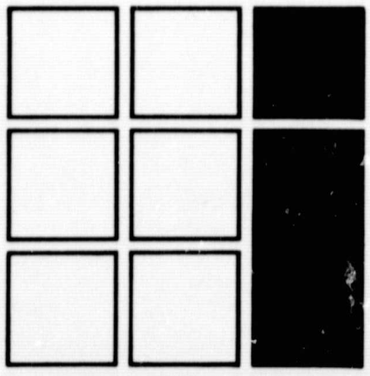


INTERMETRICS

Requirements Analysis for a Manned Spaceflight Programming Language and Compiler

MSC-01845

INTERMETRICS, INC.
380 Green Street
Cambridge, Mass. 02139

FOREWORD

This report was prepared by Intermetrics, Inc. in partial fulfillment of contract #NAS-9-10542 from the Manned Spacecraft Center of the National Aeronautics and Space Administration. The Technical Monitor of this contract is Mr. Jack Williams/FS5

The publication of this report does not constitute approval by the National Aeronautics and Space Administration of the findings or the conclusions contained therein. It is published only for the exchange and stimulation of ideas.

Table of Contents

List of Figures

1.0 INTRODUCTION

This report contains the preliminary results of the Language Requirements Analysis task as part of contract NAS 9-10542. It contains a description of the functional, general, and specific requirements for a manned spaceflight programming language and compiler. It is not a specification of the language or compiler, but it does contain a description of the principal needs of and important considerations for manned spaceflight software development and application. Subject to review, these requirements will be used as the framework for the design and specification of the language and compiler to be developed during this contract.

1.1 Organization of the Report

The report is organized into five sections. Section 2, Functional Requirements, describes the scope and orientation of the language, lists the manned spaceflight software functional areas and associates each with implied language characteristics.

Section 3, General Requirements, discusses some of the key problems in developing software for manned spaceflight projects. These problems include Software Reliability, Communication and Documentation, Fixed Point Arithmetic and Code Optimization, Software Management and Control, and Application Language. There follows a list of general goals and guidelines for the language and compiler design.

Section 4, Specific Requirements, contains a discussion of the specific features required in the language such as form, data and description, operations and executable statements. The list is not exhaustive, but covers some of the basic features of the language design.

Section 5, Preliminary Compiler Requirements, contains a description of the important technical features and capabilities required in the compiler, such as diagnostics, documentation and output, environmental considerations, debugging aids, macros, and optimization.

1.2 Summary of Requirements

The following is a summary list of language and compiler requirements developed as a result of an analysis of manned spaceflight programming requirements.

1.2.1 GENERAL REQUIREMENTS

1.2.1.1 The principal application of the language is for the development of manned spaceflight computer software for the 1972-1980 period and this includes shuttle and space station applications. (Initial orientation will be towards the shuttle system.)

1.2.1.2 Software applications should include:

a. Navigation, guidance, targeting and general mission programming.

b. Vehicle control and stabilization.

c. Operating Systems.

d. Data Management.

e. Communications and displays.

f. Compiler and support software.

1.2.1.3 The language and compiler should be designed for a wide range of flight computer systems and should be capable of supporting simplex configurations as well as advanced multi-computer and multi-processor computer systems.

1.2.1.4 The language should be machine-independent with a minimum of exceptions restricted to clearly identified areas.

1.2.1.5 The language and compiler must contain specific features to aid in achieving high software reliability. The design shall:

a. Strive toward clarity and readability in the language.

b. Enforce programming standards and conventions.

c. Perform extensive automatic checking.

1.2.1.6 As general goals, the language should, in descending order of importance:

a. Enable the programming of software for a wide variety of manned spaceflight applications.

b. Be easy to read and understand.

c. Be easy to debug.

d. Be easy to modify.

e. Be easy to use.

f. Be easy to learn.

g. Be easy to transfer to another computer.

h. Enable the enforcement of standards and conventions.

1.2.1.7 The output format of the language should strive toward presenting data types, attributes and operations in an unambiguous way. An equation will look like an equation. A character string (or text) will be easily differentiated from a vector, or array. The compiler will annotate output listing.

1.2.1.8 Language should be oriented toward a general class of technical personnel involved in manned spaceflight projects, not solely highly trained programmers.

1.2.2 Specific Requirements-Language.

1.2.2.1 To enhance readability, the language should possess distinct names and labels.

1.2.2.2 The language should possess the following data types: integers, fixed/floating point scalars, vectors, matrices, booleans, bit and character strings, status variables, and labels.

