' UPDATE blocks is allowed. The nature of such overlap must be such as to provide exclusive use of the lock group by the process requiring its change between the point where the variable is changed and the close of the UPDATE block.

4. If both processes only require a variable of the same lock group accessed, then execution of the two processes' UPDATE block may be allowed to overlap depending upon implementation.

5. If there are several simultaneous conflicts in using shared variables because of the participation of more than two processes, or more than one lock group, then the most restrictive of Rules 1 through 4 required is applied to resolve the conflicts.

9. ERROR RECOVERY AND CONTROL

References to so-called 'run time errors' have been made elsewhere in this Specification. Such errors arise at execution time through the occurrence of abnormal hardware or system software conditions. Each HAL/S implementation possesses a unique collection of such errors. The errors in the collection are said to be "system-defined". In any implementation every possible system-defined error is assigned a unique "error code". In addition, a number of other legal error codes not assigned to system-defined errors may exist. These can be used by the HAL programmer to create "user-defined" errors. All run time errors, both system- and user-defined, are classified into "error groups". The error code for an error consists of two positive integer numbers, the first representing the error group to which it belongs, and the second uniquely identifying it within its group. The method of classification is implementation dependent. E

At run time an Error Recovery Executive (ERE) senses errors, both system-defined and user-defined, and determines what course of action to take. For every error group, a standard system recovery action is defined which the ERE will take unless error recovery has been otherwise directed by the user. Depending on the error and the implementation, the standard system recovery action may be to terminate execution abnormally, to execute a fix-up routine and continue, or to ignore the error.

In a real time programming context, every process in the process queue has a separate, independent "error environment" which is continuous from the time of initiation of the process to the time of its termination. At any instant of time the "error environment" of a process is the totality of error recovery actions in force at that time for all possible errors. At the time of initiation of the process, the standard system recovery action is in force for all errors.

HAL/S possesses two error recovery and control statements. The ON ERROR statement is used to modify the error environment of a process at any time during its life. The SEND ERROR statement is used for the two-fold purpose of creating user-defined error occurrences, and simulating system-defined error occurrences. E

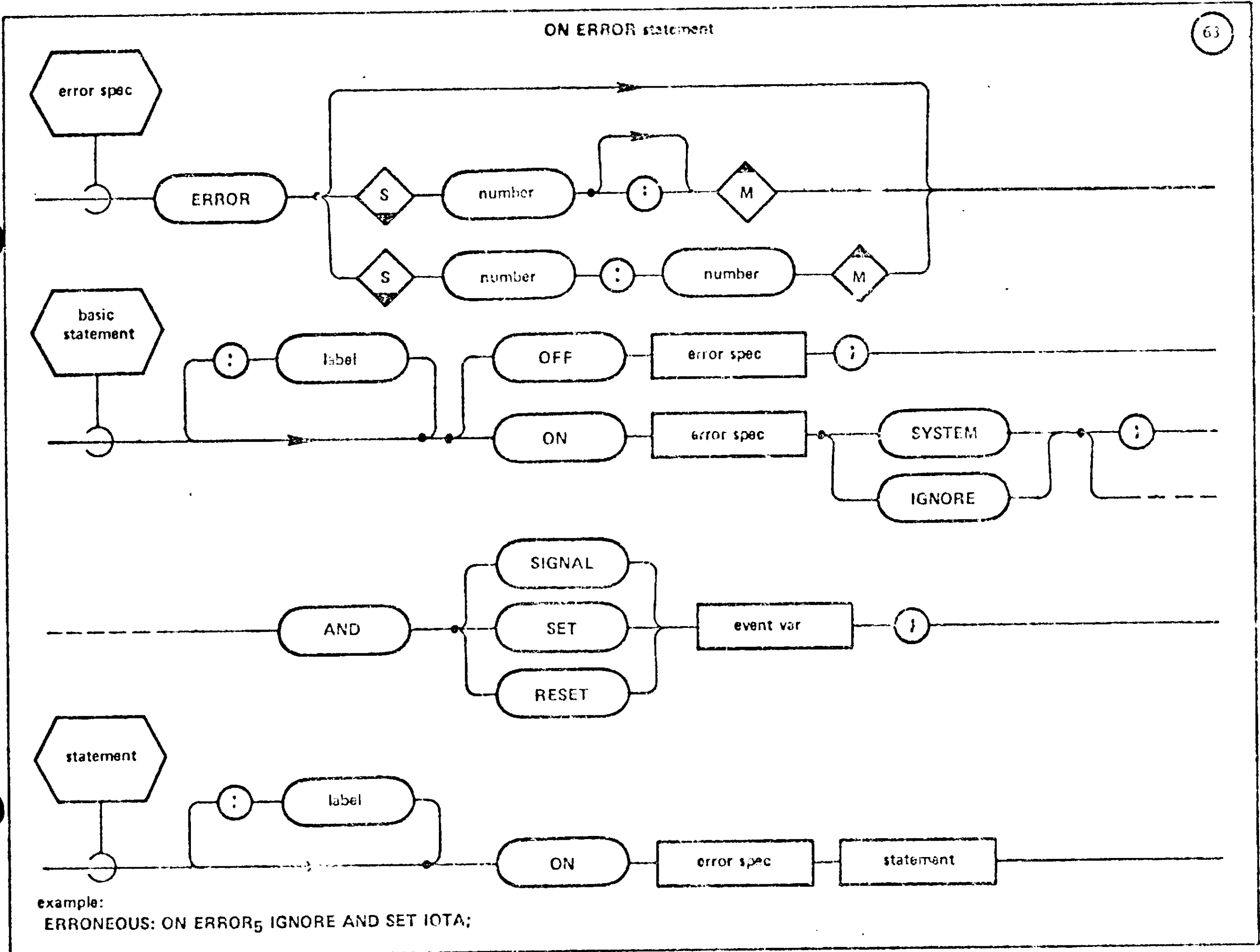
9.1 The ON ERROR Statement.

The ON ERROR statement is used to change the error environment prevailing at the time of its execution. It can change the error recovery action for one selected error code, for one selected error group, or for all groups simultaneously. There are two basic forms of the statement: ON ERROR and OFF ERROR.

Error environment modification operates according to HAL/S name scope rules. If an ON ERROR with a given error specification is executed in a particular code block, then the modified recovery action remains in force until one of three things happen:

- the modification is superseded by execution of a second ON ERROR with the same error specification.
- the modification is removed by execution of an OFF ERROR with the same error specification, the recovery action thereupon reverting to that in force on entry into the code block.
- the modification is automatically removed by exit from the code block.

SYNTAX:



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

SEMANTIC RULES:

1. The ON ERROR statement consists of two parts: a specification of an error action to be taken by the ERE, preceded by an <error spec> specifying the error number, error group or groups to which the action is to apply.

2. There are three forms of <error spec>, for specifying either all error groups, or a selected error group, or a selected error code.

   - The form of <error spec> without subscript is used to specify all error groups.

   - The subscript construct <number> with optional following colon is used to specify a selected <error group>. The value of <number> is restricted to the set of error group numbers defined for a particular implementation.

   - The subscript construct <number>: <number> is used to specify a selected error code. The leftmost <number> designates the error group number; the rightmost <number> the selected error number within the group. Values are restricted to the set of error codes defined for a particular implementation.

3. The form ON ERROR .... specifies the modification of the error recovery actions for the given <error spec>. OFF ERROR .... specifies the removal of a modification previously activated in the same name scope for the same <error spec>. If no such modification exists, the OFF ERROR is effectively a no-operation.

4. The presence of the IGNORE clause specifies that in the event of occurrence of a specified error, the ERE is to take no action other than allow execution to proceed as if the error had not occured. The IGNORE action may not be permitted for certain errors.

5. The presence of the SYSTEM clause specifies that in the event of the occurrence of a specified error, the ERE is to take the standard system recovery action.

6. The form ON ERROR ... <statement> specifies that <statement> is to be executed on the occurrence of a specified error. <statement> may optionally be labelled. However, such labels may only be referenced by EXIT or REPEAT statements within the (compound) <statement> thus labelled. After execution of <statement>, execution normally restarts from the executable statement following the ON ERROR statement. Execution of <statement> itself may of course modify this.

129

7. It is important to note that the form ON ERROR .... <statement> is itself a <statement> whilst other forms of ON ERROR are <basic statement>s. The form ON ERROR ... <statement> may therefore not be the true part of an IF...THEN...ELSE statement.

8. If an ON ERROR possesses a SYSTEM or IGNORE clause, it may also possess an additional SIGNAL, SET, or RESET clause. The purpose is to cause the value of an <event var> to be changed on the occurrence of a specified error. Its semantic rules are the same as those described for the corresponding SIGNAL, SET and RESET statements in Section 8.8. Note that if <event var> contains a subscript expression, then that expression will be evaluated at the time of execution of the ON ERROR statement, not on the occurrence of the error.

PRECEDENCE RULE:

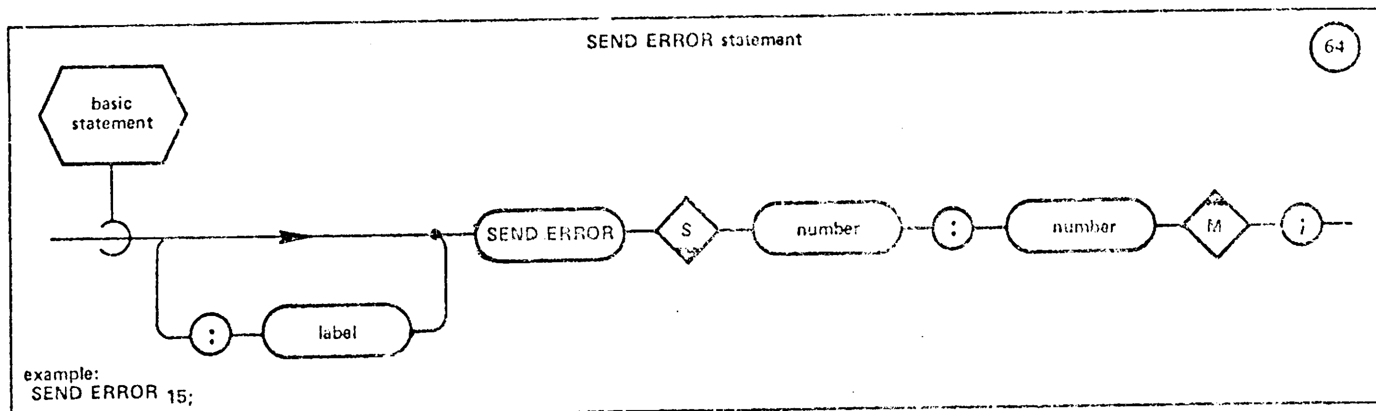
In a code block the action specified by an ON ERROR is only superseded by another if the two <error spec>s are of identical form. Similarly an OFF ERROR nullifies the effect of a previous ON ERROR only if the two <error spec>s are of identical form. However, different forms of <error spec >may involve the same error group or error code. It is logically possible for up to three ON ERRORs, each with a different form of <error spec> as described in Rule 2 above, to be active simultaneously and involve the same error code. The ON ERROR precedence order for determining the recovery action in the event of an error occurrence is as follows:

Error Specification	<error spec> subscript construct	Precedence
		LAST
all groups	-	1
selected group	{ <number> : <number> }	2
selected error code	<number>:<number>	3
		FIRST

9.2 The SEND ERROR Statement.

The SEND ERROR statement is used to announce a selected error condition to the ERE. If the error selected is 'system-defined' then in effect that error is being simulated.

SYNTAX:



SEMANTIC RULES:

1. <number> : <number> is a subscript construct consisting of two unsigned integer literals. The leftmost <number> designates the error group to which the selected error condition belongs. The rightmost number denotes the error number within the designated group. Values are restricted to the set of error codes defined for a particular implementation. If the error code corresponds to a system-defined error, then that error is simulated by the ERE. Simulation of certain system-defined errors may not be permitted.

2. The action taken by the ERE after announcement of the selected error condition is dictated by the error environment prevailing at the time of execution of the SEND ERROR statement.

10. INPUT/OUTPUT STATEMENTS

The HAL/S language provides for two forms of I/O: sequential I/O with conversion to and from an external character string representation; and random-access record-oriented I/O.

All HAL/S I/O is directed to one of a number of input/output "channels". These channels are the means used to interface HAL/S software with external devices in a run time environment. In any implementation each channel is assigned a unique unsigned integer identification number.

The input/output statements described in this Section are intentionally general-purpose. They provide a basic support facility for applications programming on the Shuttle project. Specialized hardware-oriented I/O commands may be created via features of the HAL/S Systems Language.

10.1 Sequential I/O Statements.

All sequential I/O in HAL/S is to or from character-oriented files. HAL/S pictures these files as consisting of lines of character data similar to a series of printed lines or punched cards. An "unpaged" file simply consists of an unbroken series of such lines. In a "paged" file the lines are blocked into pages, each a fixed, implementation dependent number of lines in length. The choice of paged or unpaged file organization for each sequential I/O channel is specified in an implementation dependent manner.

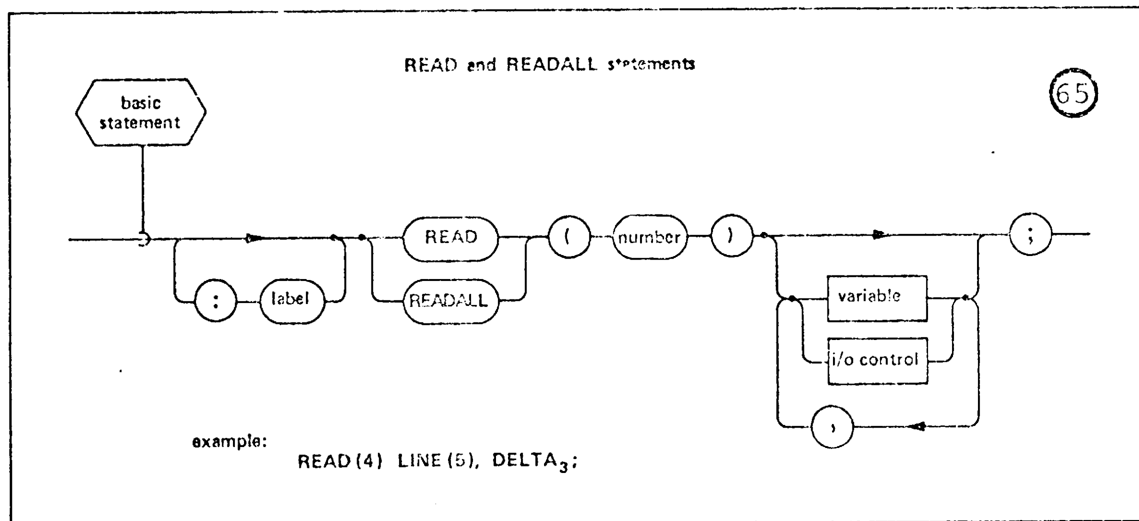
HAL/S pictures the physical device as moving across the file a read or write "device mechanism" which actually performs the data transfer. The device mechanism has at every instant a definite column and line position on the file. The action of transmitting one character to or from the file is followed by the positioning of the device mechanism to the next column on the same line. When the end of the line is reached the device mechanism moves on to the first (leftmost) column of the next line.

The HAL/S sequential I/O statements are the READ, READALL, and WRITE statements. Within these statements I/O control functions can be used to cause explicit positioning of the device mechanism on the file.

10.1.1 *The READ and READALL Statements.*

The sequential input of data is accomplished in HAL/S by employing either a READ or a READALL statement. The choice depends upon the format of the character input and the conversions (if any) which are to be performed. A READ statement is used wherever data in a standard external format is to be input; the READALL is used wherever arbitrary character string images are to be input without conversion.

SYNTAX:



GENERAL SEMANTIC RULES:

1. <number> is any legal I/O channel number.

2. <i/o control> is any legal I/O control function used to position the device mechanism explicitly.

3. Unless overridden by explicit <i/o control> before the first <variable>, the device mechanism is automatically moved to the leftmost column position and advanced to the next line prior to reading the first <variable>. A SKIP, LINE, or PAGE <i/o control> before the first <variable> overrides the automatic line advancement. A TAB or COLUMN <i/o control> overrides the automatic column positioning.

4. An unexpected end of file reached during the reading of data from the input file causes a run time error.

SEMANTIC RULES (READALL Version):

1. <variable> may be any character or structure variable in an assignment context. This specifically excludes input parameters of functions and procedures. If it is of structure type, all the terminals of the template it references must be of character type. In this case, also no nested structure template references are allowed.

2. If <variable> is an array or structure each element thereof is filled sequentially in its "natural sequence".

3. Data is read from the input file character by character from left to right, each <variable> element being filled in turn. Filling of an element is completed either when the end of a line on the file is reached, or when the element has reached its declared maximum length, whichever happens sooner.

SEMANTIC RULES (READ Version):

1. <variable> is any variable which may be used in an assignment context. This specifically excludes input parameters of functions and procedures.

2. If <variable> is a vector or matrix, or an array or structure, each element thereof is filled sequentially in its "natural sequence".

3. The device mechanism (subject to <i/o control>) scans the input file left to right, from line to line, looking for fields of contiguous characters separated by commas, semicolons or blanks. Each field found is in turn transmitted and converted from its standard external format to an appropriate HAL/S data value. Fields may not cross line boundaries except when reading character strings.

4. A semicolon field separator encountered during a normal sequential scan to fill a variable element terminates the READ statement as follows:

   - The current variable element is left unchanged;
   - All remaining <variable>s in the statement are unchanged;
   - All remaining control functions in the statement are ignored.

   <i/o control> functions can force the device mechanism over the semicolon without causing early termination.

ERMETRICS INCORPORATED • 701 CONCORD AVENUE • CAMBRIDGE, MASSACHUSETTS 02138 • (617) 661-1840

5. A null field is transmitted whenever a comma or a semicolon is detected when data is expected. This occurs when a comma or semicolon is:

   - preceded by a comma or semicolon;
   - preceded by one or more blanks following the last comma or semicolon.

111

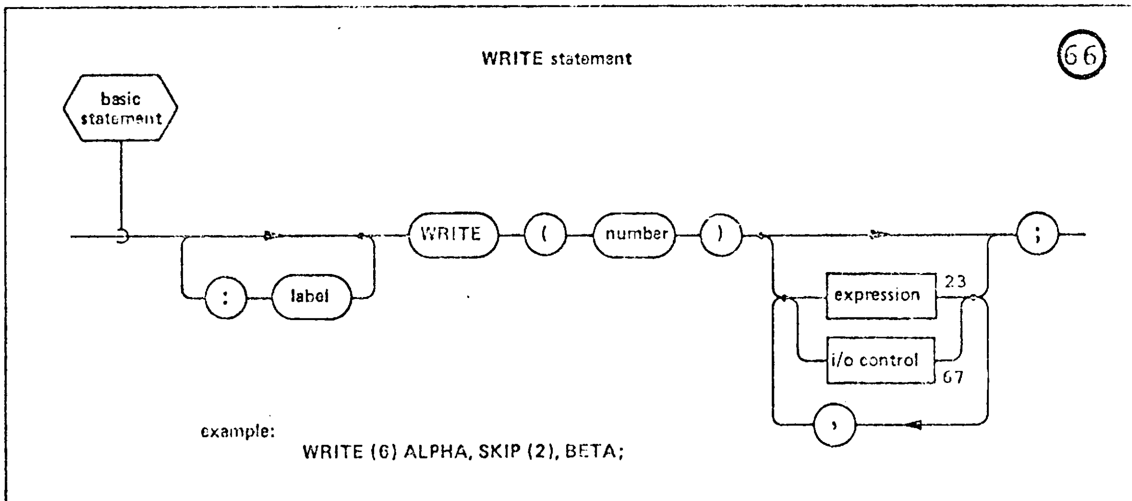
   A null field causes the corresponding variable element to remain unchanged following transmission.

6. Fields are assumed to be in a standard external format matching the type of each corresponding type of variable element. A list of standard external formats is given in Appendix E. A mismatch between standard external format and element type causes a run time error.

10.1.2 The WRITE Statement.

The sequential output of data is accomplished in HAL/S by employing the WRITE statement.

SYNTAX:



SEMANTIC RULES:

1. <number> is any legal I/O channel number.
2. <i/o control> is any legal I/O control function used to position the device mechanism explicitly.
3. There are no semantic restrictions on <expression>.
4. If <expression> is of vector or matrix type, or is an array or structure, then each element thereof is transmitted sequentially in its "natural sequence".

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

5. Unless overridden by explicit <i/o control> before the first <expression>, the device mechanism is automatically moved to the leftmost column position and advanced to the next line prior to transmitting the first <expression>. A SKIP, LINE, or PAGE <i/o control> before the first <expression> overrides the automatic line advancement. A TAB or COLUMN <i/o control> overrides the automatic column positioning.

6. Each element in turn is converted to its standard external format before being transmitted to the output file. A list of standard external formats is given in Appendix E.

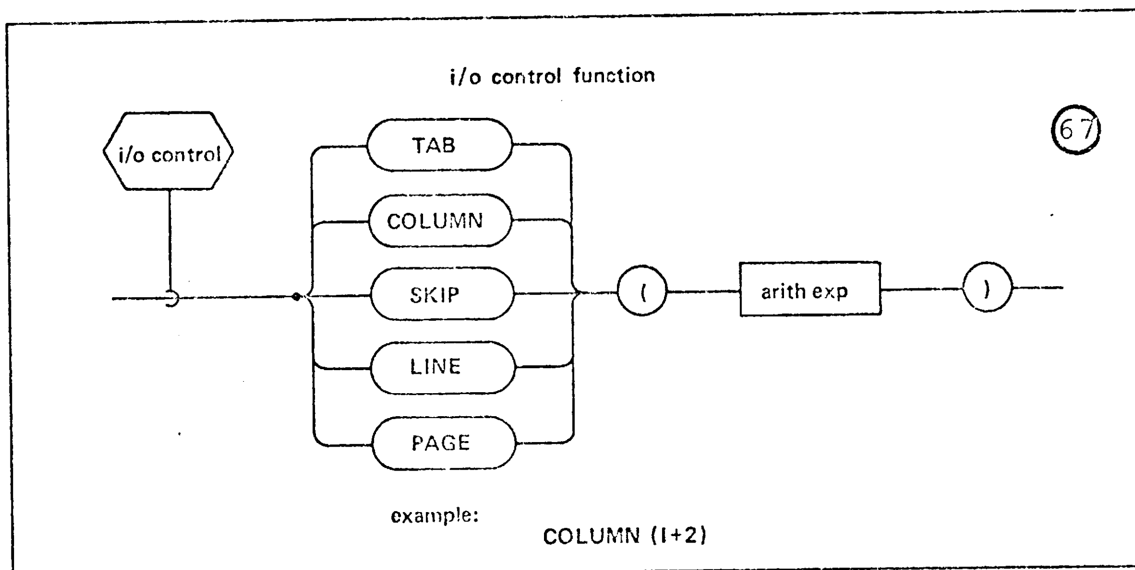
7. Between the transmission of two consecutive elements, the device mechanism is moved to the right by an implementation dependent number of columns. If a TAB or COLUMN <i/o control> separates two consecutive <expression>s then this overrides the automatic movement between transmission of the last element of the first <expression> and the first element of the second <expression>.

8. When a line has been filled to the point where the next converted output field will not fit in the remaining columns, a wrap-around condition occurs. The actions taken in such a case are implementation dependent.

10.1.3 *I/O Control Functions.*

An I/O control function is introduced into a READ, READALL, or WRITE statement to cause explicit movement of the device mechanism. Note that the interpretation of each I/O control function differs depending upon whether the file is paged or unpaged.

SYNTAX:



SEMANTIC RULES:

1. <arith exp> is an unarrayed scalar or integer arithmetic expression specifying a value to the control function. The value is treated as an integer: scalar values are rounded to the nearest integer prior to use. In the following rules, let the value of <arith exp> be denoted by *K*.

2. TAB (*K*) specifies relative movement of the device mechanism across the current line by *K* character positions (columns). Motion is to the right (increasing column index) if *K* is positive, to the left if *K* is negative. Positioning to negative or zero column index values, or to a positive index greater than an implementation dependent maximum causes a run time error.

3. COLUMN (*K*) specifies absolute movement of the device mechanism to column *K* of the current line. Values of *K* may range from 1 to an implementation dependent maximum value. Column indices outside the legitimate range cause run time errors.

4. SKIP (*K*) specifies line movement relative to the current line of the file. A positive value of *K* will cause forward movement. Subject to implementation and hardware restrictions, backward movement is indicated by a negative value of *K*. Error conditions will be indicated if a skip causes movement past either end of the file, or movement in violation of any implementation restriction on the direction of the skip.

