Report No. 72-0005
Contract NAS8-26990

FLIGHT PROGRAM LANGUAGE REQUIREMENTS

VOLUME II

REQUIREMENTS AND EVALUATIONS

CR-123570

Prepared for:

NATIONAL AERONAUTICS AND SPACE ADMINISTRATION
George C. Marshall Space Flight Center
Marshall Space Flight Center, Alabama 35812

M&S COMPUTING, INC.

PREFACE

This report summarizes the efforts and results of a study to establish requirements for a flight programming language for future onboard computer applications. This study was performed by M&S Computing under contract NAS8-26990 from the Marshall Space Flight Center of NASA. The technical monitor was Mr. Richard Jenke, S&E-CSE-LI.

Several government-sponsored study and development efforts have been directed toward design and implementation of high level programming languages suitable for future aerospace applications. As a result, several different languages were available as potential candidates for future NASA flight programming efforts. The study centered around an evaluation of the four most pertinent existing aerospace languages. Evaluation criteria were established and selected kernels from the current Saturn V and Skylab Flight Programs were used as benchmark problems for sample coding. An independent review of the language specifications incorporated anticipated future programming requirements into the evaluation. A set of detailed language requirements was synthesized from these activities.

This report is the final report of the study and is provided in three volumes. This second volume describes the details of program language requirements and of the language evaluations.

Distribution of this report is provided in the interest of information exchange and should not be construed as endorsement by NASA of the material presented. Responsibility for the contents resides with the organization that prepared it.

Participating personnel were:

T. T. Schansman
R. E. Thurber
L. C. Keller
W. M. Rogers

Approved by:

J. W. Meadlock

TABLE OF CONTENTS

TABLE OF CONTENTS
(continued)

4. DETAILED LANGUAGE REQUIREMENTS

The general language requirements have been described in Section 3 of this report. This section discusses these requirements in detail. Some of the detailed requirements have more background information or implications which are too lengthy to be conveniently included in this section. Such requirements are specified in this section, but the supporting information is deferred to a discussion of special topics in Section 5.

To organize the presentation, the language requirements have been categorized into seven major groups. The groups and the major paragraphs in which their associated requirements are described are as follows:

A General Characteristics (Paragraph 4.1)

B Data Descriptions (Paragraph 4.2)

C Data Manipulation (Paragraph 4.3)

D Internal Program Sequencing and Control (Paragraph 4.4)

E Program Structure (Paragraph 4.5)

F External Data Access (Paragraph 4.6)

G Compiler Directives (Paragraph 4.7)

These major categories have been subdivided into more specific items under which the detailed requirements are described. Figure 4-1 is a complete list of the items covered. These are the same items under which the characteristics of the four evaluated languages were organized during Phase II of the study (Section 7).

These requirements are presented at three levels:

- A mandatory requirement is one which should be included in the initial specification and implementation of the language.

- A deferred requirement covers capabilities which are cost-effective for a large programming effort, but which are not immediately required in an initial implementation of the language.

LANGUAGE CHARACTERISTICS COMPARISON ITEMS

Par.	Title	Par.	Title
4.1.1	Statement Format	4.4.1	Direct Unconditional Transfers
4.1.2	Names, Labels, and Character Set	4.4.2	Variable Transfers
4.1.3	Interaction with Other Languages	4.4.3	Conditional Statements
		4.4.4	Decision Tables
		4.4.5	Iterative Loop Control
		4.4.6	Conditional Loop Control
4.2.1	Numeric Data Items	4.4.7	Statement Groups
4.2.2	Logical, Bit, Character Data Items	4.4.8	Stop Execution
4.2.3	Other Data Items	4.5.1	Overall Structure
4.2.4	Numeric Data Values	4.5.2	Programs
4.2.5	Logical, Bit, Character Data Values	4.5.3	Procedures
		4.5.4	Real Time Tasks
4.2.6	Other Data Values	4.5.5	Functions
4.2.7	Numeric Data Item Precision Attributes	4.5.6	Closes
		4.5.7	Program and Subroutine Returns
4.2.8	Presetting Data Values	4.5.8	Subroutine Parameters
4.2.9	Other Data Item Attributes	4.5.9	Priority Assignments
4.2.10	Alternate Data Declarations	4.5.10	Exclusive Subroutines and Interrupt Controls
4.2.11	Data Organizations	4.5.11	Error Recovery
4.2.12	Index Types	4.5.12	Library Subprograms
4.2.13	Default Data Item Characteristics	4.5.13	Scope of Names and Labels
4.2.14	Hardware Registers		
		4.6.1	Common Data
4.3.1	Arithmetic Operations	4.6.2	Compile Time Data Protection Features
4.3.2	Logical Operations		
4.3.3	Relational Operations	4.6.3	Execution Time Data Protection Features
4.3.4	Boolean Operations		
4.3.5	Explicit Data Conversions and String Operations	4.6.4	Conventional Input/Output
		4.6.5	Real Time Input/Output
4.3.6	Operations on Data Organizations	4.7.1	Optimization Directives
		4.7.2	Memory Allocation Controls
4.3.7	Assignment Statements	4.7.3	Program Debug Aids
4.3.8	Scaling of Intermediate Results	4.7.4	Compile Time Identifiers
		4.7.5	Macro Statements

Figure 4-1

- A desirable requirement is a nice-to-have feature whose cost-effectiveness is marginal and whose implementation must be judged in the light of actual cost.

The requirements discussions in the following paragraphs are segregated by the foregoing levels. Requirements are mandatory unless otherwise designated.

Three verbs are used in the description of these requirements and have very explicit meanings in this context:

- "Will" describes an explicit feature or characteristic required of the language, e.g., the language will include floating point data items.

- "May" describes an option which the language will allow the programmer to exercise if he wishes, e.g., a program may have multiple entry points.

- "Must" describes a rule to be followed by the programmer and enforced by the language syntax and/or its processor, e.g., each entry point to a program must be explicitly declared.

4.1 General Characteristics

These requirements reflect the general features and appearance of the language, and have an overall effect on the total language.

4.1.1 Statement Format

A source program consists of a sequence of individual statements, and all of the language requirements are ultimately reflected in what can or cannot be included in statements. The statement format requirements establish the overall appearance and structure of these statements.

The statement format requirements are:

- Each statement will be concluded by a delimiting character which is a natural punctuation mark. This requirement prohibits the use of symbols such as the currency symbol ($), percent sign (%), or commercial at sign (@) as statement delimiters.

- More than one statement may appear on the same

input line and any statement may extend over multiple input lines without continuation indicators.

- Each statement label will be concluded with a natural delimiting character which is different from the statement delimiter.

- Comments may appear anywhere that a blank may appear, within or outside a statement, except that comments may not appear within a string literal (Paragraph 4.2.5).

- All comments will be enclosed in begin and end delimiters whether they appear on the same input line as a program statement or on a separate line.

- There will be a facility controlled (as opposed to programmer-controlled) limit on the length of a comment. The intent of this limit is to allow the compiler to detect a missing end delimiter, and not pass over a large number of subsequent statements under the assumption that they are part of the comment.

These requirements are intended to result in statement syntaxes which are convenient to write, natural in appearance, easy to distinguish one from another, and prone to compiler detection of format errors.

4.1.2 Names, Labels, and Character Set

Names are the identifiers which are created by the programmer and assigned to the data items which he defines. Similarly, labels are identifiers which are created and assigned to statements and procedures. The character set is the set of symbols used to build the commands, operations, and other primitive elements of the language. A subset of these characters is made available for the programmer to create identifiers. The requirements are:

- Names and labels will be of limited character length, but the limit will be greater than 28 characters. This requirement allows the identifiers sufficient length to be descriptive.

- Names and labels may be made up of decimal digits (0-9), upper case letters (A-Z), and the "break" character (_). The "break" character is provided because

many identifiers are created as multi-word mnemonics, and the "break" character improves readability by separating the individual words within a single identifier.

- o Names and labels must begin with a letter. This requirement makes names and labels more readily distinguished from literal numeric values.

- o There will be two character sets in the language: restricted and extended. The language will be fully expressible in either character set. The restricted character set will include at least:

  - Decimal digits (0-9)
  - Upper case letters (A-Z)
  - Arithmetic symbols (+, -, *, /, =)
  - Delimiters (comma, period, apostrophe, left and right parentheses, blank, and break character).

  The extended character set will include the restricted character set and at least:

  - Relational symbols ($\geq$, $<$)
  - Logical symbols (&, $|$, $\neg$)
  - Delimiters (semicolon, colon)

The above requirements are intended to give the programmer sufficient freedom to generate expressive names and labels, and to provide a rich enough symbology that the operators and keywords of the language can also be expressive.

4.1.3 Interaction with Other Languages

There will be no direct interaction with other languages. That is, there will be no capability to write statements of another language in the same procedure with statements of the flight programming language. However, there will be nothing in the language to prevent a full two-way interaction with separate procedures written in other languages.

This isolation of other languages is necessary to allow the desired level of compiler verification for procedures. The use of assembly language, for example, in the midst of a procedure could easily violate the effectiveness of the usage rules built into the flight programming language.

4.2 Data Descriptions

4.2.1 Numeric Data Items

Numeric data items are those used primarily to represent quantitative values. The following are required:

- There will be floating point data items with programmer-specified precision.
- There will be fixed-point data items with programmer-specified precision and scaling.

Requirements for explicit precision controls are described in Paragraph 4.2.7. The integer data item has been specifically excluded from the requirements, because an integer is equivalent to a fixed point number scaled zero (i.e., the binary point is placed at the right end of the computer word, resulting in no fractional part). Therefore, the integer capability is easily made available through a compile-time identifier which is defined to mean a fixed point number scaled zero (see Paragraph 5.1).

4.2.2 Logical, Bit, Character Data Items

These data items are generally used to represent natural language messages and other coded information rather than numeric quantities. The following types are required:

- There will be bit-string data items with programmer specified length.
- There will be character-string data items with programmer-specified length.
- String lengths may be declared as fixed or variable. In a fixed length string, the value of each element of the string will always be defined. String-expression values will be "padded" as required with additional binary values or character-codes to fully define the receiving data item in every assignment. A variable length string will take on the length of the expression to which it is assigned. The values of elements beyond the current string length will be undefined.

- Variable length strings will have a programmer-specified maximum length. Any expression which exceeds the maximum length will be truncated to fit.

The variable length string capability is included, so that string values may be built up by concatenation and so that the execution overhead of "padding" the data item is not required. Boolean data items are not provided because they are equivalent to bit-strings of length one. Therefore, the Boolean capability is easily made available through a compile time identifier (see Paragraph 5.1).

4.2.3 Other Data Items

In addition to the numeric and string data, two types of location variables are required. Location variables allow a single statement to reference different data items or different statements depending upon the value of the location variable through which the reference is being made.

The following requirements specifically restrict the usage of location variables to retain static checking capabilities in the compiler:

- Data location variables will allow indirect reference to data items.

- Location values may be changed only by simple assignment to a data item name. This requirement is to ensure that the compiler (and the human reader) can determine, at the time of assignment of the location value, which data item will be indirectly referenced. Validity of the reference can be verified at location value assignment time, rather than indirect reference time.

- A data location variable must have the same data type and attributes as the data items being referenced through it. This requirement prevents implicit data conversion through indirect reference.

- Statement label arrays will allow a variable transfer statement (Paragraph 4.4.2) to pass control indirectly to any of several statements.

- The declaration of the statement label array will include the complete list of statement labels to which

control may be transferred through it.

These requirements are intended to provide some of the flexibility and programming power which comes from indirect addressing techniques, but to also restrict the capabilities sufficiently that the static checking capabilities of the compiler are not significantly degraded.

The real time task control functions (Paragraph 4.5.2) depend upon an event variable which can be used for intertask coordination. The requirements are:

- The language will include event variables which can be turned on or off (i.e., assigned one of two states) under program control.

- Event variable values (states) will be valid arguments of conditional statements. This requirement allows them to be easily tested under program control.

- Event variables will be valid conditions for the scheduling of a task or for a task to resume execution after it has been suspended.

4.2.4 Numeric Data Values

Numeric data values are the direct literal representation of numeric quantities in the language. These literal values are reflected in the compiled program as fixed point or floating point binary numbers.

The following requirements apply:

- All numeric data value literals will have the same general form, whether they are ultimately to be compiled in fixed point or floating point format. The compiled format will be determined from the context in which the literal is used.

- All numeric digits will be decimal digits. This requirement prohibits the use of binary octal and hexadecimal bases for numeric data values, but bit-string values (Paragraph 4.2.5) may incorporate them.

- A decimal point will be required only if the digit string includes a fractional part.

- The minus sign (-) will be used to indicate negative values and the plus sign (+) or omission of the sign will indicate positive values.

- The format will provide for specifying a power of 10 by which the digit string is to be multiplied to derive the final numeric value.

4.2.5 Logical, Bit, Character Data Values

The string data values must include both bit-string literals and character-string literals, as follows:

- Bit-string literals will be expressible as binary digit patterns, octal digit patterns, or hexadecimal digit patterns enclosed between begin and end delimiters.

- An explicit indication of the digit type will be part of the literal.

- Character-string literals will be enclosed between begin and end delimiters and may include any character recognized by the target system. This may include characters outside the character set defined for the language.

- The character-string delimiter may be represented within the character-string by including two adjacent copies of the delimiter character. (If, for example, the apostrophe were selected for the delimiter, then a double apostrophe ('') within a character-string literal would represent a single apostrophe character).

- Repetition factors for string literals will be provided to represent repeated patterns.

Boolean literals such as TRUE, FALSE, ON, OFF, are not provided because they can be defined through compile time identifiers (Paragraph 5.1).

4.2.6 Other Data Values

In addition to the numeric and string literals, there is also a need for literal values to be assigned to location variables:

- o Data location constants will be provided to represent the internal memory locations of data items.

- o Statement label constants will be provided to represent the internal memory locations of program statements.

- o The literal representation of a location constant will include the name of the data item, or label of the statement, being referenced.

4.2.7 Numeric Data Item Precision Attributes

Precision attributes of numeric data items determine the number of bits which will be used to represent values stored in the data item. For fixed point data items, location of the binary point (scaling) may also be specified. The requirements are:

- o Fixed point data item declarations will include specification of total word length (precision) and number of fractional bits (scaling).

- o Floating point data item declarations will include total word length.

- o All precision specifications will have default values.

4.2.8 Presetting Data Values

As data items are declared, it is frequently useful to assign them initial values through the declaration rather than through executable assignment statements. This is particularly true for constant data items which need not be changed during program execution. Requirements for presetting data values are:

- o Data items may be preset in their declaration statements with literal values, compile-time identifiers (Paragraph 4.8.4), or previously preset data items.

- o Non-scalar (multiple-element) data items may be initialized by a list of the above listed types of values.

A related deferred requirement is:

- o Data items may also be preset with arithmetic formulas consisting of the above listed types of values and the

arithmetic operators +, -, *, /, **

Related desirable requirements are:

- o Formulas for presetting data values may include standard arithmetic functions, such as trigonometric and logarithmic, as well as matrix and vector operations.

- o Data items may be declared constant if they are preset. This requirement allows the compiler to sometimes generate more efficient code and also allows the compiler to detect inadvertent attempts to reassign a constant.

4.2.9 Other Data Item Attributes

The protection of data which is external to multiple compilations (Paragraph 4.6.1) is enhanced if the compiler can determine which procedures are permitted to read and write which data items. Furthermore, during concurrent execution of separate tasks, legitimate accesses may be in temporary conflict because of the concurrency of execution.

The following requirements are intended to support the system data protection features required:

- o External data items will include, in their declarations, identification of each procedure within whose scope reading of the item is permitted, and each procedure within whose scope writing of the item is permitted. This requirement will allow the compiler to validate external data accesses.

- o Data items which are subject to access by separate tasks that may be in execution concurrently may be so identified. This identification will be used by the compiler to ensure that real time accesses to these data items are properly protected from conflicts. (Paragraph 4.7.3).

In addition, other requirements described in other paragraphs may be best implemented through data attributes, but that form of implementation is not intended to be specifically required.

4.2.10 Alternate Data Declarations

Several alternate forms of declaring data items may be incorporated into a language to make the coding of declarations easier.

The following requirements apply:

- Factored declarations will be provided, allowing common data types or attributes to be written only once but apply to multiple data items in the same declaration.
- Data item attributes will have defaults, most of which may be changed by the programmer.
- No implicit data item declarations will be permitted.

4.2.11 Data Organizations

Non-scalar data items are organizations of individual data elements into aggregates which can be treated as a whole.

The following non-scalar data items are required:

- Arrays of at least four dimensions will be provided.
- Matrices and vectors will be provided and will be structured and accessed in the same manner as arrays but will be subject to special arithmetic operations (Paragraph 4.3.1) which differ from the array operations.
- Data structures will be provided, allowing a hierachy of data levels and any combination of data types within a structure.

4.2.12 Index Types

Index values are used to identify specific elements or partitions within a data organization.

The following requirements apply to indices:

- Any scalar (i.e., single valued) arithmetic expression will be valid for an index.

- The capability will be provided to select the full range of any index value (e.g., an entire row of an array) in addition to selecting individual values.

4.2.13 Default Data Item Characteristics

Requirements for defaults for characteristics of data items are identified in Paragraph 4.2.10.

4.2.14 Hardware Registers

There are no requirements for data items declared within the language to identify specific hardware registers.

4.3 Data Manipulation

The language requires a variety of features to perform a broad spectrum of operations on the data types and organizations described in Paragraph 4.2. The following paragraphs describe the requirements for forming expressions with, and making assignments to, the required data types.

4.3.1 Arithmetic Operations

Arithmetic operations are considered in three classes:

- Scalar Operations
- Matrix/Vector Operations
- Array/Structure Operations

Scalar operations will apply to any combination of floating point, fixed point, and bit-string data elements and will include the following:

- Prefix minus
- Addition
- Subtraction
- Multiplication
- Division
- Exponentiation

Matrix and Vector operations will apply to combinations of matrices and/or vectors of either fixed point or floating point data elements and will include:

- o Prefix minus
- o Addition
- o Subtraction
- o Vector Dot Product (between two vectors)
- o Vector Cross Product (between two vectors)
- o Multiplication by another matrix or vector
- o Inverse of a matrix
- o Transpose of a matrix

The scalar operations will also apply between:

- o A scalar and an array
- o A scalar and a structure
- o Two arrays of identical dimension
- o Two structures of identical organization

The applicability of two-operand operations is summarized in Figure 4-2.

4.3.2 Logical Operations

Logical operations apply to bit-strings and character-strings. They manipulate these strings as patterns without attaching quantitative values to them. The following logical operations are required for bit-strings:

- o Complement, which changes all "zero" bits of the string to ones, and all "ones" to "zeroes".
- o Logical "AND" which combines two strings, bit-by-bit, generating a "one" bit where corresponding bits of the source strings are both "ones", and generating a "zero" bit otherwise.

APPLICABILITY OF ARITHMETIC OPERATIONS

Operand$_1$ Operation Operand$_2$

Operand$_1$ \ Operand$_2$	SCALAR	VECTOR	MATRIX	ARRAY	STRUCTURE
SCALAR Fixed Floating Bit-String	Scalar Operations	Scalar Operations[1] (Excluding exponentiation and division)	→	→	→
VECTOR Fixed Floating	Scalar Operations[1]	Cross Product Dot Product Addition Subtraction	Addition Subtraction Matrix Multiplication	None	None
MATRIX Fixed Floating	↓	Matrix Multiplication Addition Subtraction	Matrix Multiplication Addition Subtraction	None	None
ARRAY	↓	None	None	Scalar Operations[2]	None
STRUCTURE	↓	None	None	None	Scalar Operations[2]

NOTE 1: The operation will be performed between the scalar and each element of the non-scalar independently, resulting in a new non-scalar of the same organization and dimension.

NOTE 2: The operation will apply only to non-scalars of identical organization and dimension.

Figure 4-2

- Logical "OR", which combines two strings bit-by-bit, generating a "zero" bit where corresponding bits of the source strings are "zero", and generating a "one" bit otherwise.

- Concatenation, which attaches one source string on the end of another, creating a new string with length equal to the sum of the two source string lengths. Individual bit values are unchanged in the process.

In addition, the concatenation operation is required for character strings.

Operations which apply to strings will also apply to arrays of strings, so long as arrays to be combined are of identical dimension. They will also apply to structures of strings, so long as structures to be combined are of the same organization.

A desirable requirement for bit-strings is:

- "Exclusive OR" operation which combines two bit-strings, bit-by-bit, generating a "one" bit where corresponding source string bits are different, and a zero bit where they are the same.

4.3.3 Relational Operations

Relational operations compare the values of different operands and result in values of "one" if the relational expression is true, or "zero" if it is false. The result of such an operation is a bit-string of length one.

The following relational operations will be provided:

- Equal
- Not Equal
- Less Than
- Greater Than
- Less Than or Equal
- Greater Than or Equal
- Not Less Than
- Not Greater Than

All relational operators will apply to fixed point, floating point and bit-string operands in any combination. "Equal" and "Not Equal" will be the only operations which may compare character-strings and non-scalar data items; non-scalar data items being compared must be of the same organization and dimension.

Desirable requirements are the two relational operations:

- o Within Limits
- o Outside Limits

which would compare a single operand against a pair of operand limits.

4.3.4 Boolean Operations

Boolean variables are bivalued variables which can take on one of two states (e.g., one or zero, on or off, true or false). There is no language requirement for a data type explicitly called a Boolean variable, because a bit-string of length one (single bit) provides the full capability. Consequently, there is no requirement for operations which are explicitly called Boolean. However, all of the bit-string logical operations (Paragraph 4.3.2) will be applicable to single bit bit-string operands (i.e., Boolean values), including relational expressions. The relational operations (Paragraph 4.3.3) will also be applicable to single bit bit-string operands.

4.3.5 Explicit Data Conversions and String Operations

Implicit data conversions will be made among bit-strings, fixed point, and floating point data elements when they are combined through the arithmetic operations of Paragraph 4.3.1. The order of conversion will be from:

- o Bit-string (interpreted as an integer) to
- o Fixed point, to
- o Floating point

Furthermore, implicit conversion will be made across any legitimate assignment statement (Paragraph 4.3.7).

In addition there are requirements for explicit data conversions and manipulation of portions of string values. These requirements include:

- The ability to interpret any data element as a bit-string without modification of its bit pattern.

- The ability to extract any single contiguous substring from a character-string or bit-string.

- The ability to replace any single contiguous substring of a character-string or a bit-string with a string of the same type and length, leaving the remainder of the receiving string unchanged.

4.3.6 Operations on Data Organizations

The requirements for application of data manipulation operations to vectors, matrices, arrays, and structures are described along with the operation descriptions (Paragraphs 4.3.1 through 4.3.3).

4.3.7 Assignment Statements

Assignment statements are used to assign the value(s) of a formula to one or more data items, which include partitions of non-scalar data items, or substrings of strings. The following requirements apply to assignment statements:

- Assignment statements will be of the form:

  variable, variable, . . . variable = formula

- The formula may be any data item or any valid combination of data items using the data manipulation operations.

- One or more variables may be listed, where each listed variable must be of the same organization and dimension, and each will be identically assigned the the same value or set of values.

- If the variables are non-scalar, the formula may be a non-scalar formula, a list of scalar formulas, or a single scalar formula whose value is assigned to each element of the non-scalar variable(s).

- Any scalar data type may be assigned any other scalar data type and the necessary conversions will be made implicitly.

- Statement label arrays must not appear in assignment statements. They may be assigned values only through the initialization capabilities of their data declarations.

- Data location variables may be modified only by direct assignment to a single data item name.

- Vectors, matrices, and arrays, or partitions thereof, may be assigned to each other in any combination, but must be of the same dimension.

- Structures may be assigned only to other structures of the same organization.

A desirable requirement for assignment statements is the following:

- A structure assignment "by name" which would assign values only to those elements of the receiving structure which had the same name as elements in the source structure. Other elements of the receiving structure would remain unchanged.

4.3.8 Scaling of Intermediate Results

Fixed point data items will include, in their declarations, the precision (number of bits) and scaling (number of fractional bits) of values to be stored in the data item (see Paragraph 4.2.7). The compiler uses this information to align data values before performing arithmetic operations in expressions. In some cases, the programmer needs to generate an intermediate result in an expression which has scaling and/or precision different from that dictated by the precision and scaling of the operands. The following language requirements are included to fulfill that need:

- The language will provide the capability to specify a new precision for any intermediate value of an expression.

- The language will provide the capability to specify the precision at which any arithmetic operation will be carried out.

4.4 Internal Program Sequencing and Control

In addition to the data manipulation operations described above, the programmer needs to specify the sequence in which these operations are to be performed and the decisions which must be made to control that sequence. This paragraph describes requirements for language features which specify decision making and sequencing controls within a single procedure. The controls and interfaces among separate procedures are discussed in Paragraph 4.5.

4.4.1 Direct Unconditional Transfers

The direct unconditional transfer statement will be provided in the form of a "GO TO Statement-label" which will transfer control directly to the statement whose label appears as an argument. The following restrictions will apply to the statement-labels which may be used as arguments of the GO TO statement:

- The argument statement must be internal to the procedure which contains the GO TO statement.
- The GO TO statement may not transfer control into:
  - A DO group
  - A BEGIN block
  - A nested procedure

These requirements are intended to provide the programmer with flexibility in transfer of control within a procedure, but restrict the interfaces among procedures and selected statement organizations within procedures to reduce the complexity of the overall program structure.

4.4.2 Variable Transfers

A powerful part of the internal control flexibility which can be provided by the programmer is the ability to execute a variable transfer. This capability allows a single statement to transfer to any one of a list of alternative statements depending upon conditions at the time of execution. The language requirements are as follows:

- A label array (Paragraph 4.2.3), with an index, may be the argument of a GO TO statement.

- The value of the index at the time of execution of the GO TO statement will determine which statement-label from the label array will be selected for the transfer.

It should be noted that the label array elements cannot be assigned new statement label values; all values are established in the declaration and are fixed throughout program execution. The only variable is the label array index.

4.4.3 Conditional Statement

Conditional statements are provided in the language as the basic decision making tool. They allow either one of two responses to be executed depending upon the state of a logical-condition. The requirements are:

- The conditional statement will have three clauses:
  - IF clause
  - THEN clause
  - ELSE clause
- The IF clause may be any scalar expression and is interpreted as false if it evaluates to a zero value and true if it evaluates to a non-zero value.
- The THEN clause will be a statement or statement group which is executed when the IF clause is true.
- The ELSE clause will be optional. If included, it will be a statement or statement group which is executed when the IF clause is false.
- THEN and ELSE clauses which are statement groups must be enclosed between the delimiting DO and END statements.

4.4.4 Decision Tables

Decision tables provide the capability to compactly and graphically portray a complex combination of decisions and responses.

Decision tables will be included in the language and will fulfill the following requirements:

- Any number of relational conditions may be listed.
- Decision rules may be generated from any combination of the true or false or "don't care" states of the relational conditions listed.
- Any number of actions in the form of assignment statements or direct transfer statements may be listed.
- A response, consisting of any combination of the actions listed, may be attached to each decision rule and be executed if the decision rule is fully satisfied.

A desirable requirement associated with decision tables is the following:

- The ability to declare a decision table with a label and execute it remotely through reference to the label, rather than placing all decision tables in the direct line of program execution.

4.4.5 Iterative Loop Control

Iterative loop control provides a technique for executing groups of statements several times in succession using different values of a loop variable for each iteration. The requirements for iterative loop control are:

- The sequence of values to be assigned to the loop variable will be expressed as a list of independent values or value sets.
- A value set will be expressed as a specified initial value, increased by a specified increment for each iteration, until a specified final value is exceeded.
- Iterative loops may be nested within other iterative loops.
- Each iterative loop, whether nested within another iterative loop or not, must have its own unique END statement as its final statement.

- At termination of the iterations, the loop variable will retain its last assigned value, whether the termination was caused by a transfer out of the loop or by completion of the list of values.

- The initial value, increment, and final value of a loop variable value set may each be any valid scalar arithmetic expression, and such expressions will be reevaluated at the completion of each iteration.

- The loop variable may be assigned new values by statements within the loop.

A desirable requirement is the ability to specify multiple independent loop variables with independent sets of values for a single loop. These loop variables would each be assigned a new value for each iteration of the loop.

4.4.6 Conditional Loop Control

Conditional loop control allows the specification of a logical condition upon which to base termination of the iteration of a loop. The requirements are:

- Any iterative loop control statement may include a logical-condition which will be tested before each iteration of the loop. If the logical-condition is not true, the loop will terminate.

- A conditional loop may be iterated without specifying a loop control variable.

- A logical-condition may include any relational expression or logical combination of relational expressions.

4.4.7 Statement Groups

Statement groups allow multiple statements to be considered as a single execution unit. Statement groups are used in multiple statement THEN and ELSE clauses and in iterative loops. The requirements are:

- All statement groups must be delimited by the beginning delimiter DO and the terminating delimiter END.

- o A properly delimited statement group may be used at any place in the program where a single statement is valid.

4.4.8 Stop Execution

The language will not include a specific statement to halt execution of the computer. However, an individual task may suspend or terminate itself, returning control to the operating system (Paragraph 4.5.2).

4.5 Program Structure

The preceding paragraphs have discussed data access and statement execution sequence control within a programmed procedure. The ensuing discussions present the requirements for combinations of and interactions among groups of procedures. Again, it is emphasized that the terminology used here to describe the requirements is not intended to impose a syntactical requirement on the language. The concepts described are required, but there is no specific requirement for the names which must be used to reference them.

4.5.1 Overall Program Organization

The overall program organization capabilities must allow for multi-tasking through a highly modular structure which allows any module, at any level of the structure, to be further modularized into a substructure. It must also allow portions of the organization to be compiled separately and be efficiently linked together after compilation. In addition, the data and control interfaces among modules must be straightforward so that the organization and module interactions are easily understood and can be reliably maintained. The specific requirements are:

- o An overall program organization will include one or more external programs, which are separately compilable procedures that may contain internal procedures nested to any level. (Note: Unless explicitly restricted to internal procedures, any requirement which will apply to a "procedure", will also apply to a "program".)