1.2.2.3 The language should possess the following data organizations: arrays of similar data types, "collections" of different data types (e.g., structures).

1.2.2.4 The language should possess at least the following data attributes: precision, dimension, initialization, global variable lock and unlock, and static and automatic storage.

1.2.2.5 The language should possess a complete set of scalar and matrix-vector arithmetic operations.

1.2.2.6 Boolean operations in the language should be the logical AND, OR and NOT operators and should include a convenient method of setting, resetting and inverting booleans. The language should possess, as a minimum, the following set of relational operators:

a. equal

b. not equal

c. less than ,and less than-or equal

d. greater than ,and greater than-or equal

INTERMETRICS INCORPORATED • 380 GREEN STREET • CAMBRIDGE, MASSACHUSETTS 02139 • (617) 868-1840

1.2.2.7 The language will possess a flexible set of conditional and unconditional program transfer instructions, and data calls.

1.2.2.8 The language syntax must be accomplishable with a defined common character set. This common set is:

A - Z

0 - 9

+ - =

" ' . , ; : ! ?

< > ( ) / _

* # $ % & @

plus blank or space. The compiler will not reject an expanded character set; e.g. ^, ~, [ ], etc., where the expanded set provides convenient alternate forms when available.

1.2.2.9 The language will possess the capability of dealing with I/O operations, conditional error procedures, and real-time tasking. These may not be an integral part of the language syntax.

1.2.2.10 The language will include iteration statements nested to any level.

1.2.2.11 Basic machine language coding will not be permitted everywhere but will be restricted to clearly identified areas such as special subroutines.

1.2.2.12 Simple replacement type macros will be provided by the compiler.

1.2.2.13 The language should allow definition of global and local data which can be applied to independent program sub-sections.

1.2.2.14 The language should provide for data sharing indicators and control of global information for real-time use among program sub-sections.

1.2.2.15 The language will be designed for a 2-D input stream. (An optional 1-dimension input will also be provided.)

1.2.2.16 For character strings, the language will possess the string operator CONCATENATE and a form of deconcatenation. For bit strings the language will possess the logical AND, OR,

NOT, and a method of shifting in addition to the string operator CONCATENATE and a form of deconcatenation.

1.2.2.17 The language will not provide for complex number arithmetic or data declarations.

1.2.2.18 Language will not include any specific code optimization directives.

1.2.3 Specific Requirements-Compiler.

1.2.3.1 The compiler should allow independent compilation of sub-sections of the total program.

1.2.3.2 The compiler will possess a full library of mathematical functions.

1.2.3.3 A system to handle a collection of shared data in an orderly fashion is required.

1.2.3.4 The language will not provide for extensive or ambiguous mixed data-type operations.

1.2.3.5 The compiler should provide execution checking with reference to indexed data organizations; e.g. arrays.

1.2.3.6 The compiler will annotate the output listing to increase readability.

1.2.3.7 The output listing of the language will be in 2-D format.

1.2.3.8 The compiler will produce reentrant code when needed, but recursive code is not required.

1.2.3.9 The output character set will not be restricted. In order to achieve a maximum of self-expression the compiler will be capable of utilizing the full character set of the output device.

INTERMETRICS INCORPORATED • 380 GREEN STREET • CAMBRIDGE, MASSACHUSETTS 02139 • (617) 868-1840

2.0 FUNCTIONAL REQUIREMENTS

2.1 Scope of the Language

Past experience has shown that developing software for manned space projects such as Apollo is a task of major proportions. Heavy penalties in cost and time have been paid for underestimating the manpower and time necessary to produce effective, qualified, and documented software. The problems of design, control, and management of software have not been easy to determine; techniques and procedures to cope with them have been slow to evolve. The application of a higher order programming language is an essential step toward achieving a more orderly and controlled software production effort. It is the task of this section to define the functional requirements and aims for the programming language. These must reflect the characteristics of manned space flight software development and the major programs for which the language will be used.

The current effort involves developing a language and compiler with a code generator for the IBM 360/75 by mid-1971. Presently planned manned spaceflight projects for which the language will be designed are the Space Shuttle Program and the Earth Orbital Space Base. Although these are the two major programs which will dictate the requirements of the language, the capabilities of the language and compiler are expected to satisfy other manned and unmanned aerospace programming needs which may occur during this period.