5. LINE (*K*) specifies line movement to a specified line number, *K*. Two interpretations occur depending upon whether the file is paged or unpaged.

   - Paged files - LINE (*K*) advances the file unconditionally. *K* may not be less than 1 or greater than the implementation and hardware dependent number of lines per page, otherwise an error condition will be indicated. If *K* is not less than the current line number, the new print position is on the current page; if *K* is less than the current line number, the device mechanism is advanced to line *K* of the next page.

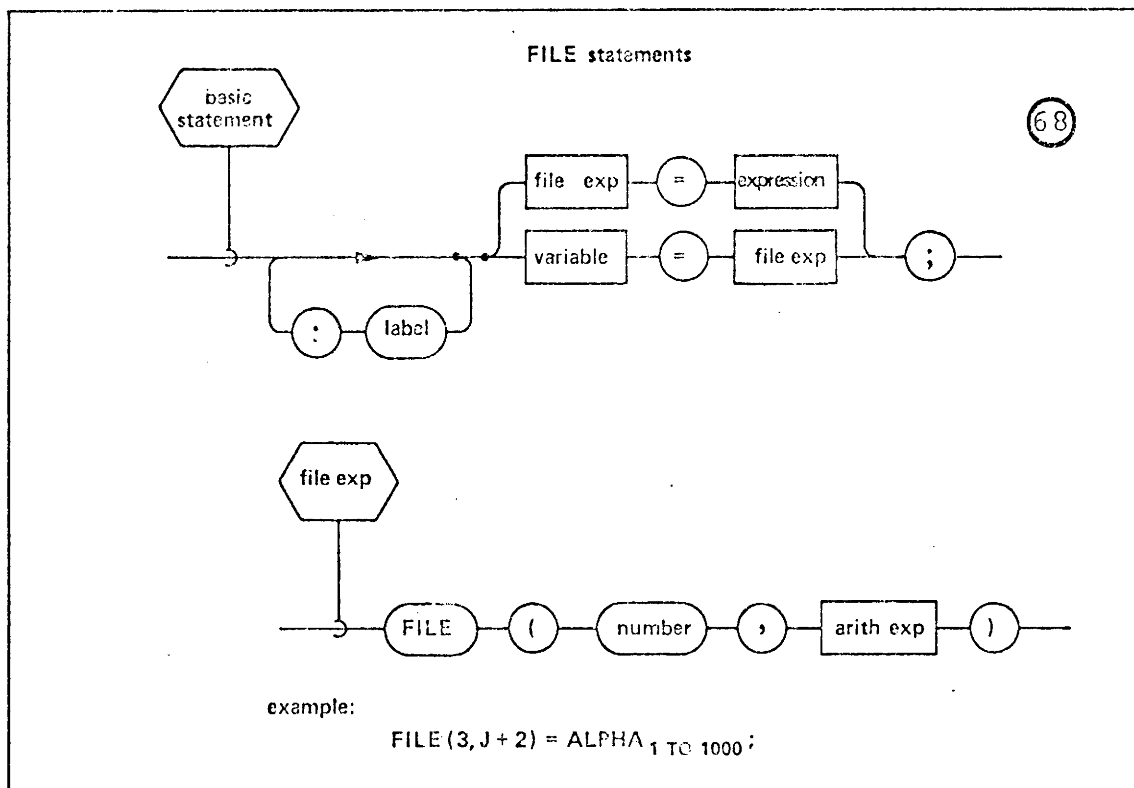
   - Unpaged files - LINE (*K*) positions the device mechanism at some absolute line number in the file. On input *K* must be greater than zero, but not greater than the total number of lines in the file. On output, *K* must merely be greater than zero. In either case, values outside the indicated ranges cause run time errors. Depending on the implementation, values of *K* causing backwards movement may be illegal.

6. PAGE (*K*) is only applicable to paged files and specifies page movement relative to the current page. If *K* is positive the movement is forward, towards the end of file. Depending upon the implementation, negative page values may or may not be legal. The line value relative to the beginning of the page remains unchanged.

10.2 Random Access I/O and the FILE Statement.

Random access I/O is handled by means of the FILE statement. In this access method individual records on a file may be written, retrieved or updated. A unique "record address" is used to specify the particular record on the file referenced.

SYNTAX:



SEMANTIC RULES:

1. The statement is an output FILE statement if <file exp> is on the left of the assignment. If <file exp> is on the right, then the statement is an input FILE statement.

2. <file exp> specifies the random access I/O channel and record address to be referenced. <number> is any legal random access channel number. <arith exp> is any unarrayed integer or scalar expression. If the expression is scalar, its value is rounded to the nearest integer before use. A run time error occurs if its value is not a legal record address.

3. Any record on a random access file may be transmitted by a FILE statement.

4. In the input FILE statement, <variable> is any variable usable in an assignment context. This specifically excludes input parameters of function and procedure blocks. Moreover, <variable> is also subject to the following rules:

   - No component subscripting for bit and character types.

   - If component subscripting is present, <variable> must be subscripted so as to yield a single (unarrayed) element of the <variable>.

   - If no component subscripting is present, but array subscripting is, then all arrayness must be subscribted away.

   - BIT type structure terminals which have the DENSE attribute may not be used, due to packing implications. However, an entire structure with the DENSE attribute may be used.

   - If the <variable> is a structure terminal or a minor structure node (but <u>not</u> if it is a major structure) and if the structure possesses multiple copies, then the number of copies must be reduced to one by subscripting.

5. In the output FILE statement, there are no semantic restrictions on <expression>.

6. Compatibility between data written by an output FILE statement, and later reference to it by an input FILE statement is assumed. The exact interpretation of compatibility is implementation dependent. In general, the FILE statement transmits binary images of the internal data forms, so that compatibility will be guaranteed if the <expression> of the output FILE statement and the <variable> of the input FILE statement have the same data type and organization.

11. SYSTEMS LANGUAGE FEATURES †

11.1 INTRODUCTION

The systems language features of HAL/S are described in this section. The features presented here are in three sections. The new Program Organization features are "Inline Function Blocks" and "%-macros". A data-related feature of this systems language extension is the concept of "TEMPORARY variables". The NAME Facility concerns a new concept in HAL/S, the addition of NAME variables pointing to data or blocks of code. | E

The information contained in this section constitutes an extension of material presented earlier. Accordingly, many of the syntax diagrams presented here are modified versions of earlier diagrams reflecting the extended features. Such modified diagrams are indicated by appending the small leter "s" to the diagram number.

11.2 PROGRAM ORGANIZATION FEATURES

The addition of Inline Function Blocks and "%-macros" to HAL/S extends the information presented in Section 3 concerning program organization. Inline functions are a modified kind of user function in which invocation is simultaneous with block definition. %-macros may be viewed as a class of special purpose implementation dependent built-in functions.

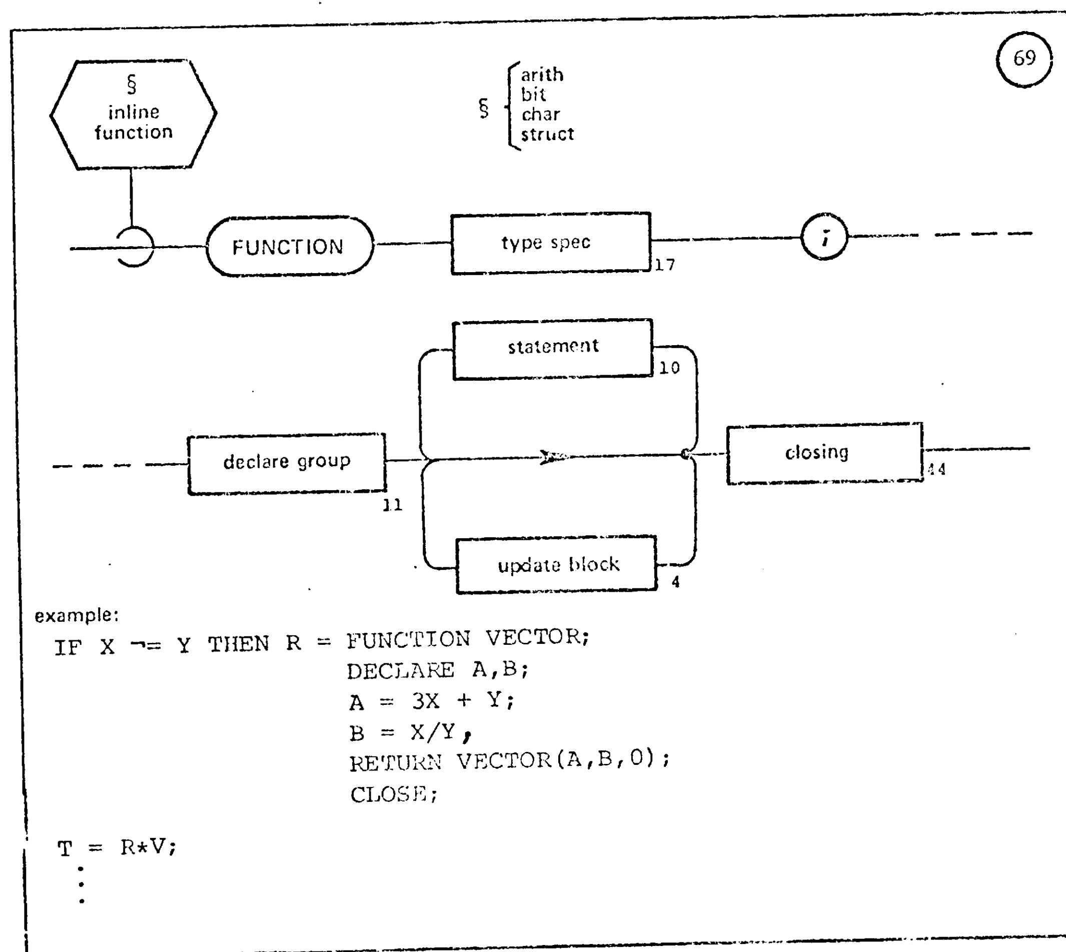
---

† The title indicates that the usage of these constructs is more suited to systems programming rather than applications programming. The programmer is warned that unrestrained and indiscriminate use of certain of these constructs can lead to software unreliability.

11.2.1 *Inline Function Blocks*

The HAL/S Inline Function Block is a method of simultaneously defining and invoking a restricted version of the ordinary user function construct. Its primary purpose is to widen the utility of the parametric REPLACE statement described in Section 4.2. Its appearance is generally in the form of an operand of an expression.

SYNTAX:



SEMANTIC RULES:

1. The syntactic form is actually equivalent to that of a function block except that:

   a) The <§inline function> has no label;

   b) The <§inline function> has no parameters;

   c) The <§inline function> definition becomes an operand in an expression.

2. The semantic rules for an <§inline function> block definition are the same as those for the <function block> definition described in Section 3.3, subject to restrictions listed below.

3. A <§inline function> may not contain the following syntactical forms:

   - All forms of I/O statements;
   - All forms of reference to user-defined PROCEDURE and FUNCTION blocks;
   - Real Time statements.

4. A <§inline function> may only contain one form of nested block, the <update block>. The following block forms are thus excluded:

   - <function block> definitions;
   - <procedure block> definitions;
   - Further nested <§inline function>s.

5. In use, the following semantic restriction holds: <§inline function>s may not appear as operands of the subscript or exponent expressions.

6. The <§inline function> falls into one of the following four categories:

<arith inline>	- <type spec> specifies an inline function of an arithmetic data type: SCALAR, INTEGER, VECTOR or MATRIX.
<bit inline>	- <type spec> specifies an inline function of a bit type: BOOLEAN or BIT.

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

<char inline>	- <type spec> specifies an inline function of the CHARACTER data type.
<struct inline>	- <type spec> specifies an inline function with a structure type specification.

The use of inline functions as operands of HAL/S expressions is discussed in Section 11.2.3.

11.2.2 %-macro References

The HAL/S %-macro facility provides a means of adding functional, special-purpose extensions to the language without requiring syntax changes or extensive rewriting of the compiler programs. The details of the implementation of any given %-macro will depend upon its nature and purpose. Possible options include inline generation of code or links to an external routine performing the processing of the %-macro.

The syntax of the %-macro reference is presented in this section. The invocations of %-macro routines in various expression or statement contexts is described below in Sections 11.2.3 and 11.2.4.

Defined %-macros are described in Appendix I.

SYNTAX:

70

§ % macro

§ arith bit char struct typeless

(<% macro-arg> ,)

% label

SEMANTIC RULES:

1. The %-macro reference falls into one of the following five categories based upon data type:

   - <arith %-macro> is a reference to a %-macro which returns an arithmetic value of INTEGER, SCALAR, VECTOR or MATRIX data type.

   - <bit %-macro> is a reference to a %-macro which returns a bit string value.

   - <char %-macro> is a reference to a %-macro which returns a value of the CHARACTER data type.

   - <struct %-macro> is a reference to a %-macro which returns a structure data value.

   - <typeless %-macro> is a reference to a %-macro which performs some systems function but returns no value and may only be referenced from a <%-macro call statement>. (See Section 11.2.4 below).

   A list of defined %-macros appears in Appendix I. Available %-macros in any implementation will be provided in the appropriate User's Manual.

117

2. The <%label> is a reserved word beginning with the character "%" which identifies the %-macro in question. The character "%" distinguishes %-macro names from all other reserved words in the HAL/S language.

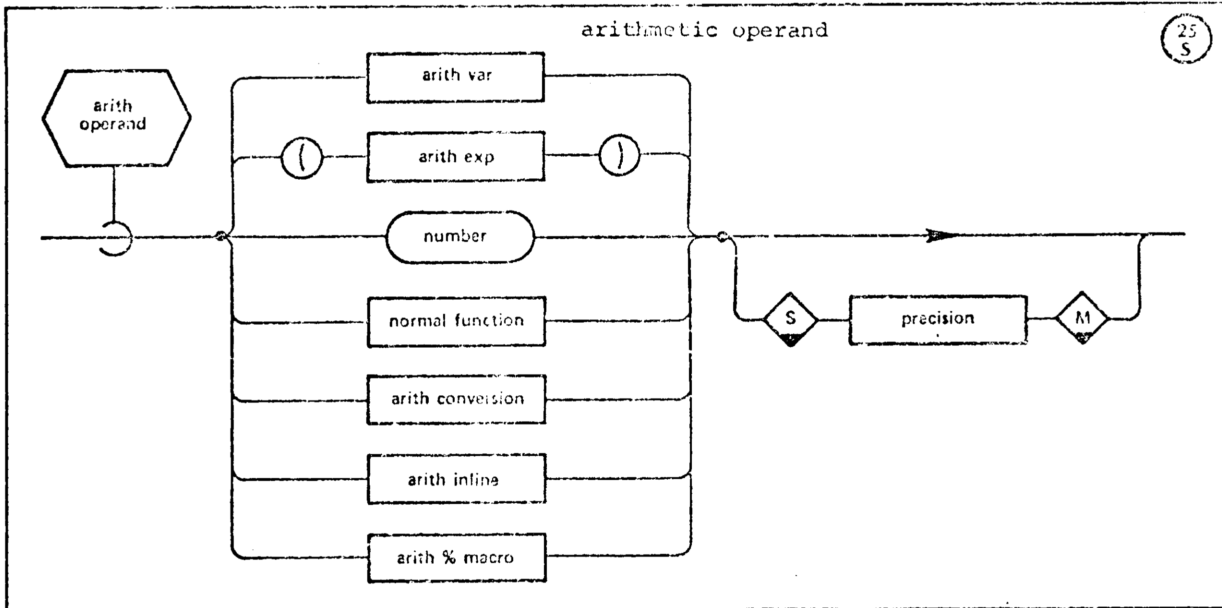
3. A series of one or more arguments of the %-macro reference may be supplied. The type, organization and number of the arguments supplied to the %-macro must be consistent with the requirements of the routine.

4. Details of <%-macro arg>s will be supplied with the definition of a given %-macro.

11.2.3 *Operand Reference Invocations*

Inline Function Blocks are always invoked at the point of their definition as operands of <expression>s. %-macros are also invoked as operands of <expression>s when they are of a definite data type and thus return a value. Similar modifications of several syntax diagrams from Section 6 add these features to arithmetic, bit, and character operands, and to structure expressions.

SYNTAX OF ARITHMETIC OPERAND:



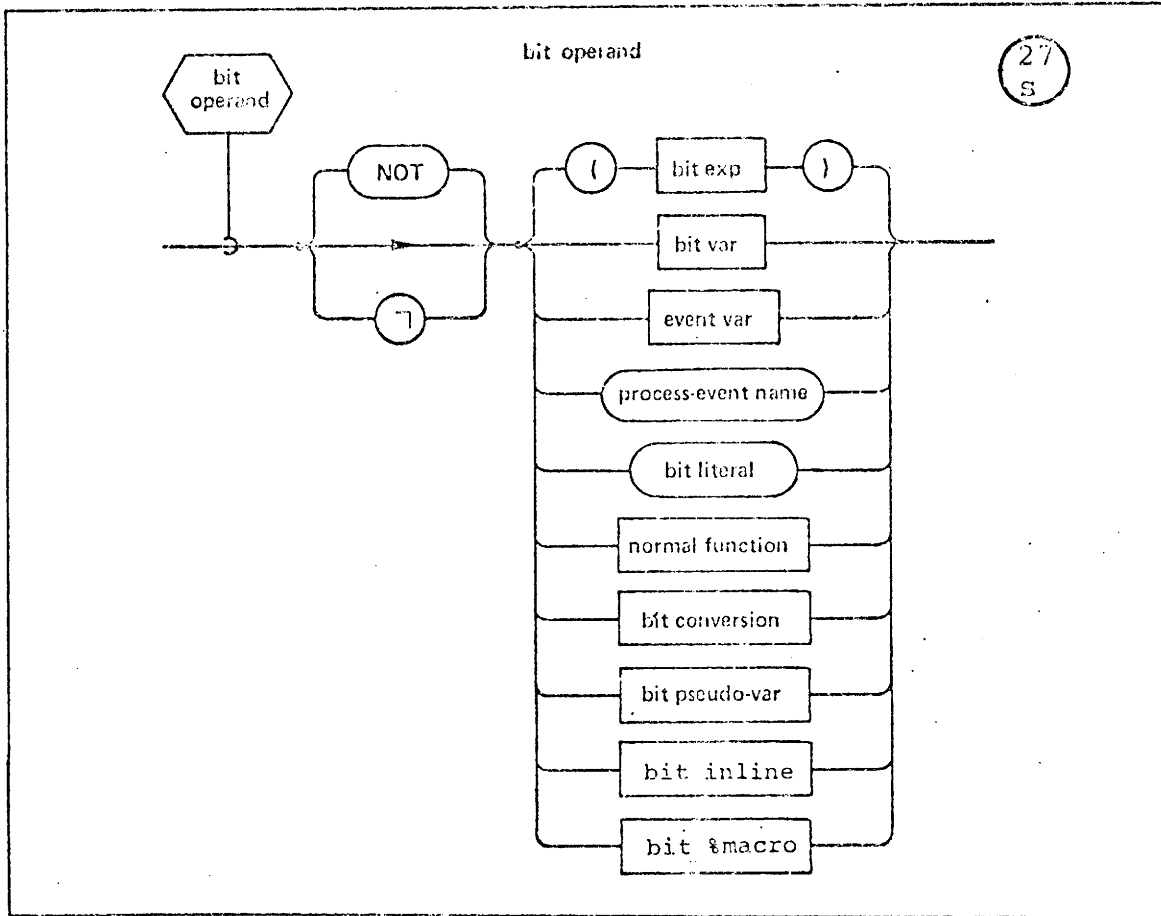
SEMANTIC RULES:

1. This syntax diagram is a systems language extension of the arithmetic operand diagram in Section 6.1.1. The semantic rules of Section 6.1.1 apply to this revised diagram.

2. <arith inline> is an inline function block which has an arithmetic <type spec> in its header statement.

3. <arith %-macro> is a reference to a %-macro which returns an arithmetic value (See 11.2.2 above).

SYNTAX OF BIT OPERAND:



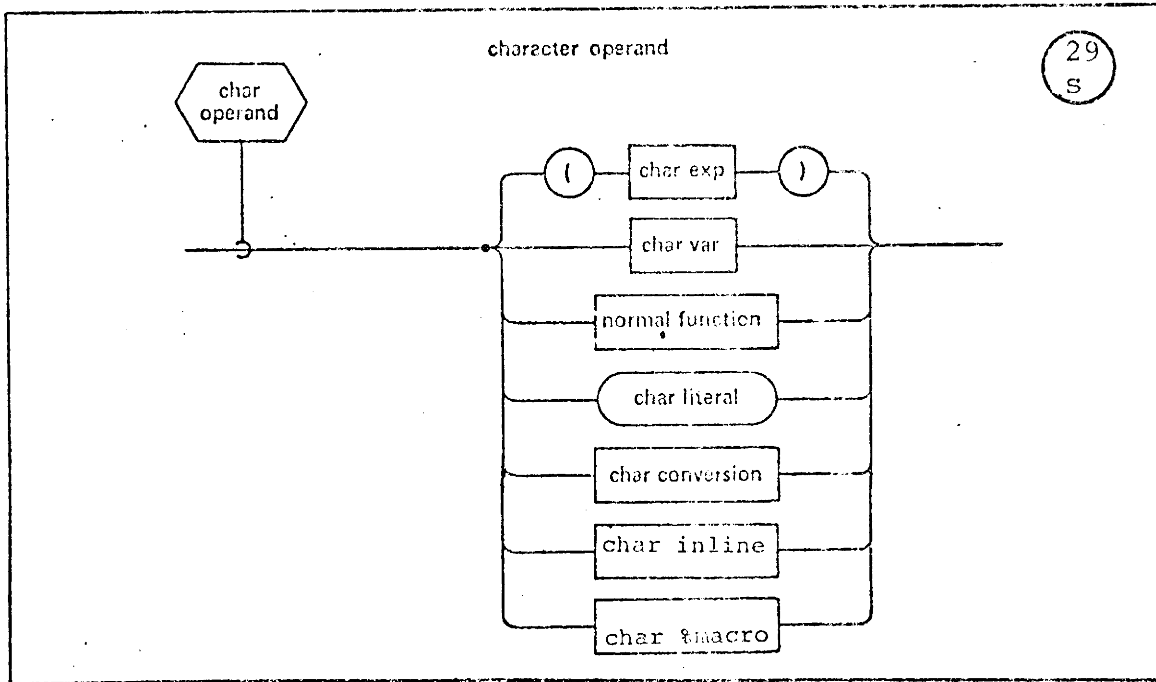
SEMANTIC RULES:

1. This syntax diagram is a systems language extension of the bit operand diagram in Section 6.1.2. The corresponding semantic rules found in Section 6.1.2 also apply to this revised diagram.

2. <bit inline> is an inline function block which has a bit string (BOOLEAN or BIT) <type spec> in its header statement.

3. <bit %macro> is a reference to a %-macro which returns a value of the BIT or BOOLEAN data types.

SYNTAX OF CHARACTER OPERAND:



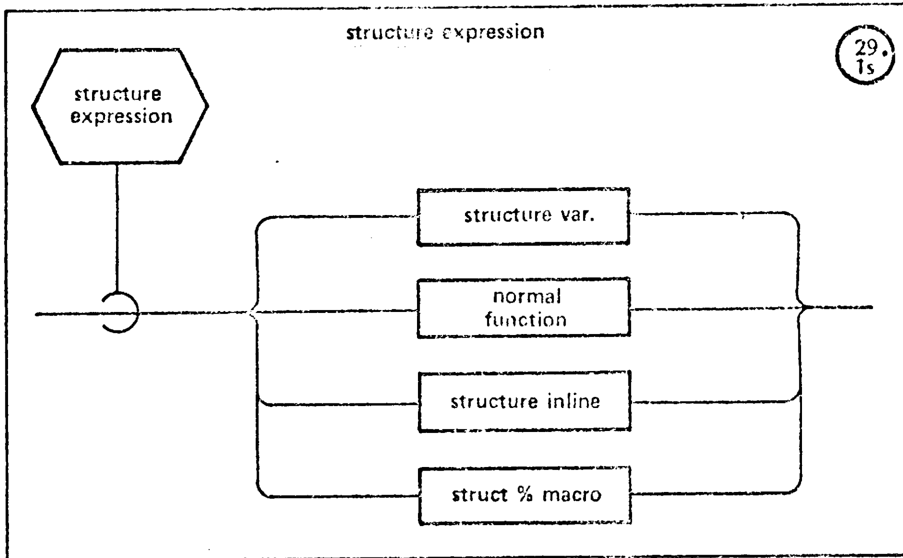
SEMANTIC RULES:

1. This syntax diagram is a systems language extension of the character operand diagram in Section 6.1.3. The corresponding semantic rules found in Section 6.1.3 also apply to this revised diagram.