- o A procedure (internal) or program (external) may be invoked as a:

  - Called procedure through a calling statement, allowing multiple parameters to be communicated

to and from the called procedure (Paragraph 4.5.3).

- Function procedure by appearance of the procedure name in an expression, allowing multiple parameters to be transmitted and a single parameter to be returned (Paragraph 4.5.5).

- Task through a task calling statement, allowing for concurrent execution with the invoking procedure and the passing of parameters to the invoked procedure (Paragraph 4.5.4).

o An overall program organization may also include a set of external data items which are accessible to all procedures (assuming proper authorization, Paragraph 4.6.2). Any data accessible to multiple tasks which can be in concurrent execution must be external data.

4.5.2 Programs

The program level of organization will be the highest declared level in the structure, and multiple programs may exist in the same organization, although they will be separately compilable. The following requirements apply to programs:

o The language statements which specify a program will be enclosed between a "begin delimiter" statement and an "end delimiter" statement.

o The "begin delimiter" statement will be labeled with the program label and the "end delimiter" statement will reference that label.

o A program may have multiple entry points, and each entry point must be explicitly declared. For each declared entry point, there must be a statement within the program which has the declared entry point label.

o A program may be scheduled (as a task) to be executed at some time in the future as a function of:

- An absolute time

- An increment of time from the instant of scheduling

- A combination of programmer declared as well as system defined events.

o A program may be scheduled (as a task) to be executed repeatedly at some periodic rate.

o While in execution, a program may suspend its own execution until any one of the above three types of occurrences.

o A program which is scheduled for execution as a task, and has not yet started execution, may be cancelled by another task. Such cancellation will remove the schedule request.

o A program in execution may terminate the task under which it is running.

o The language statements within a program must be organized in the following sequence:

- Beginning delimiter
- Declarative statements
- Imperative statements
- Nested procedures
- Ending delimiter

o The program attributes recursive and re-entrant will be provided in the language.

4.5.3 Procedures

The language requirements for a procedure definition capability are the same as for a program, except that:

o A procedure must be nested within a program, and may not be separately compiled.

A deferred requirement for procedures is:

o Procedures may be declared with the inline attribute, in which case the compiler will duplicate the object

code at the point of invocation, rather than generating a calling sequence to object code.

4.5.4 Real Time Tasks

Real time tasking will be provided through the ability to schedule programs and procedures for future and concurrent execution (Paragraph 4.5.2). The following restriction will apply to procedures which are to be scheduled as tasks:

- Data which is to be accessed by multiple tasks which can be in concurrent execution must be external data.

4.5.5 Functions

A function is a procedure which is invoked by the appearance of its label, and input parameters, within an expression. The following requirements apply:

- A procedure may be invoked as a function, as well as through a separate calling statement.
- When a procedure is invoked as a function, its only output is the value assigned to the function label where it appears in the invoking expression.

4.5.6 Close

A close is a mechanism for providing very simple and efficient subroutine interfaces. There is no explicit language requirement for closes, but there is a compiler consideration to provide a more efficient linkage for procedures which neither declare data internally nor communicate input and output parameters.

4.5.7 Program and Subroutine Returns

When a program or a procedure (subroutine) is invoked it can become part of the task which invokes it, or if it is scheduled (Paragraph 4.5.2), a new task is created. Corresponding to the two types of invocation, there will be two ways to return control when a program or procedure has completed its operations:

- A terminate statement will return control to the operating system, removing the task under which it was executing.

- o A return statement will return control to that program or procedure which directly invoked it. If it was invoked through a schedule statement, then the return is to the operating system and the effect will be the same as for the terminate statement.

- o If the invocation was a function invocation, the return statement must include an expression which represents the function value to be returned.

4.5.8 Subroutine Parameters

When a procedure is invoked it may operate upon, and generate values for, data items from two general classes: data items which are "known" to it through declarations (Paragraph 4.5.13) and data items which are passed to it or returned by it as parameters at the time of invocation and return. The following requirements apply to the transmission of these parameters:

- o A procedure invoked directly through a separate procedure call statement may receive multiple parameters and may return multiple parameters, as named in the calling sequence.

- o A procedure invoked directly through a function invocation may receive multiple parameters, as named in the calling sequence, but may return only one parameter.

- o A procedure invoked by scheduling for future concurrent execution may receive multiple parameters, but may not return any parameters.

4.5.9 Priority Assignments

In a multi-tasking environment, a priority assignment is frequently used to establish a precedence order for initiating and/or executing tasks. The absolute execution priority should be established outside the coding of the procedure. However, it is sometimes useful for a procedure, when scheduling a task, to establish a priority relative to itself and to other tasks it schedules. Therefore, the following language requirement is imposed:

- o The statement which is used to schedule a task will include provision for a numerical priority to be

assigned to the scheduled task. The precise meaning of the priority number will depend upon the real time operating system implementation.

4.5.10 Exclusive Subroutines and Interrupt Controls

In addition to the normal priority controls used for sequencing tasks there is a need for selected temporary overriding of these priorities. Two specific techniques are required in the language; exclusive procedures and interrupt controls. The requirements are:

- An exclusive attribute will be provided for procedures, such that if one task invokes an exclusive procedure, no other task regardless of priority may execute the procedure until the first task has completed execution of it.

- The capability will be provided to selectively inhibit interrupts individually or in groups.

- The capability will be provided to selectively enable inhibited interrupts individually or in groups.

4.5.11 Error Recovery

When a system error, such as arithmetic overflow, occurs the computer hardware or operating system frequently has a standard response. Sometimes it is useful to replace the system response with a programmer defined response. It is also useful to have error conditions defined and controlled by application programs in addition to the system defined errors. The requirements are as follows:

- A set of system-defined errors will be named in the language so they can be referenced in program statements.

- The programmer will be able to declare error conditions which he may then cause to occur through program statements.

- The language will include the facility for assigning a statement sequence to an error condition or an interrupt.

A desirable requirement is:

- The ability to reassign the response to an error condition within a procedure at execution time. This capability would obviate the need for deciding, within a single response sequence, which action should be taken.

4.5.12 Library Subprograms

Several classes of subprograms (procedures) are normally used in the development of a software system. Earlier paragraphs have described capabilities the programmer will have for writing his own procedures. In addition, there are built-in functions, which are an integral part of the compiler. Many of the requirements specified in this document are implementable through built-in functions. Between these extremes there is a need for commonly used procedures which can be defined and controlled centrally. The specific requirements are:

- The language will provide for an interface with a library of common procedures.

- Library procedures will be accessible by programmers through the normal procedure invoking capabilities of the language, without the programmer defining the procedures or explicitly declaring their existence.

4.5.13 Scope of Names and Labels

The scope of a name or label is that set of procedures in which the name or label is "known". That is, the set of procedures whose statements can use the name or label as an argument. Name and label scopes will be as follows:

- Data item names will be known within the procedure where they are declared and within all procedures nested in the declaring procedure.

- External data item names will be known to all procedures.

- Internal statement labels will be known only to the procedure in which they are declared. They will not be known in nested procedures no in procedures in which the declaring procedure is nested.

- o Procedure entry point labels will be known to:
  - The procedure to be entered by the label (the host procedure).
  - All procedures nested within the host procedure.
  - The procedure in which the host procedure is declared.
  - All procedures declared within the same procedure (and at the same level) as the host procedure.
- o Entry points to external procedures (programs) will be known to all procedures.

4.6 External Data Access

External data is data which is defined outside the compilation of a program module. It falls into two classes: data outside the computer (input/output) and data which is stored in computer memory but is common to two separate program compilations or two concurrently executing tasks. Special language features are required to ensure that this data can be made available where it is needed and that data communications can be maintained reliably.

4.6.1 Common Data

Data which is common to separately compilable program modules requires special attention because the separate compilations must be linked to the data outside the compiler and because it is necessary to ensure that separately compiled procedures are attempting to access the same data in the same manner.

Data which is common to procedures which may execute under separate concurrent tasks also needs special attention because improper sequences of independent access can produce erroneous results.

The language must allow for centralization of this data through the following requirements:

- o The language will allow for a common pool of external data, potentially accessible by all procedures of a

program organization.

- The language will provide a mechanism for retrieving a common data pool, by name, from a central source and incorporating it into the compilation process. Data declared in that common data pool will not require additional declaration within the procedures being compiled.

4.6.2 Compile Time Data Protection Features

Frequently, individual data items declared in a common data pool are actually accessed by only a few of the procedures with which they are compiled. To protect against erroneous access by other procedures, common data items need associated authorizations to identify procedures which have legitimate access to them. The requirements are:

- Data which is to be accessed by two or more separately compiled procedures must be declared external and may be retrieved through the common data pool.

- All external data items will include read authorization and write authorization attributes.

- Any procedure identified in a read (write) authorization, or any of its nested procedures, will be permitted to read from (write into) the corresponding data item.

4.6.3 Execution Time Data Protection Features

Although two procedures may be compiled together or may have independent authorization to read and/or write a data item, access conflicts can arise at execution time if these two procedures are in concurrently executing tasks. Therefore, execution time protection features are required to ensure that authorized data accesses are executed in non-conflicting sequences. The requirements are:

- An access protection attribute will be provided for external data items which indicates that they are accessible by two or more tasks which are subject to concurrent execution. A data item with such an attribute will be accessible only from within blocks of statements called access protection blocks.

- o Beginning and ending delimiters will be provided to delimit instructions within access protection blocks. These delimiters will dictate compiler generation of object code, which at execution time will suspend execution of a protection block, if necessary, until other protection blocks with conflicting accesses have completed execution.

- o The following restrictions will be imposed on access protection blocks to avoid deadlocks:

    - No branches into or out of access protection blocks except through the beginning and ending delimiters.

    - No nesting of access protection blocks.

    - No task suspensions in an access protection block.

    - No protection blocks in exclusive procedures.

4.6.4 Conventional Input/Output

A variety of conventional peripheral equipment can be anticipated for future spaceborne computer systems. Printers and on-line terminals with keyboard inputs and printed or CRT outputs are expected for man-machine interfaces. Record-oriented devices such as tape, disk, and drum are expected for bulk data storage. To support this range of devices, two broad classes of input/output are required:

- o The language will provide for character-string input/output which will treat the data as a continuous string of characters and will convert groups of characters to specified data types (and vice versa). Data positioning, such as line numbers, page controls, and character positions, will be provided.

- o The language will provide for record-oriented input/output which will treat data which has been structured into records, which in turn are organized into files.

Record-oriented input/output will include the following capabilities:

- Blocked records so that many logical records may be maintained in a single physical record.

- Synchronous operations, where the input or output operation is complete before control is returned to the procedure.

- Asynchronous operations, where control is returned to the procedure as soon as the input/output operation is started, and an event is created through which the procedure can determine when the operation is completed.

- Consecutive files, where each input/output operation refers to the next consecutive record on the file.

- Direct access files, where each operation can specify the relative number of the record desired and access it directly, independent of the last record accessed.

- Indexed files, where records can be accessed based on their data content in a "key" field. Access may be sequential, where the next operation refers to the record with the next highest value in the key field, or it may be direct where the operation specifies the key field value of the record to be accessed.

- Buffered operations for sequential input/output, where the operating system can maintain overlapped input/output because the next record to be accessed is predictable.

- The ability to declare data files centrally and associate them logically with devices, so that the input/output operations themselves refer only to data file names and are fully dissociated from any "hard coded" device identifications.

4.6.5 Real Time Input/Output

Real time input/output refers to communication with the avionics equipment and other space vehicle related devices which interface with the onboard computer but are not included in the "conventional" class. The input/output requirements described in Paragraph 4.6.4 are suffiencient for performing the real time input/output, so long as

the device names can be used in file declaration statements. The real-time devices can be treated as individual files or as separate records within a large file, depending upon implementation.

However, there are some desirable language requirements which would enhance the real time device interface in some environments:

- o Subscripted file names and/or record identifications. One common characteristic of spaceborne hardware is redundancy. The availability of a subscript (as an array subscript) in an input/output operation could simplify the process of switching to a backup device in case of failure. Only the value of the subscript variable would need to be changed to switch devices. It could also improve the quality of expression and readability of procedures which must read multiple redundant devices and compare their values.

- o File names in decision tables, which would imply obtaining a new value from the device each time the decision table is executed. This capability could greatly simplify the specification of monitoring requirements on vehicle test parameters.

4.7 Special Compiler Directives

Special directives to the compiler provide selected additional capabilities beyond those described in preceding requirements categories. They are described in the following paragraphs.

4.7.1 Optimization Directives

Code optimization remains an important issue for onboard programming, although it is primarily a compiler consideration rather than a language consideration. That is, the compiler must devote some of its efforts to object code optimization, independent of specific features in the language. What the language needs is the capability for the programmer to tell the compiler when to optimize and when not to optimize. The requirements are:

- o The language will provide a directive which directs the compiler to perform a specified level of optimization for a given compilation.

- At least one of these levels will direct the compiler to incorporate optimization techniques which will specifically reduce the execution time of the compiled code at the expense of increased memory. This is in addition to the memory allocation requirements of Paragraph 4.7.2.

A desirable requirement is the ability to specify different levels of optimization for different procedures within the same compilation.

4.7.2 Memory Allocation Controls

Memory allocation controls are a special class of optimization directives. They direct the compiler in allocating computer memory to declared data items, allowing the programmer to select trade-offs between program memory size and program execution time. The requirements are:

- Data packing directives will be provided to select the packing density of data items within data organizations.

- Automatic storage allocation will be provided, whereby selected data items may be assigned storage only while the declaring procedure is in execution. While the procedure is idle, the storage will be available for other uses.

4.7.3 Program Debug Aids

Some type of on-line debug capability which does not depend upon compile-time debug directives is required as part of the total flight programming system. This capability must be on-line so the programmer can modify his debug requests during program testing without returning to the language compiler. However, the on-line capability by itself is insufficient. There will be some cases where batch mode operations will be the most economical approach to debug, at least initially, and compile time debug directives will be needed. They should be fully compatible with the counterpart on-line controls. The debug directives in the language will include:

- Statement trace capability which will list identifications of statements in the sequence that they are executed.

- Branch trace capability which will list identification of branch statements in the sequence that they are

encountered and will indicate which branch was taken.

- o Output of selected data values by the following criteria:
  - On command
  - Whenever the data item is assigned a value
  - Whenever the value goes out of (or into) an indicated range between two limits.

4.7.4 Compile Time Identifiers

The compile time identifier capability allows the programmer to centrally define values, terms, expressions and other character-strings which may be used many places elsewhere in his program source statements. He may assign a simple identifier to the defined character-string and then use the identifier throughout his program. The language processor, in a pre-compilation phase, will substitute the defined character-string for each appearance of the identifier in the source program. This capability is useful because the compile time identifier can be more descriptive or easier to write than the defined character-string, and because the central definition allows the character-string to be easily redefined without modifying all the source statements where it is used. The requirements are:

- o Compile time identifiers will have the same syntax as data item names.
- o Compile time identifiers can be defined as equivalent to any character-string and will be replaced directly by the equivalent character-string prior to compilation.
- o The compile time identifier declaration will have the same scope rules as a data item declaration.

4.7.5 Macro-Definitions

As any large-scale programming effort gets under way many common operations, each performed in many procedures, become apparent. It is generally desirable to have such operations always performed in the same manner, readily specified by the programmer and easily recognized in the source program listing. Macro-statements are an effective way of accomplishing these goals. The following

deferred language requirements are intended to provide a general macro-statement capability:

- The language will provide a macro-statement capability, through which new language statements may be defined as sequences of other language statements.
- Macro-statements will be expanded in line where they appear in the source listing.
- Macro-statement definitions may include formal parameters. Actual argument identifiers, specified in the use of the macro-statement will be directly substituted for the formal parameters, when the macro-statement is expanded.

5. SPECIAL TOPICS

The benefits of the detailed language requirements, described in the preceding paragraphs, can be realized fully only if supporting requirements are imposed on other components of the overall flight program development system. The language compiler and the on-board computer operating system are the two components to which the success of the language is most sensitive. Some of the more significant requirements imposed by the language on these two software components are described in the following paragraphs.

5.1 Language Processor

Certainly the overriding requirements on the language processor are reflected directly in implementation of the language which fulfills the requirements of Sections 3 and 4. Since many of the requirements reflect rules for using language capabilities, rather than just additional capabilities, an important characteristic of the compiler is its validity checking. The language has been designed to provide the compiler with explicit information about what the programmer is intending to do. The compiler must determine that related program statements are consistent with one another and should be able to provide informative diagnostic data to the programmer when an error does occur. Indeed, advanced compiler versions may well provide error correction capabilities for the simpler punctuation omissions, misspellings, and other easily diagnosed errors. Although this capability can tend to encourage "sloppy coding", it can be more beneficial than harmful if supporting management tools are available to enforce programming disciplines.

The compile time identifiers of Paragraph 4.7.4 are particularly useful for very simple extensions to the language. For example, the following very useful key words can be defined into the language at a particular installation or by an individual programmer:

- INTEGER (a fixed point data item with default precision and scaled with zero fractional bits)
- BOOLEAN (a bit-string with a fixed length of one bit)
- TRUE (a binary "one")
- FALSE (a binary "zero")

- PI (3.14159)

To maximize the benefits of this capability the compiler (and possibly other support software) must provide a simple technique by which a facility may introduce a standard set of compile time identifier definitions directly available to (or forced upon) application programmers. The language processor must also provide efficient compile-time substitutions so that the cost of compiler execution does not render compile time identifiers uneconomical.

Since the macro statement capability is essentially an extension of the compile time identifier capability, the above arguments apply at least as much to macro statement processing.

Compiler optimization is an essential ingredient in the economics of flight programs. The computer memory and processing time savings for the object program can make the optimization capability very cost effective. However, the costs of using an optimizer (above and beyond the compiler development cost) can be high:

- Compiler execution time is higher
- Object code is less easily related to source statements, so the program debug is more complicated.

Therefore, at least during early development of a program module, optimizing functions should be inhibited through the capabilities described in Paragraph 4.7.1. The time-oriented optimization is required for frequently executed procedures or those with a critical response time requirement.

5.2 Operating System

Several of the language requirements imply a real time operating system interface to fulfill their intent. Fulfillment of these requirements involves trade-offs between object code inserted by the compiler and resident operating system routines which are invoked through calling sequences inserted by the compiler. These trade-offs are not discussed here, but requirements are established for functions which should be provided by a combined effort of the compiler and the operating system.

One of the more obvious of these requirements is multi-tasking, including real time task scheduling. It is assumed that, through an

operating system interface, a procedure can request that a new task be created, and that a specified procedure be executed under that task. The request can be for immediate creation of the task or creation at some future time or event. Parameters may be passed to the task by the invoking procedure whether the task creation is immediate or in the future. After a task has been created, it is assumed that a procedure executing under that task may terminate the task or temporarily suspend it. A task schedule request can be cancelled by any procedure before the scheduled task begins execution. Some sort of numeric priority associated with task execution is also assumed.

Of course these assumptions may not all be valid for a given operating system. Therefore, the language should be designed with these features independent of each other so that when the capabilities are provided in an operating system the language can invoke them, but when selected capabilities are not provided, the corresponding features can be eliminated from the language without reducing other features.

Multiprocessing is another capability anticipated in future computer systems and their operating systems. However, there are no language features directed specifically at multiprocessing. They are omitted because the individual program modules should not be directly concerned with multiprocessing considerations. There are sufficient opportunities to exploit parallelism among separate program modules without attempting to describe parallel paths within a module. Furthermore, the parallel execution of separate modules should be described outside the coding of the modules, and therefore outside the flight programming language.

Another area of operating system interface is the real time protection of external data. This protection is invoked by the language through the data access protection blocks of statements (Paragraph 4.6.3). The implementation of this protection requires execution time checks of data accesses to prevent conflicting accesses from separate tasks. While these checks could be performed entirely with object code inserted by the compiler, they may be accomplished more efficiently through centralized code in the operating system. Some current approaches to multi-tasking advocate creating private copies of external data for tasks when they are created and updating the common copy only when the task terminates. Of course, such an approach would obviate the need for special language designation of data access protection blocks.

Input/output is another area where operating system capabilities

vary widely. The full set of language requirements for input/output (Paragraph 4.6.4) implies an operating system data management system of considerable sophistication. As with real time task scheduling, the requirements are separable and can be implemented incrementally to correspond to the features available in the operating system.

6. LANGUAGE EVALUATION BY CATEGORY

The evaluation summaries provided in the preceding discussion showed the ratings of the languages in each of the evaluation categories and briefly introduced some of the key factors which influenced those ratings. The following paragraphs provide a more detailed discussion of the evaluations in each category. These discussions are not intended to expose every detail which influenced the evaluation ratings. Rather they are intended to emphasize and explain those features which most influenced the evaluation and which best represent the overall capability of the languages. Further details, which form much of the basis for these discussions, are provided in Section 7, which summarizes and compares detailed features of the language, and Section 8, which reviews the kernel coding effort.

6.1 Quality of Expression

The quality of expression of a language is its most important evaluation criteria category, because it reflects how easily and reliably the programmer can describe a desired process. It determines what programming techniques can be invoked by the programmer in describing his problem. This description activity is the first step in the program development process to be directly influenced by the programming language. Furthermore, the product of this activity is the source language program which is the object of the other evaluation criteria. If effective programming techniques cannot be invoked or if the description of their use is not well expressed, then every evaluation category is adversely affected.

Phase I of the study identified three major problems associated with developing the expression quality of a language. These problems and reflections on them resulting from the evaluation are summarized below:

- A written program usually must specify "how" a process is to be performed rather than simply "what" is to be done.

  Trigonometric and matrix and vector operations are the only significant language capabilities found which attempt to specify high level functions frequently used in flight programming. For example, no special features were provided for specifying digital filters

or integration techniques. This problem relates to readability and is discussed further in Paragraph 6.5.

- Quality of expression is in conflict with other evaluation categories.

  As expected, readability and expression found many beneficial features in common, and readability was even improved by some of the features which reduced the amount of writing required. Decision tables, and factored declarations are notable examples. The major conflict among evaluation criteria arose between expression and process reliability. The greater the freedom and power available to the programmer, the more opportunities there are to introduce programming errors and the more difficult they are to detect. The power of the indirect addressing capability provided by SPL leads the list of these conflicts.

- Constraints imposed by the host computer input/output media reduce the power of expression of the language.

  This problem was in evidence primarily in limiting the length of comments which were on the same line as statements. Line length did not restrict expression in the statements themselves for the sample coding, because each statement generally fit well within available characters on a card image. All the languages provided some capability for continuing the statement on the next line if necessary. However, when a comment must be continued, it loses much of its impact. The scope of such a comment is frequently unclear. Does it apply to a preceding statement (or several statements) or a following statement? If several statements, how many? The result is that comments are frequently abbreviated so they will fit on the same line as the statement, and the abbreviation reduces the value of the comment. The very rich HAL character set created some minor problems with equipment and operators, as discussed in Paragraph 6.1.2.5.

The comparative evaluations presented below are categorized under the following evaluation criteria questions:

- What programming techniques can be expressed in the language?

- How naturally or easily can the available techniques be expressed?

- How general is the form of the concepts presented by the language?

The major issues associated with each of these questions, and features of the selected languages which contribute to or detract from these issues, are discussed in the following paragraphs.

6.1.1 Techniques Expressed

The major issues concerning programming techniques which can be expressed are the following:

- Decision Tables

- Location Pointers

- Vector and Matrix Operations

- Data Preset Symbolically and by Formula

- Subroutine Control and Communication

The differences among the languages is these areas are discussed and evaluated in the following paragraphs.

6.1.1.1 Decision Tables

SPL is the only one of the evaluated languages which provides Decision Tables. This is one of the outstanding differences in expression among these languages, and has been used very effectively in commercially available languages in the past.

Decision tables were used advantageously several times in four of the 11 kernels coded. They were particularly beneficial to the Switch Selector kernel because of the many complex combinations of decisions and actions associated with Switch Selector sequences.

The power and operation of the Decision Table are best described by example. Figure 6-1 shows a simple Decision Table taken from the SPL coding of the Switch Selector Processor. Below

DECISION TABLE EXAMPLE

SPL Decision Table

```
                                                          (Rules)
CONDITIONS                                                ┌─────┐
      VSNA1 EQ MSKSSHIG                          , (Y,  ,  ,  ,  )
      VSNA1 EQ MSKSSLOG                          , (  , Y,  ,  ,  )
      VSNA1 EQ MSKSSOMG                          , (  ,  , Y,  ,  )
      VSNA1 EQ MSKSSSIVB                         , (  ,  ,  , Y, Y)
      FBRNI EQ 'FIRST'                           , (  ,  ,  , Y, N)
ACTIONS
      DVMC7 = DVMC7 LOR MSKMC7HIG   , (Y,  ,  ,  ,  )
      DVMC7 = DVMC7 LOR MSKMC7LOG   , (  , Y,  ,  ,  )
      DVMC7 = DVMC7 LOR MSKMC7OMG   , (  ,  , Y,  ,  )
      DVMC5 = DVMC5 LOR MSKMC54B1I  , (  ,  ,  , Y,  )
      DVMC6 = DVMC6 LOR MSKMC68BRI  , (  ,  ,  ,  , Y)
ELSE  RETURN
```

Equivalent Logic Using IF. . . . THEN . . . ELSE

```
IF VSNA1 EQ MSKSSHIG
      THEN DVMC7 = DVMC7 LOR MSKM7HIG
           GOTO SSEXIT
END
IF VSNA1 EQ MSKSSLOG
      THEN DVMC7 = DVMC7 LOR MSKM7LOG
           GOTO SSEXIT
END
IF VSNA1 EQ MSKSSOMG
      THEN DVMC7 = DVMC7 LOR MSKM7OMG
           GOTO SSEXIT
END
IF VSNA1 EQ MSKSSSIVB
      THEN IF FBRNI
                THEN DVMC5 = DVMC5 LOR MSKM54B1
                ELSE DVMC6 = DVMC6 LOR MSKM68BR
           END
END
GOTO SSEXIT
```

Figure 6-1

the table is the same logic expressed in a sequence of typical (CLASP) IF . . . THEN . . . ELSE statements. The CONDITIONS portion of the table describes five equality (EQ) relations each of which is either true or false on a given execution of the table. Below the conditions, there are five ACTIONS (there could be any number), which are performed selectively depending upon the truth or falsity of the individual conditions.

On the right side of the table, are the "rules" that determine which actions will be taken for various combinations of conditions. Each rule is a column of Y (yes), N (no), or blank indicators. Each rule (column) is read as follows:

> "If all the conditions marked Y are true, and if all the conditions marked N are false, then perform (sequentially from top to bottom) all the actions which are marked Y."

Rules are evaluated from left to right until the first satisfied rule (Y-conditions all true; N-conditions all false) is encountered. The action statements specified in that rule are executed and (unless one of the action statements is a GOTO) execution proceeds at the statement following the table. If none of the rules are satisfied, the ELSE statement at the end of the table is executed and (unless the ELSE statement is a GOTO) execution proceeds at the statement following the end of the table.

Figure 6-1 is a very simple table where only the fourth and fifth rules combine combinations. The fourth rule executes the fourth action only if both the fourth and the fifth conditions are true. (An implied additional constraint is that the first three conditions be false because they are covered by preceding rules).

Decision tables have the slight disadvantage that they constitute a new concept to many programmers. However, they are very straight-forward in principle and in application, and give SPL a distinct advantage in quality of expression.

3.1.1.2 Location Pointers

The topic of location pointers represents a broad spectrum of programming capability which covers both data access and control of the execution sequences. Each of the four languages took a somewhat different approach and each provided a different level of capability.

SPL provided the most extensive capability through its location variables. A location variable is a data item which takes on integer values and whose values represent locations in computer memory. It can be used in two ways:

- Indirect data access is provided by the phrase

  IND(location-variable, index)

  which treats the value of the location variable as a memory address, adds the value of the index to it, and uses the result as the address of the data item to be accessed. The index is optional. This phrase can be used anywhere in an SPL statement where a variable can occur.

- Indirect transfer of execution control is provided through the phrase

  GOTO location-variable

  which treats the value of the location variable as a memory address and transfers control to it.

To assist in assigning memory location values to location variables, location constants:

```
LOC'data-item-name'
LOC'statement-label.'
```

represent the address of the memory location of the data item or statement which is identified between the apostrophes. If the data item is an array or table, the name can include index values to get the location of a specific data element. There are no special constraints on the formulas which can be used to assign values to location variables in SPL so complete flexibility is provided. (As discussed in Paragraph 6.3, this extent of flexibility creates some problems for process reliability.) The indirect addressing capability is particularly useful for efficiently maintaining and searching large quantities of data. SPL even allows a location variable to be assigned a "secondary data type", so the compiler can know what type of data is being indirectly accessed and can make any required data conversions when combining data through formulas.

The indirect GOTO capability is useful, but all four languages have some type of dynamically variable transfer which provides virtually all of the capability required and is considered less detrimental to process reliability.

Indirect addressing was used to particular advantage in accessing Switch Selector tables in the Switch Selector kernel coding (Paragraph 8.3.2).

CLASP does not provide nearly the location pointer flexibility of SPL. Neither location variables nor indirect addressing are provided. However, the location constant capability is provided and can be used to calculate array indices and provide more flexibility in data access. This was done in the sample coding to access Switch Selector tables. CMS-2 provides essentially the same capability as CLASP with a core address (CORAD) operator.

HAL conscientiously avoids the explicit use or manipulation of computer memory addresses by the programmer.

6.1.1.3 Vector and Matrix Operations

HAL has put the greatest emphasis on vector and matrix operations by explicitly defining a vector data type and a matrix data type, and by defining several operations which are performed upon them. Therefore, in HAL a two dimensional array and a matrix are two very different things. (As a logical consequence, arrays of matrices, where each array element is a matrix, can be constructed.) Multiplication between two matrices is the conventional matrix multiplication. The same multiplication symbol (a blank in HAL) between two arrays is a simple element-by-element multiplication where each element of the resulting array is the product of the corresponding elements of each of the operand arrays.