A large manned spacecraft project is comprised of many software activities, including: a) software for the onboard computer system, b) the analysis, simulation and test programs necessary to develop this onboard software, c) software for ground-based systems for in-flight mission control and support, d) mission planning and analysis software, e) simulation and flight training software, f) post-flight mission data reduction and analysis software. The language requirements for all of these cover a broad spectrum. The emphasis in this contract will be to formulate a language principally oriented at the development and maintenance of reliable software for onboard applications, and ground-based real time control and support. Although designed for space application, the language should also be well suited for general aerospace engineering problems, and directly applicable to mission planning and analysis.

It should be noted that the size and complexity of potential flight computer systems which are targets for the language could vary considerably from a simplified candidate for the space shuttle to more sophisticated and complex systems for the space

base. The language should be designed with sufficient features to enable it to be "scaled" up or down according to the complexity of the system without having to develop a completely new language for each level of complexity. Therefore, the initial language design should encompass a broad spectrum of objectives, yet be applicable to the possibly limited requirements of the near future.

2.2 Flight Computer Software

Manned spaceflight computers, thus far, have been special purpose machines performing tasks principally in guidance, navigation, control and pilot displays. The computer has been provided with a restricted instruction set, small working memories with no secondary storage capability, and established interfaces to a limited number of output devices or special-purpose displays. For the most part, programming has been accomplished in basic machine language. Although a number of higher-order programming languages has been developed, none has been applied to manned spaceflight computer software development.

Future programs such as the shuttle and space station, will require more complex software within the flight computer. The functional processing requirements for the onboard system will increase in scope. In addition to performing guidance, navigation, and control, the computer will handle centralized data management functions while responding to requests from a number of general-purpose display and control units. Other functions will include in-flight monitoring, flight planning and management, the control and collection of data from a number of experiment sensors, and in the case of the space base, provision for a modest amount of onboard software development. The advanced spaceborne computer will perform functions common to a large ground-based data processing facility, providing many diverse computational services, and will involve extensive man-machine interfacing.

Evolving flight computer system hardware will also impact software design. Distributed multicomputers as well as large centralized multiprocessing systems are being proposed as candidate flight computer systems. These systems attempt to provide high reliability and flexibility as well as increased computational power. They portend a more complex environment for the software involving problems of resource conflicts, data protection, error detection and recovery, and parallel processing.

2.3 Functional Requirements

A detailed list follows of the software functions to be performed in a flight computer for future manned space missions. It is representative of the functional areas for software application.

a. Executive and operating systems (e.g. resource allocator, scheduler, dispatcher, etc.)

   - System Monitor
   - System Service Routines (e.g. I/O)
   - Error Detection and Recovery

b. Display processing and pilot interface.

c. Communications (up/down telemetry).

d. Stabilization and control (autopilots)

e. Guidance and steering.

f. Navigation and position determination.

g. Subsystem control and monitoring (radars, power, target tracking, etc.).

h. Onboard checkout.

i. Data management.

j. Scientific experiment maintenance, control and data collection.

k. Configuration and sequencing control.

l. Utility and support software (compilers, simulators, test and diagnostic aids).

m. Mission programs such as targeting , docking, rendezvous, mission planning, etc.

Seven comprehensive functional categories have been selected as requirements and these appear in the table below (with comments). The intention here is to present the language/compiler implications associated with each function.

INTERMETRICS INCORPORATED • 380 GREEN STREET • CAMBRIDGE, MASSACHUSETTS 02139 • (617) 868-1840

Table 2-1 Functional Requirements

Software Functions	Description/Comments	Predominant Language/Compiler Characteristics
1. Navigation, Guidance, Targeting, General Mission Programs	1. Orbit determination of position and velocity using sensor measurements. 2. Outer loop steering commands. 3. Maneuver calculation & trajectory control. 4. Position and velocity extrapolation. 5. Abort trajectory calculations. 6. Radar tracking. 7. Boost and entry aerodynamic control.	1. Highly mathematically oriented - extensive use of vector and matrix algebra. 2. Wide dynamic range of numeric data variables (i.e. accuracy, precision needed). 3. Boolean variable and flags for logical decisions (e.g. hardware, mission and system status checks). 4. Real Time Control. • Time critical computations • Multi-programming • Precise event timing • Task synchronization
2. Control and Stabilization	1. Attitude control 2. Accelerated flight 3. Balance control 4. Automatic landing	Same as above with somewhat more emphasis on boolean logical decisions.