2. <char inline> is an inline function block which has a CHARACTER <type spec> in its header statement.

3. <char %macro> is a reference to a %-macro which returns a value of the CHARACTER data type.

SYNTAX OF STRUCTURE EXPRESSION:



SEMANTIC RULES:

1. This syntax diagram is a systems language extension of the structure expression diagram found in Section 6.1.4. The semantic rules found in Section 6.1.4 also apply to this revised diagram.

2. <struct inline> is an inline function block which has a structure <type spec> in its header statement.

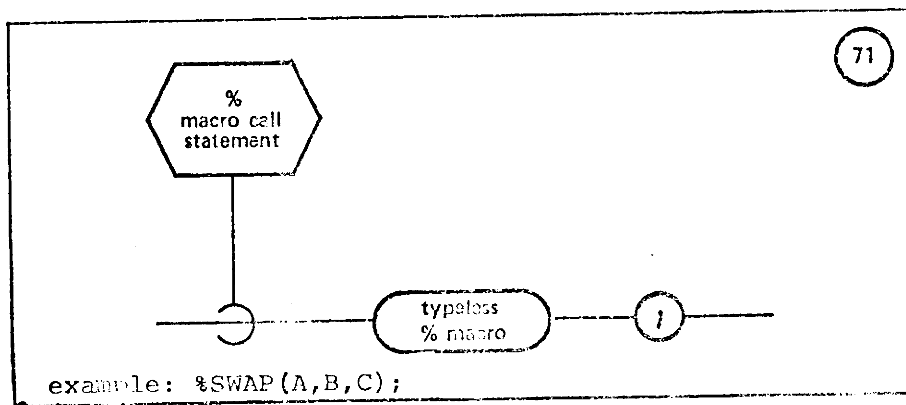
3. <struct %macro> is a reference to a %-macro which returns a value of a structure data type.

NTERMETRICS INCORPORATED • 701 CONCORD AVENUE • CAMBRIDGE, MASSACHUSETTS 02138 • (617) 661-1840

11.2.4 *The %-Macro Call Statement*

The invocation of a typeless %-macro is performed by a <%-macro call statement>.

SYNTAX



SEMANTIC RULES:

1. The <%-macro call statement> invokes execution of the typeless %-macro being referenced.

2. The effect of this statement is dependent upon the details of the %-macro being referenced.

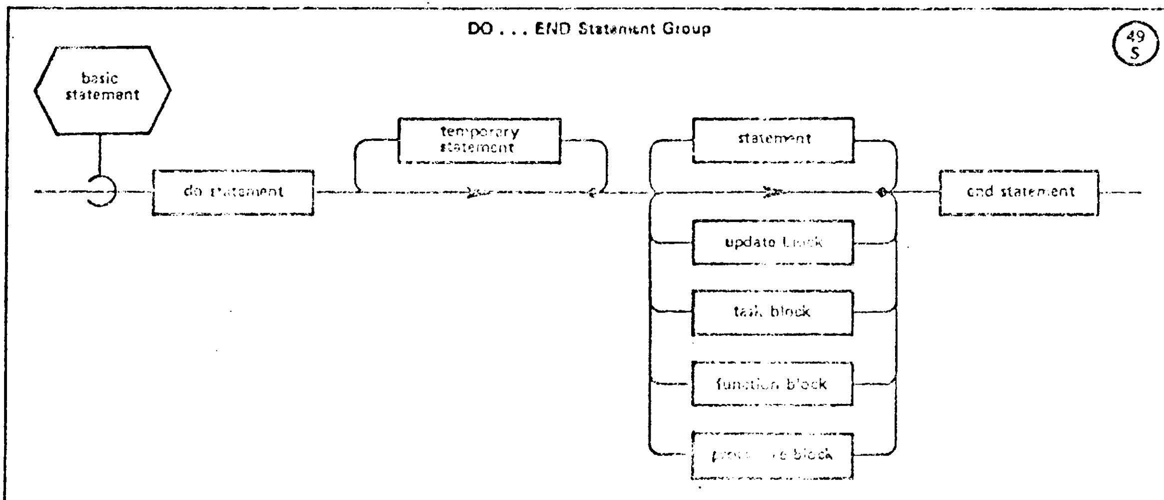
11.3 Temporary Variables

The extension of HAL/S data concepts to include a TEMPORARY variable form for use within DO groups is defined within the systems language facilities. The object of incorporating the TEMPORARY variable is to increase the optimization and efficiency of the object code produced by the compiler. Depending upon the details of the object machine, a temporary variable might be stored in a CPU register or a high speed, scratchpad memory location rather than in the slower main storage. Coding efficiency may also be achieved with temporary variables because the instructions needed to access register or scratchpad memory values are generally more compact. Since the existence of a temporary variable is confined to a DO group (from DO header statement to the END statement), these forms become highly localized control variables.

11.3.1 Regular TEMPORARY Variables

Regular TEMPORARY variables are declared in TEMPORARY statements following the DO statement which begins a DO ... END statement group and preceding the first executable statement of the DO ... END statement group. The following diagram is a systems language extension of the DO...END statement group in Section 7.6.

SYNTAX:

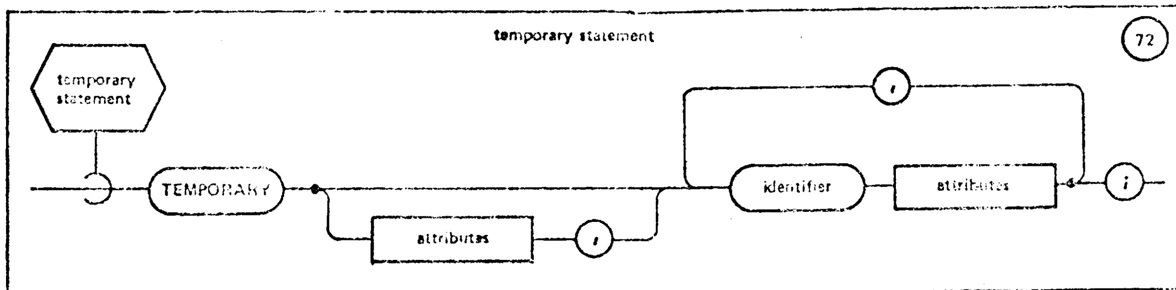


SEMANTIC RULE:

1. The TEMPORARY declaration may be included as part of any DO group except a DO CASE group. Use of TEMPORARY variables within nested DO groups of a DO CASE is allowed.

The TEMPORARY statement is a special purpose data declaration used to create TEMPORARY variables for general use within the DO group syntax as described above. Its form compares very closely to that of the DECLARE statement in Section 4.4.

SYNTAX:



SEMANTIC RULES:

1. In the <temporary statement>, <attributes> may define the <identifiers> to be of any data type except EVENT.

2. <attributes> may only specify type, precision and arrayness.

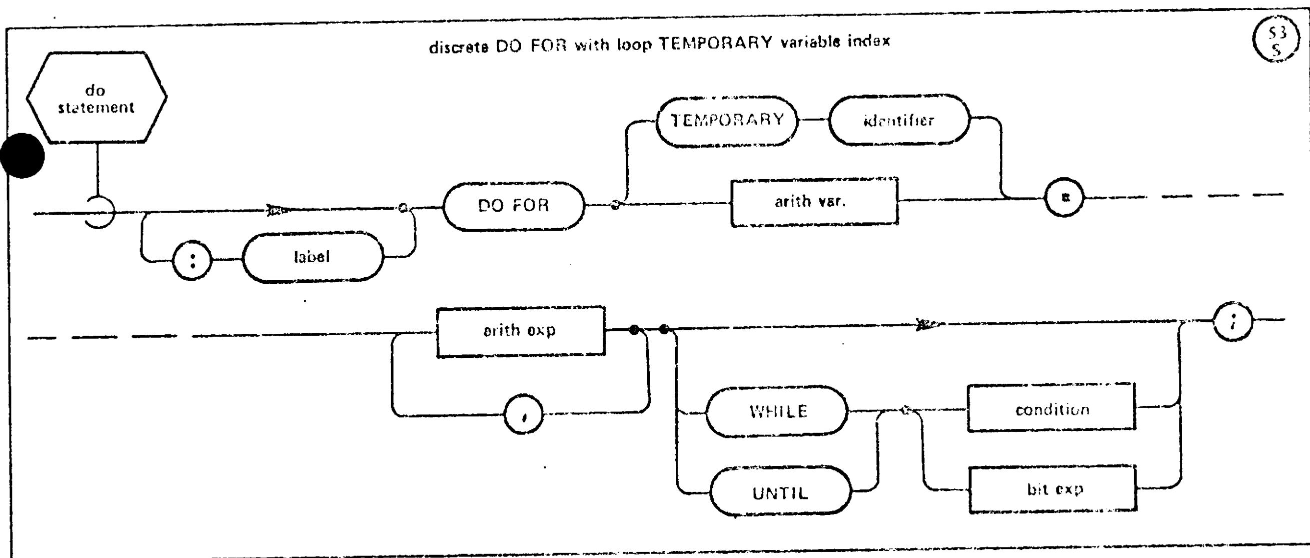
3. No minor attribute is legal.

4. The name of <identifier> may not duplicate the name of another <identifier> in the same name scope (procedure, function, or other block) or of another temporary in the same DO ... END group.

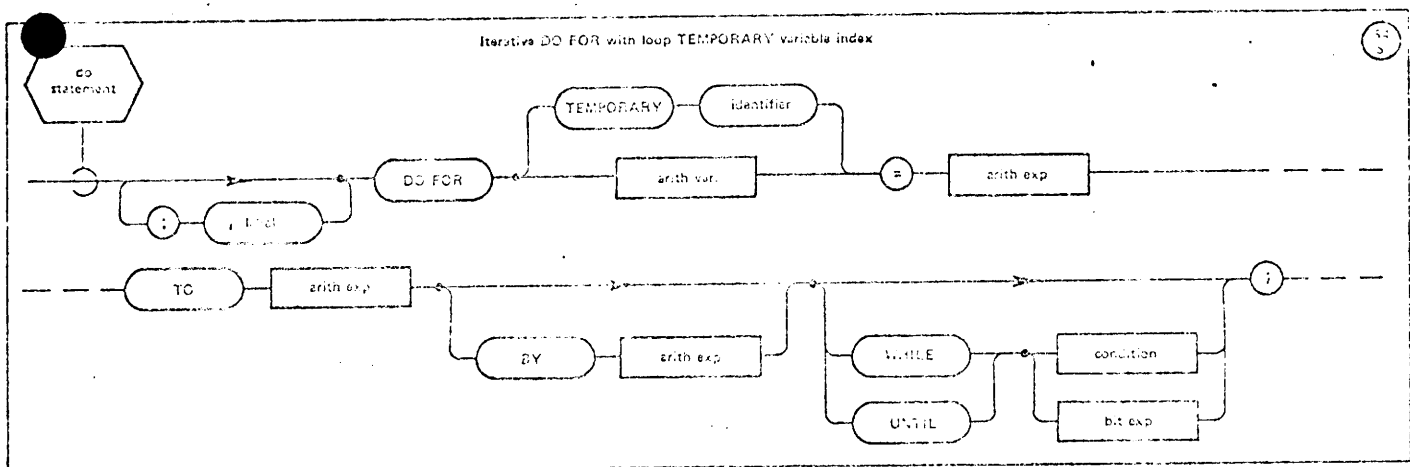
11.3.2 *Loop TEMPORARY Variables*

The Loop TEMPORARY variable form is used in the context of the DO FOR group and is declared by its specification in a DO FOR statement. The following two syntax diagrams are modifications of the discrete DO FOR and the iterative DO FOR syntax diagrams.

SYNTAX:



SYNTAX:



ERMETRICS INCORPORATED • 701 CONCORD AVENUE • CAMBRIDGE, MASSACHUSETTS 02138 • (617) 661-1840

SEMANTIC RULES:

1. All the semantic rules for DO FOR statements which are given in Section 7.6.4 and 7.6.5 apply as well to the corresponding Loop TEMPORARY forms. Additional rules for Loop TEMPORARY variables are given below.

2. The Loop TEMPORARY variable is defined in the DO FOR statement; a loop TEMPORARY variable is always a single precision INTEGER variable.

3. The scope of the Loop TEMPORARY is the DO FOR group of the DO FOR statement which defines the variable.

4. The <identifier> name used for the loop TEMPORARY may not duplicate the name of another <identifier> in the same name scope, nor may it duplicate the name of another TEMPORARY variable in the same DO ... END group.

11.4 The NAME Facility

This section gives a definitive description of the HAL/S NAME facility. This facility is designed to fill the system programmer's need for a "pointer" construct. Its basic entity is the NAME identifier: a NAME identifier "points to" an ordinary HAL/S identifier of like attributes. The "value" of the NAME identifier is thus the location of the identifier pointed to. (An "ordinary" identifier is a HAL/S identifier without the NAME attribute).

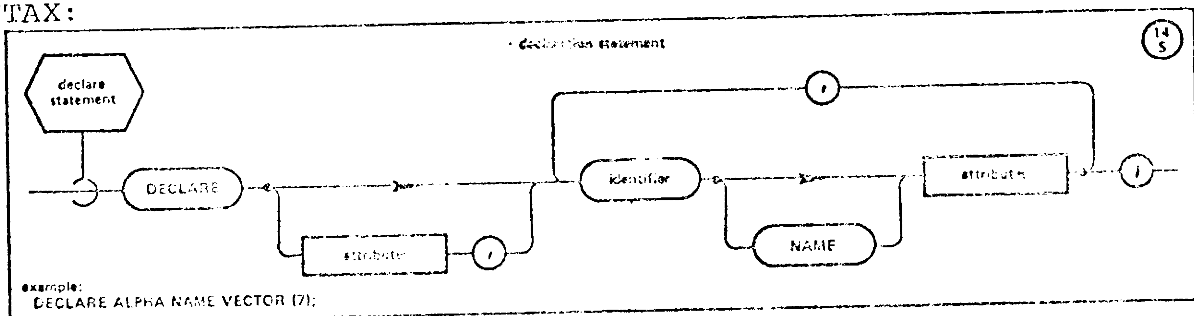
11.4.1 *Identifiers with the NAME Attribute*

Identifiers declared with the NAME attribute become NAME identifiers. NAME identifiers may be declared with the following data types:

INTEGER
SCALAR
VECTOR
MATRIX
BIT
BOOLEAN
CHARACTER
EVENT
STRUCTURE
PROGRAM
TASK

The following diagram is an extension of the DECLARE statement syntax diagram in Section 4.4. The modification shows how the keyword NAME is used in such a declaration to state the NAME attribute.

SYNTAX:



GENERAL SEMANTIC RULES:

1. The following <attribute>s apply to the NAME variable itself and bear no relationship to the ordinary identifier which is pointed to at any given time during execution:

   - The <initialization> attribute (if supplied) refers to the initial pointer value of the NAME variable itself.

   - STATIC/AUTOMATIC refer to the mode of initialization of the NAME variable itself on entry into a HAL/S block.

   - DENSE/ALIGNED apply to the actual NAME variable when it is defined by inclusion in a structure template.

   All other legal attributes describe the characteristics of the ordinary variables to which the NAME variable may point. Except as noted below, these other attributes must always match the corresponding attributes of the ordinary variables to which the NAME variable points; compilation errors will ensue if this is not the case.

2. The ACCESS attribute is illegal for NAME variables; its absence does not prevent NAME identifiers from pointing to ordinary identifiers with the ACCESS attributes and matching is not required in this case.

3. There must still be consistency between declared type, attributes, and factored attributes just as is the case for ordinary identifiers as described in Chapter 4 of this Specification.

examples

```
DECLARE VECTOR(3) DOUBLE LOCK(2), X, Y NAME;
DECLARE P NAME TASK;
```

Y may point to X
P points to any task block

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

SEMANTIC RULES (Data NAME Identifiers):

1. Arrayness Specification - in general the arrayness specification of a NAME identifier must match that of the ordinary identifiers pointed to, in both number and size of dimensions.

2. Structure Copy Specification - in general the number of copies of a NAME identifier of a structure type must match that of the ordinary identifiers pointed to.

3. The use of the "*" array specification or structure copies specification is excluded from declarations of NAME formal parameters.

4. Structure Type - if a NAME identifier is a structure type it may only point to ordinary identifiers of structure type with the same structure template.

examples of data NAME variables

```
DECLARE X ARRAY(3) SCALAR,
        Y ARRAY(4),
        Z NAME ARRAY(4) SCALAR;
DECLARE P EVENT;
DECLARE EVENT LATCHED, V, VV NAME;
```

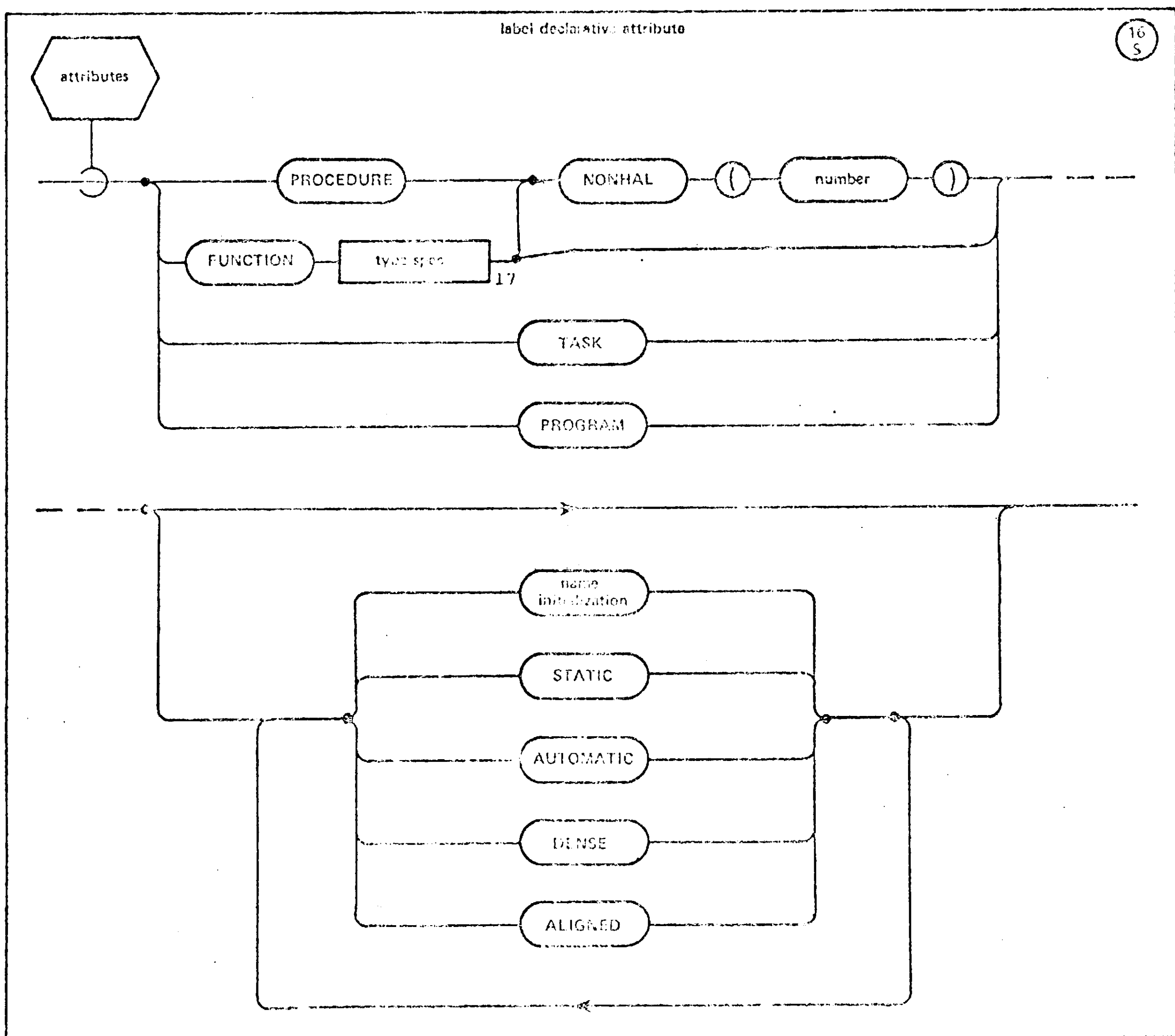
Z may point to Y but not X

5. For any unarrayed character string name variable, the "*" form of maximum length specification may be used. This is an extension of the use of the "*" notation which applies now in general to character name variables as well as to formal parameters.

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

The Label Declarative Attributes available for use in declaring NAME identifiers which point to HAL/S block forms have been extended to include PROGRAM and TASK keywords. The following syntax diagram is a modification of the Label Declarative Attributes diagram in Section 4.6.

SYNTAX:



SEMANTIC RULES (Label NAME Identifiers):

1. <initialization>, STATIC or AUTOMATIC, DENSE or ALIGNED may only be applied to the <label declarative attributes> of identifiers with the NAME attribute. They are properties of the NAME and not of the identifiers pointed to.

6

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

4. The following rules apply to NAME <identifiers> of the PROGRAM and TASK types:

   - The NAME <identifier> of a PROGRAM or TASK type always points to a PROGRAM or TASK block, respectively. A corollary of this rule is that <process event>s are never referenced by NAME identifiers of the PROGRAM or TASK types.

   - The only form of PROGRAM label declarations allowed are those with the NAME attribute.

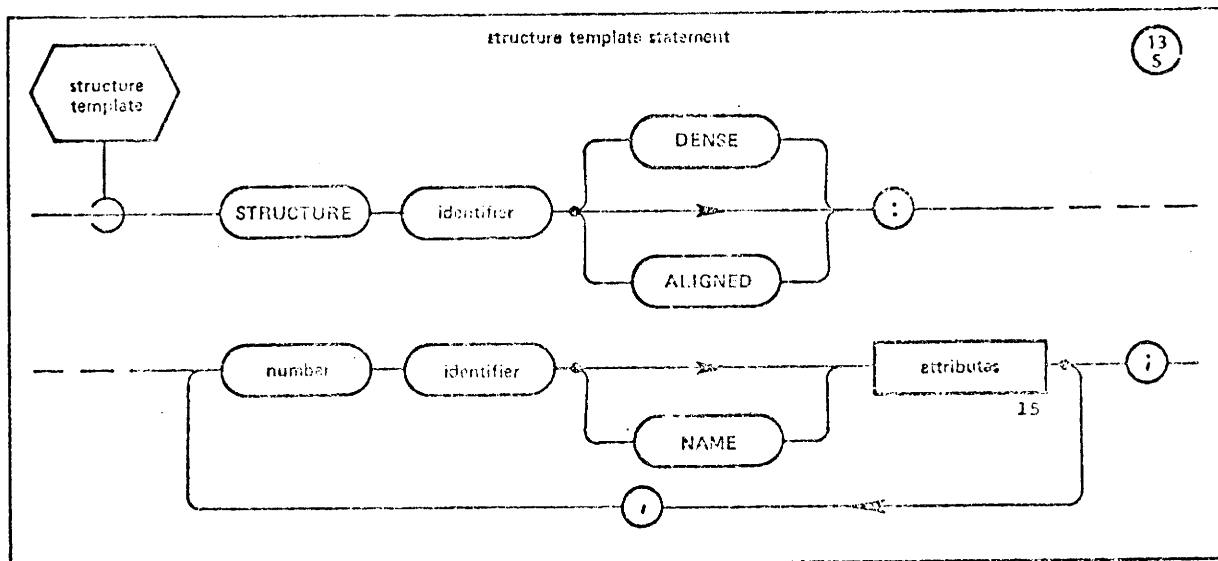
   - The program NAME <identifier> must always point to an external PROGRAM block name; therefore a block template is required for each PROGRAM which may be referenced by a NAME value.

126

11.4.2 The NAME Attribute in Structure Templates

The NAME attribute may appear on any structure terminal of a structure template. The following syntax diagram shows how the keyword NAME is used to state the NAME attribute. This diagram is a systems language extension of the Structure Template diagram.

SYNTAX:



In general, the rules governing the formation of the structure template remain unchanged (see Section 4.3).

GENERAL SEMANTIC RULES:

1. Restrictions on attributes discussed in Section 11.4.1 generally also apply to structure terminals with the NAME attribute.

2. No <initialization> may be applied to terminals; neither may the attributes STATIC/AUTOMATIC appear.

3. NAME identifiers of any type (including program, task, procedure & function) may appear as structure terminals. Note that the NAME of an EVENT may appear in a structure even though the EVENT itself may not.

4. The REMOTE attribute may be applied to a structure terminal with the NAME attribute unless it is of EVENT type.

SEMANTIC RULES: Nested Structure Template References

1. Nested structure template references are special instances of structure terminals. The manner of their incorporation into structure template definitions is as described in Section 4.3 via the <type spec>.