SPL takes the approach that any one-dimensional array is a vector and any two-dimensional array is a matrix. HAL's matrix and vector operations are available, but there is no simple element-by-element array multiplication, division, or exponentiation.

The only operation which CLASP performs on multi-element data organizations is matrix multiplication. CMS-2 provides no operations for multi-element data organizations.

6.1.1.4 Data Preset Symbolically and by Formulas

All four languages provide the capability to assign specific values to data items at compile time rather than having to do it through assignment statements at execution time. However, the flexibility of expression varies. Two specific capabilities are considered mandatory:

- The ability to symbolically represent a data value at compile time, without that value occupying core memory at execution time.
- The ability to express a preset data item with formulas combining literal values, symbolically represented values (as described above), and data constants to be used at execution time.

These needs and the corresponding language capabilities are exemplified by mission time values in the Saturn flight program. The Saturn real time clock accumulates time at the rate of 4,063.492 counts per second. There are many data items in the flight program which represent mission times and must be preset to constant or initial values. It should be possible to express these values in seconds, even though clock counts are manipulated at program execution time.

CLASP allows the programmer to declare a constant data item, preset it with a value, and then use that value in formulas to preset other data items. For example, the CLASP statement:

```
DECLARE FIXED KRTCONV 10 CONSTANT = 4063.492
```

defines the data item KRTCONV as a constant and assigns it the value 4063.492. CLASP then allows the statement:

```
DECLARE FIXED KDELTA 5 CONSTANT = 3*KRTCONV
```

to preset the constant KDELTA to a mission time value which represents 3 seconds. This capability provides three distinct advantages:

- The KDELTA declaration statement is decoupled from the precision of the real time clock so if the clock is changed, only the KRTCONV statement need be updated, rather than every time value in the program.

- The formula avoids an error prone manual multiplication by the programmer.

- The preset value is more readable.

CLASP's disadvantage is that the data item KRTCONV occupies a memory location at execution time even though the value is never required after compilation. CLASP provides no capability to symbolically declare a literal value for compile time use only.

HAL, SPL, and CMS-2 provide a compile time declaration capability but will not allow presetting of data items with formulas. In all these languages, a name can be assigned to any character-string and that character-string is substituted directly into the source code wherever the name appears. For example, in SPL

```
DEFINE KRTCONV AS '4063.492'
```

would allow KRTCONV to be used (alone) to preset a data item, or to be used as part of a formula in an assignment statement performed at execution time. However, it cannot be used as part of a formula to preset a data item.

6.1.1.5 Subroutine Control and Communication

An important aspect of program modularity is the control and communication among program modules. This discussion is restricted to program modules which are compiled and linked together prior to loading in the computer. Language features to perform equivalent functions for separately compiled and loaded tasks is part of the environment interaction discussion of Paragraph 6.2.

There are three distinct parts to the subroutine control and communication problem:

- Invoking ("calling") the subroutine

- Communicating information between the caller and the subroutine

- Returning control to the caller

All four languages have both procedures which are invoked by separate

calling statements, and functions which are invoked whenever their names appear in executable statements. All languages also provide for the declaration of data at a high enough level that it can be accessed by both the caller and the subroutine. However, the following differences are exhibited:

- o SPL is the only language which allows a subroutine (procedure or function) to have multiple entry points. This capability was extremely useful in coding the kernels and is a useful concept in general.

- o All languages except HAL allow a procedure to return control to the caller at an alternate exit which is different from the statement immediately following the calling statement.

- o CMS-2 and SPL allow parameters to be passed between separately compilable program modules. HAL does not allow this. (CLASP does not allow separately compilable modules to even call each other.)

- o CMS-2 allows only one input parameter to be passed to a function.

- o CMS-2 is the only language which provides a variable procedure call. This capability allows a single calling statement to invoke any one of a list of procedures, depending upon the value of a list index.

6.1.2 Naturalness and Ease of Expression

Among the four languages evaluated, there are many minor differences in the way a given concept or technique is expressed. For most of these differences, one form is as easy or natural as another once it has been learned. However, it is an unfortunate training and communication burden, and it is surprising that four languages apparently based on such similar lineage can look so different in describing the same things. Some of these differences have more significance to readability than to expression and are discussed in Paragraph 6.5.

There are, however, a few significant characteristics which give one language a distinct advantage over the other in writing a program. These are discussed below.

6.1.2.1 General Statement Format

SPL provides the most flexible statement format and was clearly the easiest to write during the kernel coding. No character positions are reserved, a statement can continue over many input lines without any special indication, and there is no statement terminator even for multiple statements on the same line. The compiler determines the beginning and ending of statements from the statements themselves.

CLASP was somewhat less convenient to write primarily because a continuation character (X) is required in column 73 to continue a statement on the next line. A dollar sign ($) delimiter is required between separate statements on the same line. Otherwise it is as free as SPL's.

The HAL format presents more problems for the program writer than either SPL or CLASP. The most striking difference in appearance among these languages is HAL's two-dimensional statement format, where exponents and subscripts are written on separate statement lines. The appearance of this format and its contributions to readability are discussed in Paragraph 6.5. However, the writing burdens introduced are discussed here. The first burden is simply that it is different. Experienced programmers are accustomed to writing

```
B**2 + C(5)
```

to represent "B squared added to the fifth element of array C". They require some adjustment to become accustomed to writing instead:

$$B^{2} + C_{5}$$

Experience overcomes this initial transient for the individual programmer, but so long as the programming community as a whole is using predominantly the FORTRAN-format, each programmer introduced to the language will need to go through the readjustment.

Specific annoyances which were encountered were the requirement to identify exponent and subscript lines with an E or S in column 1, and the need to terminate every statement. HAL terminates statements with a semicolon. While the semicolon is a very natural symbol to be used for a delimiter, it is not available on all high speed printers.

CMS-2 and CLASP use the less natural, but more available dollar sign ($) as a delimiter.

CMS-2's format was the least natural and convenient for several reasons. Every statement is terminated by a dollar sign ($). The first ten columns of an input line are reserved for sequence numbering, and selected compiler directives take on special meaning if they begin in certain columns. For example, COMMENT beginning in column 11 can be used to control the output listing. Such features should be provided by separate command, not by location of a statement.

6.1.2.2 Status Variables

SPL and CMS-2 provide the capability to declare status variables. These data items take on multiple states which the programmer can represent by symbolic names. In execution, it is equivalent to an integer variable which is assigned a different integer value for each state. However, in the source language, the use of symbolic names is a much more expressive and convenient approach for such variables as program flags, switches and some indices. Status variables were used extensively in the SPL and CMS-2 kernel coding. For example, the statement

```
ITEM DFLT STATUS (FLIGHT, SIM, REP)
```

creates the status variable DFLT, representing the operating mode of the flight program. DFLT can take on the three states, FLIGHT (actual flight), SIM (simulated flight), and REP (repeatable simulated flight). The initialization statement,

```
IF DFLT EQ 'FLIGHT' THEN . . .
```

can easily test and react to the mode. The same object code could be accomplished by declaring DFLT as an integer and using the values 1, 2, and 3 in place of 'FLIGHT', 'SIM', and 'REP'.

The compile time declaration capability of HAL provides a similar capability, but it is more difficult to declare and restricts the use of common symbolic names.

6.1.2.3 Bit and Text Manipulation

All of the languages except CMS-2 provided the capability

to treat a data word as a bit-string and perform logical operations (AND, OR, etc.) on bit-strings. These capabilities proved very useful in coding the kernels. They all provide the capabilities to selectively extract one or more bits from a data item and all but CMS-2 can insert binary patterns into selected portions of data items. All but CLASP can also perform these extraction and insertion operations on characters and character-strings. Character extraction and insertion were not required in the kernel coding, but in a message formatting or decoding activity for a man-machine interface it would be very useful.

A similar capability very useful to message formatting is concatenation, which is provided only by HAL. A new character-string or bit-string can be created by combining other strings in sequence, so that complex phrases can be easily built-up from simpler building blocks.

6.1.2.4 Factored Declarations

A language feature which greatly simplifies writing of a program specification without detracting from its readability is the factoring of data declarations. Rather than each data item being declared separately with all its attributes spelled out, factoring allows data items with some common attributes to be declared together and common attributes need be specified only once. For example, the SPL declaration sequence:

```
ITEM AGE INTEGER 24 =37
ITEM SIZE INTEGER 24 =42
ITEM WEIGHT INTEGER 24 =172
```

declares three 24-bit INTEGER data items named AGE, SIZE, and WEIGHT and initializes their values to 37, 42 and 172, respectively. The same declarations could be made by the factored statement:

```
DECLARE INTEGER 24, AGE =37,
                    SIZE =32,
                    WEIGHT =172
```

Common attributes are listed first and additional attributes can be declared with each item. Also, the common attributes can be overridden by specifying different attributes with given data items. For example, SIZE could have been declared a floating-point number instead of an integer by including FLOAT after its name. AGE and

WEIGHT would have been unaffected.

SPL, CLASP, and HAL provide essentially the same factoring capability, but CMS-2 requires that all data items in a factored declaration have identical attributes, including any stated preset value.

6.1.2.5 Miscellaneous

Several minor characteristics of the languages contributed to, or detracted from, naturalness and ease of expression. These are discussed below.

CMS-2 had several omissions or deviations relative to the other languages. The following are the more bothersome ones:

- There are no mnemonics for Boolean values. The other languages provide terms such as ON, OFF, TRUE, FALSE to be used as values for assignment to or testing of Boolean variables. CMS-2 provides only numeric literals '1' and '0'.

- There is no ELSE clause on an IF statement. The other languages provide the basic capability to write:

  IF condition THEN response ELSE response

  Either the THEN response or the ELSE response will be executed depending on the truth or falsity of the condition. Execution then proceeds following the IF statement. CMS-2 provides no ELSE responses, so execution proceeds without any action if the condition is false. In many decision environments, the problem is to execute one or two responses, not one response or nothing. Additional branching must be explicitly coded in CMS-2 to describe a dual response. An additional restriction unique to CMS-2 is that the THEN response cannot be another IF statement, so conditions cannot be nested.

- It has become almost universal in programming and other algorithmic languages to express assignment statements as:

  A = B

which normally means A takes on the current value of B. CMS-2 accomplishes this with the statement:

SET A TO B

It is difficult to envision the programmer or reader for whom this verbosity would be helpful.

- Because most arithmetic data values are conceived as decimal numbers, it is standard practice to interpret a string of digits as decimal unless some other indication like OCT or HEX is provided. CMS-2 takes the approach that octal representation is the standard, and requires that every decimal number be followed by the character D.

HAL provides an attractive feature by allowing blanks to be inserted in literals without affecting their values. (Character-string literals are excepted of course, because the blank is a meaningful character in the literal value.) Where long strings of digits are required, like the octal constant:

OCT'374 105 631'

the blanks make it easier to write the value correctly and verify it. The other languages do not permit blanks in literals other than character-strings.

HAL provides a very rich character set including very expressive operation symbols. In most cases a mnemonic equivalent, made up of alphabetic characters, is available for the less popular symbols (e.g., AND for &). However, there is usually a desire on the part of the programmer to use the easiest form of expression, which is the symbol. The richness of the character set contributed two minor irritations during kernel coding. First, the IBM 029 punch is capable of generating the input characters, but the keypunch operators were not familiar with them, and resulted in a higher density of keypunch errors. Second, none of several printers available for listing the language could print the entire character set and each one printed a slightly different subset. This is no major problem for a single facility, after it has established the character set it will use and has tailored the compiler and personnel to that set. However, transients are introduced when related personnel are working at two different

sites with different equipment. Many irritations were introduced into the Saturn ground computer programming effort, because different host computers were being used between MSFC and KSC. The extended character set of HAL seems to be a contributing irritant in this type of environment.

Another unique feature of HAL operator symbology is the use of a blank for multiplication. That is, the statement:

```
A = B C
```

assigns the product of B and C to the variable A. This notation derives from the rule of conventional algebra that adjacency of two symbols means multiplication as in:

```
y = 5w(a+b)
```

This is a very useful construct in conventional algebra, where variable names are almost universally single characters (perhaps subscripted), and physical adjacency (no intervening blank) is possible without ambiguity. However, in a flight program with multicharacter variable names, the intervening blank looks more like an oversight than an arithmetic operation.

6.1.3 Generality of Concepts

The level of generality with which a language concept is presented, or can be used, is valuable both in learning the language and in applying the concept to future applications. Unfortunately, some of the features provided by some of the languages have special restrictions which must be learned and remembered, lest they be violated, and which make the concept more difficult to apply, or totally inapplicable, in some situations.

HAL provides the greatest level of generality in its concepts. Once an operation or data type is defined, it can be applied almost anywhere that the application can have a reasonable meaning. SPL has some restrictions applied to several available concepts and techniques. CLASP and CMS-2 provide the greatest level of restrictions.

Several concepts which have different levels of generality among the languages are described below. It should be emphasized that the restrictions described below generally represent two shortcomings in a language:

- First, a reduced capability
- Second, a set of special restrictions to be learned and remembered, which represents a burden in training and use.

6.1.3.1 Index Expressions

Index expressions are the subscripts or parenthetical entries following a data item name which identify a particular element within an array, or a single bit within a bit-string, or some other specific selection from an ordered group. SPL and HAL both allow any valid arithmetic expression to be used as an index. The expression is evaluated and truncated,if necessary, to an integer value.

CLASP restricts the index expression to an integer variable or integer constant or a simple formula of the form:

variable * constant ± constant

or

variable / constant ± constant

where all operands must be integers.

CMS-2 allows non-integer constants and variables, but the formula format is restricted to:

variable ± constant

6.1.3.2 Numeric Data Items and Values

SPL, CLASP, and CMS-2 all provide for declaration of integer, fixed-point, and floating-point data types. HAL presents the more general concept of integers and scalars. A scalar is the numeric data type which can have a fractional part. Whether it is fixed point or floating-point is an implementation problem, but is not a distinction which needs to be made in every data declaration. If the target computer has a floating-point arithmetic capability, then all scalar data items are to be made floating-point by the compiler. Otherwise, the compiler makes them fixed-point.

HAL provides an even greater generality to the specification of literal numeric values. Most languages make a distinction in syntax among numeric values which are to be stored as fixed-point,

floating-point, or integer. However, HAL specifies all numeric values in the same format:

$$\text{dd.dd} \left\{ \begin{matrix} \text{E}\pm\text{dd} \\ \text{B}\pm\text{dd} \\ \text{H}\pm\text{dd} \end{matrix} \right\}$$

where each "dd" is a string of zero or more decimal digits. The E, B, and H expressions are optional, may be included in any combination with a given literal, and represent multiplication by powers of 10, 2, and 16, respectively. If the literal evaluates to a whole number, the item is interpreted as an integer. Otherwise, it is a scalar. Precision is determined by context of use, rather than by a specification with the literal, as in SPL. For example, if the literal appears in a formula in an addition to a double precision data item, with a double precision result, it will be a double precision value.

By contrast, SPL requires that integer literals may not have decimal points, but fixed-point and floating-point literals must have decimal points, even if they are whole numbers. Fixed-point literals must include a scaling designator, even if the scaling is zero, and even though it probably can be determined by context. For example,

4.23E1A6

is interpreted as 42.3 and is aligned in computer memory so that it has six fractional bits.

CLASP and CMS-2 both forbid a decimal point in integer literals and require it in fixed and floating-point literals, but the scaling designator (Axx)is optional in CLASP and not used at all in CMS-2.

6.1.3.3 Arrays

HAL allows arrays of data to be used in most of the contexts where single data items may be used, with the interpretation that the context (e.g., an arithmetic operation) be applied independently to each element in the array. All of the arithmetic and logical operations are applicable to arrays of the proper data types.

CLASP apparently applies arithmetic and logical operations on arrays, but SPL imposes severe restrictions. CMS-2 has no

array operations.

HAL also allows functions to be defined as operating on a single data element and then be applied to an array. If an array name is supplied as an operand of such a function, the function is performed independently on each element of the array, and the output is an array of the resulting values.

6.1.3.4 Loop Variables

In iterative loop control statements the loop variable has a starting value, a final limit, and an increment (decrement) by which it is increased (decreased) for each iteration. In SPL and HAL these values can be any numeric formulas. However, CLASP restricts the loop variable to an integer data item, and the other values to integer variables or constants. CMS-2, without specifying how the restriction is imposed, restricts the loop variable to integer values.

6.2 Expression of Environment Interaction

A flight program can be characterized as a set of synchronous and asynchronous program modules which have a continuous, and frequently time critical, interaction with the vehicle and each other. Each program module thus has a distinct environment necessary (in general) to provide meaning to its execution. The major functions that a language needs to provide to express this interaction are described below.

- Communication with Vehicle Systems/Subsystems

  The program modules need to obtain information from various systems (e.g., Inertial Measurement Unit) to be able to perform computational and decision processes, necessary to provide information to other vehicle systems (e.g., Control System). This communication is provided through various Input/Output Registers.

  None of the languages reviewed provided special commands to address "real time devices" (i.e., the Vehicle Systems/Subsystems). However, it is sufficient that the language contains an input/output command which includes a device name and a storage area. Input/output alone, however, is not sufficient to perform the total communication functions. The language must also

have sufficient capability to access and manipulate one or more bits of the coded data communicated. The coding of this data is generally dependent on the particular device addressed. Encoding and decoding of the data is, therefore, very much a part of this function.

- Synchronization with the Vehicle Mission

The tasks to be performed by the software are dependent upon the progress of the mission and various asynchronous events occurring during the mission. The prime means by which the software is synchronized with the vehicle are:

1) Discrete signals - To signal that specific events have occurred. These may or may not result in computer interrupts.

2) Real Time Clock - To signal the progress of the mission in time to control time dependent tasks.

3) Interval timer - To measure elapsed time for various purposes, independent of the elapsed mission time.

The language needs first of all to express the ability to detect whether these events occurred. More important, however, it needs to be able to specify which instruction(s) should be executed if and when these events occur, without continuously and explicitly testing these events.

- Intertask Communication.

A flight program generally consists of a set of repetitively executed processes, each of which may be executed at a different rate, and a set of processes that are executed when events, whose times of occurrence are unpredictable, actually occur. These processes access a common set of data and may "concurrently" progress. That is, a process may start execution before another process has completed execution.

The common term in use to describe these processes

is the term Task. In more complex systems the trend is to impose the following requirements upon the task:

1) Tasks should be separately compilable.

2) Tasks should be able to control the scheduling of other tasks. That is, they should be able to schedule and terminate execution of other tasks.

3) It should be possible to synchronize the progress of execution of related, concurrently executing tasks.

4) Tasks should be able to, reliably, access a set of data elements common to all tasks. Reliability implies that the integrity of the data is preserved during access.

Not included in the evaluation of the languages are the requirements for intertask communication in a multi-processing system. Concurrent execution in the context of this study refers to a multi-programming or multi-tasking system. The reason for this is that none of the languages has explicitly included execution on a multi-processing system in their design objectives.

- Mass Storage Access

  Aerospace software has, traditionally, not made any extensive use of mass storage.

  As the amount of data, programs, and the complexity of tasks increases, a very definite need will exist to manipulate mass storage data. The capabilities required are anticipated to be identical to the more rudimentary capabilities used by ground based computers.

In the following paragraphs each of the languages will be separately discussed. Within each language discussion, the manner in which each of the major functions can be expressed will be considered to evaluate the "Expression of Environment Interaction" of the language.

6.2.1 SPL

This language provides good capabilities to express the required functions.

6.2.1.1 Communication with Vehicle Systems/Subsystems

By considering a system/subsystem a "file of records," the basic read/write operations can be used. Each read or write operation results in a direct data read-in or read-out with no data conversion, which enables the programmer to convert the data as required at his own discretion.

The basic input/output statement consists of a READ or WRITE command followed by a file name and a (optional) data buffer name. The latter is optional because it can also be specified in a "file declaration".

A file declaration is associated with each file name. This allows the symbolic file name to be associated with a (system defined) device name. Other "file attributes" may also be specified in the file declaration. Such attributes may, for example, specify the type of conversion to be performed and the routine to be executed upon abnormal indications from the device addressed.

An additional feature in SPL which may be useful in this area is "Dynamic File Conversion". Frequently, the format of the input data is dependent upon variables within the input record. SPL allows all possible formats to be specified in the file declaration. It is then a simple matter to test the control variable and then interpret the input record within the appropriate format.

Decoding and encoding of the real time data is fully supported by the strong bit manipulation capabilities of the language.

6.2.1.2 Synchronization with the Vehicle Mission

The prime means of expression in SPL to perform this function are "Chronic Statements". These are statements that are not part of the normal (i.e., inline) sequence of statement execution.

These statements are preceded by ON. . . (Boolean formula) . . .

The Boolean formula is automatically evaluated whenever the

first operand acquires a new value, either by assignment or by hardware action. Either discrete events (including interrupts) or real time clocks may be used as the variable to be evaluated. It is not clear from the specification whether chronic responses may be reassigned dynamically. However, there is no language design limitation that would prevent a different (and proper) implementation.

Hardware interrupts may be enabled or disabled selectively. Unfortunately, no means is provided to test whether a particular interrupt has been enabled or disabled. Another inconvenience is that no means is provided to group all or some hardware interrupts under a single name. This forces each individual hardware interrupt to be named separately.

Program action may be delayed for an interval of time or until a specific event occurs by the use of a WAIT statement. The WAIT statement is always used in conjunction with either an IF . . ., LOOP WHILE . . ., or LOOP UNTIL . . . statement and indicates that while the logical condition is true, the program execution will not continue to the next statement.

6.2.1.3 Intertask Communication

A system in SPL may consist of a set of independently compilable Programs and a "Compool" (a common accessible set of data). The Program in SPL is its closest concept to a Task.

Tasks may call (i.e., initiate scheduling) other Tasks and pass parameters to them. By using the chronic statement described in 6.2.1.2 the initiation of scheduling may be attached to the occurrence of various conditions. Once a task is associated with an event through a chronic statement, it cannot dynamically be disassociated from that event.

Execution synchronization between tasks is provided through the WAIT statement described in Paragraph 6.2.1.2 and flags (e.g., status variables) set/reset in the compool.

The integrity of common data may be insured by a task reading the data through LOCK/UNLOCK statements. A LOCK statement prevents any other task from writing into a specific portion of memory until a subsequent UNLOCK frees it up. It is intended to be implemented only if the computer has a memory protect feature. The

disadvantage of this scheme is that there is no positive check that LOCK's are used where they should be used. That is, they may be omitted and not cause errors until after many, many runs.

Another major shortcoming is that deadlocks may occur. The language specification does not explain how deadlocks are resolved.

6.2.1.4 Mass Storage Access

The basic input/output statements available in SPL are described under Paragraph 6.2.1.1. Notable is that the capabilities are adequate only for sequential input/output files; that is,devices such as tape, card reader/punch, and printer.

Mass storage is likely to consist of some sort of direct access storage. Accessing such storage requires the identification of individual records. This capability is not now a part of the language specification. Note, however, that it is possible to define direct access storage as containing several sequential files, thereby providing a limited direct access to such storage. This may be sufficient for some applications.

6.2.2 CLASP

Expression of the required functions discussed here is the main deficiency in CLASP as it is currently defined.

6.2.3.1 Communication with Vehicle Systems/Subsystems

No input/output capability whatsoever is provided in CLASP. Any communication with external devices is meant to be expressed by lapsing into basic assembly language. Bit manipulation capabilities within the language are sufficient to express encoding/decoding functions.

6.2.2.2 Synchronization with the Vehicle Mission

The chronic statement ON, is used to indicate the statements to be executed upon the occurrence of a specified enabling condition. This enabling condition could be a discrete event coming on (if detected by a central executive or the hardware), a hardware interrupt, or an interval timer interrupt.

Interrupts may be enabled/disabled selectively. No means is

provided to detect the disable/enable status of the interrupts. No direct means is provided to delay program execution for a period of time; neither is access to an elapsed time clock provided.

6.2.2.3 Intertask Communication

No means of communication between independently compilable modules is available.

6.2.2.4 Mass Storage Access

The same comments as those noted under Paragraph 6.2.2.1 apply.

6.2.3 HAL

A significant characteristic of HAL in the support of this function is that it assumes the existence of a Control Executive. It is the only language that provides an explicit interface with a Control Executive. This does reduce the applicability of the language in small applications where the Control Executive and application programs are, for simplicity, integrated.

6.2.3.1 Communication with Vehicle Systems/Subsystems

By considering a system/subsystem a "file of records", the basic FILE statement can be used to express this. A FILE statement has the appearance of an assignment statement and may be used for both reading and writing data, depending upon which side of the equal sign (=) the term FILE appears. Device name and data area to be used can be indicated.

Decoding and encoding of the real time data is fully supported by the strong bit manipulation capabilities of the language.

6.2.3.2 Synchronization with the Vehicle Mission

The prime means of expression in HAL to perform this function is the SCHEDULE statement. The SCHEDULE statement is used to request initiation of a program module based on any of three criteria:

- at a specified time

- after an interval of time
- on events or combination of events

These events may be defined to be hardware interrupts or other detectable discrete events.

As previously mentioned, HAL assumes the existence of a Control Executive. Therefore, no provision is made to enable/disable interrupt conditions nor is any other direct control of interrupts provided.

Direct access to a real time clock is also not provided as a part of the language.

Program execution may be delayed through a WAIT statement. This statement can be used by an active program module to suspend and reactivate itself based on any one of three criteria:

- until a specified time
- for a specified time interval
- until the occurrence of an event or combination of events

6.2.3.3 Intertask Communication

Extensive capabilities are provided to perform this function. Among the reviewed languages HAL is the only one that explicitly recognizes the task concept.

Independently compilable programs as well as subprograms can be considered tasks. All tasks can access a common set of data: the Compool. Only a task as an independently compilable program will be considered here.

Tasks can initiate scheduling of other Tasks through the SCHEDULE statement previously discussed in Paragraph 6.2.3.2. No parameters can be passed. Tasks can terminate scheduling of other Tasks through a TERMINATE statement. Tasks may have priorities (relative to other tasks) assigned to them. Priorities may be changed dynamically.

Synchronization between Task executions is performed through EVENTS. EVENTS may be turned ON or OFF or may be pulsed through a SIGNAL statement.

An elaborate scheme is available to protect the integrity of dynamically shared data. Each data element in the COMPOOL may have access rights associated with it to indicate which task may access it. Each data element in the COMPOOL may have either one or two LOCKTYPES associated with it.

- LOCKTYPE(1)

  Prevents a Task from updating the statement if another task is reading or updating it.

- LOCKTYPE(2)

  Prevents a Task from updating the data element if another Task is updating it.

A Task may only access a data element that is declared to be LOCK'ed if the access is preceded by an UPDATE statement. This gives a positive check that the LOCK mechanics are indeed invoked if a Task attempts to use a protected data element. The language specification describes in detail the algorithm to be used to insure minimum task execution delays and prevent deadlocks.

6.2.3.4 Mass Storage Access

The FILE statement in HAL contains sufficient information to obtain an identified record from a device. A limitation is that it is associated with a device and not with a file. It, therefore, does not allow any "File Management" to take place. Neither does it provide sufficient input/output capabilities to design such a package within the language. Therefore, although HAL ostensibly may directly access a record from some type of direct access storage, the capability is not anticipated to be sufficient for practical applications.

6.2.4 CMS-2

This language, like HAL, explicitly assumes the existence of a Control Executive. The language itself does not contain any real time control features as discussed in the following paragraphs.

6.2.4.1 Communication with Vehicle Systems/Subsystems

By considering a system/subsystem a "file of records" the basic input/output statements can be used to express all information required to perform real time input/output. Bit manipulation capabilities are sufficient to accomplish decoding of the input data. Encoding of output data, however, is difficult mainly because the BIT conversion function cannot be used on the left side of an assignment statement. That is, bits cannot be inserted easily into a portion of a data word.

6.2.4.2 Synchronization with the Vehicle Mission

The intent of CMS-2 is to let the Control Executive perform any real time synchronization and let these requirements be communicated to the Control Executive outside the language. No specific features are therefore provided in the language.

6.2.4.3 Intermodule Synchronization and Communication

No real time features that aid this function can be expressed within the language.

6.2.4.4 Mass Storage Access

Currently the input/output is meant to be sequential files. No direct access capabilities are therefore provided within the language.

6.3 Process Reliability

The reliability of the program development process is reflected in the reliability of its final product, the object program. A higher level language's greatest contribution toward a reliable object program is in the generation of more reliable code in the beginning. That is, most of the language features which contribute to effective expression also naturally contribute to reliable code. However, program reliability is at least confirmed, and usually established, during the verification step of the process. There are specific features within a language which can assist the verification step directly. Verification, in this context, includes all activities directed toward finding and correcting errors in a program after it has been coded.

The designers of HAL took a very conscientious approach to providing a language which would produce reliable code. Indeed, increased reliability was one of the major design objectives of the language. This objective was pursued primarily through providing the opportunities for the compiler to static check the source code. Static checking allows the compiler to detect and diagnose some errors in the source code which otherwise would not be detected until execution time. The attempt to provide a language which produces static checkable source code is reflected in the language in two ways:

- Several features have been included to allow the compiler to perform static checking of the source code. Foremost examples are data access rights and lock types on compool data, which is available to all tasks.

- Several features whose execution time performance are difficult to predict, and therefore difficult to check, at compile time have been omitted. The most notable of these is indirect addressing.

HAL's efforts in this area have achieved the reliability objective to a sufficient degree that HAL is expected to produce more reliable object code than any of the other languages.

SPL, due primarily to the flexibility and power in some of the features provided, is expected to produce the least reliable object code. Even if the use of this flexibility and power is administratively restrained, SPL still does not provide some of the built-in safety features of HAL. CLASP and CMS-2 lie between SPL and HAL in process reliability, primarily because they do not provide the flexibility

and power of SPL.

To present and discuss the pertinent characteristics of the languages, the evaluation criteria have been organized into the chronological order of error generation and detection. Following are the four major issues under which the language characteristics are discussed:

- Does the language include characteristics which encourage the generation of reliable source code while it is being written?
- How easily are program errors detected by visual inspection of the program listing?
- How well can the compiler static check the source code?
- How much control can the user exercise over the verification process?