Table 2-1 Functional Requirements (cont.)

Software Functions	Description/Comments	Predominant Language/Compiler Characteristics
3. Operating Systems	1. Executive control functions: scheduling, resource allocation, dispatching, and queue processing. 2. Supervisory functions: • I/O control • Interrupt processing • Central service routines	1. Simple scalar arithmetic, no vectors or matrices. 2. Boolean variables and logical decision making. 3. Lists and pointers. 4. Arrays and tables of data. 5. Hardware interfaces. 6. Character processing for text, names and messages. 7. Relational operators for all data comparisons. 8. Storage allocation. 9. I/O statements. 10. Interlocking of data storage.
4. On-board Checkout and System Monitoring	1. Status monitoring of subsystem hardware. 2. On- and off-line diagnostics.	1. Simple mathematical operations. 2. Hardware interfacing. 3. Bit manipulation for complex logical decisions.

Table 2-1 Functional Requirements (cont.)

Software Functions	Description/Comments	Predominant Language/Compiler Characteristics
4. Onboard Checking and System Monitoring (continued)	3. Fault predictions, e.g. trend analysis. 4. Subsystem testing. 5. Reconfiguration control.	4. Data comparisons, relational operators. 5. Interfaces with machine language code.
5. Data Management	1. General file handling for stored data. · Retrieval · Updating · Reformating · Deleting · General file maintenance 2. Processing of large volumes of measurement data.	1. Hierarchical organizations of data, lists and arrays of data. 2. Mathematical reduction & processing of data. 3. Pointers - file directories 4. Test and character handling. 5. Storage allocation.
6. Communications & Display	1. Displays · Terminals, keyboard, and console support programs	1. Bit manipulation. 2. Character strings and operations. 3. Logical operations.

Table 2-1 Functional Requirements (cont.)

Software Functions	Description/Comments	Predominant Language/Compiler Characteristics
6. Communications & Display (continued)	2. Communications. • Message processing, formating, etc. • Uplink, downlink • Telemetry processing 3. CRT's and graphics. 4. Special pilot displays.	4. Arrays and data tables. 5. Hardware I/O.
7. Compiler & Utility Support Software	1. Compilers, assemblers, and diagnostic packages.	1. Arrays and lists of data. 2. Character and text processing. 3. Boolean variables and operations. 4. Relational operators for comparisons. 5. Input/output control.

NOTE:

The functions above refer to flight software as well as ground support and analysis, including simulation. For example, design and development of guidance techniques require many of the same language characteristics as the final flight software product.

3.0 GENERAL REQUIREMENTS

Experience has shown that approximately 5 or more years are required to design, develop, and verify software for a project of the magnitude of the Space Shuttle or Space Base systems. It is evident that the management of such an effort will require considerable insight, visibility and control to assure timely production of reliable software. System planners must be careful to employ the preventive measures necessary to avoid the pitfalls of poor design and implementation.

Accordingly, the programming language and compiler will serve as tools to assist in the program preparation. The following sections present a description of some of the key problems in developing manned spaceflight software, which have influence on general requirements of the programming language and compiler.

3.1 Software Reliability and Verification.

3.1.1 Discussion of Problem. A unique characteristic in the development of flight computer software is the required degree of reliability and the amount of validation necessary to achieve that reliability. The language and compiler should contain specific features to aid in this development.

Apollo software demanded the highest pre-flight confidence, and because neither the computer, the language, nor the programming techniques were designed with checkout in mind, verification was accomplished only through the philosophy of laboriously defining and carrying out tests for every logical path, every logical state. For any complicated program this task is essentially impossible and the approach reduces to "the more tests, the more confidence". All too often, success criteria are subjective and inconclusive. What is needed is the development of a technique that ensures more deterministic behavior of the software in its operational environment within the flight computer. The technique should attack specifically the problem of minimizing the test effort involved in the integration of individual program modules. Checkout of individual modules is not as significant a problem as groups of modules in a system or mission sequence. The objective should be to specify a finite number of tests for each function, and to achieve a definable level of confidence once all tests are successful. The technique should be aimed at limiting the possible number of "states" of each software element by ensuring that its interaction with the environment is predetermined, well defined and bounded in some sense. To a large extent this can be met by the incorporation of features into the compiler or language which ensures "reliability" of the

software and lessens the amount of testing.