2. Such references are permitted to use the NAME attribute. If the NAME attribute is present, the following points are to be noted:

   - Specification of multiple copies is still not permitted.

   - The reference may be to the structure template being defined (and of which the reference is a part). The implications of this are discussed later.

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

examples of structure NAME identifiers:

```
STRUCTURE A:
    1 X NAME PROGRAM,
    1 Y SCALAR,
    1 Z NAME SCALAR,
    1 ALPHA NAME A-STRUCTURE;

DECLARE P A-STRUCTURE;
DECLARE PP NAME A-STRUCTURE;
```

P.Z is a NAME identifier which may point to P.Y

PP is a NAME identifier which may point to P

PP.Z is a NAME identifier which may point to P.Z which is itself a NAME identifier pointing somewhere. This is an instance of double indirection.

P.ALPHA is a NAME identifier of A-structure type. The consequences of this are discussed later.

11.4.3 *Declarations of Temporaries*

No identifier declared in a TEMPORARY statement may possess the NAME attribute. No such identifier of structure type may have a template which contains one or more structure terminals bearing the NAME attribute.

[INT]ERMETRICS INCORPORATED • 701 CONCORD AVENUE • CAMBRIDGE, MASSACHUSETTS 02138 • (617) 661-1840

11.4.4 The 'Dereferenced' Use of Simple NAME Identifiers

Simple NAME identifiers are those which are not parts of structure templates.

If a simple NAME identifier appears in a HAL/S expression as if it were an ordinary identifer, then the value used in computing the expression is the value of the ordinary identifier pointed to by the NAME identifier. Similarly, if a simple NAME identifier appears on the left-hand side of an assignment, as if it were an ordinary identifier, then the value of the right-hand side is assigned to the ordinary identifier pointed to by the NAME identifier. These are examples of the 'dereferenced' use of NAME identifiers.

Whenever a NAME identifier appears in a HAL/S construct as if it were an ordinary identifier, the dereferencing process (to find the ordinary identifier pointed to) is implicitly being specified. Specifically this still takes place when a subscripted NAME identifier appears as if it were an ordinary identifier. Here the dereferencing takes place first, and then the subscripting is applied to the ordinary identifier pointed to:

examples of dereferenced NAME variables

```
DECLARE VECTOR(3), X, Y NAME;
DECLARE P NAME TASK;
Q: TASK;
    :
    CLOSE;
    :
```

if Y points to X, and P to Q then -

TERMINATE P;	Means terminate Q.
Y = Y*Y;	Puts the cross product of X with X in X.
$Y_1 = Y_3$;	Puts the third element of X into the first element.

E

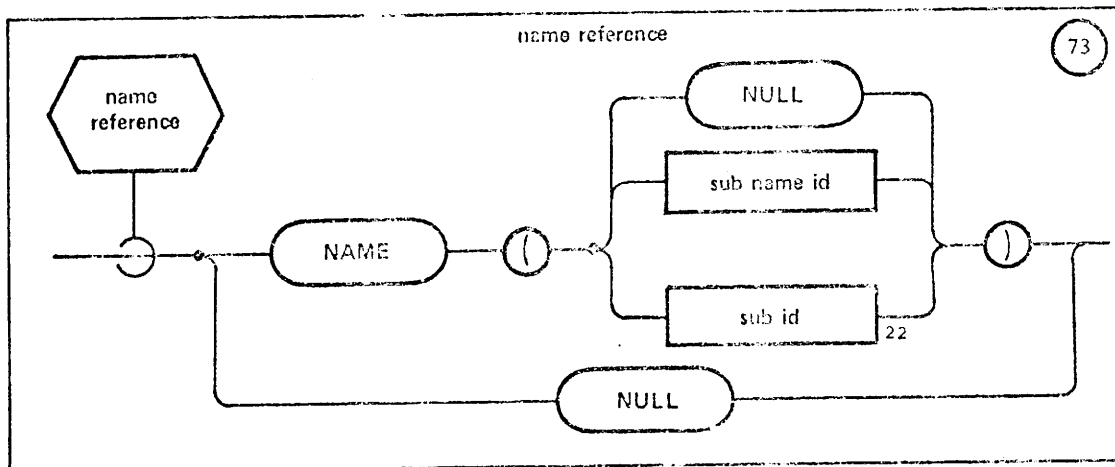
A special construct to be described in Sections 11.4.5 and 11.4.6 is required to reference or change the value of a NAME identifier (as opposed to referencing or changing the value to which it points).

11.4.5 *Referencing NAME Values*

The value of a NAME identifier is referenced or changed by using the NAME pseudo-function. This pseudo-function must also be used in order to gain access to the locations of ordinary HAL/S identifiers. The locations or values so indicated will be called NAME values. The necessity also arises for specifying Null NAME values.

The following syntax diagram shows both the NAME pseudo-function construct as used for referencing NAME values, and the construct for specifying Null NAME values.

SYNTAX:



SEMANTIC RULES:

1. <sub id> is any ordinary identifier, except an input parameter, a minor structure, an identifier declared with CONSTANT initialization, or an ACCESS-controlled identifier to which assignment access is "denied" or not asked for. <sub name id> is any NAME identifier.

2. Either of the above forms may possibly be modified by subscripting legal for its type and organization. Note, however, the following specific exceptions:

- No component subscripting is allowed for bit and character types.

- If component subscripting is present, <sub id> or <sub name id> must be subscripted so as to yield a single (unarrayed) element.

- If no component subscripting is present, but array subscripting is, then all arrayness must be subscripted away.

> example:
>
> DECLARE V NAME ARRAY(3) VECTOR;
>
> :
>
> NAME($V_{*:1}$) is illegal since it violates the second exception of semantic rule 3 above.

3. Any <sub id> must have the ALIGNED attribute.

4. NAME <identifier>s may not be declared with the ACCESS attribute (see Section 11.4.1, rule 2). This does not, however, imply that the NAME facility is independent of the ACCESS control: NAME references to <sub id>s with ACCESS control will compile without error only if implementation dependent ACCESS requirements for <sub id> are satisfied.

5. If <sub id> is unsubscripted, the construct delivers the location of the ordinary identifier specified. If it is subscripted, the construct delivers the location of the <u>part</u> of the specified identifier as determined by the <u>form</u> of the subscript. Subscripting can change the type and dimensions of <sub id> for matching purposes.

6. If <sub name id> is unsubscripted, the construct delivers the value of the NAME identifier specified. If it is subscripted, the value of the NAME identifier is taken to be the location of an ordinary identifier of compatible attributes, and the subscripting accordingly modifies the location delivered by the construct.

7. The two equivalent forms NULL and NAME(NULL) specify null NAME values.

examples:

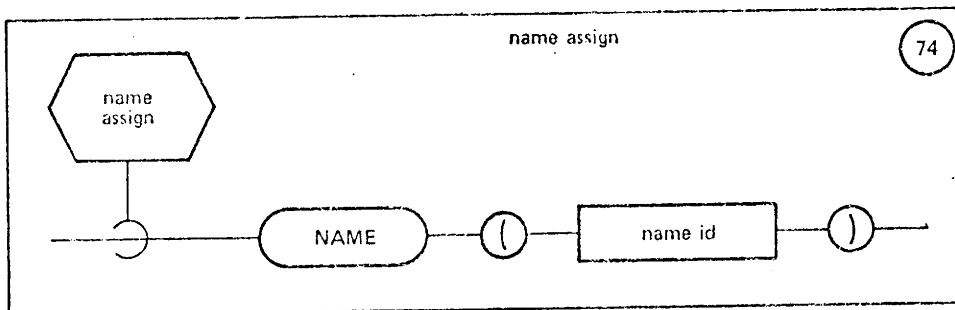
```
DECLARE X  SCALAR,
        V  VECTOR(3),
        NX NAME SCALAR,
        NV NAME VECTOR(3);
   .
   .
   .
```

NAME(X)	yields the location of X.
NAME(NX)	yields the value of NX (i.e. the location pointed to by NX).
NAME(V_2)	yields the location of the second element of V.
NAME(NV_3)	yields the location of the third element of the vector pointed to by NV.

11.4.6 Changing NAME Values

The value of a NAME identifier is changed by using the NAME pseudo-function in an assignment context. The following syntax diagram shows the NAME pseudo-function used for assigning NAME values:

SYNTAX:



SEMANTIC RULE:

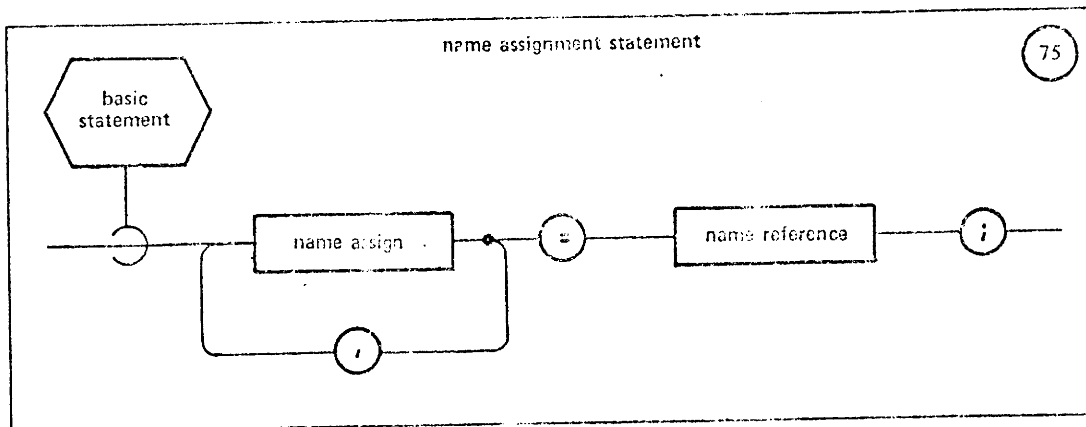
1. <name id> specifies any NAME identifier except an input parameter, whose NAME value is to be changed. <name id> may not be subscripted (except as noted in Section 11.4.29). | E

> example:
>
> DECLARE X NAME MATRIX;
>
> NAME(X) in assignment context specifies that a new value is to be given to X.

11.4.7 NAME Assignment Statements

The NAME assignment statement is the construct by which NAME values are assigned into NAME identifiers.

SYNTAX:



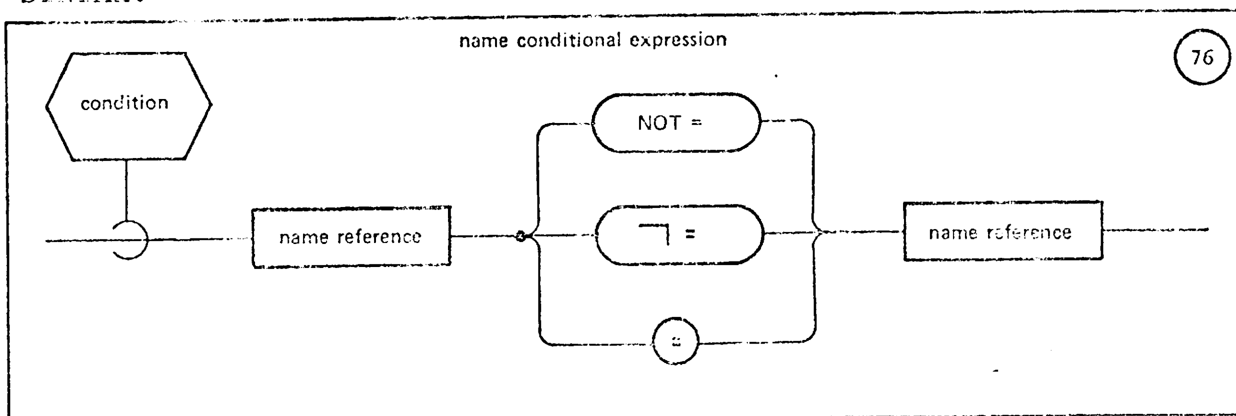
SEMANTIC RULES:

1. The <name reference> and <name assign>s must possess arguments whose attributes are compatible in the sense described in Section 11.4.1.

11.4.8 NAME Value Comparisons

The values of two <name reference>s may be compared to one another.

SYNTAX:



SEMANTIC RULES:

1. This <comparison> may be used in any syntax where other forms of <comparison> may be used, for example in a <conditional operand> or as the <condition> controlling a DO WHILE.

2. Both <name reference>s must possess arguments whose <attributes> are compatible in the sense described in Section 11.4.1.

examples:

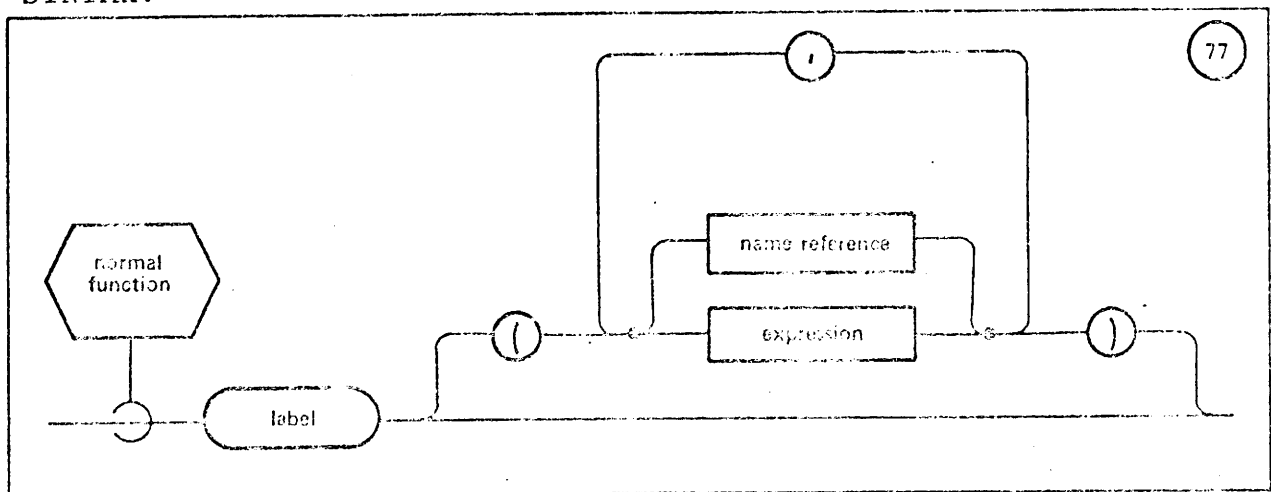
```
DECLARE X SCALAR,
        NX NAME SCALAR:
  ⋮
NAME(NX)= NAME(X) ;        value of NX is location
  ⋮                        of X (NX points to X).
IF NAME(NX)= NULL THEN RETURN;

                           if NX points nowhere,
                           then return.
```

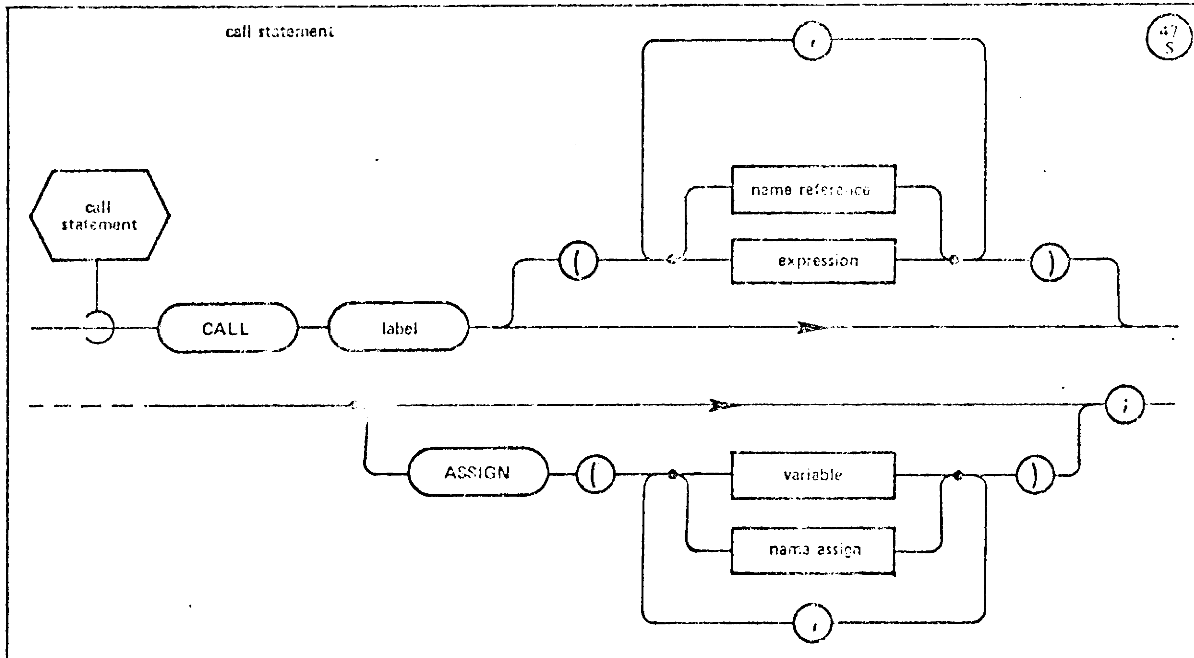
11.4.9 *Argument Passage Considerations*

NAME values may be passed into procedures and functions provided that the corresponding formal parameters of the blocks in question have the NAME attribute. The following two syntax diagrams are systems language extensions of the earlier <normal function> and <call statement> syntax diagrams.

SYNTAX:



SYNTAX:



SEMANTIC RULES:

1. The formal parameters corresponding to <name reference> or <name assign> arguments of these block invocations must possess the NAME attribute.

2. The attributes of <name reference> and <name assign> arguments supplied in the <normal function> reference or <call statement> must be compatible with those of the formal parameters in the same sense as described in Section 11.4.1.

3. If the argument of the procedure or function invocation is not a <name reference> then the corresponding formal parameter must <u>not</u> have the NAME attribute.

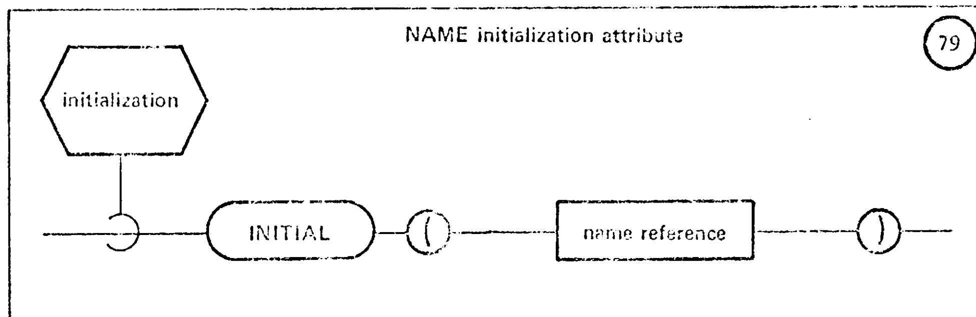
examples:

```
DECLARE X1 SCALAR,
        X2 NAME SCALAR;
:
P:  PROCEDURE(A, B) ASSIGN(C, D);
    DECLARE SCALAR, A NAME,
                    B,
                    C NAME,
                    D;
    NAME(C) = NAME(A);
    NAME(C) = NAME(B);          illegal - B is an
    :                           input parameter
CLOSE;
:
NAME(X2) = NAME(X1);
CALL P (NAME(X1), X1) ASSIGN(NAME(X2), X1);
```

11.4.10 Initialization

NAME identifiers may be declared with initialization to point to some particular identifier. The form of NAME initialization is as follows:

SYNTAX:



SEMANTIC RULES:

1. The argument of the <name reference> must be a previously declared <sub name id> or <sub id> with <attributes> compatible with the NAME identifier being declared.

3. Uninitialized NAME identifiers will have a NULL NAME value until the first NAME assignment.

4. The argument of a <name reference> may not itself possess the NAME attribute.

11.4.11 *Notes on NAME Data and Structures*

The previous sections have introduced the various syntactical forms and uses of the NAME attribute, <name assign>s, and <name reference>s. The use of these NAME facilities with structure data merits further explanation since the implications of the various legal combinations are not always immediately apparent. Therefore, the purpose of this section is to continue further discussion of various aspects of NAME and structure usage by providing several examples.

STRUCTURE TERMINAL REFERENCES

Consider the structure template and structure data declaration below:

```
STRUCTURE A:
   1 C SCALAR,
   1 B NAME A-STRUCTURE;

DECLARE A-STRUCTURE, Z1, Z2, Z3;
```

Z1.B is a NAME identifier of A-structure type: its NAME value may be set to point to Z2 by the assignment

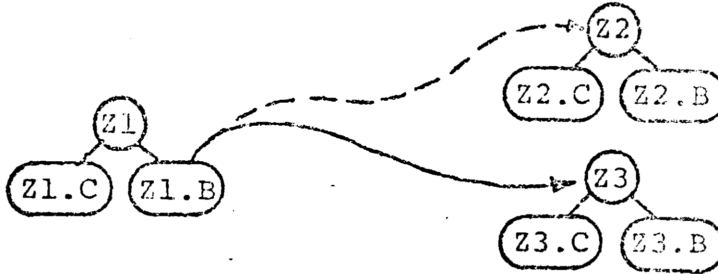
```
NAME(Z1.B) = NAME(Z2);
```

If this is done then it is legal to specify Z1.B.C as a qualified structure terminal name. The appearance of B in the qualified name causes an implicit dereferencing process to occur such that if Z1.B.C is used in a dereferencing context, the ordinary structure terminal actually referenced is Z2.C. If the NAME value of Z1.B is changed by

```
NAME(Z1.B) = NAME(Z3);
```

then the appearance of Z1.B.C in a dereferencing context causes Z3.C to be referenced.

Pictorially:

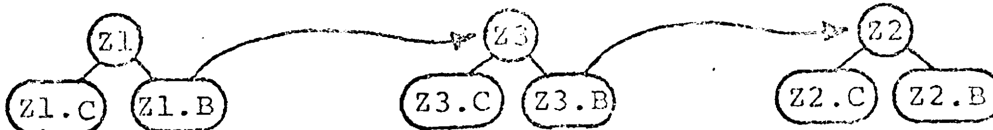


Now Z1.B.B is itself in turn a NAME identifier of A-structure type, so that if the NAME assignment

```
NAME(Z1.B.B) = NAME(Z2);
```

is executed, then Z2.C may be referenced by using the qualified name Z1.B.B.C in a dereferencing context.

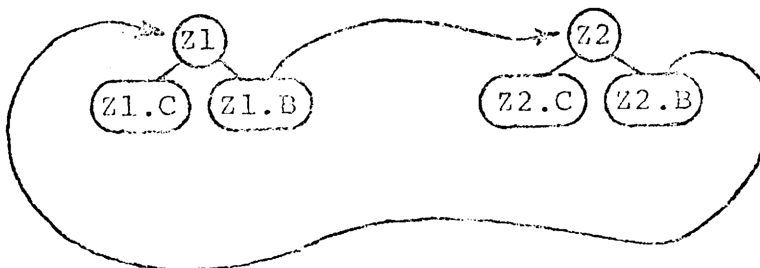
Pictorially:



Clearly this implicit dereferencing in qualified names can extend chains of reference indefinitely. A particular consequence is the creation of a closed circular chain. If the following NAME assignment statements:

```
NAME(Z1.B) = NAME(Z2);
NAME(Z1.B.B) = NAME(Z1);
```

are executed, then pictorially the following closed loop is set up:



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

Care must clearly be taken when using this implicit multiple dereferencing, so that all links in the chain have previously been set up.

IMPLICATIONS OF SUBSCRIPTING STRUCTURE TERMINALS

Using the same A-structure template as before, the following declarations are legal:

DECLARE A-STRUCTURE(3), Y1,Y2,Y3,Y4;

One or more copies of Y1.C may be referred to by subscripting, for example:

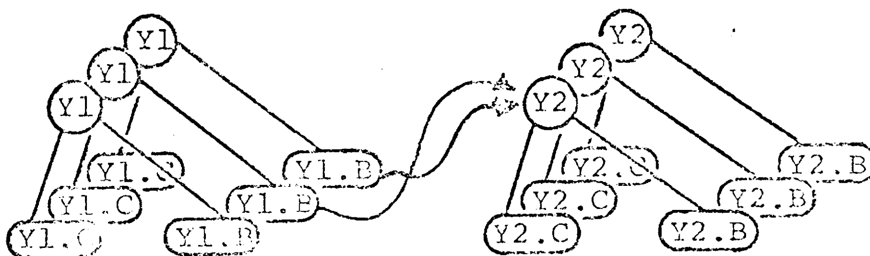
$Y1.C_{2\ AT\ 2};$ (optional semicolon for clarity)

Note that now Y1.B is a NAME identifier of A-structure type with 3 copies. One or more copies of it may therefore be assigned a NAME-value at one time. For example:

$NAME(Y1.B_{2\ AT\ 2}) = NAME(Y2_{2\ AT\ 1});$

In this assignment, the left hand side has arrayness: two copies of the Y1 structure. As a result, two values will be defined by the statement. However, the right hand side has no arrayness, because the object pointed to is $Y2_{2\ AT\ 1}$. This is a two copy section of the structure Y2, with a unique starting location.