6.3.1 Generation of Reliable Source Code

Of course the elimination of program errors before they are ever made is the most effective contribution to process reliability. Even errors which can be easily detected by the compiler slow down the program development process because they force additional iterations of the compilation cycle. Therefore, elimination of an error is one step better than detection of it. The characteristics which tend to prevent errors derive primarily from naturalness and consistency in language features and in rules for applying those features.

6.3.1.1 Loop Variable Limits

Each of the languages provides a capability to iterate a group of statements for specific values of a loop variable. The following statements represent (almost) the same logic in each of the four languages:

```
(SPL)       FOR LOOP = 1 BY 2 UNTIL 7
(CLASP)     FOR LOOP = 1 BY 2 TO 7
(HAL)       DO FOR LOOP = 1 TO 7 BY 2
(CMS-2)     VARY LOOP FROM 1 THRU 7 BY 2
```

These statements each call for execution of a group of statements,

first for LOOP = 1, then again for LOOP = 3, and again for LOOP = 5. For all languages except SPL, the group is iterated a fourth time with LOOP = 7. However, SPL does not iterate the statement group for the final value of the loop variable. This is an unnatural way to express iterative control and certainly reduces readability. However, its greatest effect is that it is misleading enough to cause confusion and errors.

6.3.1.2 Re-entry into Chronic Statements

SPL and CLASP both provide chronic statement sequences which specify the actions to be taken when an interrupt occurs or, in the case of SPL, when some logical condition is satisfied. These statements are described in the specifications as outside the normal sequence of execution, but there are no special protections to insure that they stay outside the normal sequence. There is no inherent protection against a GOTO statement in the normal sequence branching to a labelled statement in the middle of a chronic statement sequence. A subsequent interrupt could suspend that processing and start executing from the beginning of the chronic statement sequence. The result is re-entrance into a portion of the chronic statement sequence.

In HAL, the real time task, which is the counterpart of the chronic statement sequence, is a separate program module, with a separate name and label scope and with only one entry point. It is much better protected from inadvertent re-entrance than is the chronic statement sequence. Re-entrance is also possible in HAL, because the same task can be scheduled on two different priority levels. In fact, the priority level can be changed after the task has been scheduled, or even after it has started execution. While these HAL capabilities are potentially hazardous and must be used with considerable caution, they are considered less error prone than the totally unprotected chronic statements.

CMS-2 provides no real time control capability.

6.3.2 Ease of Scanning for Errors

There were no major advantages among the languages in ease of scanning the source code for errors. However, a few minor distinctions bear mention.

SPL's decision tables were certainly easier to review for errors than were the equivalent IF statements.

HAL's program, task, and subroutine declarations all begin with the label of the program module being declared. For example:

```
ALPHA:      PROGRAM
BETA:       PROCEDURE...
```

begin declarations of program ALPHA and procedure BETA, respectively. Therefore, scanning the listing to find the declaration is fairly easy because the label begins the statement. However, in SPL and CLASP, the declarator proceeds the label. For example in SPL:

```
START       .ALPHA
PROC        .BETA
```

begin the declarations of program ALPHA and procedure BETA. It was much more difficult in this format to pick a declaration out of a listing.

6.3.3 Static Checking

By far, the majority of the language contributions to process reliability are in the area of providing features which allow the compiler to better interpret the programmer's intent and therefore better detect and diagnose errors in the source coding. The HAL design has paid particularly close attention to providing such features, while SPL has incorporated features which make it very difficult for the compiler to interpret a programmer's intent. The following topics are discussed:

- Indirect Data Access
- Indirect GOTO
- Direct Code
- Contextual Variables
- Overlays
- Compile Time Data Access Control
- Execution Time Data Access Controls
- Implicit Data Declarations

o Compiler Controlled Scaling

6.3.3.1 Indirect Data Access

SPL provides an indirect data access through location variables. The content of the location variable is interpreted as a memory address and is used to locate the data to be accessed. As explained in Paragraph 6.1.1.2, this is an extremely powerful technique for data access, but is also a hazardous approach because the compiler cannot monitor the data access to determine that it is valid. Since the language allows any type of data item or formula to be assigned to a location variable, there is no way for the compiler to make even a gross validity check. If indirect data access through a location variable is to be provided at all in a language, severe restrictions must be placed on the types of formulas which can be assigned to the location variable.

The use of indexed tables and arrays, as provided by all of the languages, is considered a much more reliable approach to data access than SPL's indirect addressing.

6.3.3.2 Indirect GOTO

SPL extends its indirect addressing capability beyond data access to execution control through its indirect GOTO. The statement:

GOTO location-variable

branches control to the memory location indicated by the contents of the location variable. The SPL specification warns the user that when using a location variable in an indirect GOTO statement:

> "... the programmer should make certain that the contents of the variable points to an imperative statement and not to a data item, since the compiler has no way of checking this."

Of course, given that the variable does point to executable instructions rather than data, it may point to operations in the middle of a higher level language statement, or it may point to instructions in a completely separate procedure or program which could not be legitimately accessed through defined statement labels. It is highly unlikely that a programmer would calculate the number of computer instructions that the compiler will generate in expanding statements so that he could

establish numeric values or formulas to represent the location of a statement. The most practical approach (and the only reliable one) is to label the statement and use the location constant:

LOC'statement-label'

to assign the location variable. Therefore, it seems reasonable from the point of view of expression capability, and imperative from the point of view of reliability, to restrict the assignment of a location variable which is to be used for an indirect GOTO. It should be restricted to assignment to a simple location constant.

However, SPL has a switched GOTO statement which already provides that capability. The switched GOTO references a list of statement labels and, through an index value, selects one of those statement labels for the GOTO. Therefore, rather than assigning the statement label to a location variable and executing an indirect GOTO, the programmer assigns a value to an index and uses a switched GOTO.

This is a much more reliable approach and affords the required capability. Each of the four languages has a switched GOTO or nearly equivalent capability.

6.3.3.3 Direct Code

Another popular feature which obscures intended program action from the compiler is "direct code" capability provided by all of the evaluated languages except HAL. Through this capability, the programmer may, anywhere in his program, lapse into machine language or assembly language programming. Since all of the data items and statement labels available to the higher level language statements are also available to the direct code, there is very little capability to control or monitor modifications to data items or the execution sequence.

There is undoubtedly an occasional desire to perform some operation in assembly language rather than a higher level language. However, the interface between the direct code and the higher level language should be at least as carefully controlled as the interface between two separate procedures both in the higher level language. That is, unrestricted data access and statement label access should not be allowed for the assembly language code. A subroutine interface

where named parameters are passed back and forth and control is returned in an orderly manner, would greatly reduce the likelihood of direct code circumventing the static checking which is performed by the compiler.

6.3.3.4 Contextual Variables

SPL allows the programmer to declare contextual variables, whose attributes vary with the context of the data item's use. Each time it is assigned a value, it takes on the attributes of the formula by which it was assigned. When it is subsequently used, as part of a formula for example, the attributes of the last assignment are used to determine the conversions and formatting to be applied. Thus, a contextual variable may be used to store and retrieve a Boolean value (true or false) in one part of a program and a floating-point number in another part.

CLASP provides a similar capability but limits it to fixed-point data items. The only variable attribute is the scaling, which aligns the radix point of the data value.

For both languages, "subsequent use" of the variable is determined by the compiler from the physical sequence of statements in source code. This, of course, is not necessarily the logical sequence of execution of these statements as indicated by the following example:

```
        TEMP = INTEG1
           '
           '
           '
        IF TEMP GT 3 GOTO L2
           '
           '
           '
L1.     FLOAT2 = TEMP

        TEMP = FLOAT1
           '
           '
           '
L2.     FLOAT2 = TEMP
```

Assuming that TEMP is contextual, INTEG1 is an integer, and FLOAT1 and FLOAT2 are floating-point, then TEMP is treated by the compiler

as an integer in statement L1 and as a floating-point number in statement L2. This is because L1 follows a TEMP assignment by an integer, and L2 follows a TEMP assignment by a floating-point number.

When the compiler expands statement L1, it includes a conversion from integer format to floating-point format. However, in statement L2, the expansion is a simple transfer of (unconverted) value because both sides of the assignment statement are floating-point. Note that when TEMP, as assigned by INTEG1, has a value greater than 3, control is transferred directly to statement L2. In this case, the executed sequence of assignment statements takes the value contained in INTEG1 and transfers it via TEMP to FLOAT2. However, since no conversion is made by either assignment statement, FLOAT2 does not receive a floating-point representation of the integer value contained in INTEG1, as it would if TEMP were not contextual.

CLASP warns of this potential error with a compiler diagnostic message. Any time a labeled statement appears between the assignment of a temporary data item and its subsequent use, there is a possibility in that in execution, control will be transferred to that statement. The compiler does not know what data format will have been stored in the temporary variable. Therefore, the CLASP specification states that the compiler will output a diagnostic message at the subsequent use of the temporary variable and will identify the preceding assignment statement which is dictating the data characteristics assumed. In the example, a diagnostic message at statement L2 would warn that the current characteristics of TEMP are determined from the assignment at statement L1 + 1. CLASP also allows the programmer to specify explicitly the label of the assignment statement he wishes the compiler to use for determining characteristics. These two features constitute an improvement over SPL's complete lack of regard for the problem. However, errors still can occur, and a more stringent control seems desirable. Neither HAL nor CMS-2 allow contextual data.

6.3.3.5 Overlays

A situation very similar to that of contextual data is created by the capability for data overlay. All of the evaluated languages except HAL allow the programmer to specify that multiple data items are to occupy the same computer memory locations. This is an effective memory conserving capability, and could probably be used

reliably if it were restricted to constants (of the same value, of course) and to temporary data used by independent program modules which could never be in execution during the same time interval. However, the general case of two different variable data items occupying the same memory location is an invitation to very unreliable object code. The specifications which describe the capability warn of its hazards, but neither the languages nor the compilers appear to provide any protection.

The HAL approach of automatic and static storage is a more reliable approach to conserving memory.

6.3.3.6 Compile Time Data Access Control

The concept of a common data pool allows all tasks and subroutines to directly access data which must be communicated among them. However, any single module usually requires access to only a portion of the total compool.

HAL allows the programmers to describe selective access to compool data through access rights. Each program can be assigned an identification, and each compool data item with that identification (access right) can be accessed by the program. Each compool data item can be assigned as many access rights as necessary to identify all the programs which need to access it. The compiler can then verify at compile time that all compool data accesses within each program have been properly authorized. None of the other languages provide such access control.

The access rights concept provides a very valuable static check capability; that ability should be extended to include a distinction between read-accesses and write-accesses. In many cases a module needs to read a variable, but should not be permitted to modify it.

6.3.3.7 Execution Time Data Access Controls

CLASP and SPL both provide a LOCK and UNLOCK data capability which allows a program to selectively protect data from outside interference while it is being used within the program. The commands are intended to enable and release the hardware memory protect capabilities provided by the object computer. While this is an execution time feature, it has compile time static checking implications. There is no way for the compiler to determine which data items

require protection and, therefore, which data accesses should be surrounded by LOCK and UNLOCK statements.

HAL provides an opportunity for static checking by declaring LOCKTYPE data attributes. Any access, by any program module, to a data item which has a LOCKTYPE attribute can be checked to insure it has the proper real time locking controls surrounding it.

6.3.3.8 Implicit Data Declarations

Each of the languages provides a capability to explicitly declare a data item by specifying its name, and the data attributes which it possesses. The name then references the defined data item when it is used in executable statements. All the languages except CLASP also allow for implicit data declarations. That is, a variable name may appear in an executable program statement without having been previously declared in an explicit declaration statement. The first appearance of an undeclared data item causes it to be implicitly declared by the compiler and to be assigned default attributes. This is provided as a convenience to the programmer, but it removes a redundancy which is important to static checking. For example, if the programmer (or the keypunch operator) misspells the name of a previously declared data item, the compiler creates a new data item, compiles the executable statement with a data reference different from what the programmer intended, and proceeds to the next statement. The compiler detects no error, and may compile a working program module which will successfully execute to completion. If the unintended data item is used in an infrequently executed branch of the program, the error could avoid detection for some time.

The convenience afforded the programmer by implicit declarations does not warrant the increased probability of an error undetected by the compiler. This is especially true, because the ability to factor declarations makes explicit declaration so convenient anyway.

6.3.3.9 Automatic Scaling of Fixed-Point Data

One of the biggest problems associated with flight programming on a fixed-point computer is the scaling of numeric data items. Since the length of a data word in the computer is limited, there is a limit on the precision of a value which can be retained. The problem is one of aligning a variable such that its maximum value will not overflow the left-hand end of the computer word, but still have sufficient bits on the right to prevent truncation of significant precision. This alignment is usually described by specifying the location

of the binary point which separates the integer part from the fractional part of the value.

In all four languages, the declaration of fixed-point data scaling is approximately the same. The significant difference is that only in CMS-2, SPL, and HAL does the declaration specify exactly how values will be aligned in the data item. CLASP is defined so that the programmer's precision declarations are treated as minimum precision requirements and the compiler determines the actual data alignment within those precision constraints. The compiler picks the alignment so that when this data item is combined with other data items as specified in formulas and assignments in the program, the number of scaling shifts can be minimized, resulting in a more efficient program execution.

The problem is that the scaling of a data item in one procedure may be changed by the compiler because of a change made in a separate procedure. One data item common to both procedures is all that would be necessary to propagate the requirement for rescaling. If the programmer could guarantee that his minimum scaling specification was sufficient, the change would not create an error. However, flight program verification is historically, and will continue to be, an empirical process on object code. That is, the object program is exercised over various ranges of parameters to determine that it operates properly. This level of verification can demonstrate that the compiler selected the proper scaling for that compilation, but it does not verify that the programmer's minimum specification is sufficient. Therefore, it is considered very hazardous for the compiler to modify the scaling of a data item without an explicit declaration change by the programmer.

6.3.4 User Control Over Verification

One of the major problems in verification of a program written in a higher level language is the difficulty in correlating object code with source language statements. The only significant features found among the languages were some compiler directives intended for debug aids. HAL did not include these directives.

While debug directives are useful, they are extremely difficult to use because the program module must be recompiled to change the controls. Effective user control over program verification is essential if the program development is to proceed efficiently.

To be effective, this control must be provided through the verification facility itself, and must not be restricted to those controls and test cases which occur to the programmer at compile time.

6.4 Object Program Efficiency

Computer storage space and processing power have been, traditionally, limited to minimize weight and power requirements. Consequently, the efficiency of the utilization of these computer resources have been a major concern. Existing and near future computer system designs provide a many-fold increase in processing power over previous aerospace computers. Although this relieves the heavy emphasis on efficiency somewhat, it by no means negates it completely. It should be realized that on-board applications continue to increase in scope as well as complexity. It is naive to assume that the available processing power will outrun the required processing power. Efficiency of resource utilization therefore remains a factor to be considered.

The efficiency of an object program resulting from the description in a high level language is primarily a result of the design and implementation of the compiler. That is, the object program may be highly efficient or inefficient in spite of the design of the language.

However, the language may contain features that can aid the compiler in generating efficient object code, if the compiler indeed utilizes the information provided. By the same token, that language may contain features that prevent or make it difficult for the compiler to generate efficient object code. It is these features that are considered in this study.

It is important to re-emphasize at this point that an optimizing compiler should be considered mandatory for aerospace software and should, therefore, be the primary means to obtain object program efficiency. The features described here still help in obtaining efficiency objectives, but should not be considered the primary method used.

From the languages reviewed, CLASP is significantly different in that it assumes the availability of a compiler that has highly sophisticated optimization capabilities. HAL is the only language that pays little or no attention to time-efficiency. It should be pointed out, however, that the primary means by which other languages attempt

to obtain time-efficiency is through providing the programmer access to machine registers. This is first of all a highly machine dependent method that significantly depends on the programmer's insight. Secondly, it may interfere with the automatic assignment of these registers by an optimizing compiler, thereby possibly causing an overall decrease in efficiency.

In the following paragraphs we will discuss each language separately. For each language, the discussion will be divided into two evaluation criteria.

1) Language features that aid in the generation of efficient object code.

2) Language features that hinder the generation of efficient object code.

6.4.1 SPL

More specific features are provided in SPL than in any of the other languages.

6.4.1.1 Efficiency Aids

Language features that aid in the generation of time efficient object code are:

- Constant attribute

  Data items which values remain constant during execution can be indicated to the compiler. This allows the compiler to generate code to use immediate instruction, whenever they are available. An immediate instruction contains the data in the instruction rather than in a memory location.

- Index declaration

  This allows a data element to be assigned to a hardware register because it is frequently referenced during a particular section of the program. Memory references for each time the data item is manipulated are thus unnecessary.

- Hardware declaration

  This allows machine registers (if available) which are normally only available to the programmer in basic assembly language, to be used as storage locations for variables thus decreasing the memory accesses required.

- Decision tables

  Reference Paragraph 6.1.1 for a description. These facilitate the optimization of complex decision logic over a large segment of code.

- Inline attribute for procedures

  This provides for the generation of code at the point in the program that the procedure is referenced. This eliminates the execution of instructions normally required to process input and output parameters in invoking the procedure.

- Timing directive

  This only aids indirectly in the generation of efficient code. It provides the programmer with the ability to have a section of code timed. He may thus check the reasonableness of the execution time for his application and check upon the results of possible improvements he may make.

Language features that aid in the generation of storage efficient code are:

- Data packing attributes

  This allows data elements in tables/arrays to be stored with various packing densities. Normally a single data item is stored in one computer word. To save storage (at the cost of execution time) SPL allows the programmer to specify packing of two or more data items into one computer word where possible.

- Overlay declaration

  This provides a means by which more than one data element can occupy core storage previously used by another data element. This is simply done by declaring a data element to "overlay" the storage of another data element that is no longer in use.

6.4.1.2 Efficiency Hinderances

No features were identified that specifically hinder the efficiency of the object code. However, two of the features that aid in efficiency as described in Paragraph 6.4.1.1 may, in effect, cause inefficiencies if improperly handled. These are:

- Decision tables

  Unless logic optimization techniques are indeed implemented, these can cause very inefficient code to be generated as compared against the direct use of IF... THEN... statements. The latter allows the programmer to perform some logic optimization.

- Index/hardware declaration

  If the compiler performs automatic optimization, improper use of this feature may interfere with it, actually resulting in less efficient code.

6.4.2 CLASP

The main emphasis of the CLASP optimization directives is to enable the programmer to optimize selected portions of his program and to explicitly specify whether time or core efficiency is of major concern for a specific section of code. This provides it with the most powerful efficiency feature, assuming that the implementation is practical.

6.4.2.1 Efficiency Aids

The following time-efficiency features in CLASP are equivalent to their SPL counterparts:

- Constant attribute
- Index declaration
- Timing Directive

- o Hardware declaration
- o Inline attribute for procedures

CLASP also provides an OPTIMIZE TIME(n) directive, which allows the specification of the relative importance (n) of time optimizing to the (optimizing) compiler for various sections of code.

Storage efficiency is aided by an overlay declaration identical to SPL's (Paragraph 6.4.1.1), and an OPTIMIZE SPACE(n) directive. This directive allows the specification of the relative importance (n) of storage optimization to the (optimizing) compiler for various sections of code.

6.4.2.2 Efficiency Hinderances

The only significant language feature that might hinder the generation of efficient object code is the index hardware declaration. The explicit use of these registers by the programmer may interfere with the time optimization automatically performed by the (optimizing) compiler.

6.4.3 HAL

Little or no attention is paid to time-efficiency features in this language. It does, however, provide various means to obtain efficiency in storage used.

6.4.3.1 Efficiency Aids

The only significant language feature that aids in the generation of efficient object code through time efficiency is the constant attribute described for SPL (Paragraph 6.4.1.1).

Storage efficiency is provided through:

- o Data packing attributes

  This is similar in principle to SPL's capability (Paragraph 6.4.1.1). The main difference is that SPL only accepts packing attributes for tables/arrays while HAL allows it to be applied to any data element.

- o AUTOMATIC storage allocation attribute

  This allows storage to be allocated to data elements only when the corresponding program module is entered for

execution. This allows data elements to be overlaid in the same storage if they are not used by concurrently executing program modules.

6.4.3.2 Efficiency Hinderances

The significant language feature in HAL that may hinder the generation of efficient object code is shared data protection. The algorithm used by HAL to protect the integrity of data shared between asynchronous program modules may result in relatively inefficient code. However, whether a more efficient, simpler algorithm can be designed, or whether protection is better accomplished by program design is an open question.

6.4.4 CMS-2

The features available in this language are largely identical to those provided in SPL, but only part of them are provided.

6.4.4.1 Efficiency Aids

CMS-2 language features that aid in the generation of efficient object code for time efficiency are:

- o Constant attribute (Paragraph 6.4.1.1)

  CMS-2 provides a similarly usable feature through its EQUALS/MEANS declaration.

- o Hardware declaration (Paragraph 6.4.1.1)

  CMS-2 provides this capability through its SYS-INDEX and LOC-INDEX declarations.

Storage efficiency features include data packing attributes and an overlay declaration equivalent to SPL's capabilities (Paragraph 6.4.1.1).

6.4.4.2 Efficiency Hinderances

CMS-2's SYS-INDEX and LOC-INDEX declarations may interfere with a time optimizing compiler in the same manner as SPL's hardware declarations.

6.5 Source Program Readability

The primary conclusion reached from the evaluation of the readability of the languages is that none of them result in a program listing which, by itself, contributes significantly to the reader's understanding of the problem being solved. The most that can be said is that a reader who thoroughly understands the problem and the constraints on it, can after considerable effort understand how it is being solved, and with some additional effort can determine to a large extent if it is being solved correctly. The inability of the language to communicate a description of the problem itself relates back to the expression discussion of Paragraph 6.1. The statements of the language describe operations at such a low level of detail that the broader, more informative, elements of the problem are never articulated concisely. Until the major high level functions performed in a flight program can be clearly identified and isolated, and until standard techniques for performing those functions can be established and widely accepted, the programmer's task will continue to be one of describing the detailed steps to solve very esoteric problems. In that environment only the reader who is concerned about those detailed steps should be reading a program listing.

As mentioned in Paragraph 6.1, decision tables and matrix/vector operations are the only innovative features presented by any of the evaluated languages which help to raise the level of detail described by a program listing.

The readability features of the language were provided at such a detailed level that the overall evaluation criteria questions of the Phase I Report revealed very little distinction among the languages as indicated below:

- How dependent is the readability of the language upon externally imposed writing disciplines?

  All of the languages reviewed have fairly rigid syntaxes for individual statements. The only significant discipline beyond that level concerns the ordering of statements within an overall program organization.

  The following list indicates the relative level of discipline imposed by each of the languages:

  - HAL (most disciplined)

- CLASP
- CMS-2
- SPL

These differences reflected primarily where declaration statements must be placed relative to executable statements. The only advantage to readability is that the reader can be more certain where he should look in a listing for a given data declaration.

This relatively minor consideration is the most significant writing discipline imposed by any of the languages.

o Does the language provide notational aids to enhance readability?

Each of the languages provides for comments to be inserted by the programmer. The only other significant notational aid is HAL's data type notation attached to data item names by the compiler. The effect on readability is discussed in Paragraph 8.3.3.

o Are "language extensions" available to enhance readability?

All four languages provide for "extensions" in the form of procedures and functions which can consolidate detailed program steps under a descriptive label.

Aside from the above issues, the features of the language communicate information to the reader at a very detailed level.

Consequently, the differences exhibited by the languages are limited almost exclusively to how well they help a reader, who thoroughly understands the problem being solved, to see what steps are being taken to solve that problem. As demonstrated by the sample coding performed in this evaluation, several overall problems exist even in trying to read the details of a program. Most of the individual program statements in the coded kernels are readable. However, the relationships among statements, and therefore, the program itself are very difficult to determine from the coding. The most significant

stumbling blocks to program readability are:

- Understanding the purpose and meaning of variables being manipulated. In many cases, the manipulation is very easy to understand, but the use for the result and the physical interpretation of the operands are not clear.

- Determining the sources of update values for variables. Even when an appropriate assignment statement is found, it is not clear that this is the only update source. Variable cross reference listings are helpful, but cumbersome, and do not differentiate between updates and "read-only" accesses.

- Understanding the sequence of program execution through branch points. It is very frequently difficult for the reader to determine, understand, or remember the reasons for alternate execution paths.

- Understanding the relationships among separate program modules. The overall program structure was not at all clear from the individual kernels. Even the supporting descriptions in the Equation Defining Document were lacking in this respect.

Because of the above shortcomings there is an unavoidable need for documentation aids such as flowcharts, textual descriptions, dictionaries of data items, and data file format diagrams. Some of these aids can be embedded in the program listing as comments, but this is not necessarily the easiest form to read. Appendix A of this report includes textual descriptions, flowcharts, and glossaries to assist the readers of the listings in Appendix B.

Within this context, several language features were found which enhanced or detracted from source program readability. These features are discussed below, organized by the languages in which they appear.

6.5.1 SPL

SPL's most significant contribution to readability is its decision tables, demonstrated in Paragraph 6.1.1.1. It also provides matrix and vector operations which expressed the Iterative Guidance Mode

equations more concisely than element by element operations. The third significant contribution to readability made by SPL is its status variable with the associated status constants (Paragraph 6.1.2.2).

Several features of SPL detract in a minor way from its overall readability. Nested assignment statements are confusing especially to someone new to them. As an example:

A = B + C = D + F

which is interpreted as "Set C equal to D + F. Then add B and set A equal to the result." Less confusing but still difficult to read is the format of the multiple assignment statement:

A, B, C = D, E, F

which is equivalent to the three statements A = D, B = E, and C = F. As the strings of multiple variables grow, the reader's task of counting commas to match corresponding entries becomes more formidable.

The program and subroutine declaration and calling sequence also detract from readability. The fact that the declaration format does not begin with the label of the module has already been discussed in Paragraph 6.3.2. The statement which calls the module PROG1, for example, is:

.PROG1

followed, if necessary, by input and output parameters. The use of a command word such as CALL (used by HAL) seems more descriptive. Another confusion in SPL's program structure is in finding the entry point to a program. Nominally, the main entry point to a program is its first executable statement. However, the TERM statement which concludes the source language statements of a program, may contain a label identifying some other statement as the entry point. Consequently, the reader must scan the listing to find the TERM statement before he can identify the entry point.

Either a statement label or a location variable (indirect address) may be the argument of a GOTO statement, and there is no difference in appearance between the two in the statement. Therefore, the reader does not know whether to scan statement labels or data declarations to find the destination.

6.5.2 CLASP

CLASP provides one readability feature which was very helpful in reading the kernel coding. The ability to preset data with formulas (Paragraph 6.1.1.4) made date values much easier to read. The features which contribute to SPL's readability are scarce in CLASP. There are no decision tables and no status variables, and matrix/vector operations are restricted to matrix multiplication.

CLASP shares some of SPL's detractions from readability, notably multiple assignment and the procedure declaration and calling sequence. CLASP has an additional readability restriction in that names and labels may not exceed eight characters. This did not affect the kernel coding greatly because most of the names and labels were taken directly from the existing Saturn flight program and did not exceed eight characters anyway. However, some names were created for the effort and these had to be curtailed in the CLASP coding.

6.5.3 HAL

HAL's primary contributions to readability are its two-dimensional statement format and its data type annotations (Item A2) generated by the compiler. The kernel coding in HAL (Appendex B) demonstrates both these features. A significant confusion factor is introduced in the HAL kernel coding, (Appendix B) because there is no special spacing to isolate separate statements. It is sometimes difficult to determine, at a glance, if a given character or string of characters is a subscript on the line above, or an exponent on the line below. To diminish this confusion, two alternative line spacings were attempted on a short sample of coding, as indicated in Figure 6-2. The first sample simply double spaces between statements, which improved separations somewhat. However, exponents and subscripts still did not seem closely enough associated to the main line. The second sample simulates the output of a printer which would half-space subscripts and exponents (assuming such a printer were available). The association is quite clear in this case, but the data type notation on the exponent line (the period over T2STAT) merges with the main line character and tends to lose its effectiveness. The samples demonstrate that the benefits of the two-dimensional format are considerably restricted by the mechanics of presenting it.

Even under these restrictions, there is a noticeable format

HAL 2-D FORMAT SPACING ALTERNATIVES

Double Space Between Statements

```
            IF EPTABLE              =0
S                       1,EPTINDX

                THEN DO

                     VTOLD = EPTABLE
S                                    2,EPTINDX

                     DLTTL       = DVTMR + VTOLD DKRTCSEC/40
S                         DQST2

E                    •
                     T2STAT      = TRUE
S                         DQST2

                END

E                    •
                ELSE T2STAT      = FALSE
S                         DQST2
```

Half-Space for Subscripts and Exponents

```
IF EPTABLE 1,EPTINDX  =0

     THEN DO
          VTOLD = EPTABLE 2,EPTINDX

          DLTTL DQST2 = DVTMR + VTOLD DKRTCSEC/40

          T2STAT DQST2 = TRUE

     END

     ELSE T2STAT DQST2 = FALSE
```

Figure 6-2

advantage to HAL in complex mathematical expressions. However, this is the least of the self-documenting problems, because the equations of a flight program are typically so well documented anyway. Even with programming in a conventional one-line format, it is a fairly straightforward process to take the written equations and verify that the coding accurately reflects them.

The more complicated readability problem is with such operations as decision logic, execution flow, and data retrieval. In these areas, the conventional single-line format for program statements seems as readable as HAL's.

The data type notation provided with HAL is useful. However, as mentioned early in this readability discussion, the reader needs to know the functional meaning of an operand and its sources of update, before he can fully understand the significance of a program statement. The data type notation tells only a small part of that story. Its primary utility is in serving as a clue to remembering the rest of the information about the data item.

HAL's matrix and vector notations improve its readability, but HAL provides neither the decision table nor the status variables of SPL. The break character in names and labels makes some of the longer ones (e.g., CHI_BAR_STEER) much more informative at first glance. The appearance of blanks in long strings of digits is also helpful (Paragraph 6.1.2.5). HAL's CALL NAME statement is much more readable than the .NAME procedure call used by the other languages. The overall program organization is more readily absorbed if the CLOSE statements,which conclude program and subprogram declarations, include the label of the module being concluded. HAL provides the option; it should be a requirement as in CMS-2.

