DEPARTMENT OF DEFENSE
REQUIREMENTS FOR HIGH ORDER
COMPUTER PROGRAMMING LANGUAGES

"STEELMAN"

June 1978

PREFACE

The Department of Defense Common High Order Language program was established in 1975 with the goal of establishing a single high order computer programming language appropriate for DoD embedded computer systems. A High Order Language Working Group (HOLWG) was established to formulate the DoD requirements for high order languages, to evaluate existing languages against those requirements, and to implement the minimal set of languages required for DoD use. As an administrative initiative toward the eventual goal, DoD Directive 5000.29 provides that any new defense systems should be programmed in a DoD approved and centrally controlled high order language. DoD Instruction 5000.31 gives an interim list of approved languages: COBOL, FORTRAN, TACPOL, CMS-2, SPL/1, and JOVIAL J3 and J73. Economic analyses that were used to quantify the benefits from increased use of high order languages, also showed that the rapid introduction of a single modern language would increase the benefits considerably. The requirements have been widely distributed for comment throughout the military and civil communities, producing successively more refined versions from STRAWMAN through WOODENMAN, TINMAN, IRONMAN, and the present STEELMAN. During the requirement development process, it was determined that the single set of requirements generated was both necessary and sufficient for all major DoD applications. Formal evaluation was performed on dozens of existing languages concluding that no existing language could be adopted as a single common HoL for the DoD but that a single language meeting essentially all the requirements was both feasible and desirable. Four contractors were funded to produce competing prototype designs. After analysis of these preliminary designs the number of design teams was reduced to two. Their designs will be completed and a single language will emerge. Further steps in the program will include test and evaluation of the language, production of compilers and other tools for software development and maintenance, control of the language, and validation of compilers. Government-funded compilers and software tools, as well as the compiler validation facility, will be widely and inexpensively available and well maintained.

THE TECHNICAL REQUIREMENTS

The technical requirements for a common DoD high order programming language given here arc a synthesis of the requirements submitted by the Military Departments. They specify a set of constraints on the design of languages that are appropriate for embedded computer applications (i.e., command and control, communications, avionics, shipboard, test equipment, software development and maintenance, and support applications). We would especially like to thank the phase one analysis teams, the language design teams, and the many other individuals and organizations that have commented on the Revised Ironman and have identified weaknesses and trouble spots in the technical requirements. A primary goal in this revision has been to reduce the complexity of the resulting language.

This revision incorporates the following changes. Care has been taken to ensure that the paragraph numbers remain the same as in the Revised Ironman. There have been several changes in terminology and many changes in wording to improve the understandability and preciseness of the requirements. Several requirements have been restated to remove constraints that were unintended but were implied because the requirement suggested a particular mechanism rather than giving the underlying requirement. The requirements for embedded comments (2I), unordered enumeration types (3-2B), associative operator specifications (7D), dynamic aliasing of array components (10B), and multiple representations of data (11B) have been deleted because they have been found unnecessary or are not adequately justified. The minimal source language character set has been reduced to 55 characters to make it compatible with the majority of existing input devices (2A). The do together model for parallel processing-has been found inadequate for embedded computer applications and has been replaced by a requirement for parallel processes (section 9). The preliminary designs have demonstrated the need for additional requirements for explicit conversion between types (3B), subtype constraints (3D), renaming (3-5B), a language distinction between open and closed scopes (5G), and the ability, but preferably not special mechanisms, to pass data between parallel processes (9H), to write nonverifiable assertions (10F), to wait for several signals simultaneously (9J), and to mark shared variables (9C).

The Steelman is organized with an outline similar to that expected in a language defining document. Section 1 gives the general design criteria. These provide the major goals that influenced the selection of more specific requirements in later sections and provide a basis for language design decisions that are not otherwise addressed in this document. Sections 2 through 12 give more specific constraints on the language and its translators. The Steelman calls for the inclusion of features to satisfy specific needs in the design, implementation, and maintenance of military software, specifies both general and specific characteristics desired for the language, and calls for the exclusion of certain undesirable characteristics. Section 13 gives some of the intentions and expectations for development, control, and use of the language. The intended use and environment for the language has strongly influenced the requirements, and should influence the language design.