Pictorially:



Notice that in the above NAME assignment a subscripted <name id> appears as argument of the left-hand side NAME pseudo-function. Subscripts so appearing are legal only if they can have the interpretation exemplified. The subscripting employed must also be unarrayed, as was mentioned earlier.

Further indirection may then be set up: thus for example:

$$NAME(Y1.B.B_2) = NAME(Y3_1);$$

Here the subscript 2 on the left-hand argument refers to copies of Y1 (this can be its only interpretation). Hence, by virtue of the fact that $Y1.B_2$ has previously been set up to point to $Y2_1$, this assignment causes $Y2.B_1$ to point to $Y3_1$.

Arrayness will appear on both sides of a NAME Assignment Statement only when the assigned reference terminals of both sides possess the NAME attribute within structure variables with copies.
Consider the template:

```
STRUCTURE AA:
      1 C NAME SCALAR,
      1 D NAME VECTOR;
```

And the declaration:

```
DECLARE AA-STRUCTURE(3), YY1,YY2;
```

If the terminal element YY2.D is assigned to the terminal element YY1.D, the NAME assignment is arrayed since both sides contain three copies.

Thus:

```
NAME(YY1.D) = NAME(YY2.D);
```

causes the name values of YY2.D found in the three copies of YY2 to be transfered to the corresponding name variables in YY1.D. All the usual rules governing arrayed assignments apply in this case.

MANIPULATING STRUCTURES CONTAINING NAME TERMINALS

Since the NAME attribute may be applied to structure terminals, a definition of operations performed on such NAME terminals in ordinary structure assignments, comparisons and I/O operations is required. The following general rules are applicable:

- For assignment statements and comparisons involving structure data with NAME terminals, operations are performed on NAME values without any dereferencing.

examples:

```
STRUCTURE IOTA:
  1 LAMBDA NAME VECTOR,
  1 KAPPA SCALAR;
DECLARE ALPHA IOTA-STRUCTURE(10);
DECLARE BETA IOTA-STRUCTURE;
   :
ALPHA  = BETA;
     4
```

As a part of this assignment, the vector identifier (or NULL) pointed to by BETA.LAMBDA becomes the vector identifier pointed to by $ALPHA.LAMBDA_4$ as if a <name assignment statement> had been used.

```
IF ALPHA  = BETA THEN CALL QUE_UPDATE;
        5
```

In this IF statement, the structure comparison between the two variables ($ALPHA_5$ and BETA) is performed terminal by terminal as usual. For the NAME terminal LAMBDA of each structure operand, the effect is the same as if a <name comparison> had been used: Equality for the corresponding NAME terminals exists if they both point to the same ordinary identifier.

- For sequential I/O Operations, all NAME terminals are totally ignored. Name terminals can take part in FILE I/O. 13

examples:

```
STRUCTURE OMICRON:
  1 ALPHA SCALAR,
  1 BETA ARRAY(25) INTEGER SINGLE,
  1 GAMMA NAME MATRIX(10, 10);
STRUCTURE TAU:
  1 ALPHA SCALAR,
  1 BETA ARRAY(25) INTEGER SINGLE;
DECLARE X OMICRON-STRUCTURE;
DECLARE Y TAU-STRUCTURE;
  ⋮
READ(5) X;
```

The structure variable X is an OMICRON-STRUCTURE, whose template includes the NAME of a 10 x 10 matrix (GAMMA). Only the ordinary terminals are transferred from Channel 5 by this READ operation --- the value of X.ALPHA and the 25 values required for X.BETA. The NAME terminal X.GAMMA is ignored.

```
READ(5) Y;
```

The structure variable Y is a TAU-STRUCTURE, whose template omits the NAME terminal GAMMA found in the OMICRON-STRUCTURE, but is otherwise identical. The effect of this READ statement is the same as the previous statement as far as Channel 5 is concerned --- one value is read for Y.ALPHA and 25 values are read for Y.BETA.

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

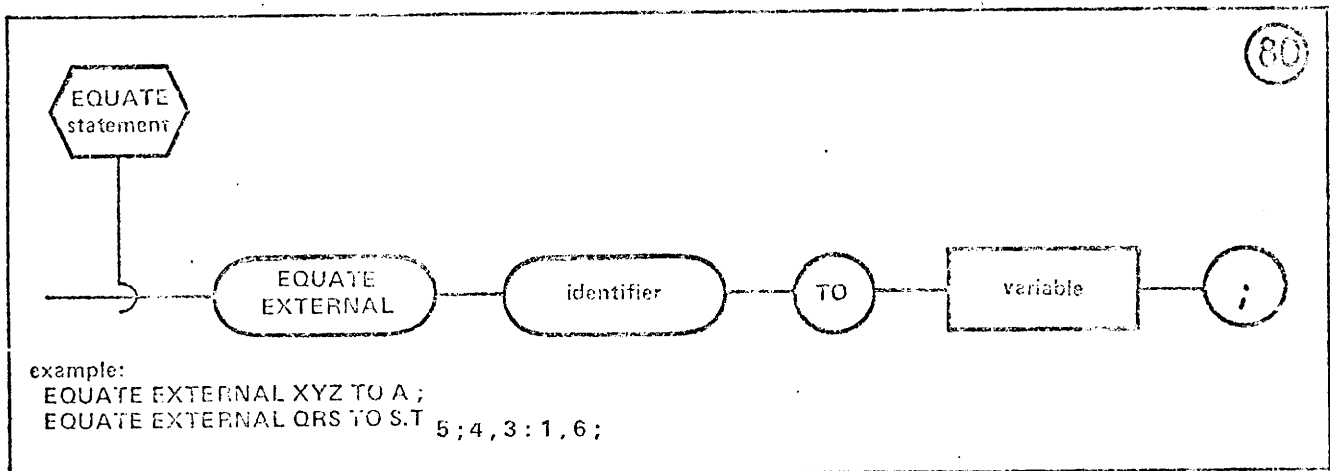
11.5 The EQUATE Facility

This section describes the HAL/S EQUATE facility which allows a system programmer to assign an external name to an element of a HAL/S data area.

Reference to HAL/S data items by HAL/S code is achieved by use of HAL/S identifiers. When such references occur across compilation unit boundaries, the Block Template provides the information necessary to generate the reference properly. If, however, the unit making reference to a HAL/S data item is not a HAL/S code block, the Block Template facility is unavailable. It is under these latter circumstances that the HAL/S EQUATE facility may be used to make the location of a HAL/S data item available to an external, non-HAL/S code block.

11.5.1 The EQUATE Statement

SYNTAX:



SEMANTIC RULES:

1. The EQUATE statement causes <identifier> to become an externally recognizable label of the HAL/S <variable>. The manner in which this is done is implementation dependent. The EQUATE statement has the effect of raising the name of <identifier> to a global external level such that it is known to whatever binders, loaders, link-editors, etc., are used by an implementation.

2. The number of characters of the <identifier> which participate in the external name created as implementation dependent.

3. The EQUATE statement does not constitute a HAL/S declaration. This implies that <identifier> may appear in a declare statement and be used in any manner consistent with that declaration. In the absence of such a declaration, <identifier> is not declared and may not be used anywhere else in the HAL/S code.

4. Duplication of <identifier>s among multiple EQUATE statements within a single compilation unit is subject to implementation dependent rules.

5. <variable> may be any HAL/S data item previously declared in the innermost scope containing the EQUATE statement.

6. If <variable> is subscripted, all subscripts must be computable at compile time.

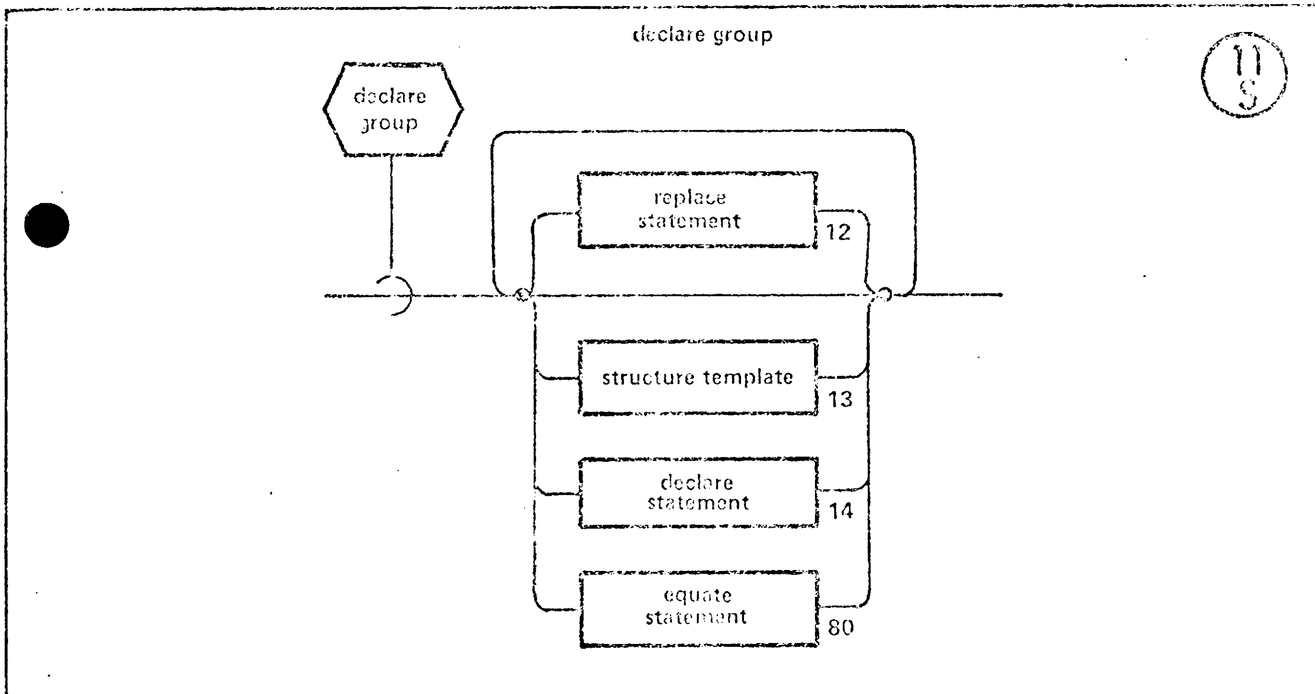
7. The external name created by the EQUATE statement will be associated with the memory location of the first (or only) element specified by <variable>.

8. Attempts to associate external names with HAL/S data items which are not located at intergrally addressable memory locations or discontiguous memory locations are subject to implementation restrictions.

11.5.2 EQUATE Statement Placement

The following diagram is a system language extension of the Declare Group syntax diagram in Section 4.1. The modification shows how the EQUATE statement fits into the declaration structure of HAL/S.

SYNTAX:



APPENDICES

A. SYNTAX DIAGRAM SUMMARIES

A.1 SYNTAX PRIMITIVE REFERENCES

The syntax diagrams in this Specification are numbered sequentially. The CONTENTS of the Specification state which diagrams are in each Section.

The following table shows where the HAL/S syntactical primitives (excluding reserved words and special characters) are referred to.

NOTES:

1. Primitives are listed in alphabetical order.

2. Numbers enclosed in [ ] denote indirect references to the primitive. Explanations are given in the accompanying Semantic Rules.

Syntactical Primitive	Diagram Number	Page
<arith var name>	[19]	[5-5]
	20	5-5
<argument>	12.1	4-6
<bit literal>	19	5-5
	20	5-5
<char literal>	[18]	[4-23]
	29	6-11
<char var name>	[18]	[4-20]
	19	5-5
	20	5-5
<event var name>	19	5-5
	20	5-5
<identifier>	8	3-15
	9	3-17
	12	4-4
	13	4-9
	14	4-12
	15	4-13
	14s	11-16
	13s	11-22
<label>	2	3-4
	3	3-6
	4	3-8
	5	3-10
	6	3-11
	10	3-19
	[18]	[4-23]
	38	6-23
	45	7-3
	46	7-5
	47	7-9

Syntactical Primitive	Diagram Number	Page
<label> (continued)	48	7-12
	50	7-15
	51	7-16
	52	7-17
	53	7-19
	54	7-21
	55	7-23
	56	7-24
	57	8-4
	58	8-9
	59	8-11
	60	8-12
	61	8-14
	62	8-15
	63	9-3
	64	9-7
	65	10-3
	66	10-6
	68	10-10
	53s	11-14
	54s	11-14
	77	11-31
	47s	11-32
<number>	13	4-9
	15	4-13
	16	4-18
	[18]	[4-23]
	25	6-6
	63	9-2
	64	9-3
	65	10-3
	66	10-6
	68	10-10
	16s	11-19
	13s	11-22
<process-event name>	27	6-8
	37	6-22
<template name>	17	4-19
<text>	12	4-4
	12.1	4-6

A.2 SYNTAX DIAGRAM CROSS REFERENCES

The following table shows where non-primitive syntactical terms are defined and referenced.

NOTES:

1. Terms are listed in alphabetical order.

2. <radix> is included even though it has no syntactical diagram, because for the purposes of the Specification it was not regarded as a primitive. Its definition is included in the Semantic Rules accompanying the syntax diagrams where it is referred to.

3. Note that an "s" suffix identifies a modified systems Language Diagram.

Syntactical Definition	Defined in			References
	Diagram	Section	Page	
<arith conversion>	39	6	6-27	25,25s
<arith exp>	24	6	6-3	15,17,18,22,23,25,26,32,39,51, 53,54,57,60,61,67,68,25s,54s
<arith operand>	25	6	6-6	24,25s
<arith var>	19	5	5-5	20,25,53,54,25s,54s
<array sub>	22	5	5-11	21
<arith inline>	25s	11	11-7	25
<arith % macro>	25s	11	11-7	25
<attributes>:				
data	15	4	4-13	
label	16	4	4-18	16s
name	16s	11	11-19	44,45
<basic statement>:				
assignment	46	7	7-5	
name	75	11	11-27	47s
CALL	47	7	7-9	47s
name	47s	11	11-32	
CANCEL	58	8	8-9	
DO...END	49	7	7-14	
EXIT	56	7	7-24	
FILE	68	10	10-10	
GO TO	56	7	7-24	
name assign	74	11	11-27	
null	56	7	7-24	75,76,77,47s
ON ERROR	63	9	9-3	

Syntactical Definition	Defined in			References
	Diagram	Section	Page	
READ	65	10	10-3	
READALL	65	10	10-3	
REPEAT	56	7	7-24	
RESET	62	8	8-15	
RETURN	48	7	7-12	
SCHEDULE	57	8	8-4	
SEND ERROR	64	9	9-7	
SET	62	8	8-15	
SIGNAL	62	8	8-15	
TERMINATE	59	8	8-11	
UPDATE PRIORITY	61	8	8-14	
WAIT	60	8	8-12	
WRITE	66	10	10-6	
<bit conversion>	40	6	6-31	27,27s
<bit exp>	26	6	6-7	23,27,33,41,45,52,53,54,27s,54s
<bit inline>	27s	11	11-8	27
<bit % macro>	27s	11	11-8	27
<bit operand>	27	6	6-8	26,27s
<bit pseudo-var>	42	6	6-35	20,27,27s
<bit var>	19	5	5-5	20,27,27s
<char conversion>	41	6	6-33	29,29s
<char exp>	28	6	6-10	23,29,34,30,29s
<char operand>	29	6	6-11	28,29s
<char var>	19	5	5-5	20,29,29s

Syntactical Definition	Defined in			References
	Diagram	Section	Page	
<char inline>	29s	11	11-9	29
<char % macro>	29s	11	11-9	29
<closing>	10	3	3-19	2,3,4,5,6,69
<comparison>:				31
arithmetic	32	6	6-15	
bit	33	6	6-17	
character	34	6	6-18	
structure	35	6	6-19	
<compilation>	1	3	3-2	
<component sub>	22	5	5-11	21
<compool block>	5	3	3-10	1
<compool header>	7	3	3-14	5,6
<compool template>	6	3	3-11	1
<condition>	30	6	6-13	31,45,52,53,54,54s
name	76	11	11-30	
<conditional operand>	31	6	6-14	30
<declare group>	11	4	4-3	2,3,4,5,6,69
<declare statement>	14	4	4-12	11,14s
name	14s	11	11-16	
<do statement>:				49,49s
CASE	51	7	7-16	
discrete FOR	53	7	7-19	
temporary var	53s	11	11-14	
iterative FOR	54	7	7-21	

Syntactical Definition	Defined in			References
	Diagram	Section	Page	
temporary var	54s	11	11-14	
simple	50	7	7-15	
UNTIL	52	7	7-16	
WHILE	52	7	7-17	
<end statement>	55	7	7-23	49,49s
<event exp>	36	6	6-21	37,57,60
<event operand>	37	6	6-22	36
<event var>	19	5	5-5	20,27,37,62
<expression>	23	6	6-2	18,38,39,40,41,42,46,47,48,66,68,70
<file exp>	68	10	10-10	68
<function block>	3	3	3-6	1,2,3,4,49,49s
<function header>	9	3	3-17	3,6
<function template>	6	3	3-11	1
<inline function>	69	11	11-2	
<initial list>	18	4	4-20	18
<initialization>	18	4	4-23	15
name	79	11	11-33	16s
<i/o control>	67	10	10-8	65,66
<name>	14s	11	11-16	
<name reference>	75	11	11-30	
<normal function>	38	6	6-23	25,27,29,77
name	77	11	11-31	
<precision>	43	6	6-38	1,2,3,4,49,49s
<procedure block>	3	3	3-6	3,6

Syntactical Definition	Defined in			References
	Diagram	Section	Page	
<procedure header>	8	3	3-15	3,6
<procedure template>	6	3	3-11	1
<program block>	2	3	3-4	1
<program header>	7	3	3-14	2
<% macro>	70	11	11-5	25s
typeless % macro	71	11	11-11	
<radix>	Note 2.	6		40,41
<replace statement>	12	4	4-4	11
parametric	12.1	4	4-6	
<statement>:				
basic	44	7	7-2	
IF	45	7	7-3	
temporary	72	11	11-13	
<structure exp>	29.1	6	6-12	
<structure sub>	22	5	5-11	21
<struct inline>	29.1s	11	11-10	29.1
<struct % macro>	29.1	11	11-10	29.1
<structure template>	13	4	4-9	11
name	13s	11	11-22	
<structure var>	19	5	5-5	20,23,35,29,15,29.1
<sub exp>	22	5	5-11	22
<sub name id>	73	11	11-26	
<subscript>	21	5	5-8	19,39,40,41,42
<task block>	3	3	3-6	2,49,49s

Syntactical Definition	Defined in			References
	Diagram	Section	Page	
<task header>	7	3	3-14	3
<type spec>	17	4	4-19	9,15,16,69
<update block>	4	3	3-8	2,3,49,69,49s
<update header>	7	3	3-14	4
<variable>	20	5	5-5	42,46,47,65,68
<temporary statement>	49s 72	11 11	11-12 11-13	53s,54s

A.3 SYNTAX DIAGRAM LISTING *

DIAGRAM #	TITLE	PAGE
1	unit of compilation	3-2
2	PROGRAM block	3-4
3	PROCEDURE, FUNCTION and TASK blocks	3-6
4	UPDATE block	3-8
5	COMPOOL block	3-10
6	block templates: PROGRAM, PROCEDURE, FUNCTION and COMPOOL templates	3-11
7	simple header statement	3-14
8	PROCEDURE header statement	3-15
9	FUNCTION header statement	3-17
10	Closing of block	3-19
11	declare group	4-3
12	REPLACE statement	4-4
13	structure template statement	4-9
13s	structure template statement/NAME attribute	11-22
14	declare statement	4-12
14s	declaration statement/NAME attribute	11-16
15	data declarative attributes	4-13
16	label declarative attributes	4-18
16s	label declarative attributes/PROGRAM-TASK	11-19
17	type specification	11-19
18	initialization specification	4-23
19	<var>: arithmetic, bit, character, structure, event variables	5-5
20	variable	5-5
21	subscript construct	5-8
22	component, array, and structure subscripts	5-11
23	expression	6-2
24	arithmetic expression	6-3
25	arithmetic operand	6-6
25s	arithmetic operand inline function block/ %-macros	11-7

*Note that an "s" suffix identifies a modified Systems Language diagram.

DIAGRAM #	TITLE	PAGE
26	bit expression	6-7
27	bit operand	6-8
27s	bit operand inline function block/ %-macros	11-8
28	character expression	6-10
29	character operand	6-11
29s	character operand inline function block/ %-macros	11-9
29.1	structure expression	6-12
29.1s	structure expression inline function block/ %-macros	11-10
30	conditional expression	6-13
31	conditional operand	6-14
32	arithmetic comparison	6-15
33	bit comparison	6-17
34	character comparison	6-18
35	structure comparison	6-19
36	event expression	6-21
37	event operand	6-22
38	normal function	6-23
39	arithmetic conversion function	6-27
40	bit conversion function	6-31
41	character conversion function	6-33
42	SUBBIT pseudo-variable	6-35
43	precision specifier	6-38
44	basic statement	7-2
45	IF statement	7-3
46	assignment statement	7-5
47	CALL statement	7-9
47s	CALL statement with NAME	11-32
48	RETURN statement	7-12
49	DO...END statement group	7-14
49s	DO...END statement group/temporary variable	11-12
50	simple DO statement	7-15
51	DO CASE statement	7-16
52	DO WHILE and UNTIL statements	7-17
53	discrete DO FOR statement	7-18
53s	discrete DO FOR statement/temporary variable	11-14
54	iterative DO FOR statement	7-21
54s	iterative DO FOR statement/temporary variable	11-14
55	END statement	7-23

DIAGRAM #	TITLE	PAGE
56	other basic statements: GO TO, "null", EXIT and REPEAT statements	7-24
57	SCHEDULE statement	8-4
58	CANCEL statement	8-9
59	TERMINATE statement	8-11
60	WAIT statement	8-12
61	UPDATE PRIORITY statement	8-14
62	SET, SIGNAL, and RESET statement	8-15
63	ON ERROR statement	9-3
64	SEND ERROR statement	9-7
65	READ and READALL statements	10-3
66	WRITE statement	10-6
67	i/o control function	10-8
68	FILE statements	10-10
69	inline function block	11-2
70	%-macro statement	11-5
71	%-macro call	11-11
72	temporary statement	11-13
73	NAME reference	11-26
74	NAME assign	11-29
75	NAME assignment statement	11-30
76	NAME conditional expression	11-30
77	normal function reference	11-31
79	NAME initialization attribute	11-33

B. HAL/S KEYWORDS

The following table of keywords excludes built-in functions and %-macro names.

ACCESS
AFTER
ALIGNED
AND
ARRAY
ASSIGN
AT
AUTOMATIC

BIN
BIT
BOOLEAN
BY

CALL
CANCEL
CASE
CAT
CHAR
CHARACTER
CLOSE
COLUMN
COMPOOL
CONSTANT

DEC
DECLARE
DENSE
DEPENDENT
DO
DOUBLE

ELSE
END
EQUATE
ERROR
EVENT
EVERY

EXCLUSIVE
EXIT
EXTERNAL

FALSE
FILE
FOR
FUNCTION

GO

HEX

IF
IGNORE
IN
INITIAL
INTEGER

LATCHED
LINE
LOCK

MATRIX

NAME
NONHAL
NOT
NULL

OCT
OFF
ON
OR

PAGE
PRIORITY
PROCEDURE
PROGRAM

READ
READALL
REENTRANT
REPEAT
REPLACE
RESET
RETURN
REMOTE

SCALAR
SCHEDULE
SEND
SET
SIGNAL
SINGLE
SKIP
STATIC
STRUCTURE
SUBBIT
SYSTEM

TAB
TASK
TEMPORARY
TERMINATE
THEN
TO
TRUE

UNTIL
UPDATE

VECTOR

WAIT
WHILE
WRITE

| 135

C. BUILT-IN FUNCTIONS

HAL/S typically supports the following set of built-in functions. Minor variations may arise between implementations.