3.1.2 Methods of Achieving Software Reliability. The primary language and compiler contributions toward achieving software reliability are three:

3.1.2.1 The language can make it easier for the programmer to express what he wants done, thereby reducing the likelihood of errors at the local level. (Local level refers to the operation of a program section without consideration of the environment in which it actually functions. A programmer should not be required to comprehend fully the environment in which his code will operate. This environment can be extraordinarily complex.)

3.1.2.2 The language and its implementation can enforce programming standards. An important example of such a convention relates to the orderly sharing of common data between processes.

3.1.2.3 The compiler implementation can isolate the effects of program errors to the set of programs logically involved, and can detect certain instances where program behavior deviates from the apparent intention of the author.

3.1.3 Reliability in Data Transfers. In the following subsections, requirements imposed by the necessities of global-data sharing and dynamic error detection will be discussed.

3.1.3.1 Data Sharing. There is a clear trend in information processing systems to pool resources and data to achieve both economy and accuracy in operation. This centralization creates the need for an infallible technique to control processes desiring simultaneous access to the same data. Further, it is also necessary to establish control over which processes have access at all.

Consider the problem of data security. It seems highly desirable to define sets or groupings of data and a mechanism to authorize access. The author of a program requiring access to data in a given group (designated "glodap" for global data pool) would specify the glodap name to the compiler, which would, by reference to the glodap definition during program compilation, verify that the "key" of the program fits the "permission-lock" of the data element. If not, an error message would be generated, and access would not be granted. Identity codes would be required of each program or application in order to retrieve global data.

In some cases, it might be necessary to allow modification of access-permission attributes of glodap contents at execution time.

To do this, an attribute would be affixed to the data elements involved, which would cause the compiler to generate code at run-time in order to obtain access permission. Refusal of permission would be treated like other program faults. The glodap thus appears to take the form of a data set whose function is to describe other data sets.

Another potential glodap data attribute is one which provides the means of controlling data-interlock conventions between processes. Types of sharing which would be regulated include: uncontrolled data, uncontrolled data-reading with exclusive access writing; access for only one user at a time, etc. Again, this information would be utilized by the compiler to generate appropriate code to ensure that interlock requirements are observed.

It must not be overlooked that this technique offers a potentially serious hazard to the operation of the system. This hazard is the "deadlock problem". Deadlock can arise when program 1 has locked variable A and wants now to gain access to variable B. This variable, however, is presently locked by program 2, which now wants access to A. Neither program can progress; the only way to break the jam is to terminate one of the programs.

This prospect, of course, is not only unsatisfactory, but requires solution of the problem, which can be very difficult when the deadlock involves a larger group of dependent programs. The avoidance of this hazard requires a systematic resource-requesting scheme. One suitable technique is for the total required resources to be requested at once. If all resources are available, the program is guaranteed to be able to finish. If not all are available, none are granted; this prevents a stalled program from owning a resource, the lack of which may stall some other program.

It is not at all clear that the implementation of such a technique can be direct and uncomplicated, nor is it clear that a non-hazardous alternative exists.

3.1.3.2 Execution Checking. Although the language will contain features which facilitate program checkout, some program bugs will still survive. The discussion in this section is oriented primarily towards a single kind of program error - invalid indexing. Use of out-of-bounds subscripts on arrays can cause addressing of other variables than the program listing indicates, because of their adjacent storage locations in memory, unlike the attempt to take the square root of a negative number, which compels an alternate path to be followed. When a group of programs or processes are operating in the same system, whether cooperatively or independently, it seems crucial to insure that a program error does not affect the entire system. The spurious subscript

is a most serious problem, but is one which lends itself to automatic detection.

The concept proposed is the obvious one; namely, the enforcement by the compiler of the declared intention of the programmer. Specifically, it is simple to cause the compiler to generate code which will check whether a subscript would or would not cause the value referenced to be a member of the named array. If it does not, a program fault condition can be raised, and the reference suppressed. The implementation of this requires a limit check every time the address of a variable-subscripted array member is computed. It is clear there will be an overhead cost of additional memory for the code to perform these checks. However, it may be necessary. Several weaker alternatives are possible which have lower overhead; for example, read-accesses could be unchecked, write-accesses could be limited to the program's own data area rather than to the named array, or subscript checking could be specified by the programmer on a statement, block, or program level. Whether to introduce subscript checking at all is a matter which must be resolved.