A precise and consistent use of terms has been attempted throughout the document. Many potentially ambiguous terms have been defined in the text. Care has been taken to distin-

guish between requirements, given as text, and comments, given as bracketed notes.
The following terms have been used throughout the text to indicate where and to what degree individual constraints apply:

shall:	indicates a requirement placed on the language or translator
should:	indicates a desired goal but one for which there is no objective test
shall attempt:	indicates a desired goal but one that may not be achievable given the current state-of-the-art, of may be in conflict with other more important requirements.
shall require:	indicates a requirement placed on the user by the language and its translators (language is subject)
shall permit:	indicates a requirement placed on the language to provide an option to the user (language is subject)
must:	indicates a requirement placed on the user by the language and its translators (user is subject)
may:	indicates a requirement placed on the language to provide an option to the user (user is subject)
will:	indicates a consequence that is expected to follow or indicates an intention of the DoD; it does not in any case by itself constrain the design of the language
translation:	refers to any processing applied to a program by the host or object machine before execution; it includes lexical analysis, syntactic error checking, program analysis, optimization, code generation, assembly, and loading
execution:	refers to the processing by the object machine to carry out the actions prescribed by the program.

1. General Design Criteria

1A. Generality. The language shall provide generality only to the extent necessary to satisfy the needs of embedded computer applications. Such applications involve real time control, self diagnostics, input-output to nonstandard peripheral devices, parallel processing, numeric computation, and file processing.

1B. Reliability. The language should aid the design and development of reliable programs. The language shall be designed to avoid error prone features and to maximize automatic detection of programming errors. The language shall require some redundant, but not duplicative, specifications in programs. Translators shall produce explanatory diagnostic and warning messages, but shall not attempt to correct programming errors.

1C. Maintainability. The language should promote ease of program maintenance. It should emphasize program readability (i.e., clarity, understandability, and modifiability of programs). The language should encourage user documentation of programs. It shall require explicit specification of programmer decisions and shall provide defaults only for instances where the default is stated in the language definition, is always meaningful, reflects the most frequent usage in programs, and may be explicitly overridden.

1D. Efficiency. The language design should aid the production of efficient object programs. Constructs that have unexpectedly expensive implementations should be easily recognizable by translators and by users. Features should be chosen to have a simple and efficient implementation in many object machines, to avoid execution costs for available generality where it is not needed, to maximize the number of safe optimizations available to translators, and to ensure that unused and constant portions of programs will not add to execution costs. Execution time support packages of the language shall not be included in object code unless they are called.

1E. Simplicity. The language should not contain unnecessary complexity. It should have a consistent semantic structure that minimizes the number of underlying concepts. It should be as small as possible consistent with the needs of the intended applications. It should have few special cases and should be composed from features that are individually simple in their semantics. The language should have uniform syntactic conventions and should not provide several notations for the same concept. No arbitrary restriction should be imposed on a language feature.

1F. Implementability. The language shall be composed from features that are understood and can be implemented. The semantics of each feature should be sufficiently well specified and understandable that it will be possible to predict its interaction with other features. To the extent that it does not interfere with other requirements, the language shall facilitate the production of translators that are easy to implement and are efficient during translation. There shall be no language restrictions that are not enforceable by translators.

1G. Machine Independence. The design of the language should strive for machine independence. It shall not dictate the characteristics of object machines or operating systems except to the extent that such characteristics are implied by the semantics of control structures and built-in operations. It shall attempt to avoid features whose semantics depend on characteristics of the object machine or of the object machine operating system. Nevertheless, there shall be a facility for defining those portions of programs that are dependent on the object machine configuration and for conditionally compiling programs depending on the actual configuration.

1H. Complete Definition. The language shall be completely and unambiguously defined. To the extent that a formal definition assists in achieving the above goals (i.e., all of section 1), the language shall be formally defined.

2. General Syntax

2A. Character Set. The full set of character graphics that may be used in source programs shall be given in the language definition. Every source program shall also have a representation that uses only the following 55 character subset of the ASCII graphics:

```
 %&'()*+,-./:;<=>?
0l23456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ_
```

Each additional graphic (i.e., one in the full set but not in the 55 character set) may be replaced by a sequence of (one or more) characters from the 55 character set without altering the semantics of the program. The replacement sequence shall be specified in the language definition.

2B. Grammar. The language should have a simple, uniform, and easily parsed grammar and lexical structure. The language shall have free form syntax and should use familiar notations where such use does not conflict with other goals.

2C. Syntactic Extensions. The user shall not be able to modify, the source language syntax. In particular the user shall not be able to introduce new precedence rules or to define new syntactic forms.

2D. Other Syntactic Issues. Multiple occurrences of a language defined symbol appearing in the same context shall not have essentially different meanings. Lexical units (i.e., identifiers, reserved words, single and multicharacter symbols, numeric and string literals, and comments) may not cross line boundaries of a source program. All key word forms that contain declarations or statements shall be bracketed (i.e., shall have a closing as well as an opening key word). Programs may not contain unmatched brackets of any kind.