6.5.4 CMS-2

CMS-2 provides the status variables of SPL but omits both decision tables and matrix/vector operations. As mentioned above, in CMS-2 the END-PROC statement which concludes a procedure declaration must include the label of the procedure. This is beneficial.

The most striking characteristic of CMS-2 is its elimination of the equal sign (=) as a character. This is an unexpected omission for an algorithmic language, and is inconsistent with its stated purpose of being developed for scientific applications. Assignment statements and loop control statements are still reasonably readable,but

unnatural without equal signs. Scanning is more difficult, because the statement begins with the term SET rather than the name of the variable being assigned. Another distraction to the reader is the dollar sign ($) which concludes every statement.

(BLANK)

7. LANGUAGE CHARACTERISTICS SUMMARIES

To perform a comparative evaluation of several programming languages, it is important to have available language descriptions in a common format and common terminology. However, there are no widely accepted documentation standards for describing a language specification, and the available specifications for the selected languages are widely divergent. Therefore, the salient characteristics of each of the four selected languages have been extracted and summarized in this section, in such a manner that direct comparisons can be made. These summaries are not intended to include evaluation of the characteristics, but simply to present them very briefly in a form which highlights the differences.

7.1 Characteristics Categories

To organize the presentation, the language characteristics have been categorized into seven major groups:

A General Language Characteristics

B Data Descriptions

C Data Manipulation

D Internal Program Sequencing and Control

E Program Structure

F External Data Access

G Compiler Directives

These major categories have been subdivided into more specific items, which are introduced in the following paragraphs. Figure 7-1 is a complete list of the items covered.

7.1.1 General Language Characteristics

Under this heading are described those general items which are reflected through many of the language characteristics. Specifically,

LANGUAGE CHARACTERISTICS COMPARISON ITEMS

No.	Title
A1	Statement Format
A2	Names, Labels, and Character Set
A3	Interaction with Other Languages
B1	Numeric Data Items
B2	Logical, Bit, Character Data Items
B3	Other Data Items
B4	Numeric Data Values
B5	Logical, Bit, Character Data Values
B6	Other Data Values
B7	Numeric Data Item Precision Attributes
B8	Presetting Data Values
B9	Other Data Item Attributes
B10	Alternate Data Declarations
B11	Data Organizations
B12	Index Types
B13	Default Data Item Characteristics
B14	Hardware Registers
C1	Arithmetic Operations
C2	Logical Operations
C3	Relational Operations
C4	Boolean Operations
C5	Explicit Data Conversions and String Operations
C6	Operations on Data Organizations
C7	Assignment Statements
C8	Scaling of Intermediate Results

No.	Title
D1	Direct Unconditional Transfers
D2	Variable Transfers
D3	Conditional Statements
D4	Decision Tables
D5	Iterative Loop Control
D6	Conditional Loop Control
D7	Statement Groups
D8	Stop Execution
E1	Overall Structure
E2	Programs
E3	Procedures
E4	Real-Time Tasks
E5	Functions
E6	Closes
E7	Program and Subroutine Returns
E8	Subroutine Parameters
E9	Priority Assignments
E10	Exclusive Subroutines and Interrupt Controls
E11	Error Recovery
E12	Library Subprograms
E13	Scope of Names and Labels
F1	Common Data
F2	Compile Time Data Protection Features
F3	Execution Time Data Protection Features
F4	Conventional Input/Output
F5	Real-Time Input/Output
G1	Optimization Directives
G2	Memory Allocation Controls
G3	Program Debug Aids
G4	Compile Time Identifiers

Figure 7-1

the following items are included:

A1 Statement Format

Describes the general format of the statements used to write programs. It includes techniques for inserting comments.

A2 Names, Labels, and Character Set

Describes the rules for forming user-defined data item names and statement labels in the language, as well as the general character set used.

A3 Interaction with Other Languages

Describes the capability to interact with programs written in another programming language.

7.1.2 Data Descriptions

These items include the types and organizations of data which can be manipulated in the language and the techniques for describing them. The manipulation of this data is described in another category.

B1 Numeric Data Items
B2 Logical, Bit, Character Data Items
B3 Other Data Items

These three items include all of the types of individual data elements which can be specified in the language. It is limited to representations of individual data elements; the ways in which these elements can be organized into structures is described in Item B11.

B4 Numeric Data Values
B5 Logical, Bit, Character Data Values
B6 Other Data Values

These three items describe the techniques for specifying actual values which can be assigned to (stored in) data items.

B7 Numeric Data Item Precision Attributes

Because the length of a computer word is limited, the precision with which it can represent a numeric value is limited. The programmer is given some control over how values are aligned in data items to take best advantage of available precision. This sometimes includes the allocation of multiple computer words to provide the desired precision.

B8 Presetting Data Values

This item describes the techniques for assigning permanent values to constants, or initial values to variables.

B9 Other Data Item Attributes

Each language provides for specification of selected data item attributes in addition to those described earlier.

B10 Alternate Data Declarations

Certain features are provided in some languages to simplify the declaration of data items.

B11 Data Organizations

This item describes the ways in which individual data elements can be organized into arrays, tables, and structures.

B12 Index Types

Data elements within data organizations are generally accessed by specifying indices whose values identify the element. This item describes the techniques for handling indices.

B13 Default Data Item Characteristics

To save the programmer from having to specify in detail every attribute of a data item, default characteristics are defined. In some cases, the programmer has control over the defaults.

B14 Hardware Registers

This item describes language capabilities to declare and directly access internal registers in the computer.

7.1.3 Data Manipulation

Programming languages typically have a variety of operations for manipulation of different types of data elements. The following items describe these operations and language features to support them:

C1 Arithmetic Operations
C2 Logical Operations
C3 Relational Operations
C4 Boolean Operations

These items describe the operations available, their applicability to different data types, and the conversions which are made when data types are mixed.

C5 Explicit Data Conversions

In addition to data conversions made implicitly in a mixed operation, there are special functions provided to convert a data element or portion of a data element to another data type.

C6 Operations on Data Organizations

Some of the data manipulating operations can be applied to arrays or tables of data.

C7 Assignment Statements

Assignment statements are the mechanism for changing the value of a data item.

C8 Scaling of Intermediate Results

In addition to the data precision attributes which can be assigned to data items at their declaration (Item B7) there are scaling operators which can be used in formulas to override declared scaling or to assign specific scaling to intermediate expressions in a formula.

7.1.4 Internal Program Sequencing and Control

Language features to specify execution control within a program module are described in this category. These features include all transfer type statements and any conditional statements, whether or not they change the sequence of statement execution.

D1 Direct Unconditional Transfers
D2 Variable Transfers

These are the simple "GOTO" type statements which transfer to a single statement or to one which may be selected dynamically.

D3 Conditional Statements

These are the statements which check relations among data items and execute different responses depending upon whether the relationship is true or false.

D4 Decision Tables

A Decision Table is a tabular combination of several conditional statements.

D5 Iterative Loop Control
D6 Conditional Loop Control

Loop control provides techniques for executing groups of statements several times in succession. This can occur for a specified number of iterations, or until some condition is satisfied, or both.

D7 Statement Groups

Conditional statements and loop control create a need for grouping statements. This item describes the techniques for delimiting groups.

D8 Stop Execution

Features are provided to stop computer execution and, in some cases, specify a restart point.

7.1.5 Program Structure

Programming languages have characteristics which imply, or explicitly direct, a particular program structure or modularity. They also include features for controlling program execution between modules. These characteristics are described in the following items.

E1 Overall Structure
E2 Programs
E3 Procedures
E4 Real-Time Tasks
E5 Functions
E6 Closes
E7 Program and Subroutine Returns

> These items describe the ways in which total programs are organized and the characteristics of the modules which make them up. Techniques for causing execution of individual modules and returning control from them are also covered.

E8 Subroutine Parameters

> Procedures and functions generally are defined to operate on data which is made available to them when they are executed. The techniques for making that data available and receiving results are described in this item.

E9 Priority Assignments
E10 Exclusive Subroutines and Interrupt Controls
E11 Error Recovery

> These items describe language capabilities for controlling an orderly execution of program modules in real time.

E12 Library Subprograms

> This item describes special purpose subroutines which are provided with the languages and can be used in the same manner as subroutines defined by the programmer.

E13 Scope of Names and Labels

Data item names and statement labels are generally recognized only within the area (scope) of the program organization where they are defined. This item describes the rules by which this scope is established and special features for extending or restricting the scope of a particular name or label.

7.1.6 External Data Access

The program modularity introduced by language features to describe program structures also introduces the problem of communication among separately compiled program modules. Language features to facilitate access to data outside a given program module, including data outside the computer, are described in the following items:

F1 Common Data

Common data is data that can be accessed by the entire program organization. The techniques for declaring such data and for access by individual program modules are described in this item.

F2 Compile Time Data Protection Features

These are features to prevent erroneous access to compool data.

F3 Execution Time Data Protection Features

In a real time application, data manipulation errors can occur when one program module interrupts another and accesses data which the interrupted module was accessing. Features to protect data from these multiple concurrent accesses are described in this item.

F4 Conventional Input/Output
F5 Real Time Input/Output

These items describe the capabilities for specifying communication with conventional peripheral equipment such as card reader, printer, and magnetic tape and disk, as well as avionics equipment and other space vehicle hardware.

7.1.7 Special Compiler Directives

Special directives to the compiler provide selected additional capabilities beyond those described in preceding categories. These are described in the following items.

G1 Optimization Directives

These are capabilities to inform the compiler or direct the compiler in such a way that it will produce more efficient object code.

G2 Memory Allocation Controls

These features direct the compiler in organizing and locating data items in memory.

G3 Program Debug Aids

These are items which direct the compiler to generate information or special object code which will aid in evaluating the operation and performance of the program.

G4 Compile Time Identifiers

The compile time identifier capability allows data such as constant data values to be represented symbolically throughout the source program and be assigned a literal meaning in one place.

G5 Macro Statement Definitions

The macro statement definition capability allows the programmer to introduce new statements into a language by defining them in terms of existing statements.

7.2 Terminology

From the different terminologies used in the four language specifications, an attempt was made to select a single set of terms and use it consistently throughout the summary characteristics descriptions. Where appropriate, the characteristics descriptions include terms which are part of the language (key words) and these are capitalized to indicate that they are key words. The most widely used non-key word terms are defined in the glossary.

7.3 Characteristics Descriptions

The following pages describe the pertinent characteristics of the languages under the comparison items introduced in Paragraph 7.1.

A1 Statement Format

HAL

Two-dimensional statement format allows exponents and subscripts to be designated by appearance on separate lines. Each line starts with a single character identifying the line type (Exponent, Main, or Subscript). Statements end with a semicolon(;) and may be of any length. Entities may be continued on next line. An alternate single line format is available where exponents are preceded by ** and subscripts by $.

Comments are delimited by slash and asterisk: /*COMMENT*/. May appear anywhere a blank is legal, even within a character-string. May appear on separate comment lines (C in column 1) without delimiters. May contain any characters except the */ sequence.

CMS-2

Columns 1 through 10 of every card are reserved for an identification field; columns 11 through 80 are available for statements. Statements may be of any length and every statement is terminated by a $. No continuation character is needed for multi-line statements. Exponents, subscripts are like SPL.

Comments are the same as in SPL, but cannot include $. Also, comments on separate lines may be introduced by the keyword COMMENT, and may omit double apostrophe.

SPL

Statements may be of any length and have no delimiters. Entities may be continued on next line. Multiple statements may appear on one line without delimiters. Exponents are preceded by ** and subscripts are parenthesized.

Comments are delimited by double apostrophe: ''COMMENT''. May appear anywhere a blank is legal except within a textual constant (character-string). May not contain multiple consecutive apostrophes or end in apostrophes.

CLASP

Statement lines must appear in columns 1 through 72. Statements may be continued to the next line by a continuation flag in column 73. Multiple statements on the same line must be separated by a $. Exponents are preceded by ** and subscripts are enclosed in parentheses.

Comments are the same as in SPL.

A2 Names, Labels, and Character Set

HAL

Names and labels consist of up to 32 alphanumeric characters, including the "break" character (_). First character must be a letter. Names of certain data types and structures are annotated by the compiler: $\overline{V}$ = Vector, $\overset{*}{M}$ = Matrix, $\dot{B}$ = Bit String, $\overset{'}{C}$ = Character String, [A] = Array, $\{S\}$ = Structure. Combinations are also possible: $[\overset{*}{A}]$ = Array of Matrices, $[\dot{B}]$ = Array of bit-strings. Statement labels are followed by colons.

The character set is the same as SPL but also includes lower case letters (a-z) and > < ¬ ↑ & ; : - # @ [ ] { } , $

CMS-2

Names and labels are the same as in SPL.

The character set is the same as SPL plus $ and ∆ (equivalent to blank in input/output formats), but = is not included.

SPL

Names and labels can be unlimited length string of alphanumeric characters, beginning with a letter. Statement labels are followed by periods.

The character set is made up of upper case letters (A-Z), decimal digits (0-9), blank, and + - * / , ( ) . ' =

CLASP

Names and labels are the same as in SPL, but are restricted to eight characters in length.

The character set is the same as SPL plus $

A3 Interaction with Other Languages

HAL

No explicit capability, but in a given implementation it should be possible to call or schedule programs written in another language.

CMS-2

Direct code is delimited by a DIRECT statement at the beginning and a CMS-2 statement at the end.

SPL

Assembly language can be inserted directly into a sequence of SPL statements; it must be delimited by a DIRECT and END statement. Variable names and statement labels declared in the SPL coding are recognized in the direct coding, and an SPL statement may GO TO a direct code label. (This implies compatible naming conventions between SPL and the assembler.) SPL provides an ASSIGN statement which can be used in direct code to transfer data between a computer hardware register and data items declared in the SPL code.

SPL provides the ability to switch between SPL and the Basic JOVIAL language at any point in the program.

CLASP

Same as SPL DIRECT and END capability but there is no ASSIGN statement.

B1 Numeric Data Items

HAL

INTEGER and SCALAR. SCALARs are floating point if floating point hardware is available in the target computer, and fixed point otherwise. Vectors and Matrices are discussed under Data Structures (Item B11).

CMS-2

Same as SPL (I, A, and F are declaratives for integer, fixed, and floating).

SPL

INTEGER, FIXED point, and FLOATING point. FLOATING point is available only if the target computer has floating point hardware.

CLASP

Same as SPL.

B2 Logical, Bit, Character Data Items

HAL

BIT-string (length fixed at declaration)
Boolean (declared as a BIT-string of length one)
CHARACTER-string (length fixed at declaration)
Variable length CHARACTER-string (maximum length fixed at declaration)

CMS-2

Boolean (declared as B)
Character-string (declared as H for Hollerith) with length fixed at declaration
There are no logical data items and no logical formulas.

SPL

LOGICAL (bit-string with length fixed at declaration)
BOOLEAN (declared as BOOLEAN)
TEXT (character-string with length fixed at declaration)
TEXT with secondary type is available for input/output (Item D4).

CLASP

BOOLEAN (declared as BOOLEAN)
TEXT (character-string with length fixed at declaration)
There is no logical data item, but logical constants (Item B5) and formulas (Item C2) are used.

B3 Other Data Items

HAL

None.

CMS-2

Status variables (declared as S) are the same as SPL's.

SPL

LOCATION variables contain addresses (memory locations) of data items or imperative statements. They are used for indirect access to data and for an indirect GOTO statement (Item D2). They are declared with a secondary data type to indicate the type of data item being indirectly addressed so the compiler may generate code to perform the proper conversions.

CONTEXTUAL data items are provided for temporary storage within a program. They are declared without fixed attributes, and each time they are assigned a value in the program they take on the attributes of the expression from which they were assigned. Only the word size (number of bits) is specified in the declaration.

STATUS variables are like BOOLEAN variables but have multiple states. The programmer may declare any number of states for a given status variable. A programmer defined name (STATUS constants - Item B6) is assigned to each of the states in the declaration of the variable.

CLASP

Location variables are not provided as data items, but location constants (Item B6) reference memory locations of data items and statements and can be used in expressions. Indirect addressing is not provided.

The TEMPorary data attribute provides the same capability as SPL's CONTEXTUAL data item but allows it only for fixed-point data items. Only the scaling of the fixed point value is determined from the assignment statement.

(BLANK)

B4 Numeric Data Values
(dd = string of zero or more decimal digits)

HAL

Both integer and scalar literals have the same form:

$$dd.dd \left\{ \begin{matrix} E\pm dd \\ B\pm dd \\ H\pm dd \end{matrix} \right\}$$

where the E, B, and H expressions represent powers of 10, 2, and 16 respectively. These expressions are optional and may be used in any combination in a single literal. If the value of the literal is a whole number, the result is an integer, regardless of the form of expression. If the value has a fractional part, the result is a scalar. The decimal point is optional if there is no fractional part. Scaling of a (fixed point) scalar is determined entirely by the context of its use.

CMS-2

Decimal numeric literals are expressed as:

integer:	ddD
fixed or floating point:	dd.ddE±ddD

where the E-expression is a power of 10 and the D indicates decimal number. Fixed-point scaling is determined by context. The same general formats are used for hexadecimal (followed by S indicator instead of D) and octal (no indicator required). In these cases, all digits are interpreted in the indicated number system and the E-expression is a power of 16 or 8 respectively. Hexadecimal numbers must begin with a numeric digit.

SPL

Integer, fixed point, and floating point literals each have different forms:

Integer:	ddEdd
Floating Point:	dd.ddE±dd
Fixed Point:	dd.ddE±dd A±dd

The E-expression represents a power of 10 and is optional in all forms. The A-expression is a required scaling factor to determine the number of fractional bit positions in the computer word. The decimal point is required in both non-integer forms.

CLASP

CLASP forms are identical to SPL except that integers may not have an E-expression, and the A-expression is optional (scaling determined by context).

B5 Logical, Bit, Character Data Values

HAL

Four forms of bit-string literals are available:

BIN 'binary digit string'
OCT 'octal digit string'
HEX 'hexadecimal digit string'
DEC 'decimal digit string'

Blanks may be embedded within digit strings but have no significance. The first three forms may include an integer repetition factor to repeat the digit string pattern.

Special forms are available for Booleans:

TRUE or ON can be used for BIN '1'
FALSE or OFF can be used for BIN '0'

Two forms of character-string literals are available:

'character-string'
CHAR 'character-string'

The CHAR form may have a repetition factor to repeat the pattern. Within a character-string double apostrophe ('') must be written to represent a single apostrophe ('), and the sequence /* starts a comment (Item A1).

CMS-2

Bit-strings may be generated as numeric integers (Item B4).

Boolean literals are expressed as the integers 1 and 0.

Hollerith (character-string) literals are expressed as:

H(character-string)

Within a character-string, two consecutive right-parentheses or left-parentheses must be written to represent a single right-parenthesis or left-parenthesis (just as HAL does for apostrophes).

SPL

Logical (bit-string) literals have the same forms as HAL bit-strings except they do not include DEC. Additionally, the characters B, O, and X may be used in place of BIN, OCT, and HEX, respectively. Blanks may not be embedded in digit strings.

Boolean literals are:

'TRUE'	'ON'	(true condition)
'FALSE'	'OFF'	(false condition)

Note that the apostrophes are part of the literal expression.

Two forms of textual (character-string) literals are available:

```
   'character string'
nT'character string'
```

where n indicates the number of characters in the string and T indicates "textual". The second form is required if apostrophes are to be embedded in the character-string.

CLASP

Logical (bit-string) literal forms are identical to SPL but only the B, O, and X keywords are permitted. Blanks may not be embedded in digit strings. Boolean literals are TRUE, ON, FALSE, OFF.

Textual (character-string) literals are identical to SPL except that the T is not required in the second form.

B6 Other Data Values

HAL

None.

CMS-2

Status constants are defined as in SPL, and can include any characters except dollar sign ($) and apostrophe (').

SPL

Location constants have three forms:

LOC 'label'	for statement label
LOC 'name'	for data items
LOC 'name(sub)'	for arrays, tables, and table items

The subscript sub (see Item B12), identifies a specific element.

Status constants have the form:

'name'

Every status variable (Item B3) has a status constant assigned to each of its defined states at declaration. The status constants are represented internally as integers, but may be represented in the source code by the defined literal 'name'.

CLASP

Location constants are defined as in SPL.

B7 Numeric Data Item Precision Attributes

HAL

If SCALARs are floating point, the only declared precision attribute is specification of the number of significant decimal digits required. This number is used only to determine whether a single precision or a multiple precision word will be used.

If SCALARs are fixed point, the declaration includes the required number of integer bits and the required number of fractional bits. These values are used both to determine single or multiple precision and to align data values within the computer word.

The word length of INTEGERs is fixed and not under programmer control.

CMS-2

Floating point data items have no precision attributes.

Fixed point and integer data items have the same precision attributes as SPL.

SPL

FLOATING point data items have the same precision attributes as HAL.

FIXED point data items have declared attributes for the total number of bits and the number of fractional bits. The total number of bits determines if single or multiple precision is required and the number of fractional bits aligns data values within the computer word.

INTEGER data items have an attribute for the total number of bits.

CLASP

FIXED and FLOATING point data items have the same precision attributes as their SPL counterparts. However, FIXED point precision attribute declarations are considered minimum precision requirements. In an effort to reduce scaling shifts during program execution, the compiler may allocate more fractional bits than are requested.

INTEGERs have no precision attribute. Maximum value is the highest computer memory address.

B8 Presetting Data Values

HAL

Any data item may be declared CONSTANT and have its permanent value assigned in the declaration, or it may be declared INITIAL and have its initial value assigned in the declaration. In either case the value must be a literal, or list of literals if the data item has multiple elements. Literal lists may include repetition factors to avoid writing the same literal many times.

CMS-2

Initial values may be preset by appearance of a literal value in the declaration, or through a separate DATA statement. Numeric values in DATA statements may be followed by an integer which places the binary point within the value, to align it with the binary point of the data item being preset.

CMS-2 data items cannot be declared constant.

SPL

Same basic declaration capability as HAL. In addition, there is a PRESET statement which allows preset (initial or constant) values to be assigned to portions (e.g. selected columns or rows) of arrays and tables. The values must be literals or lists of literals.

CLASP

Same basic declaration capability as HAL, with the additional capability of representing preset values with formulas consisting of literals and predefined CONSTANTS.

B 9 Other Data Item Attributes

HAL

See Item F3 for LOCKTYPE and Item G2 for data packing and dynamic storage allocation. See Item F2 for access rights.

CMS-2

Integers and fixed point data items may be declared signed (S) or unsigned (U) as in SPL.

SPL

Integers and fixed point data items may be declared SIGNED or UNSIGNED.

Any arithmetic data items may be declared ROUND. When a ROUNDed data item is assigned to a data item or expression of higher precision than its own, the result is rounded rather than simply truncated.

A data item may be assigned minimum and maximum values which are used to scale intermediate results of expressions.

See Item G2 for data packing.

CLASP

The PARAMETER attribute may be assigned to any data item which is a constant during any program execution, but whose precise value is not known at the time of program compilation. PARAMETER values may be changed without recompilation, but data type and attributes may not.

(BLANK)

B 10 Alternative Data Declarations

HAL

Selected data items may be declared implicitly by appearance of the item name in the program. Data type is determined by the notation on the exponent line over the item name indicating vector (_), matrix (*), character string ('), bit string (.), or scalar (blank). Default attributes (Item B13) for the data type are assigned. To avoid inadvertently using an already declared name for an implicit declaration, an OUTER statement (Item E13) is provided to selectively isolate a block of code from names declared outside it.

The data type notation on the exponent line may also be used in a DECLARE statement instead of writing out the word describing the data type.

Factored DECLARE statements may be used to simplify the declaration of many data items with similar characteristics. The common (factored) attributes are listed immediately following the DECLARE statement, and the data items follow the factor list specifying only additional attributes. These may override the common attributes for specific data items.

CMS-2

Implicit declarations are not permitted, but a MODE statement, as in SPL, defines default data type and attributes. Any data item declared (explicitly) with no type or attributes specified takes on the default data type and attributes.

Declarations can be factored only to the extent that multiple data items may all be given exactly the same attributes.

SPL

SPL also allows data items to be declared implicitly by appearance, but the notation does not permit distinction among data types. Instead, a MODE statement is used to define a data type and specific attributes, and implicitly declared data items take on the type and characteristics of the most recent MODE statement.

Factored declarations are provided.

CLASP

Implicit declarations are not permitted.

Factored declarations are provided but all items within a single declaration must be of the same data type.

B11 Data Organizations

HAL

VECTOR and MATRIX forms can be defined explicitly. Their data elements are SCALAR, by definition. A full complement of VECTOR and MATRIX arithmetic functions is provided (see Item C8).

ARRAYs can be defined to be made up of any data type, including VECTOR and MATRIX. Maximum number of dimensions is implementation dependent.

Structures allow data to be organized in a hierarchy of data levels. Each element at any level of a structure may be either a data item (including ARRAY, VECTOR, or MATRIX types) or a substructure (minor structure) which is made up of elements on the next lower hierarchical level. A structure may be QUALIFIED, or NON QUALIFIED (unique structure element names are used at each level).

Multiple copies of a structure may be declared for the major structure at the top level, as well as for any minor structure at lower levels. Specific copies are accessed by indices in the structure element names.

CMS-2

Tables of items may be defined, where table-items are further structured into named fields. Table-items are selected by an index, and fields within items are selected by name. The term TABLE1 (Index, NAME) retrieves the data element from field NAME of item number "Index" in TABLE1. A field can contain any data type (Items B1, B2, and B3). The programmer can exercise full control over field definitions and can overlap fields, providing for variable data structures.

Tables may have one, two, or three dimensions, and require the corresponding number of indices. The field name is in addition to the dimension indices.

SPL

ARRAYS can be defined to be made up of any data type. Maximum number of dimensions is implementation dependent. ARRAYs are used to represent vectors and matrices.

TABLEs are two-level data organizations with numbered entries and named items within the entries. The term NAME(Index) retrieves table item NAME from the entry number "Index". Features are provided for both variable structures and variable length table entries.

Group structures can be defined to organize unrelated data declarations under a common group name. This concept allows different data tables, arrays, and items to have the same declared name and provide unique references by qualifying it with the group name (e.g. GROUP1'ITEM and GROUP2'ITEM). The same type of qualified name applies to items within tables if items in two different tables have the same name (e.g. TABLE1'ITEM and TABLE2'ITEM).

CLASP

ARRAYs are the same as for SPL but cannot have more than three dimensions.

Groups are similar to SPL but all items within a group must be of the same data type. Qualified names are not provided and the only place a group name can be used is on the left side of a multiple assignment statement (Item C7).

B12 Index Types

HAL

Indices are written as subscripts on a data item name. They can specify either single elements or multiple-element partition of vectors, matrices, bit-strings, character-strings, arrays, and structures. Individual index expressions may be any valid arithmetic expression which results in a single value.

Three index forms are available to select a multiple-element partition of a data organization:

- An asterisk (*) in any index position selects all elements in the corresponding row, column, plane, etc. of the organization ($M_{*,4}$ selects the entire fourth column of the matrix M).
- The phrase A TO B in any index position selects elements A through B, inclusive, of the corresponding dimension ($M_{1\ TO\ 3,4}$ selects the first 3 elements of column 4).
- The phrase A AT B in any index position selects A elements beginning with element B ($M_{2\ AT\ 4,4}$ selects elements 4 and 5 of column 4).

In the most general case, structures may contain arrays which may contain vectors, matrices or strings.

CMS-2

Indices are included in parentheses following a table name. Any literal value, data item name, tag (Item G4), or hardware index name (Item B14) may be used. The only formula allowed is a variable $\pm$ a constant.

The only multiple-element partition capability (such as HAL's asterisk and SPL's blank) is the ability to access an entire table item (all fields) within a TABLE (Item B11).

SPL

Indices are enclosed in parentheses following a data name. They can specify either single elements or multiple element partitions of tables, table items, arrays, bit-strings, and byte (character) strings. Any valid arithmetic expression which results in a single value may be used. A blank in any index position is treated the same as HAL's asterisk.

Implicit indices may be named for an array at the time it is declared. If the array name appears without an index expression, the implicit indices are used. Note that ARRAY has no index expression so implicit indices are used, but ARRAY( ) has blank indices so the entire array is referenced. This notation applies only to arrays which have implicit indices declared. If the array has no implicit indices, then the term ARRAY references the entire array.

CLASP

Indices are enclosed in parentheses following an array name, and are used only with arrays. The index must be an integer constant or variable or a simple integer formula of the form:

$$\text{variable} \left\{ \begin{matrix} * \\ / \end{matrix} \right\} \text{constant} \pm \text{constant}$$

where all operands must be integers.

An asterisk (*) in an index position is treated as in HAL. SPL's implicit indices are also available.

B13 Default Data Item Characteristics

HAL

System defaults are provided for essentially all declarable data item characteristics. A DEFAULT statement can be used to modify these default characteristics independently for each data type. The DEFAULT statement applies only to the program block in which it is defined (and sub-blocks within that block).

CMS-2

Default characteristics are implementation dependent. They are under programmer control only for MODE-defined data items (Item B10).

SPL

Default characteristics are defined but they are under programmer control only for implicitly declared data items (Item B10).

CLASP

Default characteristics are defined but there is no programmer control of them.

B14 Hardware Registers

HAL

None.

CMS-2

Index registers may be declared, assigned names, and accessed directly as if they were data items. They may be declared on the system level (SYS-INDEX) or the local level (LOC-INDEX).

SPL

HARDWARE registers within the computer such as accumulators, index registers, and base registers can be declared, assigned names, and accessed directly as if they were data items. They may also be assigned data types and attributes like any other data item. These registers are used to communicate with direct assembly language code (Item A6). LOCK and UNLOCK directives (Item E10) may be used to prevent the compiler from manipulating these registers while the programmer is using them.

CLASP

Same capabilities as SPL.