ARITHMETIC FUNCTIONS

- arguments may be INTEGER or SCALAR types
- in functions with one argument, result type matches argument type (except as specifically noted)
- in functions with two arguments, unless specifically specified, result type is scalar if either or both arguments are scalar; otherwise the result type is integer
- arrayed arguments cause multiple invocations of the function, one for each array element - arrayness of arrayed arguments must match

140

Name, Arguments	Comments
ABS(α)	$\lvert\alpha\rvert$
CEILING(α)	smallest integer $\geq \alpha$
DIV(α,β)	integer division α/β (arguments rounded to integers)
FLOOR(α)	largest integer $\leq \alpha$
MIDVAL(α,β,γ)	the value of the argument which is algebraically between the other two. If two or more arguments are equal, the multiple value is returned. Result is always scalar.
MOD(α,β)	α MOD β
ODD(α)	TRUE 1 if α odd } result is FALSE 0 if α even } BOOLEAN
REMAINDER(α,β)	signed remainder of integer division α/β (argument rounded to integer)

ARITHMETIC FUNCTIONS (CONTINUED)	
Name, Arguments	Comments
ROUND(α)	nearest integral value to α
SIGN(α)	+1 $\alpha \geq 0$ -1 $\alpha < 0$
SIGNUM(α)	+1 $\alpha > 0$ 0 $\alpha = 0$ -1 $\alpha < 0$
TRUNCATE(α)	largest integer $\leq \lvert\alpha\rvert$ times SIGNUM (integer($\bar{\alpha}$))

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

ALGEBRAIC FUNCTIONS	
• arguments may be integer or scalar types - conversion to scalar occurs with integer arguments • result type is always scalar • arrayed arguments cause multiple invocations of the function, one for each array element • angular values are supplied or delivered in radians.	
Name, Arguments	**Comments**
ARCCOS(α)	$\cos^{-1} \alpha \quad \lvert\alpha\rvert \le 1$
ARCCOSH(α)	$\cosh^{-1} \alpha \quad \alpha \ge 1$
ARCSIN(α)	$\sin^{-1} \alpha, \quad \lvert\alpha\rvert \le 1$
ARCSINH(α)	$\sinh^{-1} \alpha$
ARCTAN2(α,β)	$-\pi < \tan^{-1}(\alpha/\beta) \le \pi$ Proper Quadrant if: $\left.\begin{array}{l}\alpha = k \sin \theta \\ \beta = k \cos \theta\end{array}\right\} k > 0$
ARCTAN(α)	$\tan^{-1} \alpha$
ARCTANH(α)	$\tanh^{-1} \alpha \quad \lvert\alpha\rvert < 1$
COS(α)	$\cos \alpha$
COSH(α)	$\cosh \alpha$
EXP(α)	e^{α}
LOG(α)	$\log_e \alpha, \quad \alpha > 0$
SIN(α)	$\sin \alpha$
SINH(α)	$\sinh \alpha$
SQRT(α)	$\sqrt{\alpha}, \quad \alpha \ge 0$
TAN(α)	$\tan \alpha$
TANH(α)	$\tanh \alpha$

102

VECTOR-MATRIX FUNCTIONS

- arguments are vector or matrix types as indicated
- result types are as implied by mathematical operation
- arrayed arguments cause multiple invocations of the function, one for each array element

Name, Arguments	Comments
ABVAL(α)	length of vector α
DET(α)	determinant of square matrix α
INVERSE(α)	inverse of nonsingular square matrix α
TRACE(α)	sum of diagonal elements of square matrix α
TRANSPOSE(α)	transpose of matrix α
UNIT(α)	unit vector in same direction as vector α

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

136

MISCELLANEOUS FUNCTIONS

- arguments are as indicated; if none are indicated the function has no arguments
- result type is as indicated

Name, Arguments	Result Type	Comments
CLOCKTIME	scalar	returns time of day
DATE	integer	returns date (implementation dependent format)
ERRGRP	integer	returns group number of last error detected, or zero
ERRNUM	integer	returns number of last error detected, or zero
PRIO	integer	returns priority of process calling function
RANDOM	scalar	returns random number from rectangular distribution over range 0-1
RANDOMG	scalar	returns random number from Gaussian distribution mean zero, variance one.
RUNTIME	scalar	returns Real Time Executive clock time (Section 8.)
NEXTIME (<label>)	scalar	<label> is the name of a program or task. The value returned is determined as follows: a) If the specified process was scheduled with the REPEAT EVERY option and has begun at least one cycle of execution, then the value is the time the next cycle will begin. b) If the specified process was scheduled with the IN or AT phrase, and has not yet begun execution, then the value is the time it will begin execution. c) Otherwise, the value is equal to the current time (RUNTIME function).

116

MISCELLANEOUS FUNCTIONS (CONTINUED)		
Name, Arguments	Result Type	Comments
SHL(α,β)	Same as α	α may be integer or bit type. β must be integer type. If α is integer type, the result is an integer whose internal binary representation is that of α shifted left by β bit locations. The signed nature of the integer α is taken into account in an implementation dependent manner which depends upon the number system and word size of the target computer. If α is bit type, the result is a bit string containing the value of α shifted left by β bit locations. α is treated as an unsigned logical quantity. The size of the result is implementation dependent. Arrayed arguments produce multiple invocations of the function, one for each array element - arrayness of arrayed arguments must match.
SHR(α,β)	Same as α	α may be integer or bit type. β must be integer type. Results are as defined for the SHL function except that all shifting occurs to the right. Arrayed arguments produce multiple invocations of the function, one for each array element - arrayness of arrayed arguments must match.

CHARACTER FUNCTIONS

- first argument is character type - second argument is as indicated (any argument indicated as character type may also be integer or scalar, whereupon conversion to character type is implicitly assumed)
- result type is as indicated
- arrayed arguments produce multiple invocations of the function, one for each array element - arraynesses of arrayed arguments must match

Name, Arguments	Result Type	Comments
INDEX(α,β)	integer	β is character type - if string β appears in string α, index pointing to the first character of β is returned; otherwise zero is returned
LENGTH(α)	integer	returns length of character string
LJUST(α,β)	character	β is integer type - string α is expanded to length β by padding on the right with blanks $\beta \geq$ length (α)
RJUST(α,β)	character	β is integer type - string α is expanded to length β by padding on the left with blanks $\beta \geq$ length (α)
TRIM(α)	character	leading and trailing blanks are stripped from α

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

ARRAY FUNCTIONS	
• arguments are n-dimensional arrays where n is arbitrary • arguments are integer or scalar type • result type matches argument type and is unarrayed	
Name, Parameters	Comments
MAX(α)	maximum of all elements of α
MIN(α)	minimum of all elements of α
PROD(α)	product of all elements of α
SUM(α)	sum of all elements of α

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

SIZE FUNCTION	
Name, Argument	Comments
SIZE(α)	One of the following must hold: • α is an unsubscripted arrayed variable with a one-dimensional array specification - function returns length of array. • α is an unsubscripted major structure with a multiple copy specification - function returns number of copies. • α is an unsubscripted structure terminal with a one-dimensional array specification - function returns length of array. Result is of integer type

E

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

D. STANDARD CONVERSION FORMATS

In relatively limited circumstances HAL/S allows conversion between scalar, integer, bit and character types. The following rules govern such conversions.

CONVERSIONS TO INTEGER TYPE:

- A bit type is converted to integer type by regarding it as the bit pattern of a signed integer of the desired precision (halfword or fullword). Left padding with binary zeroes, or left truncation may occur.
- A scalar type is converted to integer type by rounding to the nearest whole number. Overflow errors may occur if the absolute value of the scalar type is too large to be represented as an integer of the desired precision.
- A character type is convertible to integer type only if its value represents a signed whole number (e.g. '-604), otherwise an error condition occurs. An error condition also occurs if the whole number is too large to be represented as an integer of the desired precision.

137

CONVERSIONS TO SCALAR TYPE:

- An integer type is converted directly to scalar form. Depending on the implementation, and the precisions, some decimal places of accuracy may be lost during conversion.
- A bit type is converted to scalar type by first converting it to double precision integer type according to the rule previously given, and then applying the integer to scalar conversion.
- A character type is convertible to scalar type only if its value represents a legal scalar- or integer-valued literal (e.g. '-1.5E-7'). See Section 2.3.3 for details of arithmetic literals. Other values cause error conditions to arise.

141

137

CONVERSIONS TO BIT TYPE:

114A

- An integer type is converted to a bit string of maximum length. The value is the bit pattern of the integer.

- A scalar type is first converted to double precision integer type according to the rule already given, and the integer to bit conversion rules is then applied.

- A character type is convertible to bit type only if its value is a string of '1's and '0's, and blanks, (but not all blanks), otherwise an error condition arises. The result of the conversion is always a maximum length bit string, irrespective of the argument type. If the argument has more than N bits, where N is the maximum allowable length of a bit operand, then only the N right-most are used. If the argument has fewer than N bits, the string is padded on the left with binary zeroes.

CONVERSIONS TO CHARACTER TYPE:

137

- An integer type is converted to the representation

```
 dddd    (positive)
-dddd    (negative)
```

  where dddd represents an arbitrary number of decimal digits. Leading zeroes are suppressed yielding a variable length result.

- A scalar type is converted to the representation

```
ƀd.ddddE±dd    (positive)
-d.ddddE±dd    (negative)
```

  (except scalar 0 is converted to 0.0).

  The number of decimal digits d in the fractional part and exponent are implementation and precision dependent. The digit to the left of the decimal point is non-zero. There are no imbedded blanks. Leading zeros in the exponent are not suppressed. The representation includes a leading blank (ƀ) if the scalar is positive. In all cases, the result is fixed in length.

- A bit type is converted to a character string of '1's and '0's corresponding to the binary representation of the bit string argument.

E. STANDARD EXTERNAL FORMATS

Corresponding to each data type there exists a "standard external format" for the representation of its values on sequential I/O files. In any implementation the standard external formal on output is fixed; on input the user has a certain flexibility in the format he can use.

OUTPUT FORMATS

1. Integer Type:

   - The value of an integer is represented by a string of decimal digits, preceded if it is negative by a - sign. Leading zeroes are suppressed.

   - The string of digits is right justified in a field of fixed width. The width depends on the implementation, and on the precision of the integer.

2. Scalar Type:

   - If the value of a scalar is positive it is represented by

     ꝥd.dddddddE±dd

     where d represents a decimal digit. One non-zero digit appears before the decimal point. The numbers of digits in the fractional part and exponent are fixed, and depend on the implementation and the precision of the scalar. Leading zeroes in the exponent are not suppressed. The representation includes a leading blank (ꝥ).

   - A negative value has the same form except that a - sign precedes the first decimal digit.

   - If the value is exactly zero, it is represented as 0.0.

   - The representation of a scalar is contained in a field of fixed width. The width is dependent on the implementation and the precision of the scalar. Justification is such that the decimal point occupies a fixed, precision dependent position in the field.

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

3. Bit Type (including BOOLEAN):

- There are two different representations of values of bit variables.

- The first representation consists of a string of binary digits corresponding to the bit variable. Leading binary zeros are not suppressed. The field width is equal to the number of binary digits in the string plus an inserted blank following every fourth digit (to enhance readability). This form is not compatible with the READ input (see Section 10.1.1).

- In the alternate representation, the string of binary digits plus inserted blanks is enclosed in the apostrophes. The field width is equal to the total of the number of digits, blanks and two apostrophes.

4. Character Type:

- There are two different representations of values of character variables.

- The first representation merely consists of the string of characters comprising the value. The field width is equal to the number of characters in the string. This representation is not compatible with READ input (see Section 10.1.1).

- In the alternate representation, the string of characters is enclosed in apostrophes, and all internal apostrophes are converted to apostrophe pairs. The field width is equal to the total number of characters in the string, including added apostrophes.

NOTE: The two alternate representations for bit and character types occur on paged and unpaged output respectively.

INPUT FORMATS

1. Scalar and Integer Types:

- There are two basic representations, whole-number and floating-point.

- The whole number representation consists of a string of decimal digits preceded by an optional - sign. The maximum number of digits allowed is implementation dependent. Conversion to mantissa-exponent form takes place for scalar types.

- The floating-point representation is either

ddd.dddd

or dddd.dddd $\begin{Bmatrix} E \\ B \\ H \end{Bmatrix}$ ±dd

where d is a decimal digit. Any number of digits is allowed in the mantissa to an implementation dependent maximum. The decimal point may appear in any position. E,B, and H represent the exponent digits to be powers of 10,2 and 16 respectively. A choice of one is indicated. The maximum number of digits in the exponent is implementation dependent. For bit and integer types, the representation is rounded to the nearest integral value. For bit types the binary representation of the result is taken.

- The floating-point representation may be prefixed by + or - signs to indicate the sign of the value. Without such prefix the value is positive.

2. Character Type:

- The representation of character type is a string of characters from the HAL/S extended set enclosed in apostrophes. The number of characters may vary between zero (a "null string") and an implementation dependent maximum. Within the string apostrophes must be represented by an apostrophe pair.

3. Bit Type:

- The representation of bit type is a string of '1's and '0's enclosed in apostrophes. Imbedded blanks are ignored. The number of digits may vary between one and an implementation maximum.

INTERMETRICS INCORPORATED • 701 CONCORD AVENUE • CAMBRIDGE, MASSACHUSETTS 02138 • (617) 661-18[illegible]

F. COMPILE-TIME COMPUTATIONS

References are made in the text to expressions which must be computable at compile time. In particular the following constructs make use of them:

- declaration of dimensions;
- initialization;
- subscripting.

Subsets of arithmetic, bit, and character expressions are guaranteed to be computable at compile time.

ARITHMETIC EXPRESSIONS (see Section 6.1.1)

1. <arith exp>s of integer and scalar type only can be computable at compile time.

2. The operators of such <arith exp>s are limited to:

   +

   -

   <> (multiply)

   /

   **

3. The <arith operand>s of such <arith exp>s may either be <number>s or unarrayed unsubscripted simple variables[1] of integer or scalar type. Such variables must previously have been declared, and initialized using the CONSTANT form.

---

[1] see Section 4.5

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

4. The following built-in functions are also legal:

SIN	EXP	DATE
COS	LOG	CLOCKTIME
TAN	SQRT	

DATE and CLOCKTIME are only computed at compile time if they appear in an <initialization> construct.

BIT EXPRESSIONS (see Section 6.1.2)

1. The operators which may appear in <bit exp>s computable at compile time are:

¬

&

|

2. The <bit operand>s of such <bit exp>s must be either <bit literal>s or unarrayed unsubscripted simple variables of bit type. Such variables must previously have been declared, and initialized using the CONSTANT form.

CHARACTER EXPRESSIONS (see Section 6.1.3)

1. The catenation operator (||) only may appear in <char exp>s computable at compile time.

2. The <char operand>s of such <char exp>s must be either <char literal>s, <arith exp>s computable at compile time, or unarrayed unsubscripted simple variables of character type. Such variables must previously have been declared, and initialized using the CONSTANT form.

In some implementations, additional forms may also be computed at compile time. They will not, however, be regarded as legal in contexts where compile time computability is enforced semantically.

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

G. WORKING GRAMMAR

Following is the Release 360-14, FC-10 version of the HAL/S working grammar, written in standard BNF notation. | 135

Edited Grammar

```
 1   <COMPILATION> ::= <COMPILE LIST> _|_

 2   <COMPILE LIST> ::= <BLOCK DEFINITION>
 3                    | <COMPILE LIST> <BLOCK DEFINITION>

 4   <ARITH EXP> ::= <TERM>
 5                 | + <TERM>
 6                 | - <TERM>
 7                 | <ARITH EXP> + <TERM>
 8                 | <ARITH EXP> - <TERM>

 9   <TERM> ::= <PRODUCT>
10            | <PRODUCT> / <TERM>

11   <PRODUCT> ::= <FACTOR>
12               | <FACTOR> * <PRODUCT>
13               | <FACTOR> . <PRODUCT>
14               | <FACTOR> <PRODUCT>

15   <FACTOR> ::= <PRIMARY>
16              | <PRIMARY> <**> <FACTOR>

17   <**> ::= **

18   <PRE PRIMARY> ::= ( <ARITH EXP> )
19                   | <NUMBER>
20                   | <COMPOUND NUMBER>

21   <ARITH FUNC HEAD> ::= <ARITH FUNC>
22                       | <ARITH CONV> <SUBSCRIPT>

23   <ARITH CONV> ::= INTEGER
24                  | SCALAR
25                  | VECTOR
26                  | MATRIX

27   <PRIMARY> ::= <ARITH VAR>

28   <PRE PRIMARY> ::= <ARITH FUNC HEAD> ( <CALL LIST> )

29   <PRIMARY> ::= <MODIFIED ARITH FUNC>
30               | <ARITH INLINE DEF> <BLOCK BODY> <CLOSING> ;
31               | <PRE PRIMARY>
32               | <PRE PRIMARY> <QUALIFIER>

33   <OTHER STATEMENT> ::= <ON PHRASE> <STATEMENT>
34                       | <IF STATEMENT>
35                       | <LABEL DEFINITION> <OTHER STATEMENT>
```


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

```
36   <STATEMENT> ::= <BASIC STATEMENT>
37                 | <OTHER STATEMENT>

38   <ANY STATEMENT> ::= <STATEMENT>
39                     | <BLOCK DEFINITION>

40   <BASIC STATEMENT> ::= <LABEL DEFINITION> <BASIC STATEMENT>
41                       | <ASSIGNMENT> ;
42                       | EXIT ;
43                       | EXIT <LABEL> ;
44                       | REPEAT ;
45                       | REPEAT <LABEL> ;
46                       | GO TO <LABEL> ;
47                       | ;
48                       | <CALL KEY> ;
49                       | <CALL KEY> ( <CALL LIST> ) ;
50                       | <CALL KEY> <ASSIGN> ( <CALL ASSIGN LIST> ) ;
51                       | <CALL KEY> ( <CALL LIST> ) <ASSIGN> ( <CALL ASSIGN LIST> ) ;
52                       | RETURN ;
                         | RETURN <EXPRESSION> ;
                         | <DO GROUP HEAD> <ENDING> ;
55                       | <READ KEY> ;
56                       | <READ PHRASE> ;
57                       | <WRITE KEY> ;
58                       | <WRITE PHRASE> ;
59                       | <FILE EXP> = <EXPRESSION> ;
60                       | <VARIABLE> = <FILE EXP> ;
61                       | <WAIT KEY> FOR DEPENDENT ;
62                       | <WAIT KEY> <ARITH EXP> ;
63                       | <WAIT KEY> UNTIL <ARITH EXP> ;
64                       | <WAIT KEY> FOR <BIT EXP> ;
65                       | <TERMINATOR> ;
66                       | <TERMINATOR> <TERMINATE LIST> ;
67                       | UPDATE PRIORITY TO <ARITH EXP> ;
68                       | UPDATE PRIORITY <LABEL VAR> TO <ARITH EXP> ;
69                       | <SCHEDULE PHRASE> ;
70                       | <SCHEDULE PHRASE> <SCHEDULE CONTROL> ;
71                       | <SIGNAL CLAUSE> ;
72                       | SEND ERROR <SUBSCRIPT> ;
73                       | <ON CLAUSE> ;
74                       | <ON CLAUSE> AND <SIGNAL CLAUSE> ;
75                       | OFF ERROR <SUBSCRIPT> ;
76                       | <% MACRO NAME> ;
                         | <% MACRO HEAD> <% MACRO ARG> ) ;

78   <% MACRO HEAD> ::= <% MACRO NAME> (
79                    | <% MACRO HEAD> <% MACRO ARG> ,

80   <% MACRO ARG> ::= <NAME VAR>
81                   | <CONSTANT>

82   <BIT PRIM> ::= <BIT VAR>
83                | <LABEL VAR>
84                | <EVENT VAR>
85                | <BIT CONST>
86                | ( <BIT EXP> )
87                | <MODIFIED BIT FUNC>
88                | <BIT INLINE DEF> <BLOCK BODY> <CLOSING> ;
89                | <SUBBIT HEAD> <EXPRESSION> )
```

```
 90                  | <BIT FUNC HEAD> ( <CALL LIST> )

 91    <BIT FUNC HEAD> ::= <BIT FUNC>
 92                      | BIT <SUB OR QUALIFIER>

 93    <BIT CAT> ::= <BIT PRIM>
 94                | <BIT CAT> <CAT> <BIT PRIM>
 95                | <NOT> <BIT PRIM>
 96                | <BIT CAT> <CAT> <NOT> <BIT PRIM>

 97    <BIT FACTOR> ::= <BIT CAT>
 98                   | <BIT FACTOR> <AND> <BIT CAT>

 99    <BIT EXP> ::= <BIT FACTOR>
100                | <BIT EXP> <OR> <BIT FACTOR>

101    <RELATIONAL OP> ::= =
102                      | <NOT> =
103                      | <
104                      | >
105                      | < =
106                      | > =
107                      | <NOT> <
108                      | <NOT> >

109    <COMPARISON> ::= <ARITH EXP> <RELATIONAL OP> <ARITH EXP>
110                   | <CHAR EXP> <RELATIONAL OP> <CHAR EXP>
111                   | <BIT CAT> <RELATIONAL OP> <BIT CAT>
112                   | <STRUCTURE EXP> <RELATIONAL OP> <STRUCTURE EXP>
113                   | <NAME EXP> <RELATIONAL OP> <NAME EXP>

114.   <RELATIONAL FACTOR> ::= <REL PRIM>
115                          | <RELATIONAL FACTOR> <AND> <REL PRIM>

116    <RELATIONAL EXP> ::= <RELATIONAL FACTOR>
117                       | <RELATIONAL EXP> <OR> <RELATIONAL FACTOR>

118    <REL PRIM> ::= ( <RELATIONAL EXP> )
119                 | <NOT> ( <RELATIONAL EXP> )
120                 | <COMPARISON>

121    <CHAR PRIM> ::= <CHAR VAR>
122                  | <CHAR CONST>
123                  | <MODIFIED CHAR FUNC>
124                  | <CHAR INLINE DEF> <BLOCK BODY> <CLOSING> ;
125                  | <CHAR FUNC HEAD> ( <CALL LIST> )
126                  | ( <CHAR EXP> )

127    <CHAR FUNC HEAD> ::= <CHAR FUNC>
128                       | CHARACTER <SUB OR QUALIFIER>

129    <SUB OR QUALIFIER> ::= <SUBSCRIPT>
130                         | <BIT QUALIFIER>

131    <CHAR EXP> ::= <CHAR PRIM>
132                 | <CHAR EXP> <CAT> <CHAR PRIM>
133                 | <CHAR EXP> <CAT> <ARITH EXP>
134                 | <ARITH EXP> <CAT> <ARITH EXP>
135                 | <ARITH EXP> <CAT> <CHAR PRIM>
```

```
136   <ASSIGNMENT> ::= <VARIABLE> <=1> <EXPRESSION>
137                  | <VARIABLE> , <ASSIGNMENT>

138   <IF STATEMENT> ::= <IF CLAUSE> <STATEMENT>
139                    | <TRUE PART> <STATEMENT>

140   <TRUE PART> ::= <IF CLAUSE> <BASIC STATEMENT> ELSE

141   <IF CLAUSE> ::= <IF> <RELATIONAL EXP> THEN
142                 | <IF> <BIT EXP> THEN

143   <IF> ::= IF

144   <DO GROUP HEAD> ::= DO ;
145                     | DO <FOR LIST> ;
146                     | DO <FOR LIST> <WHILE CLAUSE> ;
147                     | DO <WHILE CLAUSE> ;
148                     | DO CASE <ARITH EXP> ;
149                     | <CASE ELSE> <STATEMENT>
150                     | <DO GROUP HEAD> <ANY STATEMENT>
151                     | <DO GROUP HEAD> <TEMPORARY STMT>

152   <CASE ELSE> ::= DO CASE <ARITH EXP> ; ELSE

153   <WHILE KEY> ::= WHILE
154                 | UNTIL

155   <WHILE CLAUSE> ::= <WHILE KEY> <BIT EXP>
156                    | <WHILE KEY> <RELATIONAL EXP>

157   <FOR LIST> ::= <FOR KEY> <ARITH EXP> <ITERATION CONTROL>
158                | <FOR KEY> <ITERATION BODY>

159   <ITERATION BODY> ::= <ARITH EXP>
160                      | <ITERATION BODY> , <ARITH EXP>

161   <ITERATION CONTROL> ::= TO <ARITH EXP>
162                         | TO <ARITH EXP> BY <ARITH EXP>

163   <FOR KEY> ::= FOR <ARITH VAR> =
164               | FOR TEMPORARY <IDENTIFIER> =

165   <ENDING> ::= END
166              | END <LABEL>
167              | <LABEL DEFINITION> <ENDING>

168   <ON PHRASE> ::= ON ERROR <SUBSCRIPT>

169   <ON CLAUSE> ::= ON ERROR <SUBSCRIPT> SYSTEM
170                 | ON ERROR <SUBSCRIPT> IGNORE

171   <SIGNAL CLAUSE> ::= SET <EVENT VAR>
172                     | RESET <EVENT VAR>
173                     | SIGNAL <EVENT VAR>

174   <FILE EXP> ::= <FILE HEAD> , <ARITH EXP> )

175   <FILE HEAD> ::= FILE ( <NUMBER>
```

```
176   <CALL KEY> ::= CALL <LABEL VAR>

177   <CALL LIST> ::= <LIST EXP>
178                 | <CALL LIST> , <LIST EXP>

179   <CALL ASSIGN LIST> ::= <VARIABLE>
180                        | <CALL ASSIGN LIST> , <VARIABLE>

181   <EXPRESSION> ::= <ARITH EXP>
182                  | <BIT EXP>
183                  | <CHAR EXP>
184                  | <STRUCTURE EXP>
185                  | <NAME EXP>

186   <STRUCTURE EXP> ::= <STRUCTURE VAR>
187                     | <MODIFIED STRUCT FUNC>
188                     | <STRUC INLINE DEF> <BLOCK BODY> <CLOSING> ;
189                     | <STRUCT FUNC HEAD> ( <CALL LIST> )

190   <STRUCT FUNC HEAD> ::= <STRUCT FUNC>

191   <LIST EXP> ::= <EXPRESSION>
192                | <ARITH EXP> # <EXPRESSION>

193   <VARIABLE> ::= <ARITH VAR>
194                | <STRUCTURE VAR>
195                | <BIT VAR>
196                | <EVENT VAR>
197                | <SUBBIT HEAD> <VARIABLE> )
198                | <CHAR VAR>
199                | <NAME KEY> ( <NAME VAR> )

200   <NAME VAR> ::= <VARIABLE>
201                | <LABEL VAR>
202                | <MODIFIED ARITH FUNC>
203                | <MODIFIED BIT FUNC>
204                | <MODIFIED CHAR FUNC>
205                | <MODIFIED STRUCT FUNC>

206   <NAME EXP> ::= <NAME KEY> ( <NAME VAR> )
207                | NULL
208                | <NAME KEY> ( NULL )

209   <NAME KEY> ::= NAME

210   <LABEL VAR> ::= <PREFIX> <LABEL> <SUBSCRIPT>

211   <MODIFIED ARITH FUNC> ::= <PREFIX> <NO ARG ARITH FUNC> <SUBSCRIPT>

212   <MODIFIED BIT FUNC> ::= <PREFIX> <NO ARG BIT FUNC> <SUBSCRIPT>

213   <MODIFIED CHAR FUNC> ::= <PREFIX> <NO ARG CHAR FUNC> <SUBSCRIPT>

214   <MODIFIED STRUCT FUNC> ::= <PREFIX> <NO ARG STRUCT FUNC> <SUBSCRIPT>

215   <STRUCTURE VAR> ::= <QUAL STRUCT> <SUBSCRIPT>

216   <ARITH VAR> ::= <PREFIX> <ARITH ID> <SUBSCRIPT>
```