2E. Mnemonic Identifiers. Mnemonically significant identifiers shall be allowed. There shall be a break character for use within identifiers. The language and its translators shall not permit identifiers or reserved words to be abbreviated. (Note that this does not preclude reserved words that are abbreviations of natural language words.)

2F. Reserved Words. The only reserved words shall be those that introduce special syntactic forms (such as control structures and declarations) or that are otherwise used as delimiters. Words that may be replaced by identifiers, shall not be reserved (e.g., names of functions, types, constants, and variables shall not be reserved). All reserved words shall be listed in the language definition.

2G. Numeric Literals. There shall be built-in decimal literals. There shall be no implicit truncation or rounding of integer and fixed point literals.

2H. String Literals. There shall be a built-in facility for fixed length string literals.

String literals shall be interpreted as one-dimensional character arrays.

2I. Comments. The language shall permit comments that are introduced by a special (one or two character) symbol and terminated by the next line boundary of the source program.

3. Types

3A. Strong Typing. The language shall be strongly typed. The type of each variable, array and record component, expression, function, and parameter shall be determinable during translation.

3B. Type Conversions. The language shall distinguish the concepts of type (specifying data elements with common properties, including operations), subtype (i.e., a subset of the elements of a type, that is characterized by further constraints), and representations (i.e., implementation characteristics). There shall be no implicit conversions between types. Explicit conversion operations shall be automatically defined between types that are characterized by the same logical properties.

3C. Type Definitions. It shall be possible to define new data types in programs. A type may be defined as an enumeration, an array or record type, an indirect type, an existing type, or a subtype of an existing type. It shall be possible to process type definitions entirely during translation. An identifier may be associated with each type. No restriction shall be imposed on user defined types unless it is imposed on all types.

3D. Subtype Constraints. The constraints that characterize subtypes shall include range, precision, scale, index ranges, and user defined constraints. The value of a subtype constraint for a variable may be specified when the variable is declared. The language should encourage such specifications. [Note that such specifications can aid the clarity, efficiency, maintainability, and provability of programs.]

3.1. Numeric Types

3-1A. Numeric Values. The language shall provide distinct numeric types for exact and for approximate computation. Numeric operations and assignment that would cause the most significant digits of numeric values to be truncated (e.g., when overflow occurs) shall constitute an exception situation.

3-1B. Numeric Operations. There shall be built-in operations (i.e., functions) for conversion between the numeric types. There shall be operations for addition, subtraction, multiplication, division, negation, absolute value, and exponentiation to integer powers for each numeric type. There shall be built-in equality (i.e., equal and unequal) and ordering operations (i.e., less than, greater than, less than or equal, and greater than or equal) between elements of each numeric type. Numeric values shall be equal if and only if they have exactly the same abstract value.

3-1C. Numeric Variables. The range of each numeric variable must be specified in programs and shall be determined by the time of its allocation. Such specifications shall be interpreted as the minimum range to be implemented and as the maximum range needed by the application. Explicit conversion operations shall not be required between numeric ranges.

Approximate Arithmetic

3-1D. Precision. The precision (of the mantissa) of each expression result and variable in approximate computations must be specified in programs, and shall be determinable during translation. Precision specifications shall be required for each such variable. Such specifications shall be interpreted as the minimum accuracy (not significance) to be implemented. Approximate results shall be implicitly rounded to the implemented precision. Explicit conversions shall not be required between precisions.

3-1E. Approximate Arithmetic Implementation. Approximate arithmetic will be implemented using the actual precisions, radix, and exponent range available in the object machine. There shall be built-in operations to access the actual precision, radix, and exponent range of the implementation.

Exact Arithmetic

3-1F. Integer and Fixed Point Numbers. Integer and fixed point numbers shall be treated as exact numeric values. There shall be no implicit truncation or rounding in integer and fixed point computations.

3-1G. Fixed Point Scale. The scale or step size (i.e., the minimal representable difference between values) of each fixed point variable must be specified in programs and be determinable during translation. Scales shall not be restricted to powers of two.

3-1H. Integer and Fixed Point Operations. There shall be integer and fixed point operations for modulo and integer division and for conversion between values with different scales. All built-in and predefined operations for exact arithmetic shall apply between arbitrary scales. Additional operations between arbitrary scales shall be definable within programs.

3.2. Enumeration Types

3-2A.Enumeration Type Definitions. There shall be types that are definable in programs by enumeration of their elements. The elements of an enumeration type may be identifiers or character literals. Each variable of an enumeration type may be restricted to a contiguous subsequence of the enumeration.