Another point of view may be taken where language features are either very difficult or impossible to check. An attempt may be made to limit the number of users, by forcing use of a code (a number for privileged users), where the risk of undetected errors is unusually high. (For example, the list-processing elements of PL/I: based data and pointers.)

3.2 Communications and Software Documentation

3.2.1 Discussion of the Problem. In a large programming effort many individuals, some representing quite different activities, are required to communicate with each other, and each does so in a way that has meaning to himself. The manager attempts to specify his requirement; the engineer attempts to describe his design, qualitatively to the manager and quantitatively to the programmer; the programmer is plagued with documenting the code to a level that will be useful for descriptive material, debugging activities, and a quick reference, as the authority for 'what is really going on in the machine'. Clearly, all contributors want and have a need to know at least some of the intricacies of the implementation. The problem here is language. Each group has only a limited comprehension of the others' mode of expression. The engineer designs and expresses his algorithms using conventional mathematics, or perhaps FORTRAN-like statements. The programmer must take this specification and translate it into his language: traditionally a basic assembly language appropriate to the particular computer. The programmer must then explain his efforts to the user in the field, by using other media. He might have

charts drawn to describe detailed functional flow. He might use word statements or user-guides, or other apparently helpful devices.

In many projects the coding language isolates the programmers from everyone else associated with the effort. The programmer becomes too busy to learn the physics and objectives of the mission and is too busy to explain to others how the code works. He, therefore, is forced to assume an increasing share of the total responsibility. Small indispensable groups of experts direct and shape the code and become the overworked "authorities".

3.2.2 Language Orientation. A properly designed programming language will prove to be a useful analytical tool for the designer, a convenient and useful program tool for the programmers and will provide a medium for communication and documentation for technical management. It will assist in bridging the communications gap. The language will be oriented toward a general class of technical personnel in the manned spaceflight project concerned with software. Users are presumed to have technical backgrounds and some familiarity with aerospace problems. The language will not be oriented solely at the highly experienced flight computer programmer, solely at the novice or non-scientific programmer, nor solely at management personnel.

It will be a specific objective of the language to promote the ability of the user quickly and easily to learn to read, write, understand, and "think in" this language, and to document his results in a clear and unambiguous manner.

3.3 Aerospace Higher Order Languages.

Recently designed higher order programming languages have overemphasized the need for fixed point arithmetic and code optimization. These languages are oriented at current and past aerospace computing problems. The language to be applied to future manned space applications should be influenced by the now developing hardware technology and should not be constrained by previous environment. A discussion of these past problems and trends for the future is presented below.

3.3.1 Fixed Point Arithmetic. Most aerospace computers have had no floating point hardware. Consequently, the coding of guidance, navigation and other mathematical formulae was accomplished using fixed point arithmetic. This involved the scaling of the computations and performing the necessary shifting and other operations within the execution of an equation in order to maintain

accuracy. The question to address is, to what extent should the language provide fixed point arithmetic capabilities? Although fixed point computers will exist for some time to come, it is Intermetrics' opinion that floating point hardware will be mandatory for computers used in the next generation of manned space systems, if only from a cost-effectiveness point of view. Floating point is available (or optional) in many flight designed computer systems today such as the UNIVAC 1832 or IBM 4 Pi EP. Therefore, assuming that the onboard computer system will have floating hardware, what are the requirements for fixed point? That is, why would fixed point be used if floating point were available?

Integer variables, constants, and integer-arithmetic are useful for memory address computations, indexing, counting and simple equations. The requirement for scaled fixed point arithmetic when floating point hardware is available, however, is not apparent. Generally, all navigation, guidance and other mathematical equations will be implemented using floating point. Some reasons often given for not employing floating point are:

a. Fixed point will achieve shorter running times. Several routines in the flight computer such as control autopilots or other closed loop control functions are time-critical. Although it is true that processor execution times for floating point instructions are slower than fixed point, they are generally not much slower. They are approximately twice as slow for adds and subtracts, and less for multiplies and divides. The actual execution time for the floating point instructions is fast and in most cases faster than the memory cycle times of older computers such as the Apollo Guidance Computer (AGC). As an example, typical execution times are given below for the UNIVAC AN/UYK-7(V) (similar to the 1832) and CDC ALPHA multiprocessor systems.