C1 Arithmetic Operations

HAL

Arithmetic operations on scalars include:

Prefix minus (-)
Addition (+)
Subtraction (-)
Multiplication (blank)
Division (/)
Exponentiation (exponent as superscript on exponent line)

Arithmetic operations on vectors and matrices include:

Prefix minus (-)
Addition (+) and subtraction (-)
Vector Dot Product (A. B)
Vector Cross Product (A*B)
Multiplication (blank) or division (/) by a scalar
Multiplication (blank) by another vector or matrix
Matrix inversion and transpose

In any of the above operations, integers or bit-strings may be used in place of scalars. Implicit conversions are made as necessary.

CMS-2

Arithmetic operations on scalars are the same as SPL. There are no vector or matrix operations.

SPL

Arithmetic operations in SPL provide the same basic capabilities as those in HAL. Differences are primarily in notation for multiplication (*), exponentiation (**), dot product (*), and cross product (/*). While vectors and matrices are not explicitly declared in SPL, vector operations apply to one-dimensional arrays and matrix operations to two-dimensional arrays. Appropriate partitions of higher-dimension arrays can also be used for vectors and matrices.

CLASP

Arithmetic operations on scalars are the same as for SPL. The only matrix operation available is matrix multiplication (/*/).

C2 Logical Operations

HAL

Bit-string operations include:

- Complement (NOT or ¬)
- Logical AND (AND or &)
- Logical OR (OR or !)
- Concatenation (CAT or !!)

The only character-string operation is concatenation.

NOT, AND, and OR can be applied only to bit-strings. For concatenation, both operands must be bit-strings or one argument must be a character-string. Bit-strings can be concatenated on the end of character-strings but not character-strings on bit-strings.

No exclusive OR and no explicit shift operations are provided.

CMS-2

No capability.

SPL

Logical operations include:

- Logical AND (LAND)
- Logical OR (LOR)
- Exclusive OR (LXOR)
- Right Shift (RSH)
- Left Shift (LSH)

These operations may be applied to any data type or combinations of data types. Shift operations are logical shifts if the argument is logical or textual, and algebraic shifts if the argument is numeric.

Complement and concatenation are excluded.

CLASP

Operations are identical to SPL except that all shift operations are algebraic, and textual data cannot be shifted.

C3 Relational Operations

Symbologies for available relational operations are shown below:

	not equal	equal	less than	greater than	less than or equal	greater than or equal	not less than	not greater than
HAL	¬ =	=	<	>	< =	> =	¬ <	¬ >
CMS-2	NOT	EQ	LT	GT	LTEQ	GTEQ	not available	not available
SPL	NQ	EQ	LS	GR	LQ	GQ		
CLASP	NQ	EQ	LS	GR	LQ	GQ		

HAL

Vectors and matrices can be compared only with "equal" and "not equal" operations. Only integer, scalar, or bit-string formulas may be mixed in relational formulas. (NOT may be used for ¬ in any operation.)

CMS-2

Textual (Hollerith), status, and Boolean Formulas may be compared only with "equal" and "not equal". Numeric formulas may be compared with any of the operations.

SPL

Single-valued formulas of any type may be compared in any combination.

CLASP

Textual and logical formulas may be compared only with "equal" and "not equal". Numeric formulas may be compared with any of the operations.

C4 Boolean Operations

HAL

All of the bit-string logical operators (NOT, OR, AND, ConCATenate) can be applied to single-bit bit-strings or to the results of relational operations (Item C3). There are no operations which are exclusively Boolean.

CMS-2

Boolean operations include:

COMP	(complement)
OR	(true if either operand true)
AND	(true if both operands true)

Operands must be Boolean. COMP is implemented as an open function; the argument must be enclosed in parentheses.

SPL

Boolean operations include:

NOT	(complement)
OR	(true if either operand true)
AND	(true if both operands true)
EQUIV	(true if both operands equal)

Operands must be Boolean.

CLASP

Same as SPL.

C5 Explicit Data Conversions

HAL

The following data element conversion functions are provided:

INTEGER (argument)
SCALAR (argument)
BIT (argument)
CHARACTER (argument)

These functions convert single data element formulas to the indicated data type, and convert array, vector, or matrix formulas to arrays of elements of the indicated data type. Subscripts on these functions can convert lists of arguments to arrays of the indicated data type and with dimensions dictated by the subscripts. Subscripts on BIT and CHARACTER can also select a portion of an argument for conversion (Item B12).

Selected portions of bit-strings and character-strings can be assigned values without modifying the remainder of the string. This is accomplished by subscripting the string name on the left side of an assignment statement.

Other data types may also be partially assigned by using the SUBBIT (argument) operator which allows any argument to be treated as a bit-string. A subscript on the operator identifies the bits to be changed.

Two other conversion functions:

VECTOR (argument)
MATRIX (argument)

convert arguments to scalar data types and form them into vectors and arrays.

CMS-2

The same BIT and CHARacter operations as provided in HAL. CHAR may be used on the left side of an assignment statement to assign a portion of a character-string, but BIT may not.

SPL

Almost all data conversions are performed implicitly in SPL, and there are no functions provided exclusively for data type conversion. However, BIT and BYTE modifiers extract portions of data items and interpret them as bit-strings and character-strings, respectively, similar to their HAL counterparts. Indices on the modifier indicate how many consecutive bits (bytes) to extract and at which bit (byte) to start. BYTE operates only on textual data items, but BIT operates on any type of data item.

BIT and BYTE can both be used on the left side of an assignment statement to assign a portion of a data item.

The scaling operator (Item C8) converts to fixed point data.

CLASP

A built-in function UNPACK extracts selected bits from a formula. A corresponding procedure, PACK, inserts the low order bits of a numeric formula into selected bit positions of a numeric data item.

(BLANK)

C6 Operations on Data Organizations

HAL

All of the arithmetic and logical operations (Items C1 and C2) are directly applicable to arrays and array partitions of the appropriate data types. The two array operands must be the same size and shape, and the operation is performed element by element between pairs of corresponding elements in the two arrays. The "equal" and "not equal" relational operations may also be applied to arrays.

If only one of the operands is an array, then the indicated operation is performed in sequence using the single operand and each element of the array.

Data structures and multiple copies of structures may be treated as arrays in operations, provided they do not contain substructures.

CMS-2

No capability.

SPL

Arrays or array partitions which qualify as vectors (one dimension) or matrices (two dimensions) are subject to a comprehensive set of standard operations (Item C1). Other arrays are subject only to multiplication by a scalar and having scalars added to or subtracted from them. ("Scalar" refers to a single valued item, as opposed to table or array.) There are no two-array operations.

Tables (or table-entries) are subject only to "equal" and "not equal" comparisons, and can be compared only to integer zero, textual blank, or another table (or table-entry).

CLASP

All arithmetic and logical operations are apparently applicable to arrays and array partitions (the specification is not explicit). (Relational operations are not mentioned in conjunction with arrays.) Groups are not subject to any operations.

C7 Assignment Statements

HAL

Assignment statements are of the form:

variable, variable, . . . variable = formula

where multiple variable names must be all of the same data organization (single element, vector, array, etc.), and each variable is assigned the same value (or set of values if the data organization has multiple elements). The formula represents that value or set of values. If the data organization is multi-element, then either the formula is multi-element and assignment is made element by element or the formula is single-element and the single value is assigned to each element of each variable. The VECTOR and MATRIX conversion functions (Item C5) can group data element values for assignment to vectors, matrices, and arrays.

Integer, scalar, and bit-string formulas may be assigned to any data type, and conversions are implicit. Textual (character-string) formulas may be assigned only to textual variables.

CMS-2

Assignment statements for data elements are of the form:

SET variable, variable, . . . variable TO formula

where each variable is assigned the single value represented by the formula, as in HAL. Multiword assignment statements are restricted to:

SET Table TO Table
SET Table-Item TO Table-Item
SET Table or Table-Item TO formula

The first two cases assign element by element from one multiword data item to another. In the third case, the single-valued formula is assigned to each element of the table or table-item.

No data type conversions are made between arithmetic, Hollerith, Boolean, or status variables. Within an assignment statement, all variables and the formula must be of the same type. Conversions among fixed-point, floating point, and integer are made implicitly. Status variables may be assigned only to single status constants. Exchange assignment is available in the form:

SWAP variable AND variable

The same conversion and compatibility rules apply as for one-way assignments.

SPL

Assignment statements are of the form:

variable, variable, . . . variable = formula, formula, . . . formula

and are interpreted the same as for HAL if the variables are single-element and there is only one formula. If there are multiple formulas, each variable is assigned the value of the corresponding formula. Excess formulas are ignored; excess variables are each assigned the value of the last formula.

Tables (or table-entries) may be assigned only from integer zero, textual blank, or word for word from another table (or table-entry). An array (or table-item) may be assigned from another array (or table-item) or from a list of single-element formulas.

Implicit conversions are made among all data types except textual data which does not have a secondary type (Item D4). Such data can be assigned to (or from) textual or logical data only; assignments involving other types are undefined.

Nested assignment statements allow variables to be assigned values in the same statement in which they are being used. For example:

A = B + C = D

assigns the value of D to C; then adds the value of B and assigns the result to A.

Exchange assignment statements are of the form:

variable, . . . variable==variable, . . . variable

and each variable on the right exchanges values with the corresponding variable on the left.

CLASP

The basic assignment statement is identical to SPL except that the number of variables cannot exceed the number of formulas. Arrays may be assigned element by element from array formulas or may be assigned from a single-element formula (each element of the array gets assigned the same value).

Implicit conversions are made among all data types except that textual formulas may be assigned only to textual variables.

Exchange assignment is the same as SPL, but nested assignment is not available.

C8 Scaling of Intermediate Results

HAL

HAL code automatically scales the intermediate results of scalar expressions according to the declared scale factors of the operands and of the ultimate result. The programmer may force a different scaling on an intermediate result by subscripting the intermediate portion of the expression with @ followed by the desired precision expression. Either single or double precision can be specified along with the placing of the binary point.

By including the name of another variable in the precision expression, the programmer can place the binary point relative to the scaling of the named variable.

If scalars are floating point, the precision expression is simply a number specifying the total number of decimal digits to be allocated.

CMS-2

The CMS-2 scaling operator is of the form ..p where p is an integer literal or a tag (Item G4). It specifies the number of fractional bits to be allocated in the intermediate result, but cannot respecify the total number of bits.

SPL

The SPL scaling operator is of the form SCL or SCLR ("scale and round") followed by a precision expression which specifies either absolute placement of the binary point or the name of a fixed point variable whose scaling is to be used. The scaling operator can be applied to any numeric or logical expression and converts it to fixed point.

CLASP

The CLASP scaling operator is of the form .S followed by a precision expression. The capability is the same as SPL's SCL except that:

- The precision expression can be a variable name only if the scaling operator is being applied to a TEMPorary (Item B3) variable.
- The scaling operator can be applied only to fixed point expressions.

D1 Direct Unconditional Transfers

HAL

"GO TO label" causes direct transfer to the destination statement "label". The destination statement may be within the same program or subroutine as the GO TO statement, or it may be in its "outer block" (see Item E1). The GO TO statement may not branch into the interior of a nested subroutine.

CMS-2

"GOTO label" has the same characteristics as the HAL GO TO, but the destination statement must be within the same subroutine. Additional uses of the GOTO are described in Item D2.

SPL

"GOTO label" has the same characteristics as the HAL GO TO, when the label is a simple statement label. Additional uses of the GOTO are described in Item D2.

CLASP

"GOTO label" will transfer directly to any labelled statement in the program regardless of where it is located. Additional uses of the GOTO are described in Item D2.

(BLANK)

D2 Variable Transfers

HAL

The DO CASE (case expression) statement executes a single statement from the group of statements following it, based on the value of the arithmetic "case expression". If the selected statement is a GO TO, control is transferred to the indicated label. Otherwise control resumes at the end of the group of statements following the DO CASE. The "selected statement" may be any executable statement including a "DO group" (Item D7). A NULL statement is included in HAL so that "no response" can be assigned to certain values of the "case expression".

CMS-2

The switched GOTO is equivalent to SPL's, but the switch list is always declared separate from the GOTO. The switched GOTO is the only way to branch to a statement label outside a subroutine.

SPL

Two separate capabilities are provided for transferring control to one of several places from the same statement. The first is an indirect GOTO, where the argument is the name of a location variable (Item B3). The value currently stored in the named location variable is treated as a memory address and control is transferred to that address.

The second is a "switched" GOTO, where the argument is a list of statement labels, location variables (indirect addresses), or Close names, followed by an index formula. The items in the switch list are numbered consecutively in the order they appear, and the current value of the index determines which item will be used as the argument of the GOTO. Blank entries in the switch list refer to the statement following the GOTO (i.e., do not branch). The index is normally limit checked during execution to insure the value corresponds to a switch list item. Out of limit index values do not cause a branch. However, this limit check can be overridden by terminating the switch list with an asterisk (*). A switch list may be declared once and then referenced by name in many GOTO statements.

CLASP

The switched GOTO is the same as SPL but the switch-list must be defined in the GOTO statement, and the index formula must be a simple integer formula (Item B12). CLASP has no indirect addressing.

D3 Conditional Statements

HAL

The conditional statement is of the general form:

IF logical-condition THEN statement ELSE statement

It evaluates the logical-condition and performs the THEN statement if true and the ELSE statement if false. The THEN and ELSE statements may be IF statements or DO groups, as well as simple statements. The keyword ELSE and the ELSE statement are optional. If ELSE is included, the THEN statement cannot be an IF statement. (However the THEN statement may be a DO group with IF statements embedded in it).

CMS-2

The IF statement has no ELSE statement and the THEN statement cannot be a conditional or iterative statement. If the true response has multiple statements each is preceded by THEN. (They are all executed if the condition is true):

```
IF logical-condition    THEN statement 1
                        THEN statement 2
                           '
                           '
                        THEN statement n$
```

Two special cases of the logical-condition are defined. IF DATA FOUND (or IF DATA NOTFOUND) can be used following a FIND statement (Item D6), to determine if a table search was successful. Table-element VALID (or table-element INVALID) may be used to validity check subscript values in a table reference. If any subscript in the table element reference is out of range of the declared table dimensions, the reference is INVALID.

SPL

The basic form of SPL's conditional statement is the same as HAL's except that the statement must be concluded with an END:

IF logical-condition THEN statement ELSE statement END

The THEN and ELSE statements may be IF statements or statement groups as well as any single statement. The ELSE is optional, but the END is required. A simpler version has the form:

IF logical-condition single-statement

Because the true response is a single statement, and there is no false response, the primitives THEN, ELSE, and END are not required. (HAL still requires the THEN keyword in this case.)

Separate conditions and true responses may be concatenated with an ORIF clause:

```
   IF logical-condition 1 THEN response 1
ORIF logical-condition 2 THEN response 2
ORIF logical-condition 3 THEN response 3
   '                     '
   '                     '
   '                     '
ELSE default-response
 END
```

Only the response for the first true condition encountered is executed. If none of the conditions are true, the (optional) default-response is executed.

An ENDALL delimiter is discussed in Item D7.

CLASP

Same basic form as SPL. The simpler version (no THEN) and the ORIF are not provided in CLASP.

(BLANK)

HAL

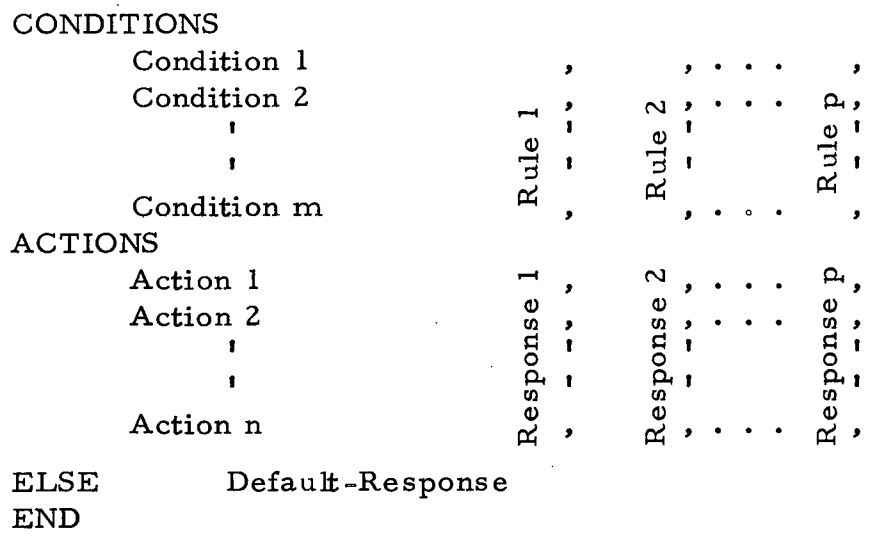
No capability.

CMS-2

No capability.

SPL

Decision tables are of the form:



Each condition is a logical-condition and each rule is a column of Y (the corresponding condition must be true), N (the corresponding condition must not be true), or blank (don't care) entries. Each action is a statement (restricted primarily to simple assignment and GOTO statement), and each response is a column of Y (this action is part of the response) or blank (this action is not part of the response) entries. The rules are tested sequentially from left to right, and the first rule for which all conditions meet the indicated (Y, N, blank) states determines the response to be taken. The response is all the actions (executed sequentially) which have a Y indication in the response column corresponding to the valid rule. If any executed action branches out of the table, the remaining actions are not executed. The default - response (required in every decision table) is executed if the conditions do not meet any of the stated rules.

CLASP

No capability.

D5 Iterative Loop Control

HAL

HAL's unconditional iterative loop control statement has the form:

```
DO FOR a = b TO c BY d, x TO y BY z, . . . ;
      '
      '
   END;
```

This statement causes repetitive execution of the statements between the DO FOR and the END. The loop variable (a) is assigned a new value at the beginning of each iteration. The values are assigned from the "for-list" of assignment values. In the example above, (a) starts at value (b) and increases by increments of (d) to the value (c). It then takes on the value (x) and is incremented by (z) to the value (y). Any of the terms (b, c, d, x, y, z) may be any integer or scalar formulas, and may take on positive or negative values, and the list of loop-variable values may be of any length. If the BY-expression is omitted, +1 is assumed for an increment. If the TO-expression is also omitted, the value (b) is used once and not incremented.

Any type of statement may appear in the iterated loop including nested loops.

CMS-2

CMS-2's unconditional iterative loop control statement has the form:

```
label VARY a FROM b THRU c BY d $
      '
      '
   END label $
```

Iterations are performed in the same manner as in HAL. The loop variable (a) takes on only integer values. The FROM-expression is optional with a default value of zero, and the BY-expression is optional with a default value of +1. Parallel loops (as in SPL) are provided through the phrase:

```
AND w FROM x THRU y TO z $
```

which follows the VARY statement.

IF THRU c is replaced by WITHIN table-name, the final value of the loop variable is the number of items defined in the named table.

SPL

SPL's unconditional iterative loop control statement is of the form:

```
FOR a = b BY d UNTIL c
 '
 '
END
```

Besides the slight difference in appearance, it has the following differences from HAL:

- The loop is not iterated for the case a=c; it is only iterated if a is less than c (or "greater than c", if b is also greater than c).
- If the BY-expression is omitted, the default increment is zero (i.e., no increment) rather than +1.
- The BY-expression can be included without the UNTIL-expression, in which case the loop variable is incremented indefinitely and iteration must be terminated by a branch from within the loop to a statement outside the loop.
- Only one set of loop variable values (b BY d UNTIL c) may follow the FOR.

SPL provides for parallel loops as well as nested loops. The ALSO-expression is used to establish parallel loop variables. If the above FOR statement had been followed immediately by:

```
ALSO w = x BY z UNTIL y
```

then each loop variable (a and w) would be incremented for each iteration. The first loop variable to reach its limit (c and y, respectively), would stop the iteration for both.

CLASP

The only loop control statement is of the form:

```
FOR a = b BY d TO c
```

and operates like the HAL statement except that:

- The loop variable (a) must be an integer variable and b, c, d must all be integer data items (no formulas).
- Control may not be transferred directly into the interior of a loop from outside the loop.
- Only one "for-list" (b BY d TO c) is permitted.
- The value of the loop variable (b) is undefined when the loop is exited.

D6 Conditional Loop Control

HAL

The DO FOR statement (Item D5) may have a logical-condition appended to it:

DO FOR a = b TO c BY d, . . . WHILE logical-condition

Iteration proceeds as in the DO FOR statement so long as the logical-condition is true. The first time it is false at the beginning of a pass through the loop, that pass is not taken and iteration stops.

If there is no need for a loop variable, the statement can be shortened:

DO WHILE logical-condition

and iteration is stopped only by the logical-condition.

CMS-2

A limited conditional iteration is provided to search a table. The form is:

label FIND logical-condition VARYING a FROM b THRU c BY d $

The first term of the logical-condition must be a subscripted table-name, and the loop control variable (a) must be one of the subscripts. The table is searched as specified by b, c, and d, or until the condition is satisfied by one of the elements found. The IF DATA statement (Item D3) is used to determine the success of the search.

SPL

The full conditional loop control statement is much like HAL's:

DO FOR a = b BY d $\left\{\begin{matrix}\text{WHILE}\\ \text{UNTIL}\end{matrix}\right\}$ logical-condition

Differences are that there is no loop variable limit (c) and UNTIL may be used instead of WHILE if it is desired to stop iteration when the condition goes false.

As in HAL, short forms are available:

LOOP WHILE logical-condition
LOOP UNTIL logical-condition

CLASP

No capability.

D7 Statement Groups

HAL

There are several cases where multiple statements need to be treated as a single group. Statements in iterative loops and multiple-statement THEN clauses and ELSE clauses are examples. The DO . . . END construction is used exclusively to delimit these groups. The iterative control statements (DO FOR, DO WHILE) and the DO CASE all delimit their statement groups with END.

CMS-2

The only CMS-2 statement which starts a statement group is the labelled VARY statement (Item D5). The group is concluded by:

```
END label $
```

where the required label references the VARY statement.

SPL

Statement groups in SPL may be started by THEN, ELSE, FOR, LOOP WHILE, LOOP UNTIL, or DIRECT.

The END delimiter may be used to conclude a group started by any of these. In an SPL conditional statement, END delimits the entire IF statement rather than THEN and ELSE groups.

Where nested loops and conditional statements imply multiple ENDs, a single ENDALL delimiter may be used to simultaneously conclude all open groups.

CLASP

Statement groups are started by THEN, ELSE, FOR and DIRECT, and concluded by END or ENDALL as in SPL. Also as in SPL, the entire IF statement is delimited rather than the THEN and ELSE groups.

(BLANK)

D8 Stop Execution

HAL

No capability.

CMS-2

STOP specifies a suspension of program execution. Details are implementation dependent.

SPL

The following statement halts the computer when it is executed:

```
STOP (label)
```

When the computer is restarted execution will proceed at the statement identified by the label or, if there is no label, at the statement following the STOP.

CLASP

Same as SPL.

E1 Overall Program Organization

HAL

An overall program organization consists of one or more independently compilable PROGRAMs and a common data pool (Item F1). Each PROGRAM can contain, in addition to its own statements, subroutines in the form of PROCEDUREs, real time TASKs and FUNCTIONs. TASKs may be defined only at the program level (i.e., they cannot be nested), but PROCEDUREs and FUNCTIONs may be defined at the program level or within other TASKs, PROCEDUREs, and FUNCTIONs, to any level of nesting.

A "block" of code is any program or subroutine and all blocks declared (nested) within it. Any given subroutine, at any level, has an "outer block" which includes all code in the direct line of subroutines in which it is nested (back to, and including, the program level).

Within any program block the statements must be ordered as follows:

- Declaration statements
- Imperative (executable) statements
- All nested blocks (in any order)

CMS-2

An overall program organization consists of one or more separately compilable System Data Designs (SYS-DD), which define system level (Compool) data, and System Procedures (SYS-PROC), which are at the same organizational level as HAL and SPL PROGRAMS. Each System Procedure can contain Local Data Designs (LOC-DD), local PROCEDUREs, and local FUNCTIONs. As in SPL PROCEDUREs and FUNCTIONs may be declared only at the SYS-PROC level. Data is declared only within Data Designs, not within FUNCTIONs or PROCEDUREs.

FUNCTIONs, PROCEDUREs, and Local Data Designs may appear in any order within a SYS-PROC, but data must be declared prior to its use.

SPL

Like HAL, an overall program consists of one or more independently compilable programs and a Compool (Item F1). PROCEDUREs and FUNCTIONs may be defined only at the PROGRAM level; they may not be nested. CLOSEs may be nested indefinitely. Data may be declared within procedures and functions, but not within CLOSEs. Declarative statements, imperative statements, and nested subroutines may be mixed in any order.

CLASP

A CLASP program organization consists of a single main program and, optionally, one or more PROCedures, functions, and CLOSEs. As in SPL, procedures and functions may be defined only at the program level and may declare local data. CLOSEs may be nested to any level but may not declare data. An entire program organization must be compiled together. The statements must be in the following order:

- Main program declarations
- Main program imperative statements
- Procedures and functions (in any order)

Closes may be declared anywhere in the source program.

HAL

A PROGRAM is the smallest separately compilable unit in the HAL organization. It has only one entry point through which it can be entered by other PROGRAMs. Parameters cannot be passed from one PROGRAM to another, so the only form of communication is through the Compool. A PROGRAM is declared and invoked as follows:

Declaration	Calling Sequence
label: PROGRAM; ' ' ' CLOSE label;	CALL label;

The label on the CLOSE statement is optional.

PROGRAMs are also subject to real-time scheduling as are TASKs (Item E4).

CMS-2

A System Procedure (SYS-PROC) is the smallest separately compilable program unit. (A System Data Definition is separately compilable to generate a Compool, but does not produce executable code.) Unlike SPL and HAL, there are no executable statements at the SYS-PROC level. There are only declarations of PROCEDUREs, FUNCTIONs, and Local Data Definitions. Every SYS-PROC contains a prime PROCEDURE, which has the same label as the SYS-PROC and which is an external entry point. Other PROCEDUREs and FUNCTIONs may be explicitly declared external. A SYS-PROC is declared as follows:

```
label SYS-PROC $
      '
      '
      '
    END-SYS-PROC label $
```

The label on the END-SYS-PROC statement is optional.

SPL

A PROGRAM is the smallest separately compilable unit in the SPL organization. A single PROGRAM can have multiple entry points accessible by other PROGRAMs. Both input and output parameters (Item E8) can be transferred between PROGRAMs. In addition, all PROGRAMs may access the Compool. Both RECURSIVE and REENTRANT programs can be declared. A program is declared and invoked as follows:

Declaration

```
START  .label (input-parameters = output-parameters) attributes
   '
   '
TERM label
```

Calling Sequence

```
.label (input-arguments = output-arguments)
```

A program may also be declared with the data type and return formula of a function (Item E5), and then may be either called as above or invoked like a function by appearance of the label in an expression. In this case, values are returned for all defined output parameters and a separate value (the return formula) is returned for the appearance of the label in the expression where the function was invoked.

CLASP

A CLASP program organization consists of one and only one program. Separately compiled programs within one organization are not permitted. A program is declared as follows:

```
START
  '
  '
TERM
```

E3 Procedures

HAL

A PROCEDURE is very similar to a PROGRAM in definition and calling sequence, but a PROCEDURE is not separately compilable and parameters can be passed between a calling module and a PROCEDURE. PROCEDUREs are not subject to real-time scheduling. PROCEDUREs are declared and invoked as follows (The label on the CLOSE statement is optional.):

Declaration

```
label: PROCEDURE (input-parameters) ASSIGN (output-parameters);
          '
          '
       CLOSE label;
```

Calling Sequence

```
CALL label (input-arguments) ASSIGN (output-arguments);
```

CMS-2

PROCEDUREs are very similar to HAL procedures with the primary difference that alternate return statement labels can be passed when the PROCEDURE is called. The PROCEDURE may return either immediately after the calling statement or to a named alternate return. PROCEDUREs are declared and invoked as follows:

Declaration

```
PROCEDURE label INPUT input-parameters OUTPUT output-
                parameters EXIT alternate-returns $
       '
       '
END-PROC label $
```

Calling Sequence

```
label INPUT input-arguments OUTPUT output-arguments EXIT
                alternate-returns $
```

PROCEDUREs may also be invoked through a procedure switch, which is a list of PROCEDURES. The current value of an index determines which PROCEDURE in the list will be invoked. All procedures must have identical number and type of formal parameters, and alternate returns may not be specified. The switched call may include an error branch to be taken if the index value is beyond the range of the switch list (see Item D2).

SPL

PROCEDUREs are very similar to HAL procedures with the following differences:

- RECURSIVE and REENTRANT procedures may be declared.
- A procedure can be declared INLINE, in which case the compiler duplicates the object code when the procedure is invoked rather than generating a branch to common object code.
- Like an SPL program, the SPL procedure can be declared with the attributes of a function and can then be either called or invoked like a function (Item E5).
- An SPL procedure may have alternate entry points.
- Labels of CLOSEs (Item E6) may be passed as input parameters and the procedure may execute a CLOSE whose label is passed.
- Alternate exit statement labels may be passed as output parameters, and the procedure may return to an alternate exit.
- Procedures may not be nested.
- The declaration and calling sequence differ in format.

Declaration

```
PROC .label (input-parameters = output-parameters) attributes
  '
  '
EXIT
```

Calling Sequence

```
.label (input-arguments = output-arguments)
```

CLASP

Identical to SPL except:

- No RECURSIVE or REENTRANT capability.
- Cannot be invoked like a function.
- No alternate entry points.

E4 Real-Time Tasks

HAL

A TASK is a subroutine which is intended to be executed in real-time through an executive system. Although it can be called directly like a PROCEDURE, it can also be SCHEDULEd to be invoked in the future as a result of:

- A specific time
- An increment of time from now
- An external EVENT such as an interrupt
- An internal EVENT which is defined and SIGNALled by program statements.

While in execution, a TASK may suspend its own execution (WAIT) until any one of the above four types of occurrences. Upon the specified occurrence, execution resumes at the statement following the WAIT statement.

CMS-2

No capability.