3-2B.Operations on Enumeration Types. Equality, inequality, and the ordering operations shall be automatically defined between elements of each enumeration type. Sufficient additional operations shall be automatically defined so that the successor, predecessor, the position of any element, and the first and last element of the type may be computed.

3-2C.Boolean Type. There shall be a predefined type for Boolean values.

3-2D.Character Types. Character sets shall be definable as enumeration types. Char-

acter types may contain both printable and control characters. The ASCII character set shall be predefined.

3.3 Composite Types

3-3A.Composite Type Definitions. It shall be possible to define types that are Cartesian products of other types. Composite types shall include arrays (i.e., composite data with indexable components of homogeneous types) and records (i.e., composite data with labeled components of heterogeneous type).

3-3B.Component Specifications. For elements of composite types, the type of each component (i.e., field) must be explicitly specified in programs and determinable during translation. Components may be of any type (including array and record types). Range, precision, and scale specifications shall be required for each component of appropriate numeric type.

3-3C.Operations on Composite Types. A value accessing operation shall be automatically defined for each component of composite data elements. Assignment shall be automatically defined for components that have alterable values. A constructor operation (i.e., an operation that constructs an element of a type from its constituent parts) shall be automatically defined for each composite type. An assignable component may be used anywhere in a program that a variable of the component's type is permitted. There shall be no automatically defined equivalence operations between values of elements of a composite type.

3-3D. Array Specifications. Arrays that differ in number of dimensions or in component type shall be of different types. The range of subscript values for each dimension must be specified in programs and may be determinable at the time of array allocation. The range of each subscript value must be restricted to a contiguous sequence of integers or to a contiguous sequence from an enumeration type.

3-3E. Operations on Subarrays. There shall be built-in operations for value access, assignment, and catenation of contiguous sections of one-dimensional arrays of the same component type. The results of such access and catenation operations may be used as actual input parameter.

3-3F.Nonassignable Record Components. It shall be possible to declare constants and (unary) functions that may be thought of as record components and may be referenced using the same notation as for accessing record components. Assignment shall not be permitted to such components.

3-3G.Variants. It shall be possible to define types with alternative record.structures (l.e., variants). The structure of each variant shall be determinable during translation.

3-3H.Tag Fields. Each variant must have a nonassignable tag field (i.e., a component

that can be used to discriminate among the variants during execution). It shall not be possible to alter a tag field without replacing the entire variant.

3-3I. Indirect Types. It shall be possible to define types whose elements are indirectly accessed. Elements of such types may have components of their own type, may have substructure that can be altered during execution, and may be distinct while having identical component values. Such types shall be distinguishable from other composite types in their definitions. An element of an indirect type shall remain allocated as long as it can be referenced by the program. [Note that indirect types require pointers and sometimes heap storage in their implementation.]

3-3J. Operations on Indirect Types. Each execution of the constructor operation for an indirect type shall create a distinct element of the type. An operation that distinguishes between different elements, an operation that replaces all of the component values of an element without altering the element's identity, and an operation that produces a new element having the same component values as its argument, shall be automatically defined for each indirect type.

3.4. Sets

3-4A. Bit Strings (i.e., Set Types). It shall be possible to define types whose elements are one-dimensional Boolean arrays represented in maximally packed form (i.e, whose elements are sets).

3-4B. Bit String Operations. Set construction, membership (i.e., subscription), set equivalence and nonequivalence, and also complement, intersection, union, and symmetric difference (i.e., component-by-component negation, conjunction, inclusive disjunction, and exclusive disjunction respectively) operations shall be defined automatically for each set type.

3.5. Encapsulated Definitions

3-5A.Encapsulated Definitions. It shall be possible to encapsulate definitions. An encapsulation may contain declarations of anything (including the data elements and operations-comprising a type) that is definable in programs. The language shall permit multiple explicit instantiations of an encapsulation.

3-5B.Effect of Encapsulation. An encapsulation may be used to inhibit external access to implementation properties of the definition. In particular, it shall be possible to prevent external reference to any declaration within the encapsulation including automatically defined operations such as type conversions and equality. Definitions that are made within an encapsulation and are externally accessible may be renamed before use outside the encapsulation.

3-5C. Own Variables. Variables declared within an encapsulation, but not within a function, procedure, or process of the encapsulation, shall remain allocated and retain

their values throughout the scope in which the encapsulation is instantiated.

4. Expressions

4A. Form of Expressions. The parsing of correct expressions shall not depend on the types of their operands or on whether the types of the operands are built into the language.