Table 3-1. Execution Timed in Microseconds

	AN/UYK-7(V)	CDC ALPHA
Fixed Add	3.0	2
Fixed Multiply	7.5	-
Fixed Divide	13.5	-
Floating Add	7	3
Floating Multiply	10	9.7
Floating Divide	16	17

As can be seen from the comparisons, the times required to perform floating point arithmetic will not be a burden to the system. This seems particularly valid in view of the infrequency of occurrence of floating point instructions.

b. Data received from other equipment subsystems will be received in a fixed point format; also, certain stored data may contain elements retained in fixed form. If only floating point were used in computation then conversions would be required. However, the amount of additional code may be less overall than the code required to mask and shift corresponding fixed point versions of the problem. Again, in general, this does not appear to be a valid reason for including fixed point arithmetic.

In summary, the need for fixed point arithmetic in general purpose aerospace computers with floating point hardware will decrease. Since the requirement for fixed point depends on the characteristics of the target computer, fixed point capability will remain a language requirement but will not be fully implemented in the initial version of the compiler. The language will be designed to include the necessary data declaration and precision statements, but the initial implementation of the compiler on the IBM 360/75 will only implement fixed point to the extent that it can be modified to include code generation for fixed point statements when a particular fixed point target computer has been identified and a code generator development planned. Integer arithmetic will be included in the initial design.

3.3.2 Code Optimization. Code optimization has received considerable attention in higher order languages. It is felt that the need for code optimization will lessen in the future. While it is true that in the past, flight programs have grown larger than the available space and unusual effort has been necessary to accommodate all programs, current opinion is that the main-memory and secondary storage structure of computers of the future will introduce a whole new set of problems such as paging, segmentation, and file handling. Thus, the problem of code reduction will be relegated to less than prime importance, and will be of real concern only for the mission phase with the greatest demand for memory capacity. Flight computer memory sizes are getting larger and secondary mass memory devices are being considered for bulk program storage. From a cost effective point of view, one wonders about the trade-off between more memory and faster processors versus the cost of testing and verification of software containing tricky coding shortcuts for a smaller, less expensive machine. The conclusion seems clear that it is cheaper in the long run to provide extensive and sophisticated hardware in the development of a manned space system. Reliability, the handling

of great quantities of program and data, and the structure of storage hierarchies will become the foremost problem areas.

Of course, the language design should not ignore implementation difficulties, nor should the compiler produce inefficient code. But this consideration is fundamentally different from the inclusion in the language of optimization controls. The structure of the language itself should not be modified or compromised in order to make efficient code easier to generate.

3.4 Software Management and Control

The technical management of software for the space shuttle or station faces problems of visibility and control. Continual design changes, short production times, and pressing operational schedules require flexibility in software design and organization. Clearly, an overall management and control plan for manned spaceflight software is required which will define the procedures for developing software design requirements, interface specifications, documentation requirements, testing requirements, change procedures and organizational responsibility. The directed use of a higher order language and the control over the language should be part of this plan.

Accordingly, the language should provide features which support the software production environment in general. It should be self-documenting to a maximum extent, provide ease in program modification, and provide a mechanism for enforcement of management rules and programmer conventions.

3.5 Flight Computer Application Languages

A manned spaceflight computer system must provide a technique for communication between crew and computer. The crew must be able to initiate and control jobs, request information, reconfigure equipment, diagnose problems, and,in general, be able to communicate with the computer for a variety of purposes. Correspondingly, the computer must be able to communicate with the crew to request a decision, ask for data, alert when problems occur, etc. This is an example of a "user-language" as opposed to a programming language. The higher order language being developed in the scope of this work is a programming language used by analysts, engineers, and programmers to code flight computer programs. This programming language will be used to develop the translators for user languages and the programs which will perform requested functions. It may be that some of these user languages are merely extensions to the basic

programming language. Even so, modifications by the compiler writing staff will still be necessary since it is not felt that the language should be programmer-extendable at the instruction level. The compiler will make it easy to extend or modify the language capabilities.

3.6 General Goals and Objectives

The preceding sections have discussed some of the major problems in manned aerospace programming. The language and compiler will provide assistance in the solution of these problems. The following general goals and objectives are defined for the language and compiler.

3.6.1 Easy to Read. A programming language must be lucid. It should be easy to read and to understand the writer's intent. A programmed equation should look like an equation, and it should possess self-defining characteristics; for example, whether a variable is a scalar, vector, or a matrix should be evident.