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

```
217   <CHAR VAR> ::= <PREFIX> <CHAR ID> <SUBSCRIPT>

218   <BIT VAR> ::= <PREFIX> <BIT ID> <SUBSCRIPT>

219   <EVENT VAR> ::= <PREFIX> <EVENT ID> <SUBSCRIPT>

220   <QUAL STRUCT> ::= <STRUCTURE ID>
221                   | <QUAL STRUCT> . <STRUCTURE ID>

222   <PREFIX> ::=
223               | <QUAL STRUCT> .

224   <SUBBIT HEAD> ::= <SUBBIT KEY> <SUBSCRIPT> (

225   <SUBBIT KEY> ::= SUBBIT

226   <SUBSCRIPT> ::= <SUB HEAD> )
227                 | <QUALIFIER>
228                 | <$> <NUMBER>
229                 | <$> <ARITH VAR>
230                 |

231   <SUB START> ::= <$> (
232                 | <$> ( @ <PREC SPEC> ,
233                 | <SUB HEAD> ;
234                 | <SUB HEAD> :
235                 | <SUB HEAD> ,

236   <SUB HEAD> ::= <SUB START>
237                | <SUB START> <SUB>

238   <SUB> ::= <SUB EXP>
239           | *
240           | <SUB RUN HEAD> <SUB EXP>
241           | <ARITH EXP> AT <SUB EXP>

242   <SUB RUN HEAD> ::= <SUB EXP> TO

243   <SUB EXP> ::= <ARITH EXP>
244               | <# EXPRESSION>

245   <# EXPRESSION> ::= #
246                    | <# EXPRESSION> + <TERM>
247                    | <# EXPRESSION> - <TERM>

248   <=1> ::= =

249   <$> ::= $

250   <AND> ::= &
251           | AND

252   <OR> ::= |
253          | OR

254   <NOT> ::= ¬
255           | NOT

256   <CAT> ::= ||
```

```
257               | CAT

258   <QUALIFIER> ::= <$> ( @ <PREC SPEC> )

259   <BIT QUALIFIER> ::= <$> ( @ <RADIX> )

260   <RADIX> ::= HEX
261             | OCT
262             | BIN
263             | DEC

264   <BIT CONST HEAD> ::= <RADIX>
265                      | <RADIX> ( <NUMBER> )

266   <BIT CONST> ::= <BIT CONST HEAD> <CHAR STRING>
267                 | TRUE
268                 | FALSE
269                 | ON
270                 | OFF

271   <CHAR CONST> ::= <CHAR STRING>
272                  | CHAR ( <NUMBER> ) <CHAR STRING>

273   <IO CONTROL> ::= SKIP ( <ARITH EXP> )
274                  | TAB ( <ARITH EXP> )
275                  | COLUMN ( <ARITH EXP> )
276                  | LINE ( <ARITH EXP> )
277                  | PAGE ( <ARITH EXP> )

278   <READ PHRASE> ::= <READ KEY> <READ ARG>
279                   | <READ PHRASE> , <READ ARG>

280   <WRITE PHRASE> ::= <WRITE KEY> <WRITE ARG>
281                    | <WRITE PHRASE> , <WRITE ARG>

282   <READ ARG> ::= <VARIABLE>
283                | <IO CONTROL>

284   <WRITE ARG> ::= <EXPRESSION>
285                 | <IO CONTROL>

286   <READ KEY> ::= READ ( <NUMBER> )
287                | READALL ( <NUMBER> )

288   <WRITE KEY> ::= WRITE ( <NUMBER> )

289   <BLOCK DEFINITION> ::= <BLOCK STMT> <BLOCK BODY> <CLOSING> ;

290   <BLOCK BODY> ::=
291                  | <DECLARE GROUP>
292                  | <BLOCK BODY> <ANY STATEMENT>

293   <ARITH INLINE DEF> ::= FUNCTION <ARITH SPEC> ;
294                        | FUNCTION ;

295   <BIT INLINE DEF> ::= FUNCTION <BIT SPEC> ;

296   <CHAR INLINE DEF> ::= FUNCTION <CHAR SPEC> ;
```


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

```
297   <STRUC INLINE DEF> ::= FUNCTION <STRUCT SPEC> ;

298   <BLOCK STMT> ::= <BLOCK STMT TOP> ;

299   <BLOCK STMT TOP> ::= <BLOCK STMT TOP> ACCESS
300                      | <BLOCK STMT TOP> RIGID
301                      | <BLOCK STMT HEAD>
302                      | <BLOCK STMT HEAD> EXCLUSIVE
303                      | <BLOCK STMT HEAD> REENTRANT

304   <LABEL DEFINITION> ::= <LABEL> :

305   <LABEL EXTERNAL> ::= <LABEL DEFINITION>
306                      | <LABEL DEFINITION> EXTERNAL

307   <BLOCK STMT HEAD> ::= <LABEL EXTERNAL> PROGRAM
308                       | <LABEL EXTERNAL> COMPOOL
309                       | <LABEL DEFINITION> TASK
310                       | <LABEL DEFINITION> UPDATE
311                       | UPDATE
312                       | <FUNCTION NAME>
313                       | <FUNCTION NAME> <FUNC STMT BODY>
314                       | <PROCEDURE NAME>
315                       | <PROCEDURE NAME> <PROC STMT BODY>

316   <FUNCTION NAME> ::= <LABEL EXTERNAL> FUNCTION

317   <PROCEDURE NAME> ::= <LABEL EXTERNAL> PROCEDURE

318   <FUNC STMT BODY> ::= <PARAMETER LIST>
319                      | <TYPE SPEC>
320                      | <PARAMETER LIST> <TYPE SPEC>

321   <PROC STMT BODY> ::= <PARAMETER LIST>
322                      | <ASSIGN LIST>
323                      | <PARAMETER LIST> <ASSIGN LIST>

324   <PARAMETER LIST> ::= <PARAMETER HEAD> <IDENTIFIER> )

325   <PARAMETER HEAD> ::= (
326                      | <PARAMETER HEAD> <IDENTIFIER> ,

327   <ASSIGN LIST> ::= <ASSIGN> <PARAMETER LIST>

328   <ASSIGN> ::= ASSIGN

329   <DECLARE ELEMENT> ::= <DECLARE STATEMENT>
330                       | <REPLACE STMT> ;
331                       | <STRUCTURE STMT>
332                       | EQUATE EXTERNAL <IDENTIFIER> TO <VARIABLE> ;

333   <REPLACE STMT> ::= REPLACE <REPLACE HEAD> BY <TEXT>

334   <REPLACE HEAD> ::= <IDENTIFIER>
335                    | <IDENTIFIER> ( <ARG LIST> )

336   <ARG LIST> ::= <IDENTIFIER>
337                | <ARG LIST> , <IDENTIFIER>
```

```
338   <TEMPORARY STMT> ::= TEMPORARY <DECLARE BODY> ;

339   <DECLARE STATEMENT> ::= DECLARE <DECLARE BODY> ;

340   <DECLARE BODY> ::= <DECLARATION LIST>
341                    | <ATTRIBUTES> , <DECLARATION LIST>

342   <DECLARATION LIST> ::= <DECLARATION>
343                        | <DCL LIST ,> <DECLARATION>

344   <DCL LIST ,> ::= <DECLARATION LIST> ,

345   <DECLARE GROUP> ::= <DECLARE ELEMENT>
346                     | <DECLARE GROUP> <DECLARE ELEMENT>

347   <STRUCTURE STMT> ::= STRUCTURE <STRUCT STMT HEAD> <STRUCT STMT TAIL>

348   <STRUCT STMT HEAD> ::= <IDENTIFIER> : <LEVEL>
349                        | <IDENTIFIER> <MINOR ATTR LIST> : <LEVEL>
350                        | <STRUCT STMT HEAD> <DECLARATION> , <LEVEL>

351   <STRUCT STMT TAIL> ::= <DECLARATION> ;

352   <STRUCT SPEC> ::= <STRUCT TEMPLATE> <STRUCT SPEC BODY>

353   <STRUCT SPEC BODY> ::= - STRUCTURE
354                        | <STRUCT SPEC HEAD> <LITERAL EXP OR *> )

355   <STRUCT SPEC HEAD> ::= - STRUCTURE (

356   <DECLARATION> ::= <NAME ID>
357                   | <NAME ID> <ATTRIBUTES>

358   <NAME ID> ::= <IDENTIFIER>
359               | <IDENTIFIER> NAME

360   <ATTRIBUTES> ::= <ARRAY SPEC> <TYPE & MINOR ATTR>
361                  | <ARRAY SPEC>
362                  | <TYPE & MINOR ATTR>

363   <ARRAY SPEC> ::= <ARRAY HEAD> <LITERAL EXP OR *> )
364                  | FUNCTION
365                  | PROCEDURE
366                  | PROGRAM
367                  | TASK

368   <ARRAY HEAD> ::= ARRAY (
369                  | <ARRAY HEAD> <LITERAL EXP OR *> ,

370   <TYPE & MINOR ATTR> ::= <TYPE SPEC>
371                         | <TYPE SPEC> <MINOR ATTR LIST>
372                         | <MINOR ATTR LIST>

373   <TYPE SPEC> ::= <STRUCT SPEC>
374                 | <BIT SPEC>
375                 | <CHAR SPEC>
376                 | <ARITH SPEC>
377                 | EVENT
```

```
378   <BIT SPEC> ::= BOOLEAN
379                | BIT ( <LITERAL EXP OR *> )

380   <CHAR SPEC> ::= CHARACTER ( <LITERAL EXP OR *> )

381   <ARITH SPEC> ::= <PREC SPEC>
382                  | <SQ DQ NAME>
383                  | <SQ DQ NAME> <PREC SPEC>

384   <SQ DQ NAME> ::= <DOUBLY QUAL NAME HEAD> <LITERAL EXP OR *> )
385                  | INTEGER
386                  | SCALAR
387                  | VECTOR
388                  | MATRIX

389   <DOUBLY QUAL NAME HEAD> ::= VECTOR (
390                             | MATRIX ( <LITERAL EXP OR *> ,

391   <LITERAL EXP OR *> ::= <ARITH EXP>
392                        | *

393   <PREC SPEC> ::= SINGLE
394                 | DOUBLE

395   <MINOR ATTR LIST> ::= <MINOR ATTRIBUTE>
396                       | <MINOR ATTR LIST> <MINOR ATTRIBUTE>

397   <MINOR ATTRIBUTE> ::= STATIC
398                       | AUTOMATIC
399                       | DENSE
400                       | ALIGNED
401                       | ACCESS
402                       | LOCK ( <LITERAL EXP OR *> )
403                       | REMOTE
404                       | RIGID
405                       | <INIT/CONST HEAD> <REPEATED CONSTANT> )
406                       | <INIT/CONST HEAD> * )
407                       | LATCHED
408                       | NONHAL ( <LEVEL> )

409   <INIT/CONST HEAD> ::= INITIAL (
410                       | CONSTANT (
411                       | <INIT/CONST HEAD> <REPEATED CONSTANT> ,

412   <REPEATED CONSTANT> ::= <EXPRESSION>
413                         | <REPEAT HEAD> <VARIABLE>
414                         | <REPEAT HEAD> <CONSTANT>
415                         | <NESTED REPEAT HEAD> <REPEATED CONSTANT> )
416                         | <REPEAT HEAD>

417   <REPEAT HEAD> ::= <ARITH EXP> #

418   <NESTED REPEAT HEAD> ::= <REPEAT HEAD> (
419                          | <NESTED REPEAT HEAD> <REPEATED CONSTANT> ,

420   <CONSTANT> ::= <NUMBER>
421                | <COMPOUND NUMBER>
422                | <BIT CONST>
423                | <CHAR CONST>
```

```
424   <NUMBER> ::= <SIMPLE NUMBER>
425              | <LEVEL>

426   <CLOSING> ::= CLOSE
427               | CLOSE <LABEL>
428               | <LABEL DEFINITION> <CLOSING>

429   <TERMINATOR> ::= TERMINATE
430                  | CANCEL

431   <TERMINATE LIST> ::= <LABEL VAR>
432                      | <TERMINATE LIST> , <LABEL VAR>

433   <WAIT KEY> ::= WAIT

434   <SCHEDULE HEAD> ::= SCHEDULE <LABEL VAR>
435                     | <SCHEDULE HEAD> AT <ARITH EXP>
436                     | <SCHEDULE HEAD> IN <ARITH EXP>
437                     | <SCHEDULE HEAD> ON <BIT EXP>

438   <SCHEDULE PHRASE> ::= <SCHEDULE HEAD>
439                       | <SCHEDULE HEAD> PRIORITY ( <ARITH EXP> )
440                       | <SCHEDULE PHRASE> DEPENDENT

441   <SCHEDULE CONTROL> ::= <STOPPING>
442                        | <TIMING>
443                        | <TIMING> <STOPPING>

444   <TIMING> ::= <REPEAT> EVERY <ARITH EXP>
445              | <REPEAT> AFTER <ARITH EXP>
446              | <REPEAT>

447   <REPEAT> ::= , REPEAT

448   <STOPPING> ::= <WHILE KEY> <ARITH EXP>
449                | <WHILE KEY> <BIT EXP>
```


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

H. SUMMARY OF OPERATORS

This section contains a series of tables which explicitly summarize the possible arithmetic, bit, character, and conditional operators used in forming expressions in the HAL/S Language.

The information found in this appendix has been abstracted from chapter 6 of this specification.

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

H.1 ARITHMETIC OPERATORS*

OPERATORS	NAME	ARITHMETIC PRECEDENCE	FORM	COMMENTS
**	Exponentiation	1	x**x m**i m**0 m**-i m**T	Ordinary exponentiation Repeated Multiplication Identity matrix Repeated mult. of inverse Transpose of matrix
(blank) < >	Product	2	m m m v v m v v x m m x v x x v x x	matrix-matrix product matrix-vector product vector-matrix product outer product scalar or integer product with matrix/vector scalar or integer product with scalar or integer
*	Cross Product	3	v*v	cross product of two 3-vectors
.	Dot Product	4	v.v	dot product of two vectors
/	Division	5	m/x v/x x/x	division of left-hand term by scalar or integer
+ -	Addition Subtraction	6	x+x m+m v+v x-x m-m v-v +x +m +v -x -m -v	Algebraic addition or subtraction; binary plus and minus

The following abbreviations apply:

i = positive integer literal
x = scalar or integer
m = matrix
v = vector

*Note that this table contains information found in Section 6.1.1.

H.2 CHARACTER OPERATOR*

OPERATOR	NAME	FORM
||	concatenation	[re] || [sult] ——→ [result]

*Note that this table contains information found in Section 6.1.3.

H.3 BIT OPERATORS*

OPERATORS	NAME	BIT OPERATOR PRECEDENCE	FORM	COMMENTS
|| CAT	concatenation	1	B||B	[11101] || [010] ——→ [11101010]
& AND	logical product	2	B&B	Parallel operation bit by bit
| OR	logical sum	3	B|B	Parallel operation bit by bit
¬ NOT	logical complement	Highest implied by syntax	¬B	Parallel operation bit by bit;

The following abbreviations apply:

B = bit string or boolean

*Note that this table contains information found in Section 6.1.2

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

H.4 CONDITIONAL AND EVENT OPERATORS*

<table>
<tr><th>OPERATOR</th><th>NAME</th><th>CONDITIONAL PRECEDENCE</th><th>FORM</th><th>COMMENTS</th></tr>
<tr><td>&
AND</td><td>logical product</td><td>1</td><td>C&C
C AND C</td><td>True if both "C"s true</td></tr>
<tr><td>|
OR</td><td>logical sum</td><td>2</td><td>C|C
C OR C</td><td>True if either "C" is true</td></tr>
<tr><td>¬
NOT</td><td>logical complement</td><td>Highest implied by syntax</td><td>¬C</td><td>Operand</td></tr>
<tr><td colspan="5">The following abbreviations apply:
"C" = any conditional operand.</td></tr>
</table>

*Note that this table contains information found in Sections 6.2 and 6.3.

H.5 COMPARISON OPERATORS*

OPERATOR	USE	COMMENTS
>	A > B	magnitude comparsions: apply only to unarrayed scalar and integer data A and B.
>=	A >= B	
<	A < B	
< =	A < = B	
¬> / NOT>	A ¬> B	
¬< / NOT <	A ¬< B	
=	A=B	equality/inequality for general data A and B.
NOT= / ¬=	A ¬= B	

*Note that this table contains information found in Section 6.2.

I. % MACROS

E

The specific details of %-macro operation are implementation dependent

1. %NAMECOPY(α,β);

   Performs the equivalent of:

   NAME(α) = NAME(β);

   This <%-macro call statement> allows a NAME identifier of structure type to be assigned the name of a structure without the requirement for template matching.

   Rules

   1. α must be a name identifier of structure type.
   2. β must be a name identifier of structure type or a <struct var>.
   3. Neither α nor β may imply multiple copies of names.
   4. For α, other rules for NAME(α) in an assign context apply. E
   5. For β, other rules for NAME(β) in a reference context apply.
   6. No template matching will take place.

2. %SVC(α);

   This <%-macro call statement> transfers flow control in an implementation dependent manner with an accompanying parameter, α. The construct is intended to invoke an external or system defined procedure, or hardware instruction.

   Rules

   1. α may be a literal, name identifier or variable of any type with the exception of <bit pseudo var>. E

2. A subscripted variable must conform to the same restrictions as for assign arguments of procedures.

3. REMOTE and TEMPORARY attributes for α are illegal.

4. The parameter is conveyed as "call-by-reference", i.e. by a pointer to the implied data.

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

BIBLIOGRAPHY

[1]. 'The Programming Language HAL - A Specification', Document #MSC-01846, Intermetrics, Inc., June 1971.

Distribution

NASA
Johnson Space Center
Houston, Texas 77058

John L. Ford, BC28
Charles M. Grant, BM2
Gilbert C. Symons, BT3
John R. Garman, FR

Kennedy Space Center
Florida 32815

J.R. Medlock, LV-CAP

Aerospace Corporation
2350 E. El Segundo Blvd.
El Segundo, Calif. 90245

R.K. Luke

CSDL
75 Cambridge Parkway
Cambridge, Mass. 02142

M. Hamilton, 73

IBM Electronics Systems Center
Bodle Hill Road
Owego, New York 13927

R.T. Smith, 002BC62

International Business Machines Corp.
1322 Space Park Drive
Houston, Texas 77058

J. Hoskins, MC69

Rockwell International
Space Division
12214 Lakewood Blvd.
Downey, Calif. 90241

E. Freddolino, FB-37
J. Ling, FA19

USAF
Unit Postal Office
Los Angeles, Calif. 90045

Capt. D.G. Keach, XRZT

INDEX

ACCESS — 3-14, 3-16, 3-18, 4-13, 4-15, 4-16, 11-17, 11-27

active process — 8-2

ALIGNED — 4-9, 4-10, 4-13, 4-16, 4-17, 11-27

AND — 6-7, 6-8, 6-13, 6-21

apostrophe — 4-7

argument type summary (chart) — 6-37

arguments — 3-17

<argument> — 4-7

arithmetic comparison — 6-15
  syntax diagram #32
  legal arithmetic comparisons — 6-16

arithmetic conversion function — 6-27
  syntax diagram #39

<arith conversion> — 6-6, 6-28

arithmetic expressions — 6-3
  syntax diagram #24

<arith exp> — 4-19, 4-20, 5-12, 5-18, 6-3, 6-15, 7-21, 8-12

<arith exp># — 4-24, 6-28

<arith inline> — 11-3, 11-7

arithmetic literals 2-8

<arith %-macro> 11-5, 11-7

<arith operand> 6-6
  syntax diagram #25

arithmetic operand 11-7
  arith %-macro
    syntax diagram #25s

<arith var> 5-16,

ARRAY 4-13, 4-14

arrayed <comparison> 6-18, 6-17

array dimension 5-17

array properties of expressions 6-12

arrayness 5-17, 5-21, 6-22, 6-24, 7-5, 7-10, 11-18

arrayed infix operations 6-12

arrayed operand comparison 6-20

array specification 4-14, 4-16, 4-17, 4-25, 4-26, 5-17

arrayness of subscript expressions 5-18

array subscripts 5-11
  syntax diagram #22

Array Subscripting 4-2, 5-7, 5-14, 7-10

<array sub> 5-7, 5-8, 5-14

ASSIGN 3-16, 7-9, 7-10,

assign parameter 7-10

assignment 7-1

assignment statement 7-5
  syntax diagram #46

asterisk, use of 4-16, 4-21, 4-22, 4-24, 4-27, 5-11, 5-12

"*" 11-18

** 2-11

AT <arith exp> 8-5

AT 5-11

AT-partition 5-12, 5-13, 5-14, 5-15

<attributes> 4-12,
  factored <attributes> 4-12

AUTOMATIC 4-13

basic statement 7-2
  syntax diagram #44

<basic statement> 7-24

BIT 4-21, 4-26, 6-31, 11-3

bit argument length 6-24

bit assignments 7-7

bit comparison 6-17
  syntax diagram #33

bit comparison function 6-31
  syntax diagram #40

<bit conversion> 6-8

bit expression 6-7
  syntax diagram #26

<bit exp> 4-26, 6-7, 6-8, 6-17, 6-34, 7-3, 7-17, 7-18,

bit expression length 7-13

<bit inline> 11-3, 11-8

bit literals 2-9

<bit literal> 6-8, 6-9

bit operand
  syntax diagram #27 6-8
  bit inline
  bit%-macro
    syntax diagram #27s 11-8

bit operator precedence 6-8

<bit %-macro> 11-8, 11-5

<bit-pseudo var> 5-6, 6-35, 7-7

<bit var> 6-9, 6-8

BIN 2-9, 6-32

@BIN 6-32, 6-34

blanks 2-13, 4-7, 4-24

Block delimiting statements 3-13

block name uniqueness 3-20

Block Templates 3-13
  Syntax Diagram #6 3-11

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

BNF Grammar of HAL/S	Appendix G
boolean	6-9, 6-14, 7-3,
BOOLEAN	4-19, 4-21, 4-26, 11-3
built-in functions	6-23, 11-1
built-in function names	2-6
built-in function parameters	6-24
BY	7-22
CALL Statement	7-10
Syntax Diagram #47	7-9
with NAME Syntax Diagram #47s	11-32
call-by-reference	6-24, 7-10
call-by-value	6-24, 7-10
CANCEL Statement	8-10
Syntax Diagram #58	8-9
cancellation	8-6, 8-7, 8-8, 8-9
CAT	6-7, 6-8, 6-10
catenation	6-10
channels	10-1
HAL/S character set	2-4
CHARACTER	4-21, 4-26 6-33, 6-34, 7-10, 11-9

character argument length 6-24

character comparison 6-18
  syntax diagram #34

character conversion function 6-33
  Syntax Diagram #41

<char conversion> 6-11

character expression 6-10
  Syntax Diagram #28

character expression length 7-13

<char exp> 4-26, 6-10, 6-18, 6-32

character initialization 4-26

<char inline> 11-4, 11-9

character length 4-21

character literal 2-10, 4-7

<char literal> 2-10, 6-11

character operand 6-11
  Syntax Diagram #29
  char inline
  char %-macro
    Syntax Diagram #29s 11-9

character operator precedence 6-14

<char %-macro> 11-5, 11-9

character string 6-10

character type 4-21

<char var> 6-11, 7-7

CLOSE Statement 3-19
  Syntax Diagram #10

CLOSE 7-12, 7-13

closing 3-4

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

<closing>	3-13, 3-19
code blocks	3-13
colon, use of	4-10, 5-14, 9-4
COLUMN	10-3, 10-7 10-8, 10-9
comma, use of	4-7, 4-10 5-11, 10-5
comments (imbedded)	2-13
<comparison>	6-13, 6-20
<compilation>	3-2, 3-20 7-9
component subscripts Syntax Diagram #22	5-11
Component Subscripting	4-2, 5-7, 6-35, 7-7, 7-10, 10-11
<component sub>	5-7, 5-8, 5-16
COMPOOL	3-2, 3-14
COMPOOL block Syntax Diagram #5	3-13, 3-19 3-10
<compool block>	4-16
COMPOOL block template	3-11
<compool header>	3-10
compool header statement	3-14
compool modules	3-1
<compool template>	4-16

conditional expression 6-1
  Syntax Diagram #30 6-13

conditional operand
  Syntax Diagram #31 6-14

<conditional operand> 6-13

<condition> 6-1, 7-4, 7-17, 7-20, 7-22

CONSTANT 4-23

conversion 6-24

cyclic execution 8-4, 8-6

Data declarative attributes 4-13
  Syntax Diagram #15

data declarative <attributes> 4-12

Data Manipulation 6-1

Data NAME identifiers 11-18

Data referencing 5-1

Data Sharing and the UPDATE Block 8-19

data types 1-2

DEC 2-9, 6-32

@DEC 6-32, 6-34

DECLARE Statement 4-12
  Syntax Diagram #14
  with NAME
    Syntax Diagram #14s 11-16
135 | <declare statement> 4-3, 11-42