4B. Type of Expressions. It shall be possible to specify the type of any expression explicitly. The use of such specifications shall be required only where the type of the expression cannot be uniquely determined during translation from the context of its use (as might be the case with a literal).

4C. Side Effects. The language shall attempt to minimize side effects in expressions, but shall not prohibit all side effects. A side effect shall not be allowed if it would alter the value of a variable that can be accessed at the point of the expression. Side effects shall be limited to own variables of encapsulations. The language shall permit side effects that are necessary to instrument functions and to do storage management within functions. The order of side effects within an expression shall not be guaranteed. [Note that the latter implies that any program that depends on the order of side effects is erroneous.]

4D. Allowed Usage. Expressions of a given type shall be allowed wherever both constants and variables of the type are allowed.

4E. Translation Time Expressions. Expressions that can be evaluated during translation shall be permitted wherever literals of the type are permitted. Translation time expressions that include only literals and the use of translation time facilities (see 11C) shall be evaluated during translation.

4F. Operator Precedence Levels. The precedence levels (i.e., binding strengths) of all (prefix and infix) operators shall be specified in the language definition, shall not be alterable by the user, shall be few in number, and shall not depend on the types of the operands.

4G. Effect of Parentheses. If present, explicit parentheses shall dictate the association of operands with operators. The language shall specify where explicit parentheses are required and shall attempt to minimize the psychological ambiguity in expressions. [Note that this might be accomplished by requiring explicit parentheses to resolve the operator-operand association whenever a nonassociative operator appears to the left of an operator of the same precedence at the least-binding precedence level of any subexpression.]

5. Constants, Variables, and Scopes

5A. Declarations of Constants. It shall be possible to declare constants of any type. Such constants shall include both those whose values-are determined during translation and those whose value cannot be determined until allocation. Programs may not assign to constants.

5B. Declarations of Variables. Each variable must be declared explicitly. Variables may be of any type. The type of each variable must be specified as part of its declaration and must be determinable during translation. [Note, "variable" throughout this document refers not only to simple variables but also to composite variables and to components of arrays and records.]

5C. Scope of Declarations. Everything (including operators) declared in a program shall have a scope (i.e., a portion of the program in which it can be referenced). Scopes shall be determinable during translation. Scopes may be nested (i.e., lexically embedded). A declaration may be made in any scope. Anything other than a variable shall be accessable within any nested scope of its definition.

5D. Restrictions on Values. Procedures, functions, types, labels, exception situations, and statements shall not be assignable to variables, be computable as values of expressions, or be usable as nongeneric parameters to procedures or functions.

5E. Initial Values. There shall be no default initial-values for variables.

5F. Operations on Variables. Assignment and an implicit value access operation shall be automatically defined for each variable.

5G. Scope of Variables. The language shall distinguish between open scopes (i.e., those that are automatically included in the scope of more globally declared variables) and closed scopes (i.e., those in which nonlocal variables must be explicitly Imported). Bodies of functions, procedures, and processes shall be closed scopes. Bodies of classical control structures shall be open scopes.

6. Classical Control Structures

6A. Basic Control Facility. The (built-in) control mechanisms should be of minimal number and complexity. Each shall provide a single capability and shall have a distinguishing syntax. Nesting of control structures shall be allowed. There shall be no control definition facility. Local scopes shall be allowed within the bodies of control statements. Control structures shall have only one entry point and shall exit to a single point unless exited via an explicit transfer of control (where permitted, see 6G), or the raising of an exception (see 10C).

6B. Sequential Control. There shall be a control mechanism for sequencing statements. The language shall not impose arbitrary restrictions on programming style, such as the choice between statement terminators and statement separators, unless the restriction makes programming errors less likely.

6C. Conditional Control. There shall be conditional control structures that permit selection among alternative control paths. The selected path may depend on the value of a Boolean expression, on a computed choice among labeled alternatives, or on the true condition in a set of conditions. The language shall define the control action for all values of the discriminating condition that are not specified by the program. The user may supply a single control path to be used when no other path 4s selected. Only the selected branch shall be compiled when the discriminating condition is a translation time expression.

6D. Short Circuit Evaluation. There shall be infix control operations for short circuit conjunction and disjunction of the controlling Boolean expression in conditional and iterative control structures.

6E. Iterative Control. There shall be an iterative control structure. The iterative control may be exited (without reentry) at an unrestricted number of places. A succession of values from an enumeration type or the integers may be associated with successive iterations and the value for the current iteration accessed as a constant throughout the loop body.