SPL

Real time task control is provided through a chronic statement which assigns a sequence of statements to be executed in response to an interrupt occurring or a Boolean-formula coming true. The formats are:

```
ON interrupt-name          ON Boolean-formula
      '                          '
      '                          '
END                        END
```

When such a sequence is encountered in program execution, the statements between "ON" and "END" are bypassed, but assigned to be executed when the named condition occurs. Interrupts can be separately inhibited and enabled (Item E10). The Boolean-formula is a simple relational expression (Item C3) and is checked whenever the first operand changes value. It is also possible for a routine to coordinate its execution in real time with another routine. The WAIT statement can be appended to a conditional (IF) statement or to a LOOP statement. So long as the stated condition is true, program execution will be held up by the WAIT statement. Execution will proceed sequentially when the condition (presumably changed by another routine) becomes false.

CLASP

CLASP has the same chronic statement capability as SPL but it applies only to interrupts. The same interrupt control capability is also provided.

E5 Functions

HAL

A FUNCTION is similar to a PROCEDURE with the primary difference that it returns only one parameter rather than the multiple parameters which can be returned by a PROCEDURE. The FUNCTION name (label) is used directly in an expression, rather than through a CALL statement. The declaration and calling sequence are as follows:

Declaration	Calling Sequence
label: FUNCTION (input-parameters) ' data-type-attributes; ' CLOSE label;	Invoked by appearance of "label (input-arguments)" in a statement.

CMS-2

FUNCTIONs are similar to HAL functions, but may have only one input parameter. The data type of the FUNCTION is not explicitly declared, and the method for returning the result is implementation dependent. The declaration and calling sequence are as follows:

Declaration	Calling Sequence
FUNCTION label (input-parameter) $ ' ' END-FUNCTION label $	Invoked by appearance of "label (input-argument)" in a statement.

SPL

Function declarations are identical to SPL PROCedure declarations with the addition of a data type and attributes for the function name and a formula in the RETURN and EXIT statements (Item E7). A PROCedure which has these additions may be called either as a procedure (.label) or as a function (label with input-arguments appears in a statement). All output parameters are calculated, in addition to the return formula, for a function call.

CLASP

Function declarations are identical to CLASP PROCedure declarations except that there are no output parameters and the function label is declared as a data item within the function. Functions are invoked by appearance of the function label with input-arguments in a statement.

E6 Closes

HAL

No capability.

CMS-2

No capability.

SPL

A CLOSE is a subroutine which has no input or output parameters and which cannot declare data items or statement labels internal to itself. It can be declared anywhere among the imperative statements of the program. It is invoked by a GO TO statement. CLOSEs may not be used recursively.

Declaration	Calling Sequence
CLOSE . label	
'	
'	GOTO label
EXIT	

CLASP

Same as SPL except that the calling sequence is the same as for a PROCedure (. label) instead of GOTO label.

E7 Program and Subroutine Returns

HAL

All programs and subroutines in HAL are delimited by a CLOSE statement. The CLOSE statement may include the label of the program or subroutine being delimited, in which case the compiler will verify that other CLOSE statements have not been left out or extraneously included. Except in the case of a function, where execution of the CLOSE is an error, the CLOSE statement returns control to the caller at the statement following the CALL. In addition, RETURN statements may be included anywhere in the program or subroutine to return control to the caller. In a function, each RETURN statement includes a formula which represents the value to be assigned to the function.

CMS-2

System procedures, procedures, and functions are delimited by END-SYS-PROC, END-PROC, and END-FUNCTION, respectively. The END-PROC and END-FUNCTION statements must include the label of the unit being delimited. These delimiters never return control to the calling routine; the RETURN statement is used exclusively for this purpose. In a procedure, the RETURN statement may indicate an alternate return label, passed in the calling sequence (Item E3).

Functions do not have return formulas; techniques for returning results from functions are implementation dependent.

SPL

An SPL program is delimited by a TERM statement and subroutines are delimited by an EXIT statement. In all cases, even for functions, the delimiter returns control to the caller. The only delimiter which references another statement's label is the TERM statement; the label identifies the primary entry point of the procedure (default is the first executable statement). The primary entry point label need not be the procedure name. As in HAL, RETURN statements can be used anywhere in the program or subroutine to return control to the caller. Unlike HAL, SPL subroutines may be passed alternate exit labels and may, therefore, return control to some point other than immediately following the calling sequence. As in HAL, function RETURNs include formulas which represent the function value. (For SPL, the function EXIT statement also includes a formula.)

CLASP

Identical to SPL with the following exceptions:

- No RETURN statements; only the delimiting EXIT statement returns control following the calling sequence. (The alternate exit capability is provided.)
- Function names are assigned values by assignment statements rather than formulas in EXIT statements.

E8 Subroutine Parameters

HAL

Parameters cannot be passed to or returned by Programs or Tasks. All procedure and function parameters are "call-by-name". That is, input and output parameters occupy memory locations named by the calling program and the subroutine communicates with those memory locations rather than physically passing data from the calling program to the subroutine and back. Arguments specified by the calling program must be identical in data type and attributes as the formal (dummy) parameters in the function or procedure declaration, with one exception: the length or dimensions of bit-strings, arrays, etc. can be declared variable for the formal parameter and have specific values for these attributes supplied by the calling programs.

CMS-2

Input data parameters and alternate return labels may be passed to Procedures, and output data parameters may be returned. Only a single input parameter may be passed to a function and only one value is returned. All parameters are call-by-value but a special core address (CORAD) operator allows tables to be passed by name.

SPL

Parameters may be passed to and returned by programs, procedures, and functions but not closes. Tables, arrays, and long character-strings ("long" is an implementation dependent quantity) are "call-by-name" parameters. Short character-strings and other simple data items are "call-by-value" parameters, which means that data is physically passed between the caller and the called program or subroutine. In addition to data parameters, close names may be passed as input parameters and statement labels may be passed for alternate exits (Item E3).

Implicit conversion is made between data types of formal parameters and actual arguments (except for textual and non-textual). As in HAL, dimensions of formal parameter arrays may be variable.

CLASP

Same as SPL, with the exception that tables do not exist in the language and programs are never called.

E9 Priority Assignments

HAL

Priorities may be assigned to real-time TASKs and PROGRAMs at the time they are SCHEDULEd or modified through a PRIOCHANGE statement after they have been scheduled.

CMS-2

No capability.

SPL

No explicit capability, but interrupts can be inhibited and enabled to dynamically control which program segments can interrupt other program segments. (Item E10)

CLASP

Same as SPL.

E10 Exclusive Subroutines and Interrupt Controls

HAL

The multiple priority levels associated with HAL create the possibility of a program or subroutine being re-entered on one level while it is concurrently in execution (suspended) on a lower level. Since some program designs do not execute properly if re-entered, HAL procides an EXCLUSIVE attribute. If any program or subroutine is declared EXCLUSIVE, it is protected from re-entrance. A higher priority task which calls it will be stalled, if necessary, until the lower priority concurrent execution is completed.

CMS-2

No capability.

SPL

Subroutines may not be declared exclusive, but any routine may LOCK (inhibit) any or all interrupts to prevent an interrupt response (Item E4) from occurring and using the routine. Similarly, an UNLOCK statement selectively enables interrupts. Each interrupt must be explicitly named in LOCK and UNLOCK statements.

In addition, SPL can declare a routine to be REENTRANT (Item E2), or INLINE as in CLASP.

CLASP

CLASP has the same LOCK and UNLOCK capability as SPL. There is no RE-ENTRANT capability, but an INLINE procedure (Item E3) can be declared so that separate inline coding can be generated for each calling sequence. This avoids multiple use of the same object code by two different calls.

E11 Error Recovery

HAL

Error conditions such as underflow, overflow, and division by zero are assumed to be detected by the system and it is assumed that each has a standard system response. The ON $ERROR_x$ statement allows the programmer to specify a particular response to the error condition identified by the subscript x. For any error (or group of errors) he may specify either the SYSTEM response or a transfer (GO TO) to his own defined response. The response is assigned when the ON $ERROR_x$ statement is encountered in execution and the assignment remains for the scope of the statement (that is, for the block in which it is specified and any nested blocks which do not modify it).

In addition to the system detected error conditions, the programmer can define his own error conditions, assign responses, and then announce their occurrence through a SEND $ERROR_x$ statement which invokes the response assigned to $ERROR_x$.

CMS-2

An error branch can be specified to occur on a divide overflow.

SPL

No capability except for target computers where the error generates an interrupt to which a chronic statement (Item E4) can be attached.

CLASP

Same as SPL.

(BLANK)

E12 Library Subprograms

HAL

HAL provides an extensive library of "Built-In" FUNCTIONs such as trigonometric and hyperbolic functions, matrix inverse and transpose, and a variety of string manipulations.

CMS-2

A limited set of library functions includes absolute value, remainder, and selected input/output controls.

SPL

These built-in functions are provided: absolute value, remainder, and integer quotient with remainder (a procedure).

CLASP

CLASP provides the three SPL built-in functions and procedures, plus six others.

E13 Scope of Names and Labels

HAL

A data item name is declared through a DECLARE statement, or implicitly through the first appearance of the name in the coding. That declaration is recognized throughout the block (PROGRAM, TASK, PROCEDURE, FUNCTION) in which it appears and throughout any nested blocks within that block. However, it is not recognized in an "outer block" (i.e., one in which the block where it is declared is nested). A declared name may be re-declared in a nested block, overriding the earlier declaration only within that block and its nested blocks.

Duplicate names in separate scopes have no connection. Compool data (Item F1) is declared outside any program and is recognized in any block.

An OUTER statement is provided which isolates a nested block from all declarations made in an "outer block", including Compool. Only those declared names explicitly listed as parameters of the OUTER statement are recognized.

Statement labels follow the same scope rules as data item names.

CMS-2

A CMS-2 program organization has two levels of name scope:

- System Data (SYS-DD) which is equivalent to a Compool
- Local Data (LOC-DD) which is declared within a System Procedure

System Data is recognized throughout the program organization. Local data is recognized only by Procedures and Functions within the System Procedure where it is declared. Data may not be declared within a Procedure or Function. A name declared at the System Data level may not be re-declared at a local level, but duplicate names may be used locally in different System Procedures.

Several features are provided to override the nominal name scope, and can selectively make any local declaration available to any other System Procedure.

Each System Procedure has one primary Procedure whose label is recognized throughout the program organization. In addition, any procedure may be explicitly declared external, making its label also known to the entire organization. Statement labels may be declared external only through a GOTO switch (Item D2).

SPL

An SPL program organization has three levels of name and label scope:

- Compool (Item F1)
- Program
- Procedure or Function

Compool declarations are recognized throughout the program organization. Each (separately compilable) program is a separate block and each procedure or function is a nested block within the program in which it is declared. Since procedures and functions must all be declared at the program level, there are no further nested blocks. Declarations within blocks and nested blocks follow the same scope rules as for HAL.

There is no OUTER statement in SPL. A DECLARE LOCAL statement allows a program to declare a separate data item for each of its procedures and functions as if each had declared its own. Each item has the same name and attributes, but they are entirely independent data items.

An external procedure declaration is provided to declare names of routines which are not part of the program being compiled. The declaration has the same format and data as a normal PROCedure declaration (Item E3) and includes the attribute EXTERNAL. The compiler prepares a linkage for later resolution.

CLASP

There are only two levels of scope for data item names in CLASP:

- Main Program
- Procedure or Function

Since there is only one main program in an overall program organization, there is no need for a separate compool level. The two available levels are treated in the same manner as their SPL counterparts, but there is no DECLARE LOCAL capability in CLASP.

There is only one level of label scope; any statement label is recognized throughout the program.

F1 Common Data

HAL

Any data which is declared outside of PROGRAM block and prior to that block is considered Compool (common data pool) data and can be shared with other, separately compiled programs. A symbolic library is available in which centrally defined and maintained Compool declarations, as well as other commonly used source code, can be stored. An INCLUDE statement allows the programmer to select a Compool or any other identified group of statements and have it inserted anywhere in his program.

CMS-2

CMS-2's System Data Designs (SYS-DD) define Compool data. A special compiler directive (CMP) is provided to create a Compool from a group of System Data Designs.

SPL

SPL has the same Compool capability as HAL, but no INCLUDE statement.

CLASP

No capability, and no separately compilable programs.

F2 Compile Time Data Protection Features

HAL

Compool variables may be assigned ACCESS rights as they are DECLAREd. Each PROGRAM can be assigned a unique IDCODE number, and each Compool data item can authorize access to any number of PROGRAMs by listing their IDCODEs in an ACCESS clause. A data declaration without the ACCESS clause is available to any PROGRAM. ACCESS rights can be assigned only to Compool declarations and they do not distinguish between read accesses and write accesses.

CMS-2

No capability.

SPL

No capability.

CLASP

No capability.

(BLANK)

F3 Execution Time Data Protection Features

HAL

Data declared at the PROGRAM or COMPOOL level may be assigned either one of two LOCKTYPEs. LOCKTYPE(2) data is protected against two independent tasks attempting to update the data concurrently. LOCKTYPE(1) data provides the above protection and also protects the reader of multiple data elements (e.g. a vector) from an intervening update which causes him to get "old" values for some of his data elements and "new" values for the rest. This protection is accomplished through UPDATE blocks which are delimited by an UPDATE and a CLOSE statement. All accesses to LOCKTYPE data must be made within these blocks and they are protected by a system of real-time locks which hold up (stall) a conflicting access until the current one is completed.

CMS-2

No capability.

SPL

A LOCK statement prevents any writing into a specified portion of memory until a subsequent UNLOCK statement frees it up. This protects data from being written while it is being read. It is intended to be implemented only if the target computer has a memory protect capability. There is no equivalent to HAL's LOCKTYPE in a data declaration, and therefore no capability for the compiler to flag data accesses which should be LOCKed but are not.

SPL also provides the ability to inhibit interrupts (Item E10). This capability can be used during shared data accesses to temporarily prevent execution of programs which might cause a conflict.

CLASP

Same capability as SPL.

F4 Conventional Input/Output

HAL

HAL provides a device-oriented input/output capability which identifies a device and record number, card-column, print line or other physical device characteristic. A FILE statement is used for record-oriented devices, READ and WRITE are used for formatted character-strings, and READALL reads any character-strings. SKIP, TAB, COLUMN, PAGE, and LINE are controls which can be included with READ and WRITE statements to position devices.

Data conversion between character-strings and other data types is performed by the READ and WRITE statements. The declared data types of the internal variables determine what type of conversion takes place.

CMS-2

Like SPL, CMS-2 provides a file oriented input/output capability. A FILE statement declares files; OPEN and CLOSE activate and deactivate them. INPUT and OUTPUT transmit data. File positioning is performed by a statement of the form:

SET POS (file-name) TO position

The position may be an absolute record location or it may be relative to the current position.

Data conversion is performed by the INPUT or OUTPUT statement which (optionally) identifies a FORMAT statement. The FORMAT statement describes the conversions to be made between character-strings and internal data types.

SPL

SPL provides a FILE-oriented, input/output capability, where FILE declarations identify a device, data storage mode, record length, label of statement sequence to be executed when end-of-file is encountered, and other implementation-dependent information. Input/output then takes place between internal core buffers and declared data FILES, using READ and WRITE statements. OPEN and SHUT statements activate and deactivate files. Files are positioned by BACKSPACE, SKIP and REWIND statements.

Data is transmitted between files and buffers without conversion. The conversion is performed implicitly as the data items from internal buffers are used in formulas and assignment statements (see Item C7).

To aid in this conversion, the buffers may be declared as TEXTual (character-string) data with a secondary data type (integer, floating point, status, etc.). This secondary data type determines how character-strings are interpreted when they are converted to other data types.

CLASP

No capability.

F5 Real-Time Input/Output

HAL

No capability other than Item F4. If the system implementation assigns device names to external avionics equipment, the FILE statement would be capable of describing the transfer of data between internal data items and external equipment. However, no explicit features beyond those described in Item F4 are defined.

CMS-2

Same adaptability as SPL, but no explicit description of a capability.

SPL

The FILE declaration capability could be used to declare file-names for external avionics equipment, and READ/WRITE statements could be used to perform I/O. However, no explicit features beyond those described in Item F4 are defined.

CLASP

No capability.

G1 Optimization Directives

HAL

HAL's optimization-related directives are memory allocation controls (Item G2).

CMS-2

Optimization is supported through memory allocation controls (Item G2) and the ability to access index registers directly (Item B14).

SPL

SPL's optimization-related directives are memory allocation controls (Item G2) and a DECLARE INDEX statement.

The DECLARE INDEX statement is used to identify a particular integer variable as a frequently used subscript. The compiler can then generate object code which keeps the value in an index register or other suitable register whenever practical to increase execution efficiency.

CLASP

In addition to the memory allocation controls (Item G2), CLASP provides OPTIMIZE TIME(n) and OPTIMIZE SPACE(n) directives. The integer n in each case indicates the degree to which one optimization criterion (space or time) will be sacrificed to optimize the other. The closing delimiters are UNSPACE and UNTIME. Within a group of statements being optimized, the statements SIC and UNSIC delimit a subgroup to be exempted from the directed optimization.

CLASP also has the DECLARE INDEX statement as in SPL.

G2 Memory Allocation Controls

HAL

Memory allocation controls are provided through two data declaration attributes. One is the storage class attribute of STATIC or AUTOMATIC. Storage is allocated to STATIC data when a program is loaded, but AUTOMATIC data items are allocated storage only when the subprogram in which they are defined is actually entered for execution. This storage is released when the subprogram returns control to the calling program or the system.

The second is the data packing attribute of DENSE or ALIGNED. DENSE data is packed to conserve storage at the expense of additional execution time to pack and unpack in real time. Aligned data is aligned on full-word (or other unit) boundaries for more efficient retrieval. DENSE and ALIGNED attributes may be applied to individual data elements as well as data organizations.

CMS-2

Three packing attributes, NONE, MEDIUM, and DENSE apply to one-dimensional tables. An EQUALS (Item G4) statement performs the same function as SPL's OVERLAY. In addition, it can assign data to specific (absolute or relative) memory locations.

SPL

Four data packing attributes are provided: NONE, MEDIUM, DENSE, and TIGHT. They have the same objective as HAL's ALIGNED and DENSE, but they apply only to tables and arrays.

An additional capability is provided by the OVERLAY directive, which can be used to cause different data items to occupy the same memory locations, thereby reducing overall memory requirements. For example, several different input/output buffers with different declared data attributes can occupy the same memory locations. The OVERLAY statement can also be used to allocate separate data items and organizations contiguously in core memory.

CLASP

The SPL OVERLAY capability is provided but there are no packing attributes.

G3 Program Debug Aids

HAL

No specific aids.

CMS-2

The following debug facilities are available:

- TRACE and END-TRACE as in SPL. SNAP is used to print data values.
- PTRACE traces all procedure calls.
- Numeric items can be RANGE tested and a message is printed each time the value goes out of the specified range.
- DISPLAY is an executable statement which prints current value of its arguments (data items and hardware registers) whenever it is encountered.

A DEBUG declarative is used to enable the debug statements; otherwise, they are ignored by the compiler.

SPL

A program declaration may include a maximum limit for size of the object code. A diagnostic message is generated by the compiler if the limit is exceeded.

TRACE and ENDTRACE delimiters cause the compiler to generate code which will print, in sequence, all statement labels encountered during execution between the delimiters. Data names may also be specified and values of the variables named will also be printed.

TIME and ENDTIME delimiters create object code which will calculate the amount of time elapsed in execution between the delimiters.

CLASP

TRACE and UNTRACE perform the same function as the SPL counterparts.

COUNT(n) and UNCOUNT(n) perform the same function as SPL's TIME and ENDTIME, but allow the user to specify separate timers (the integer n), so that overlapping sections of code can be timed.

G4 Compile-Time Identifiers

HAL

REPLACE identifier BY character-string causes the defined identifier, whenever it appears in the coding, to be replaced by the assigned character-string before compilation. The scope of a REPLACE statement is the same as that for a DECLARE statement (Item E13). That is, the replacement is in effect throughout the block in which the REPLACE occurs and throughout any nested blocks.

CMS-2

The statement, identifier MEANS character-string, performs the same function as HAL's REPLACE. The scope is that of the Data Design in which it appears.

A different type of statement, tag EQUALS arithmetic-formula, assigns the value of the arithmetic-formula to the tag named in the statement. The tag can them be used, for example, to preset data constants or to represent a literal value in an executable assignment statement. If the tag is declared elsewhere as a data item name, the EQUALS statement can be used to assign a memory location for the item. The location can be relative to the beginning of the compilation or relative to some other data item.

SPL

DEFINE identifier BY character-string performs the same function as HAL's REPLACE. However, its scope is from the point of appearance, sequentially through the source program statements to the end.

CLASP

No capability.

G5 Macro Statements

HAL

No capability

CMS-2

Same as SPL

SPL

There is no explicit capability to define macro statements. However, the programmer has the capability to insert assembly language code into his source program (Item A3), and could use an assembler macro capability if it were available.

CLASP

Same as SPL

(BLANK)

8. FLIGHT PROGRAM KERNEL CODING

A large part of the language evaluation discussion of Paragraph 6 is based on the flight program kernel coding, which was a major task performed during Phase II of the study. The kernels coded were those selected during Phase I, and the basis for their selection is described in the Introduction. Descriptions of the kernels, and the unique language characteristics required for them, are provided in Appendix A of this report. Source card listings of the kernel coding are provided in Appendix B.

The AS-509 Saturn Flight Program Equation Defining Document was the design basis for all but one of the kernels coded. ATM Task Keying was taken from the Equation Defining Document for the Skylab Apollo Telescope Mount Digital Computer Flight Program. An actual Saturn flight program listing was used to reflect detailed implementation of the Saturn kernels. In general, the kernels were coded to duplicate the functions of the corresponding modules of the Saturn flight program. However, certain deviations were made where language features permitted more efficient coding of a function or where a certain function served merely to handle unique hardware features of the Saturn computer. For example, loop capabilities have been used to eliminate the duplication of code for each of the three vehicle axes in the Minor Loop and in Accelerometer Processing.

Program symbology was also based to a large degree on the symbology of the Saturn flight program. Symbolic names for subprogram entry points and program data were obtained from the referenced program listing where applicable and were used with only slight modification. Such alterations include the deletion of the period from most program symbols and the grouping of vector component symbols into a composite vector symbol. For example, the subprogram name M.AP00 was changed to MAP00 and the three symbols D.VFX, D.VFY, and D.VFZ were replaced with array DVF. Saturn flight program names were used wherever possible to assure kernel fidelity and to provide a means for correlating the kernels with the actual flight program. In cases where additional or more appropriate symbols were required, they were designed to be as descriptive as possible. The Iterative Guidance Mode (IGM) kernel, in particular, was coded without using the original program symbols since many of the terms in the equations had been assigned to temporary storage. Thus the entire IGM kernel was coded using symbolic names derived from the Equation Defining Document.

Saturn flight program naming conventions were utilized to organize the data base for the kernels. Specifically, symbolic data names beginning with a "D", which were used in the original program to denote items accessed by more than one subprogram, were assigned to a common data pool. Those items beginning with something other than "D" were generally local in nature and were defined within the program in which they were used. Other items which did not follow the Saturn flight program naming convention were assigned to the common data pool or defined locally depending on whether or not they were referenced by more than one kernel.

Program organization is centered around the program kernel. That is, each kernel is declared at the highest level of program or subprogram provided by the language. Where permitted by the language, each kernel was declared as a separately compilable unit. Except where specifically called out in Appendix A, the internal structure of each kernel was retained.

The above groundrules were modified or extended uniquely for each of the four languages. These language peculiar considerations are discussed individually with each language.

The following paragraphs describe, for each language, the groundrules and assumptions made before and during the coding, the peculiar problems encountered with the language, and the features of the language which proved particularly useful to this coding effort.

8.1 Space Programming Language (SPL)

8.1.1 Groundrules and Assumptions

Each kernel was coded in SPL as a separately compilable program. Utility programs were assumed to be included in an SPL Compool so that no special external declarations were required for communicating with them.

Floating-point data declarations were employed for the majority of the numerical operations. Those computations performed with fixed-point arithmetic were only the processing of input and output data, where timing efficiency could not be sacrificed for the conversion to or from floating-point.

The coding of input/output functions was performed using READ and WRITE statements and associated FILE declarations under

the assumption that these would be implemented for real time input/output processing. If they are not provided, the program would have to lapse into another language to issue input/output commands to real time devices.

A set of device names was established to allow symbolic reference to hardware input/output registers and interrupts (see Paragraph A. 12). Such names would ordinarily be generated when the compiler is implemented and would then be made available for programming purposes. In cases where a hardware input was to be used immediately and not saved for future reference, the data was read into a CONTEXTUAL item used for temporary storage. The assumption was made that the CONTEXTUAL item would take on LOGICAL attributes.

A problem arose concerning activation of chronic statements for the processing of interrupts. The documentation did not make clear whether or not a given interrupt can be specified in more than one chronic statement. This is desirable in a mission such as Saturn's where the response to a given interrupt varies in time. Since dynamic reassignment of interrupt responses was required, the assumption was made that multiple chronic statements could reference a single interrupt for purposes of altering the response to the interrupt.

SPL allows a procedure or a program to have multiple entry points. It was assumed to be legal to transfer control directly to any of these entry points from any other point within the same procedure or program without destroying information required for returning control to the caller who originally invoked it. Strict interpretation of the specification allows this, but its implementation requires special attention by the compiler which was not mentioned.

SPL specifications require a fixed-point constant to be coded with an accompanying scale factor. In the case of a simple assignment statement, however, where a literal constant is assigned to a data item, it was felt that such a requirement was an unnecessary burden. Therefore, it was assumed that the compiler would scale the constant automatically based on the scaling of the receiving data item. Likewise, it was assumed that a literal constant "zero" does not require the specification of a scale factor since scaling has no meaning for a value of zero.

8.1.2 Significant Problems Encountered

One of the primary problems encountered in coding the kernels was that of interrupt control. SPL facilities for inhibiting interrupts were not designed for inhibiting a large number of interrupts simultaneously. It is a burden to have to list a long string of interrupts every time it is necessary to inhibit them. For example, the statement:

```
LOCK   T1INT, T2INT, EX2INT, EX3INT, EX4INT,
       EX5INT, EX6INT, EX7INT, EX8INT, EX9INT
```

is required to inhibit the ten interrupts listed. A 'LOCK ALL' capability or 'LOCK GROUP' capability would be useful,because frequently the initial response to an interrupt is to lock all or nearly all of the interrupts for a brief period.

A bigger problem was enabling interrupts following a momentary inhibit. While SPL's UNLOCK feature allows interrupts to be selectively enabled, it does not permit the selection to vary during real time execution as is required. The problem could be solved by maintaining a status word for interrupt control and testing it each time a momentary inhibit is to be released. Such a method would be cumbersome and inefficient at best. A better solution would be to have the language provide an indirect UNLOCK capability. No attempt was made to solve the problem in coding of the kernels; a simple UNLOCK statement with an appropriate comment was used whenever such a release was required.

The requirement for the ELSE clause in a decision table was a minor annoyance. There were times when it was not needed, because the decision table itself specified all possible conditions.

In programs which must activate other subprograms (examples include the Interrupt Processor and the Events Processor), an indexed or indirect "call" capability would be very useful. Capabilities exist for transferring control via an indexed or indirect GOTO but this does not provide the necessary linkage for a return from the invoked subprogram. The problem is illustrated by the following example selected from the Interrupt Processor.

```
GOTO (EGP12,GP21,GP22,GP23,GP24,GP25,GP26,GP27,
      GP28,GP29,GP30,GP31) DGST2

GP21.     .MUM00      GOTO EGP12
GP22.     .MLR10      GOTO EGP12
GP23.     .MEP00      GOTO EGP12
GP24.     .MTT10      GOTO EGP12
GP25.     .MNU00      GOTO EGP12
GP26.     .MEE00      GOTO EGP12
GP27.     .MCM00      GOTO EGP12
GP28.     .MCM10      GOTO EGP12
GP29.     .MCM20      GOTO EGP12
GP30.     .MEPWM      GOTO EGP12
GP31.     .MER00
EGP12.    .EGP18
```

The value of DGST2 determines which statement gets control from the switched GOTO. Provision for an indexed or indirect call, such as CMS-2's switched CALL, could replace the above series of statements with two statements.

An indirect input/output capability would eliminate the need for making explicit decisions concerning which type of input/output is to be performed. For example, in the Minor Loop it is necessary to test a flag to determine whether the fine or backup gimbals from the platform are to be read. An indirect or indexed capability could reduce the logic required for each read. In future applications where a significant amount of system reconfiguration may be encountered, the utility of such a capability should be even greater.

Restrictions on the definition of constants are inconsistent with the flexibility provided elsewhere in SPL. Program maintenance and readability would be greatly enhanced by permitting constants to be defined in terms of expressions containing other constants, as is provided in CLASP.

The use of fixed-point data is difficult from the standpoint that the programmer loses sight and control of the intermediate scalings and intermediate operations in the evaluation of a complicated expression. The scaling aids are useful in that they save the programmer some bookkeeping in many operations. However, they tend to obscure the scaling of intermediate results in an expression and make it difficult for the programmer to determine where he might be losing

precision. Special features are available to specify scaling of intermediate results, but it is difficult for the programmer to determine when they are needed because the nominal scaling is obscured.

SPL does not provide facilities for specifying a short time delay such as that required by the Switch Selector Processor. When such a delay was needed in the kernels, it was coded by using a dummy loop. If such a technique were required in actual practice, the programmer would need to adjust the loop control pass count in a trial-and-error fashion to achieve the desired delay, or know precisely the compiler expansion of his statements and the instruction execution times of the computer. While time delay facilities are sometimes provided by an operating system, stringent hardware timing requirements, as in Minor Loop, make it desirable to perform the delay under control of the application program.