DECLARE 11-13

declare group 3-4, 3-6, 4-1
  Syntax Diagram #11 4-3
  with EQUATE
135 | Syntax Diagram #115 11-42

<declare group> 3-12, 5-2

Declarations of Temporaries	11-24
DENSE/ALIGNED	4-15, 11-17
DENSE	4-9, 4-10, 4-13, 4-16, 4-17, 7-10, 7-11 11-17, 11-22
DEPENDENT	8-6, 8-12
dependent processes	8-2
DOUBLE	4-19, 4-20, 6-38, 7-6
double precision	6-15
double quotes	4-5
DO	11-12
single DO statement Syntax Diagram #50	7-15
<do statement>	7-14, 7-23
DO CASE statement Syntax Diagram #51	7-16
DO...END statement group Syntax Diagram #49	7-14
DO...END	7-23, 7-24, 7-25 11-15
DO...END statement TEMPORARY statement Syntax Diagram #49s	11-12
DO FOR	11-15
Discrete DO FOR Statement Syntax Diagram #53	7-19

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

discrete DO FOR with loop TEMPORARY variable index Syntax Diagram #53s	11-14
DO WHILE and UNTIL statements Syntax Diagram #52	7-17
DO UNTIL	7-18
DO WHILE	11-30
ELSE	7-3, 7-4, 7-16,
dangling ELSE	7-4
END statement Syntax Diagram #55	7-23
END	7-23
<end statement>	7-14, 7-23
EQUATE Statement Syntax Diagram #80	11-40 11-40
errors system-defined user-defined	9-1
error code	9-1, 9-5, 9-7
ERE	9-1, 9-4, 9-7
error environment	9-1
error groups	9-1
error number	9-7
Error precedence (chart)	9-6
<error spec>	9-3, 9-4 9-5
Error Recovery	1-2, 7-1

135

INTERMETRICS INC • 701 CONCORD AVENUE • CAMBRIDGE, MASSACHUSETTS 02138 • (617) 661-1840

Error Recovery Executive 8-9, 9-1

EVENT 4-18, 4-19, 4-21, 4-26, 11-13, 11-23

event change point 8-5, 8-7, 8-8, 8-15

Event Control 8-15

event expression 6-1
  Syntax Diagram #36 6-21

<event exp> 6-21, 8-13

event infix operator precedence 6-22

event operand 6-22
  Syntax Diagram #57

<event operand> 6-21

<event var> 6-9
  latched 8-16
  unlatched 8-16

EVERY <arith exp> 8-6

EXCLUSIVE 3-16, 3-18

executable statements 7-1

EXIT statement 7-24
  Syntax Diagram #56

EXIT 7-25

explicit conversion functions 6-26, 6-27, 6-31, 6-33

explicit type conversion 6-24, 6-38

exponent 7-6

<exponents> 2-8

exponentiation 2-12

<expression> 6-1, 7-5, 7-12

external procedure 3-1

135 | EXTERNAL 3-12, 11-40

extended character set 2-4

FALSE 2-9, 4-26, 7-4, 7-17, 8-16

FILE 10-11

<file exp> 10-10, 10-11

FILE statement
  Syntax Diagram #68 10-10

paged file 10-2

unpaged file 10-2

flow control 7-1

flow of execution 3-5, 3-7

flow path 2-3

format 1-1

formal parameters 3-17, 4-16, 4-18

FUNCTION 3-2, 3-5, 3-7, 3-8, 3-20, 4-18, 7-12, 11-3, 11-20

<function> 7-12

user-defined functions 6-27, 6-24

FUNCTION block
  Syntax Diagram #3 3-6

<function block> 3-17, 6-23, 6-24

FUNCTION block template 3-11

Function header statement
  Syntax Diagram #9 3-17

<function header> 4-4

function modules 3-1

<function template> 3-12, 3-17, 6-23

GO TO statement
Syntax Diagram #56 7-24

GO TO 7-25

@HEX 6-32, 6-34

HAL Data Types and Organizations 4-2

HAL/S Block Structure and Organization 3-1

hardware discretes 8-15

HEX 2-9, 6-32

simple header statements
Syntax Diagram #7 3-14

identifiers 2-5, 2-7

<identifier> 3-15, 3-17, 4-3, 4-10, 4-11, 4-25 11-40, 11-41

135

identifiers with NAME attribure 11-16

IGNORE 9-4

IF 7-2, 6-1

IF Statement
Syntax Diagram #45 7-3

implicit conversion 7-11, 7-13

implicit type conversion 6-26, 7-6

independent processes 8-2

infix operators 6-3, 6-4
(chart) 6-4

INITIAL	4-23
initialization	4-13
<initialization>	4-14, 4-15, 4-16, 4-23 11-17, 11-27
initial list	4-23
<initial list>	4-24, 4-25, 4-26, 4-27
initiation	8-2
IN <arith exp>	8-5
partial initialization	4-27
initialization specification Syntax Diagram #18	4-23
Inline Function Block Syntax Diagram #69	11-2
inline function	11-3
Inline Function Blocks	11-1
<§inline function>	11-2
input argument	7-10, 7-11
input/output	7-1
input/output statements	10-1
input parameters	3-15, 3-17, 6-24
I/O channel number	10-6
I/O control function Syntax Diagram #67	10-8
<I/O control>	10-3, 10-4 10-6, 10-7

INTEGER	4-19, 4-20, 4-24, 6-27, 6-28
integer	7-6
integer-valued literal	2-8
Introduction	1-1
Iterative DO FOR statement Syntax Diagram #54	7-21

iterative DO FOR
  with loop TEMPORARY variable index
  Syntax Diagram #54s
	11-14

keywords	2-6
<label>	2-7, 3-8 3-10, 3-11, 3-19, 3-20, 4-18, 6-23, 7-2, 7-3, 7-9, 7-23, 7-24, 7-25, 8-5, 8-14
<%label>	11-5
Label Name identifiers	11-20

Label declarative attributes
  Syntax Diagram #16
  with NAME
    Syntax Diagram #16s
	4-18
	
	11-19

label declarative <attributes>	4-12
LATCHED	4-13, 4-15, 4-21, 4-26
latched event	8-15
linear array	4-14
literals	2-5, 2-8
literal zero	7-6
LINE	10-3, 10-7, 10-8, 10-9

LOCK 4-13, 4-15, 4-16, 7-10, 8-19

lock group 7-10

LOCK(*) 8-19

Loop TEMPORARY variable 11-15
  Syntax Diagram #53s 11-14
  Syntax Diagram #54s 11-14

loop variable 7-19, 7-20, 7-22

machine units 8-3

mantissa 7-6

major structure 4-13, 4-16, 5-3, 5-9, 5-20

Matrix 5-21, 7-6,

MATRIX 4-19, 4-20, 4-25, 5-15, 6-27, 6-29

maximum index value 5-12

minor structure 4-8, 4-10, 4-17, 5-3, 5-20

minor structure node 7-11, 10-11

MU 8-3

multiple copies 4-22, 5-9, 5-13, 5-17, 5-20, 7-11

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

name scope 3-20, 4-3, 4-4, 4-6, 4-10, 4-12, 7-9

Name Scope Rules 3-20

name uniqueness 4-10, 4-12

NAME assign 11-29
  Syntax Diagram #74

NAME assignment statement 11-29
  Syntax Diagram #75 11-30

<name assign> 11-30, 11-32, 11-34

NAME assignment 11-35

NAME attribute 11-16
  Syntax Diagram #14s

NAME attribute in structure templates 11-22
  Syntax Diagram #13s

NAME attribure 11-23, 11-24

NAME conditional expression 11-30
  Syntax Diagram #76

NAME data and structures 11-34

NAME facility 11-16

name identifier 11-19
  Syntax Diagram #16s

NAME identifier
  label 11-20

NAME identifier 11-21, 11-23, 11-24, 11-27

simple NAME identifiers 11-25

dereferenced use of simple NAME identifiers 11-25

<NAME id> 11-29

NAME initialization attribute 11-33
  Syntax Diagram #79

NAME reference 11-26
  Syntax Diagram #73

<name reference> 11-30, 11-34

NAME (NULL) 11-28

Null NAME values 11-26

NAME pseudo-function 11-29

Referencing NAME values 11-26

NAME terminals 11-34, 11-38

NAME variable 11-17

natural sequence 4-24, 5-20, 6-28

nested blocks 3-8

Nested Structure Template References 11-23

NONHAL 4-18, 11-20

<normal function> 11-31
  with NAME
  Syntax Diagram #77

NOT 6-8, 6-9, 6-14, 6-19, 6-20

normal function 6-6, 6-24, 6-23
  Syntax Diagram #38

Null 4-7, 7-6, 11-28

Null character literal	2-10
Null field	10-5
Null statement Syntax Diagram #56	7-24
Null string	4-21
<number>	4-18, 6-6
@OCT	6-32, 6-34
object modules	3-1
OCT	2-9, 6-32
OFF ERROR	9-2, 9-4
one-dimensional source format	2-11
ON ERROR	7-2, 7-3, 9-1, 9-4 9-5
ON ERROR Statement Syntax Diagram #63	9-2 9-3
ON <event exp>	8-5
operand	5-20, 6-1
OR	6-7, 6-8, 6-13, 6-21
packing attribute	4-10
PAGE	10-3, 10-7, 10-8, 10-9
parametric replace reference Syntax Diagram #12.1	4-7 4-6
parentheses	2-12, 4-7
partitioning SUBBIT subscripts	6-36
precedence rules (chart)	6-5
%-macro	2-6, 11-1 11-6
arith	11-5
bit	11-5

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

char 11-5
struct 11-5
typeless 11-5

%-macro references 11-4
Syntax Diagram #70 11-5

%-macro CALL statement 11-11
Syntax Diagram #71

<%-macro call statement> 11-5, 11-11

pointer 11-16

<power> 2-8

precision specifier 6-38
Syntax Diagram #43

<precision> 6-6, 6-27, 6-29, 6-38

precision 6-38

precision conversion 7-6

primal process 8-2

HAL/S Primitives 2-1, 2-5

PRIORITY 8-6

PROCEDURE 3-2, 3-5, 3-7, 3-8, 3-15, 3-20, 7-9, 7-12 11-3, 11-20

PROCEDURE block 3-4
Syntax Diagram #3 3-6

PROCEDURE block template 3-11

Procedure Header Statement 3-15
Syntax Diagram #8 3-15

<procedure header> 4-4

<procedure template> 3-12, 3-15, 7-10

process events 8-18

<process event> 11-21

<process-event name> 2-7, 6-8, 6-9, 6-22

process queue 8-4, 8-10

Program 3-2, 3-14, 6-22, 7-12, 8-2, 11-19. 11-21,

PROGRAM block
Syntax Diagram #2 3-4

<program block> 3-5

Program block template 3-11

program complex 3-1

program header 3-4

program header statement 3-14

qualified structure 4-22, 5-3, 5-4

<radix> 6-31, 6-32, 6-33, 6-34

random-access I/O 10-1, 10-10

READ 6-36, 10-2, 10-4, 10-8

READ and READALL statements
Syntax Diagram #65 10-3

READALL 6-36, 10-2, 10-4, 10-8

ready state 8-4

ready 8-2

real time 7-1, 3-8, 11-3

real time control	1-2, 8-1
real time processes	8-2
Real Time Executive	8-1
REENTRANT	3-16, 3-18
regular expression	6-1
Syntax Diagram #23	6-2
referencing simple variables	5-2
referencing structures	5-3
REMOTE	4-13, 4-15, 7-10, 7-11 11-23
REPEAT Statement Syntax Diagram #56	7-24
REPEAT	7-25, 8-6
REPLACE	4-5, 4-6, 4-7 11-2
Syntax Diagram #12	4-4
<replace statement>	4-3, 11-42
reraveling	5-20, 6-28
RESET	8-16
Syntax Diagram #62	8-15
restricted character set	2-4
reserved words	2-5, 2-6
RETURN statement Syntax Diagram #48	7-12
RETURN	3-17, 3-19, 7-13
rounding	6-28
row and column dimensions (Matrix)	4-20, 6-29
RTE	8-1, 8-2 8-3, 8-4 8-11, 8-15

35

RTE-clock	8-3, 8-5, 8-6, 8-12
run time errors	9-1
S	5-9
S;	5-9
SCALAR	4-19, 4-20, 4-24, 6-27, 6-28, 7-6
scalar valued literals	2-8
SCHEDULE statement Syntax Diagram #57	8-4
SCHEDULE	8-5, 8-7
semi colon, use of	5-13, 7-24, 10-4, 10-5
SIGNAL, SET, and RESET	3-9
SET, SIGNAL, and RESET statements Syntax Diagram #62	8-15
SET, RESET, SIGNAL summary (chart)	8-17
SET, RESET, SIGNAL statements	8-18
SET statement Syntax Diagram #62	8-16 8-15
SEND ERROR Syntax Diagram #64	9-1 9-7
Sequential I/O	10-1
Sequential I/O statements	10-2
shaping functions	6-26
simple index	5-12
SIGNAL statement Syntax Diagram #62	8-16 8-15
single precision	6-15

SINGLE	4-19, 4-20, 6-29, 7-6
SINGLE (default)	4-20
SKIP	10-3, 10-7 10-8, 10-9
source macro	4-4, 4-6
source modules	3-1
source text	2-13
stall	8-2
stall state	8-4, 8-6
STATIC	4-13
STATIC (default)	4-14
STATIC/AUTOMATIC	4-14, 11-17
statement	3-4
<statement>	7-2, 9-5
STRUCTURE	4-10, 4-19, 4-21, 4-22
structure assignments	7-8
structure comparison Syntax Diagram #35	6-19
subscript construct Syntax Diagram #21	5-8
structure copies	4-27, 6-19
structure copy dimensions	5-17
structure copy specification NAME	11-18
structure expression	
Syntax Diagram #29.1	6-12
struct inline	11-10
struct & macro	11-10
Syntax Diagram #29.1s	11-10

<structure exp>	6-12, 6-19, 7-8
<struct inline>	11-4, 11-10
<struct %-macro>	11-5, 11-10
structure template statement Syntax Diagram #13	4-9
structure template tree diagram	4-8 4-8
structure template statement with NAME Syntax Diagram #13s	11-22
<structure template>	4-3, 4-10, 4-16, 4-17 11-42
structure terminal	4-10, 4-13, 4-16, 7-11
structure terminal references	11-34
subscripting structure terminals	11-36
structures containing NAME terminals	11-38
unarrayed structure terminal array structure terminal	5-9
non-qualified structure variable declaration	4-11
structure subscripts Syntax Diagram #22	5-11
structure type	11-18
structure types	4-27
Structure Subscripting	4-2, 5-7, 5-13
<structure sub>	5-7, 5-8, 5-13

E

135

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

<structure var name> 5-3

<structure var> 6-12

Subscripting 5-5
  Syntax Diagram #19

subscripting classes 5-7

component subscripting 5-15

legal subscript combinations 5-8

<subscript> 6-6, 6-27, 6-28, 6-29, 6-31, 6-34

<sub exp> 5-13, 5-14

<sub id> 11-26, 11-27, 11-33

subscript line 2-11

SUBBIT 5-6, 6-35

SUBBIT pseudo-variable 6-35
  Syntax Diagram #42

syntax diagrams 2-1, 2-2

SYSTEM 9-4

<sub name id> 11-26, 11-27, 11-33

systems language features 11-1

system-defined errors 9-7

"T" 6-4

TAB 10-3, 10-7, 10-8

TASK 3-5, 3-14, 4-18, 6-22, 7-12, 8-2, 8-6, 11-19, 11-21

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

TASK block	3-6
Syntax Diagram #3	
task block	3-4
<task block>	4-18
task header statement	3-14
TERMINATE statement	8-11
Syntax Diagram #59	
<template name>	2-7, 4-8, 5-3
TEMPORARY	11-12, 11-13
TEMPORARY statement	
Syntax Diagram #49s	11-12
Syntax Diagram #72	11-13
Temporary Variables	11-12
regular Temporary variables	11-12
THEN	7-3, 7-4
timing considerations	8-3
timing lines	8-15
TO	5-11, 7-22
TO-partition	5-12, 5-13, 5-14, 5-15
transpose	6-4
tree organization	6-19, 7-8
TRUE	2-9, 4-26, 8-16
Type Conversion	4-25

typeless %-macro	11-11
type specification Syntax Diagram #17	4-19
<text>	4-4, 4-5, 4-6
<type spec>	4-13, 4-14, 4-16, 4-17, 4-19, 4-20, 4-22, 11-3
two dimensional Source Format	2-11
unlatched event	8-15
UPDATE	3-5, 3-7, 3-14
UPDATE block Syntax Diagram #4	3-8
UPDATE PRIORITY statement Syntax Diagram #61	8-14
<update header>	3-8
update header statement	3-14
update block	3-4
<update block>	3-9
unqualified structure	4-22, 5-3, 5-4
unraveling	5-20
unit of compilation Syntax Diagram #1	3-2 3-2
UNTIL	6-1, 7-18, 7-19, 7-20, 8-6
UNTIL <arith exp>	8-7, 8-8

variable
  Syntax Diagram #20 — 5-5

unarrayed simple variable
  arrayed simple variable — 5-9

<variable> — 7-5, 7-6, 10-4, 11-40 | 135

<§var name> — 2-7, 5-2, 5-3

<§var name> (interpretation table) — 5-9

<§var> — 5-17

simple variable — 4-13, 4-16, 5-2

VECTOR — 4-19, 4-20, 4-25, 5-15, 5-21, 6-27, 6-29, 7-6

vector length — 4-20

wait — 8-2

WAIT statement
  Syntax Diagram #60 — 8-12

WAIT — 8-12

WAIT FOR — 8-13

WAIT FOR DEPENDENT — 8-13

WAIT UNTIL — 8-12

WHILE — 6-1, 7-19, 7-20, 7-22, 8-6

WHILE <event exp> — 8-7

WRITE — 6-2, 10-2, 10-8

  Syntax Diagram #66 — 10-6

/, use of 2-10, 2-13

#, use of 5-11, 5-12

¢, use of 2-10, 2-10.1
2-4

@, use of 2-4

UPDATE SHEET

Enclosed with each update package you receive is a new update sheeting listing the new Version number, the affected pages, and the date of the update.

As you receive each update, replace the old update sheet with the new one. It is important that you refer to the most recent Version number on the update sheet whenever you correspond with Intermetrics concerning this document.

VERSION	AFFECTED MATERIAL	DATE
IR-61-4		6/15/74
IR-61-5		12/5/74
IR-61-6		3/17/75
IR-61-7	This version of the HAL/S Language Specification contains material from the following approved Language Change Requests: LCR 114A BIT Function with Integer LCR 115 Extended Comparison to Allow Sorting LCR 133 Allow subscripts in NAME Initialization LCR 134 Remove Checking in DO CASE Statement LCR 135 EQUATE Statement LCR 136 HAL/S Shift Functions LCR 140 MIDVAL Function LCR 141 Document Clarifications	11/14/75

VERSION	AFFECTED MATERIAL	DATE
IR-61-7 (Con't)	The following pages have been up-dated or added: 1) Title Page 2) Table of Contents (Complete) 3) Page 3-16.1/blank page LCR 141 4) Page 3-18.1/blank page LCR 141 5) Page 3-19/3-20 6) Page 4-3/4-4 LCR 141 7) Page 4-5/4-6 editorial change 8) Page 4-13/4-14 editorial change 9) Page 5-17/5-18 editorial change 10) Page 5-19/5-20 editorial change 11) Page 6-17/6-18 LCR 115 12) Page 6-31/6-32 LCR 114A 13) Page 7-15/7-16 LCR 134 14) Page 7-19/7-20 LCR 141 15) Page 7-21/7-22 LCR 141 16) Page 8-5/8-6 LCR 141 17) Page 9-1/9-2 editorial change 18) Page 11-1/11-2 editorial change 19) Page 11-29/11-30 editorial change 20) Page 11-33/11-34 LCR 133 21) Page 11-39/11-40 LCR 135 22) Page 11-41/11-42 LCR 135 23) Page B-1/B-2 LCR 135 24) Page C-1/C-2 LCR 140 25) Page C-3/C-4 LCR 140 26) Page C-5/C-6 LCR 136 27) Page C-7/C-8 LCR 136 28) Page C-9/blank page LCR 136 29) Page D-1/D-2 LCR 141 LCR 114A 30) New Appendix G (Complete) Pages G-1 thru G-11 31) Page I-1/I-2 editorial change 32) Index - I-7/I-8, I-9/I-10, I-11/I-12, I-13/I-14, I-21/I-22, I-25/I-26, I-29/I-30 LCR 135 33) Index - I-25/I-26 editorial change	11/14/75

IBM Technical Newsletter

This Newsletter No.	GN26-0805
Date	April 30, 1976
Base Publication No.	GC28-6515-10
File No.	S360/370-25
Previous Newsletters	None

IBM System/360 and System/370 FORTRAN IV Language

© IBM Corp. 1965, 1966, 1968, 1971, 1972, 1973, 1974

This technical newsletter provides new and replacement pages for the subject publication. These replacement pages remain in effect for subsequent releases unless specifically altered. Pages to be inserted and removed are:

cover-preface	69-70
summary of amendments 1-3	79-90
contents	97-102
illustrations	117-126
11-14.1 (14.1 added)	131-136
23-24	157-160
49-58	165-back cover

Each technical change is marked by a vertical line to the left of the change.

Summary of Amendments

Changes include miscellaneous documentation changes and clarifications and also programming support for FORTRAN IV running under VSPC.

Note: Please file this cover letter at the back of the publication to provide a record of changes.

IBM Corporation, Programming Publishing, 1501 California Avenue, Palo Alto, California 94304

Printed in U.S.A.