6G. Explicit Control Transfer. There shall be a mechanism for control transfer (i.e., the go to). It shall not be possible to transfer out of closed scopes, into narrower scopes, or into control structures. It shall be possible to transfer out of classical control structures. There shall be no control transfer mechanisms in the form of switches, designational expressions, label variables, label parameters, or alter statements.

7. Functions and Procedures

7A. Function and Procedure Definitions. Functions (which return values to expressions) and procedures (which can be called as statements) shall be definable in programs. Functions or procedures that differ in the number or types of their parameters may be denoted by the same identifier or operator (i.e., overloading shall be permitted). [Note that redefinition, as opposed to overloading, of an existing function or procedure is often error prone.]

7B. Recursion. It shall be possible to call functions and procedures recursively.

7C. Scope Rules. A reference to an identifier that is not declared in the most local scope shall refer to a program element that is lexically global, rather than to one that is global through the dynamic calling structure.

Functions

7D. Function Declarations. The type of the result for each function must be specified in its declaration and shall be determinable during translation. The results of functions may be of any type. If a result is of a nonindirect array or record type then the number of its components must be determinable by the time of function call.

Parameters

7F. Formal Parameter Classes. There shall be three classes of formal data parameters: (a) input parameters, which act as constants that are initialized to the value of corresponding actual parameters at the time of call, (b) input-output parameters, which enable access and assignment to the corresponding actual parameters, either throughout execution or only upon call and prior to any exit, and (c) output parameters, whose values are transferred to the corresponding actual parameter only at the time of normal exit. In the latter two cases the corresponding actual parameter shall be determined at time of call and must be a variable or an assignable component of a composite type.

7G. Parameter Specifications. The type of each formal parameter must be explicitly specified in programs and shall be determinable during translation. Parameters may be of any type. The language shall not require user specification of subtype constraints for formal parameters. If such constraints are permitted they shall be interpreted as assertions and not as additional overloading. Corresponding formal and actual parameters must be of the same type.

7H. Formal Array Parameters. The number of dimensions for formal array parameters must be specified in programs and shall be determinable during translation. Determination of the subscript range for formal array parameters may be delayed until invocation and may vary from call to call. Subscript ranges shall be accessible within function and procedure bodies without being passed as explicit parameters.

7I. Restrictions to Prevent Aliasing. The language shall attempt to prevent aliasing (l.e., multiple access paths to the same variable or record component) that is not intended, but shall not prohibit all aliasing. Aliasing shall not be permitted between output parameters nor between an input-output parameter and a nonlocal variable. Unintended aliasing shall not be permitted between input-output parameters. A restriction limiting actual input-output parameters to variables that are nowhere referenced as nonlocals within a function or routine, is not prohibited. All aliasing of components of elements of an indirect type shall be considered intentional.

8. Input-Output, Formating and Configuration Control

8A. Low Level Input-Output. There shall be a few low level input-output operations that send and receive control information to and from physical channels and devices. The low level operations shall be chosen to insure that all user level input-output operations can be defined within the language.

8B. User Level Input-Output. The language shall specify (i.e., give calling format and general semantics) a recommended set of user level input-output operations. These shall include operations to create, delete, open, close, read, write, position, and interrogate both sequential and random access files and to alter the association between logical files and physical devices.

8C. Input Restrictions. User level input shall be restricted to data whose record representations are known to the translator (i.e., data that is created and written entirely within the program or data whose representation is explicitly specified in the program).

8D. Operating System Independence. The language shall not require the presence of an operating system. [Note that on many machines it will be necessary to provide runtime procedures to implement some features of the language.]

8E. Resource Control. There shall be a few low level operations to interrogate and control physical resources (e.g., memory or processors) that are managed (e.g., allocated or scheduled) by built-in features of the language.

8F. Formating. There shall be predefined operations to convert between the symbolic end Internal representation of all types that have literal forms in the language (e.g., strings of digits to integers, or an enumeration element to its symbolic form). These conversion operations shall have the same semantics as those specified for literals in programs.

9. Parallel Processing

9A. Parallel Processing. It shall be possible to define parallel processes. Processes (i.e., activation instances of such a definition) may be initiated at any point within the scope of the definition. Each process (activation) must have a name. It shall not be possible to exit the scope of a process name unless the process is terminated (or uninitiated).

9B. Parallel Process Implementation. The parallel processing facility shall be designed to minimize execution time and space. Processes shall have consistent semantics whether implemented on multicomputers, multiprocessors, or with interleaved execution on a single processor.