8.1.3 Desirable Characteristics

The general flexibility of SPL is reflected in the essentially free form of individual program statements. In the coding process, the programmer need not be concerned with requirements for positioning the statement within an eighty character line, for specifying punctuation to delimit statements, or for coding any other statement descriptor for the compiler. Multiple statements can be coded on a single line, or a single statement can be extended over several lines. The flexibility of SPL is best exemplified by its ability to handle expressions containing mixed data types. The language performs the required conversions and scaling adjustments automatically, thus providing the programmer with considerable freedom in formulating expressions.

The more useful features of SPL include decision tables, status variable and indirect addressing capabilities discussed in Paragraph 6.1. The data organization capabilities are also very powerful. Multiple-item data aggregates are handled in SPL via arrays and tables. Arrays are multi-dimensional data organizations which contain data elements all having the same attributes. Arrays can be used to represent vectors and matrices and can be combined with powerful matrix and vector operations. Individual array elements, denoted via subscripts, can be used as simple data items in virtually any context.

Tables are used to organize aggregates of mixed data types in a variety of ways. They provide sufficient flexibility to enable the

programmer to tailor the organization of his data to suit individual applications in an efficient manner. A table entry can be subdivided into individual elements each of which can be independently declared. Furthermore, the manner in which these elements are combined to form an entry can be specified to either conserve core or minimize access time. While individual table elements can be used as simple data items in any context, the entire table or a multiple-element table entry can be referenced only in selected assignment, comparison, or exchange statements.

Contextual variables were very useful for temporarily storing intermediate results. They help conserve memory requirements and eliminate the need for the programmer to define many different variables for storing such data. Besides providing temporary storage, contextual variables also take on the attributes of the data currently stored in them. The attributes are utilized by the compiler in subsequent references to the temporary data.

Facilities for controlling (inhibiting/enabling) and responding to interrupts are also provided by SPL. The former satisfies the requirements of certain flight program functions which require direct control of hardware interrupts. The latter is also necessary if the operating system functions are to be implemented in a high-level language.

Other SPL features make it possible to implement a flight program with a modular organization. Programs can be compiled separately and combined for execution through use of a program loader capable of resolving inter-program linkages. Included in the language is the concept of a Compool designed to contain common data accessed by more than one program and common, frequently used subroutines. Associated with program structure, the multiple entry-point feature was found to be useful. It allows the programmer greater flexibility in organizing program elements.

8.2 Computer Language for Aerospace Programming (CLASP)

8.2.1 Groundrules and Assumptions

Due to the many common features between the two languages, most of the assumptions and groundrules of SPL (Paragraph 8.1.1) also apply to CLASP. The exceptions are described below.

Since CLASP does not provide for separately compilable program modules, the individual kernels were coded as procedures within

one program. The Compool data of SPL was simply declared at the program level in CLASP so it could be accessed by all the procedures. Utility programs presented no access problem, because the entire program organization must be compiled together, anyway.

All arithmetic was performed with fixed-point data to illustrate the differences between fixed-point and floating-point coding for numerical operations via comparison with the SPL kernels. The primary difference is the additional scaling information required for the fixed-point data declaration. Because the scaling had already been determined for the flight program and documented in the available listings, writing them down in the data declarations was not a significant burden. However, if it had been necessary to derive the scaling data, the effort would have been greatly increased over the equivalent floating-point coding. The scaling factor on a fixed-point constant is optional in CLASP, so it was not necessary to make the SPL assumption that the compiler would scale by implication on a simple assignment.

Since CLASP does not provide any input/output facilities, input/output must be handled through direct code. Wherever input/output was required in the kernels, direct code directives were used along with a comment indicating the input/output operation to be performed. No actual assembly language statements were coded.

CLASP documentation specifies that several constants used in an expression will be combined into a single constant by the compiler, rather than generating object code to combine them at execution time. While location constants present a unique problem in that they are relocatable, it was assumed that an absolute expression containing location constants would be evaluated by the compiler and replaced with a single constant.

8.2.2 Significant Problems Encountered

Since CLASP and SPL are quite similar, many of the problems encountered in SPL were also encountered in CLASP. In particular, the problems concerned with interrupt control (LOCK, UNLOCK, chronic statements), indirect "call", and fixed-point intermediate scaling discussed in Paragraph 8.1.2 are common to SPL and CLASP. Additional problems, unique to CLASP, are discussed below.

CLASP allows preset constants to be defined in terms of expressions involving other, previously defined, constants (Paragraph

8.2.3). It does not, however, provide a compile-time substitution facility similar to that of the other three languages. Such facilities allow symbolic names to be defined as equivalent to literal values which are used by the compiler for substitution whenever the symbolic names are subsequently encountered in the source code. Compile-time facilities help reduce storage requirements since the definition of the symbolic name does not utilize a memory location of execution time to store the item. They are also desirable from the viewpoint of improving maintainability and readability. An example is provided in Paragraph 6.1.1.4.

CLASP requires all program code to be compiled at one time; there is no provision for link-editing separately combined programs or for utilization of a COMPOOL containing common data and subroutines as in SPL. This makes overall program generation more difficult.

Another problem associated with program structure is the restriction that procedures can have only one entry point. This required certain kernels to be divided into several procedures where such division was convenient. In other kernels, where the interrelationship of the program logic was too complex to allow splitting the kernel into several procedures, slight modifications were made to the program logic to accommodate a single entry point. A good example of this is the Switch Selector Processor kernel whose entry logic is shown here.

```
PROC .MSS00 "SWITCH SELECTOR PROCESSING"

GOTO (, MSS05, MSS10, MSS20, MSS30, MSS40,        X
      MSS50, MSS55, MSS60, MSS70, MSS80) DGSSM
```

The caller must set the index DGSSM to the proper value before calling MSS00 in order to execute the desired sections of logic. In SPL, each of the MSSxx entry points could be called directly.

CLASP does not provide an indirect addressing capability. Location constants (pointers) are available but, without indirect addressing, have limited utility. The lack of indirect addressing required alterations in program logic in several kernels. Access to the Switch Selector Tables, as discussed in Paragraph 8.3.2, is a good example of this restriction.

Facilities for multiple-element data aggregates are too restrictive

in that all elements must have the same attributes. While this is acceptable for vectors and matrices, it limits the manner in which data tables can be structured.

CLASP restricts data item names and statement labels to eight characters or less. Since existing Saturn flight program names and labels were adopted directly in most cases and were already limited, this was not a widespread shortcoming. However, in the Iterative Guidance Mode kernel more descriptive names had been generated for SPL coding, and some of these names had to be shortened to fit the CLASP restriction. This, of course, made them less descriptive in certain areas.

8.2.3 Desirable Characteristics

Some of the SPL features discussed in Paragraph 8.1.3 are available in CLASP also. Most of them, however, are limited in capability and, as a result, provide less programming power and flexibility. These common facilities are discussed first, followed by unique CLASP features.

The format of a CLASP statement is a good example of the limited flexibility of the language as compared to SPL. Statements are free form except for being limited to columns 1-72. Column 73 must be used to indicate continuation for a statement which must be continued on the following line. Columns 74-80 can be used only for sequencing. Statement delimiters are not required unless multiple statements are coded on the same line, in which case the statements are separated by a dollar sign.

SPL's location constants are available, but location variables and indirect addressing are not. Contextual variables exist only as an attribute of a fixed-point variable rather than as a separate data type. CLASP facilities for handling mixed data types are, in general, similar to those of SPL. However, the CLASP method for combining logical terms with numeric is more restricted in that the logical terms are treated as integers and scaled accordingly, whereas logical terms in SPL are used unscaled.

There are two additional features provided exclusively by CLASP. Preset values for declared data items can be expressed in terms of previously declared constant values. This facilitates program maintenance functions since updating a key value will automatically update all other values dependent upon it. Readability can also

be improved by expressing derived values in terms of their more meaningful components.

CLASP permits the programmer to specify how object code optimization is to be handled. In addition to the normal optimization performed by the compiler, the programmer can request that object code efficiency be emphasized for execution time, core requirements, or a weighted combination of time and core.

8.3 HAL

8.3.1 Groundrules and Assumptions

Most of the groundrules and assumptions made for SPL were also made for the HAL coding. Each kernel was separately compilable and utility routines were in the Compool. The same interrupt and input/output register names were used. However, the real time task control in HAL clarified the question of changing interrupt responses, so the SPL assumption about multiple chronic statements was not required.

As in SPL, READ and WRITE statements were used to perform real time input/output functions even though they were not specifically adapted for such purposes in the specification. The HAL specification indicates that application-oriented input/output capabilities are to be supplied for each compiler implementation.

No fixed-point arithmetic operations were performed, because HAL does not allow fixed- and floating-point data in the same compiler implementation. All non-floating-point operations had to be integer operations.

HAL allows the break character (_) to be used in names and labels. Some of the names and labels generated for the SPL coding were modified by inserting the break character at meaningful points to make the name or label more descriptive.

8.3.2 Significant Problems Encountered

Generally speaking, HAL is more difficult to code than either SPL or CLASP. The programmer is forced to be more explicit and, as a result, more aware of the ramifications of each statement coded. For example, in SPL a tremendous amount of flexibility was built into

the language to allow coding of expressions containing mixed data types. Implicit data conversions were performed automatically by the language. In HAL the programmer is required to use explicit built-in conversion functions in certain areas (notably for performing logical operations on numeric data). Another example of language explicitness is the statement format which requires a line descriptor coded in column one of subscript and exponent lines and a semi-colon to delimit the statement. Some of this writing burden is beneficial to process reliability (Paragraph 8.3), but it still makes coding more difficult.

Perhaps the largest single problem area in HAL is that related to the operating system. Certain features are provided for interfacing with an operating system while other features, such as those provided by SPL and CLASP for controlling interrupts, are not available in HAL. Apparently it is intended that HAL is not to be used for programming operating system functions such as processing interrupts. As a result, the kernels could not be coded realistically with regard to such functions. In particular, appropriate comments were inserted into the program wherever it was necessary to inhibit or enable interrupts, because the capability was not provided.

The CALL statement provided by HAL for invoking another program does not allow the passage of parameters between the programs or tasks. All interprogram communication must be accomplished via common data storage. A procedure CALL does permit parameters to be passed but does not permit the address of an error exit to be passed as a parameter. Control must always be returned to the statement following the CALL where potential error indicators or conditions must be examined by the calling program.

HAL does not provide for both fixed-point and floating-point arithmetic data in a given compiler implementation. Either one or the other is implemented and is the only basic arithmetic data type besides the integer. Floating-point data was employed in coding the sample kernels. Integer data had to be utilized where timing efficiency was required, and the requirement to represent everything in integer units was a significant burden.

HAL does not have a contextual data type, as SPL does, for temporary storage of intermediate results. Since this capability was not available in HAL, it was necessary to define a separate temporary data item for each data type, and then to select the proper data item

in coding the kernels.

Indirect addressing and location constants and variables (pointers) are not available in HAL. While techniques can be employed which circumvent the need for pointers and indirect addressing, such techniques are sometimes undesirable themselves. To illustrate the problems, consider the Switch Selector Processor kernel which required data retrieval from a number of individual Switch Selector Tables. The manner in which it accessed data from these tables is shown below for SPL, CLASP and HAL.

In SPL:

```
SST1PTR = LOC'SUBTABLE1'
   !
   !
   !
SSTIME = IND (SST1PTR)
```

In CLASP:

```
SST1INDX = LOC'SUBTABLE1' - LOC'SSTABLE'
   !
   !
   !
SSTIME = SSTABLE(0,SST1INDX)
```

In HAL:

```
KSUBTABLE1 = 279;
   !
   !
   !
SST1INDX = KSUBTABLE1;
   !
   !
   !
SSTIME = SSTABLE          ;
                  1,SST1INDX
```

In SPL, the individual subtables could be independently defined and indirectly addressed via the location variable SST1PTR. Each subtable could be of any length and could be located anywhere in memory.

CLASP required all of the subtables to be organized into a compound, all-inclusive master table (SSTABLE) through use of an overlay. A specific subtable could then be addressed by referencing

SSTABLE and specifying an index. The index, SST1INDX, was assigned a value, which was the difference between two location constants. The two location constants addressed the beginning locations of the subtable and of the master table.

HAL also required the subtables to be organized into one large table from which they were accessed through an indexed address. However, without the capability of generating location constants to address the subtables, the index values could not be generated automatically by the compiler. Instead, the index values had to be computed manually by the programmer counting table entries. Such manual computations are error prone, and reduce maintainability because each added table entry requires manual modification of all index values below it in the table.

In conjunction with the indirect addressing/location constant problem, the multiple copy capability for structures could have been used in HAL for the Switch Selector Table by making each subtable a copy of a structure. However, since the number of entries varied considerably from table to table, a prohibitive amount of memory would have been wasted because each copy of a structure must be the same size. This fact, combined with the initiation problem discussed in the following paragraph, eliminated the use of a structure for the Switch Selector Table. It was, instead, declared an integer array.

HAL features for organizing tabular data are quite comprehensive. Structures provide capabilities for organizing complex groups of data and, for many applications, should be sufficient. The manner in which constants or initial values are specified for structures, however, was undesirable for the Switch Selector and Events Tables. For example, if the Events Table of the Events Processor module had been declared as a structure, the event times would have been made one array substructure and the event identifications would have been another array substructure. In such a case, the format for presetting constant values would require that all time values be specified together in one list followed by the values for all the identifications. Such an arrangement would not provide a convenient means for correlating the times with the identifications, as indicated by:

```
DECLARE 1 EPTABLE
          2 EPID ARRAY(13) INTEGER INITIAL
            (1, 4#0, 2, 3, 0, 4, 5, 0, 6, 0);
             2 EPTIME ARRAY(13) SCALAR INITIAL
               (0, 16, 17, 17.5, 2#0, 1, 6, 9, 10, 14, 134.7, 0);
```

Thus, it was decided to declare the Events Table as a simple array where the two items could be specified side-by-side to provide the means for correlating an event with its associated time.

```
DECLARE EPTABLE ARRAY(2,13) INTEGER INITIAL

  (1,     0,  /*START OF TIME BASE 0 TABLE*/
   0,   160,
   0,   170,
   0,   175,
   0,     0,
   2,     0,  /*START OF TIME BASE 1 TABLE*/
   3,    10,
   0,    60,
   4,    90,
   5,   100,
   0,   140,
   6,  1347,
   0,     0);
```

An array organization, however, is not completely adequate either, since all elements in an array must have the same attributes. In the Events Table just discussed, the event identifications were integers and thus required that the event times be integers also. Since times had to be specified to the nearest tenth of a second, all times had to be multiplied by a factor of ten, so they would be integer values with measurement units of one-tenth second.

HAL,in general, does not permit multiple expressions to appear on the right-hand side of an equal sign in an assignment statement. Exceptions occur with the VECTOR and MATRIX built-in functions where multiple expressions are enclosed within parentheses for assignment to vectors, matrices, and one or two dimensional arrays. Such capability is useful in assigning values to a number of related items.

The logical operators AND and OR can be used only for string type data and no implicit conversions are made for arithmetic data. If it is desired to mask an arithmetic data item, the built-in conversion function BIT must be used to convert the item to a bit-string before the masking operation can be performed.

8.3.3 Desirable Characteristics

Except for the restrictions of line descriptors in column 1 and

semicolons to delimit statements, HAL's statement format has the same free-form as SPL's. That is, statements can be positioned in any column, and multiple statements can be coded on a single line or a single statement can be continued on multiple consecutive lines.

HAL provides a greater selection of methods for organizing multiple-item data aggregates than SPL, CLASP, or CMS-2. Vectors and matrices are handled as unique data types and, along with a more comprehensive group of associated vector and matrix operations, provide greater mathematical expression capability. Arrays and structures are provided which have capabilities similar to those of SPL arrays and tables. HAL's capabilities are more general, however, in that arrays can contain vectors and matrices, and structures can contain arrays as well as vectors and matrices. Structures and substructures may also have multiple copies, accessed through subscripts.

The ability to handle mixed data types in expressions and assignment statements exists in HAL, but it is not quite as flexible as in SPL or CLASP. A sample restriction is the inability to perform logical operations on numeric data items without the explicitly coded conversion of the numeric item to a bit-string. In general, the language will handle mixed expressions adequately but frequently requires explicit conversion specifications to be coded into the expression rather than performing the conversions implicitly.

Although interrupt inhibit and enable capabilities were not provided, the real time task scheduling capabilities of HAL were fully capable of declaring interrupt responses and assigning them to interrupts.

The Compool concept provided by SPL is also available in HAL. It serves as the only means for communicating data among tasks and programs, since parameter lists are not available in the program and task CALL statements.

8.4 Compiler Monitor System-2 (CMS-2)

8.4.1 Groundrules and Assumptions

Some of the groundrules and assumptions made for SPL were also made for the CMS-2 coding. Each kernel was a separately compilable unit and utility routines were implicitly available to all the kernels. Interrupt interfaces were not provided by the language, so

all interrupt controls and real time responses were indicated by comments, and no special groundrules or assumptions needed to be established. Real time input/output operations were also indicated by comments. Arithmetic operations were coded in floating-point, except where fixed-point operations were dictated by execution time efficiency requirements in the Minor Loop.

Some of the kernels were not coded in CMS-2. Experience gained in coding with SPL, CLASP, and HAL indicated that for CMS-2, the characteristics demonstrated by these kernels are adequately illustrated by the remaining kernels. The kernels omitted are:

- o Interrupt Processor
- o Non-Interrupt Sequencer
- o Periodic Processor
- o Events Processor
- o Accelerometer Processor
- o Initialization

8.4.2 General Problems Encountered

The CMS-2 programming language is designed to produce object code which executes under control of an operating system and, as a result, does not provide facilities for controlling or responding to interrupts. Comments were used in the kernels to indicate where interrupt inhibiting or enabling was required.

Several features of the language are unnatural. The most flagrant example is the requirement to append a "D" to decimal numbers to distinguish them from octal. This applies not only to program literal constants but also to compiler control constants such as a number describing a fixed-point scaling or a field length. Most languages distinguish between octal and decimal by considering decimal the natural form and by tagging octal constants in some fashion. The CMS-2 technique is abnormal and, therefore, error-prone from the programmer standpoint, as well as unnatural to the reader. CMS-2 uses NOT to mean "not equal to" in relational expressions and "COMP" to specify a Boolean "not" function. This also is unconventional.

While not necessarily unnatural, the specification of the starting bit position for a FIELD is inconsistent with other such specifications within the language. It references bits within a word by numbering them from right to left rather than left to right as is done in all other bit or word references.

Certain capabilities in the area of data declarations which are provided by the other languages are not available in CMS-2. For instance, data items cannot be declared constant and, consequently, there can be no checking by the compiler to detect accidental attempts to modify constant data. This omission also prevents the compiler from optimizing object code based on the fact that a data item is a constant. There is no contextual data type for use as temporary storage. Data declarations cannot be factored unless all data items in the declaration have identical attributes. The factoring capability provided by the other languages permits data items with only some attributes in common to be defined in a single statement. The common attributes are specified only once and apply to each item defined in the statement, unless overriden for a given item by explicitly specifying different attributes uniquely for that item.

Tables cannot be preset within a declaration statement. Instead a separate "DATA" statement must be coded for each individual entry in the table. There is no repetition factor or list facility for presetting multiple elements.

Limitations were also found in the dynamic statements of the language. The IF statement, for example, is quite restricted. It has no ELSE clause and requires the programmer to code a GOTO statement to transfer control around what would ordinarily be expressed as an ELSE clause. The THEN clause is restricted to a single statement, requiring the programmer to code a compound statement consisting of individual statements concatenated by the THEN primitive whenever a number of functions must be performed within a THEN clause. For example:

```
IF DVASW (0, TB6C)

    THEN SET DVASW (0, TB6C) TO 0
    THEN SET VASPI (0, TB6C) TO 1
    THEN SET DVMC6 (0, TB6B) TO 1
    THEN SET SST1PTR TO CORAD (SSTTB6C)
    THEN SSTUPD OUTPUT VATRR
    THEN GOTO SS1050 $
```

IF statements cannot be nested.

Logical operations can be performed only on Boolean data. There is no bit-string capability in general. Thus, it is difficult to perform bit manipulation. The built-in function BIT or special FIELD definitions can be used for such purposes but in certain cases they are ineffective. For example, if it is necessary to mask an input data item such as a gimbal reading, either BIT or a FIELD can be used because the bits that are to be masked are always the same. However, if the bits to be masked change during real time execution, neither of these techniques can be applied and the programmer is forced to use a prohibitively expensive loop which considers one bit each pass.

There are no vector matrix facilities in CMS-2. Such functions as vector dot and cross products, vector rotations, and matrix multiplication must be coded in long form or performed via subroutines.

In general, no padding is provided when a small item is assigned to a large item. This is true of Hollerith character-strings and also for tables. The excess data of the large receptacle is left unchanged. Of course, if the item being assigned is larger than the receiving item, the excess data is truncated and lost.

Procedures are restricted to a single entry point as was true for HAL and CLASP, so procedures had to be subdivided or additional logic provided (Paragraph 8.2.2).

CMS-2 appears to severely restrict combinations of mixed data types in expressions, because the specification makes very little mention of the topic. The only mixed expression discussed at all is the combination of fixed-point and floating-point data. Nothing is specified, for instance, about how integer arithmetic is performed or about combining arithmetic terms with Boolean or Hollerith terms. The same restrictions seem to apply to assignment statements. The data type of the receptacle must generally agree with that of the data being assigned to it.

No input/output facilities are provided for real time data. Comments were coded at points in the kernels where input/output was required.

No facilities are available for the programmer to specify small

time delays such as those required by the Minor Loop and Switch Selector Processing kernels. Comments were inserted in the coding where delays were required.

8.4.3 Desirable Characteristics

Except for the reservation of the first ten columns of a line for card identification/sequencing purposes and the requirement for coding a dollar sign to delimit statements, the CMS-2 statement format is free. Statements can be coded anywhere between columns 11 through 80 of one or more lines or multiple statements can be coded on a single line.

Several significant declarative-type features are provided by CMS-2. In addition to facilities for the standard data-types, the language also has status constants and variables like those provided by SPL (Paragraph 6.1.2). Multiple-item data aggregates are handled via tables, also similar to those of SPL. Arrays exist as a type of table rather than as a separately defined type of data organization.

Memory address constants (pointers) are provided which are similar to those of SPL and CLASP (Paragraph 6.1.1). They are designed primarily to pass data table locations to subroutines, so that the entire table need not be passed. The other languages inherently pass only the location when a table or array is a subroutine argument. Therefore, they do not need to make this distinction.

With respect to dynamic statements which generate executable code, the indexed call capability provided exclusively by CMS-2 was found to be quite useful. It utilizes a list of procedure names and an associated index which can be referenced by special call statements. Any procedure in the list can be invoked with a single call statement by setting the index to the proper value. Such a capability is desirable for use in control applications where real time decisions must be made to determine which of a number of tasks is to execute. Input and output parameters can be passed but must be identical in data type and attributes for all procedures in the list.

GLOSSARY

Data item	-	any element or organization (array, table, etc.) of data including literal, constant, and variable data.
Literal	-	a literal representation of a value.
Constant	-	a data item which always contains the same value.
Variable	-	a data item which may contain different values at different times.
Data element	-	a single valued data item (as opposed to an array or table of many elements).
Scalar	-	same as a data element (note that HAL's key word SCALAR has a more restricted definition than this).
Bit-string data	-	a group of individual data bits in which each bit, or small groups of bits, have individual meanings or values.
Logical data	-	bit-string data.
Boolean data	-	a bit-string of length one.
Formula	-	any literal, constant, variable or valid combination of them with operators.
{ }	-	indicates that alternatives from within the braces may be selected.
Logical Condition	-	a Boolean formula
Boolean formula	-	any formula of any complexity which results in a Boolean (true/false) value. In SPL (only), any value is considered true if non-zero and false if zero, when used in a Boolean context.

Identifier	-	a character-string which is created by the programmer and assigned to a data item or statement.
Name	-	an identifier which is assigned to a data item and by which it is referenced in the coding.
Entry-point label	-	an identifier which is assigned to a routine or external entry-point to a routine and by which it is referenced in the coding.
Statement label	-	an identifier assigned to an internal statement of a routine.
Routine	-	any program, procedure or function.

REFERENCES

A. PROGRAMMING LANGUAGES

1. "Flight Computer and Language Processor Study" (CLASP Specification), Logicon, Inc., March 1970, NASA/ERC Contract NAS12-2005, Document No. NASA CR-1520.

2. "CLASP - Its Role in AADC Software Development", Edward H. Bersoff, Logicon, Inc., High Level Aerospace Computer Programming Language Conference Proceedings, Naval Research Laboratory, 30 June 1970.

3. "Compiler Monitor System-2 (CMS-2) User's Manual", Computer Sciences Corporation, 9 June 1969, Navy Contract N00123-67-C-0214, Manual No. M-5012.

4. "A Technical Overview of Compiler Monitor System-2 (CMS-2)", Computer Sciences Corporation, High Level Aerospace Computer Programming Language Conference Proceedings, Naval Research Laboratory, 30 June 1970.

5. "Requirements Analysis for a Manned Spaceflight Programming Language and Compiler", Intermetrics, Inc., NASA/MSC Contract NAS9-10542, Document No. MSC-01845.

6. "The Programming Language HAL - A Specification", Intermetrics, Inc., NASA/MSC Contract NAS9-10542, Document No. MSC-01846, June 1971.

7. "Development of an MSC Language and Compiler", Intermetrics, Inc., NASA/MSC Contract NAS9-10542, Document No. MSC-01848, June 1971.

8. "Space Programming Language/MARK IV (SPL/MK IV) Reference Manual", System Development Corporation, 31 August 1970, Air Force Contract FO 4701-70-C-0022, Document No. SAMSO TR-70-349.

9. "Introduction to Space Programming Language":
   "Implementation of SPL" (SAMSO TR-70-324),
   "Development of SPL" (SAMSO TR-70-325)
   "Overview of SPL" (SAMSO TR-70-326)
   "Conception of SPL" (SAMSO TR-327),
   System Development Corporation, 11 September 1970, Air Force Contract FO 4701-70-C-0214.

10. "Space Programming Language: Flight Software Comes of Age", R. E. Nimensky, High Level Aerospace Computer Programming Language Conference Proceedings, Naval Research Laboratory, 30 June 1970.

11. "AED-0 Programmer's Guide", Clarence G. Fieldmann, Douglas T. Ross, Jorge E. Rodriquez, MIT, January 1970, Air Force Contract F 33615-69-C-1341, Report ESL-R-406.

12. "The SHOWIT System: An Example of the AED Approach", Softech, June 1970, Document No. A-AED-000-1.

13. "A Programming Language", Kenneth E. Iverson, John Wiley & Sons, Inc., 1962.

14. "Ground Operations Aerospace Language (GOAL)", NASA/MSC, September 1971, Draft.

15. "IBM System/360 Operating System PL/I (F) Language Reference Manual", IBM Corporation, June 1970, GC 28-8201-3.

B. EVALUATION CRITERIA

1. "Object Code Optimization", Edward S. Larry and C. W. Meadlock, Communications of the ACM, Volume 12/ Number 1/ January 1969.

2. "High Speed Compilation of Efficient Object Code", C. W. Gear, Communications of the ACM, Volume 8/ Number 8/ August 1965.

3. "On the Automatic Simplification of Computer Programs", J. Nievergelt, Communications of the ACM, Volume 8/ Number 6/ June 1965.

4. "Program Optimization", F. E. Allen, Annual Review in Automatic Programming, Volume 5, Pergarnon, New York.

5. "ALPHA - An Automatic Programming System of High Efficiency", A. P. Yershov, Journal of the Association for Computing Machinery, Volume 13/ Number 1/ January 1966.

6. "Programming Languages: History and Fundamentals" Jean E. Sammet, Prentice Hall.

7. "A Guide to PL/I", S. V. Pollack and T. D. Sterling, Holt, Rinehart, and Winston, Inc.

8. "Software Development and Verification Technology", H. Trauboth and B. Hodges, NASA/MSFC, Space Shuttle Technology Conference on Integrated Electronics, 13 May 1971.

9. "Space Software: At the Crossroads", Levi J. Carey and Walter A. Sturm, System Development Corporation, Space/Aeronautics, December 1968.

10. "Effective Program Development: The Choices; Expressing Program Logic", Michael S. Montalbano, Data Processing Digest, September 1968.

11. "Flight Program Language Evaluation Method", M&S Computing, Inc., NASA/MSFC Contract NAS8-26990, Document No. 71-0010, June 7, 1971.

12. "Data Processing Technologies:

Volume I - High-Level Language Evaluation
Volume II - Compiler Studies
Volume III - Special Studies (Secret)"

Teledyne Brown Engineering, May 1970, IESD-ABMDA-1144.

13. "Problems in, and a Pragmatic Approach to, Programming Language Measurement," Jean E. Sammett, AFIPS Conference Proceedings, Volume 39, November 1971.

C. FLIGHT PROGRAM CHARACTERISTICS

1. "LVDC Equation Defining Document for the Saturn V Flight Program", Revision G, IBM Corporation, 2 December 1970, NASA/MSFC Contract NAS8-14000, MSFC Document No. III-4-423-15.

2. "Apollo Telescope Mount Digital Computer (ATMDC) Program Definition Document (PDD)", IBM Corporation, 4 November 1970, NASA/MSFC Contract NAS8-20899, IBM No. 70-207-0002.

3. "Onboard Autonomy Panel Report (Preliminary)", Onboard Autonomy Panel of the Manned Space Flight Computer Study, February 4, 1970.

4. "Spaceborne Computer Executive Routine Functional Design Specification", Computer Sciences Corporation, October 1971, NASA CR-1869.

5. "Space Station Program Extension Period Final Performance Review", McDonnell Douglas Astronautics Company, November 1971, MDC G2586.

6. "Modular Space Station Computer Study", IBM Corporation, October 1971, IBM No. 71W-00345.

7. "High Energy Astronomy Observatory, Phase B Final Report", Grumman Aerospace Corporation, April 1971, DRD No. MA-082-U2.

8. "Reference Earth Orbital Research and Applications Investigations (Blue Book)", NASA, 15 January 1971.

9. "Mathematical Concepts and Historical Development of the MASCOT Guidance Technique for Space Vehicles", NASA/MSFC, NASA TM X-64608.