9C. Shared Variables and Mutual Exclusion. It shall be.possible to mark variables that are shared among parallel processes. An unmarked variable that is assigned on one path and used on another shall cause a warning. It shall be possible efficiently to perform mutual exclusion in programs. The language shall not require any use of mutual exclusion.

9D. Scheduling. The semantics of the built-in scheduling algorithm shall be first-in-first-out within priorities. A process may alter its own priority. If the language provides a default priority for new processes it shall be the priority of its initiating process. The built-in scheduling algorithm shall not require that simultaneously executed processes on different processors have the same priority. [Note that this rule gives maximum scheduling control to the user without loss of efficiency. Note also that priority specification does not impose a specific execution order among parallel paths and thus does not provide a means for mutual exclusion.]

9E. Real Time. It shall be possible to access a real time clock. There shall be translation time constants to convert between the implementation units and the program units for real time. On any control path, it shall be possible to delay until at least a specified time before continuing execution. A process may have an accessible clock giving the cumulative processing time (i.e., CPU time) for that process.

9G. Asynchronous Termination. It shall be possible to terminate another process. The terminated process may designate the sequence of statements it will execute in response to the induced termination.

9H. Passing Data. It shall be possible to pass data between processes that do not share variables. It shall be possible to delay such data transfers until both the sending and receiving processes have requested the transfer.

9I. Signalling. It shall be possible to set a signal (without waiting), and to wait for a signal (without delay, if it is already set). Setting a signal, that is not already set, shall cause exactly one waiting path to continue.

9J. Waiting. It shall be possible to wait for, determine, and act upon the first completed of several wait operations (including those used for data passing, signalling, and real time).

10. Exception Handling

10A. Exception Handling Facility. There shall be an exception handling mechanism for responding to unplanned error situations detected in declarations and statements during execution. The exception situations shall include errors detected by hardware, software errors detected during execution, error situations in built-in operations, and user defined exceptions. Exception identifiers shall have a scope. Exceptions should add to the execution time of programs only if they are raised.

l0B. Error Situations. The errors detectable during execution shall include exceeding the specified range of an array subscript, exceeding the specified range of a variable, exceeding the implemented range of a variable, attempting to access an uninitialized variable, attempting to access a field of a variant that is not present, requesting a resource (such as stack or heap storage) when an insufficient quantity remains, and failing to satisfy a program specified assertion. [Note that some are very expensive to detect unless aided by special hardware, and consequently their detection will often be suppressed (see 10G).]

10C. Raising Exceptions. There shall be an operation that raises an exception. Raising an exception shall cause transfer of control to the most local enclosing exception handler for that exception without completing execution of the current statement or declaration, but shall not of itself cause transfer out of a function, procedure, or process. Exceptions that are not handled within a function or procedure shall be raised again at the point of call in their callers. Exceptions that are not handled within a process shall terminate the process. Exceptions that can be raised by built-in operations shall be given in the language definition.

10D. Exception Handling. There shall be a control structure for discriminating among the exceptions that can occur in a specified statement sequence. The user may supply a single control path for all exceptions not otherwise mentioned in such a discrimination. It shall be possible to raise the exception that selected the current handler when exiting the handler.

10E. Order of Exceptions. The order in which exceptions in different parts of an expression are detected shall not be guaranteed by the language or by the translator.

10F. Assertions. It shall be possible to include assertions in programs. If an assertion is false when encountered during execution, it shall raise an exception. It shall also be possible to include assertions, such as the expected frequency for selection of a conditional path, that cannot be verified. [Note that assertions can be used to aid optimization and maintenance.]

10G. Suppressing Exceptions. It shall be possible during translation to suppress individually the execution time detection of exceptions within a given scope. The language shall not guarantee the integrity of the values produced when a suppressed exception occurs. [Note that suppression of an exception is not an assertion that the correspond-

ing error will not occur.]

11.Representation and Other Translation Time Facilities

11A. Data Representation. The language shall permit but not require programs to specify a single physical representation for the elements of a type. These specifications shall be separate from the logical descriptions. Physical representation shall include object representation of enumeration elements, order of fields, width of fields, presence of "don't care" fields, positions of' word boundaries, and object machine addresses. In particular, the facility shall be sufficient to specify the physical representation of any record whose format is determined by considerations that are entirely external to the program, translator, and language. The language and its translators shall not guarantee any particular choice for those aspects of physical representation that are unspecified by the program. It shall be possible to specify the association of physical resources (e.g., interrupts) to program elements (e.g., exceptions or signals).

11C. Translation Time Facilities. To aid conditional compilation, it shall be possible to interrogate properties that are known during translation including characteristics of the object configuration, of function and procedure calling environments, and of actual parameters. For example, it shall be possible to determine whether the caller has suppressed a given exception, the callers optimization criteria, whether an actual parameter is a translation time expression, the type of actual generic parameters, and the values of constraints characterizing the subtype of actual parameters.

11D. Object System Configuration. The object system configuration must be explicitly specified in each separately translated unit. Such specifications must include the object machine model, the operating system if present, peripheral equipment, and the device configuration, and may include special hardware options and memory size. The translator will use such specifications when generating object code. [Note that programs that depend on the specific characteristics of the object machine, may be made more portable by enclosing those portions in branches of conditionals on the object machine configuration.]

11E. Interface to Other Languages. There shall be a machine independent interface to other programming languages including assembly languages. Any program element that is referenced in both the source language program and foreign code must be identified in the interface. The source language of the foreign code must also be identified.

11F. Optimization. Programs may advise translators on the optimization criteria to be used in a scope. It shall be possible in programs to specify whether minimum translation costs or minimum execution costs are more important, and whether execution time or memory space is to be given preference. All such specifications shall be optional. Except for the amount of time and space required during execution, approximate values beyond the specified precision, the order in which exceptions are detected, and the occurrence of side effects within an expression, optimization shall not alter the semantics of correct programs, (e.g., the semantics of parameters will be unaffected by the choice between open and closed calls).

12.Translation and Library Facilities.

12A. Library. There shall be an easily accessible library of generic definitions and separately translated units. All predefined definitions shall be in the library. Library entries may include those used as input-output packages, common pools of shared declarations, application oriented software packages, encapsulations, and machine configuration specifications. The library shall be structured to allow entries to be associated with particular applications, projects, and users.

12B. Separately Translated Units. Separately translated units may be assembled into operational systems. It shall be possible for a separately translated unit to reference exported definitions of other units. All language imposed restrictions shall be enforced across such interfaces. Separate translation shall not change the semantics of a correct program.

12D. Generic Definitions. Functions, procedures, types, and encapsulations may have generic parameters. Generic parameters shall be instantiated during translation and shall be interpreted in the context of the instantiation. An actual generic parameter may be any defined identifier (including those for variables, functions, procedures, processes, and types) or the value of any expression.

13.Support for the Language

13A. Defining Documents. The language shall have a complete and unambiguous defining document. It should be possible to predict the possible actions of' any syntactically correct program from the language definition. The language documentation shall include the syntax, semantics, and appropriate examples of each built-in and predefined feature. A recommended set of translation diagnostic and warning messages shall be included in the language definition.

13B. Standards. There will be a standard definition of the language. Procedures will be established for standards control and for certification that translators meet the standard.

13C. Completeness of Implementations. Translators shall implement the standard definition. Every translator shall be able to process any syntactically correct program. Every feature that is available to the user shall be defined in the standard, in an accessible library, or in the source program.

13D. Translator Diagnostics. Translators shall be responsible for reporting errors that are detectable during translation and for optimizing object code. Translators shall be responsible for the integrity of object code in affected translation units when any separately translated unit is modified, and shall ensure that shared definitions have compatible representations in all translation units. Translators shall do full syntax and type checking, shall check that all language imposed restrictions are met, and should provide warnings where constructs will be dangerous or unusually expensive in execution and shall attempt to detect exceptions during translation. If the translator determines that a call on a routine will not terminate normally, the exception shall be reported as a translation error at the point of call.

13E. Translator Characteristics. Translators for the language will be written in the language and will be able to produce code for a variety of object machines. The machine independent parts of translators should be separate from code generators. Although it is desirable, translators need not be able to execute on every object machine. The internal characteristics of the translator (i.e., the translation method) shall not be specified by the language definition or standards.

13F. Restrictions on Translators. Translators shall fail to translate otherwise correct programs only when the program requires more resources during translation than are available on the host machine or when the program calls for resources that are unavailable in the specified object system configuration. Neither the language nor its translators shall impose arbitrary restrictions on language features. For example, they shall not impose restrictions on the number of array dimensions, on the number of identifiers, on the length of identifiers, or on the number of nested parentheses levels.

13G. Software Tools end Application Packages. The language should be designed to work in conjunction with a variety of useful software tools and application support

packages. These will be developed as early as possible and will include editors, interpreters, diagnostic aids, program analyzers, documentation aids, testing aids, software maintenance tools, optimizers, and application libraries. There will be a consistent user interface for these tools. Where practical software tools and aids will be written in the language. Support for the design, implementation, distribution, and maintenance of translators, software tools and aids, and application libraries will be provided independently of the individual projects that use them.