3.6.2 Easy to Use. Programming can be accomplished in less time with fewer programmers if the language is easy to use. The language should offer the programmer power, flexibility, and a closeness to his natural style that most augments his approach to the problem. Errors are easily induced if the programmer is forced to circumvent awkwardness of the language in expressing his problem.

3.6.3 Easy to Learn. It is not obvious that ease of learning differs from ease of use. The complex tool may be quite easy to use, once it is thoroughly learned. But if the language is easy to learn, it will accelerate the training of programmers and attract a broader field of users. At least, we feel that a complicated tool should have a simplified mode of operation that can be learned rapidly.

3.6.4 Easy to Modify. Recompilation by procedure or blocks must be possible, so that modifications are localized and their effects contained in the immediate area.

3.6.5 Easy to Debug. The combination of language and compiler must be designed so that most types of program errors are evident to the programmer and therefore less likely to be committed. However, for those errors which are made, the compiler should produce comprehensive diagnostic information and explicit indication of the error in terms of the source language. When no compile-time errors are found but the program does not function properly, both the language and the run time system should provide the programmer with convenient and powerful means quickly to diagnose the fault.

3.6.6 Program Transferability. The language should be general enough to use for ground systems as well as on-board systems. Further, it should not include machine-dependent features. Programs checked-out on one computer should be transferable to another computer and be expected to perform in an equivalent manner.

3.6.7 Software Reliability. The compiler could assist in producing reliable code as discussed in Section 3.1.2.

3.6.8 Conventions. The compiler should promote enforcement of programming conventions and rules established by management.

4.0 SPECIFIC REQUIREMENTS

The previous functional and general requirements must be implemented by a set of specific language features including language form, data and description, operations, basic commands, executable statements and program structure. The intention of this section is to present a partial list of specific language requirements. At this writing the list is not exhaustive, nor does it constitute a design specification.

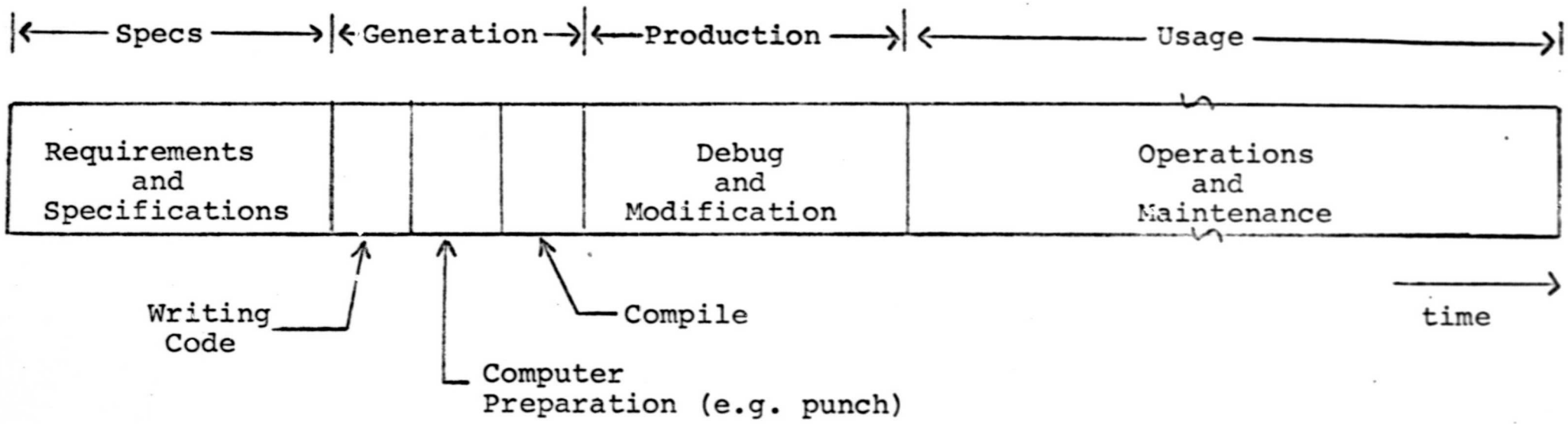
4.1 Appearance of the Language

4.1.1 Format. The format of a space programming language; i.e. its source input and printed appearance, should be designed to achieve maximum readability, ease in transfer of knowledge and understanding, and it should provide a basis for program documentation. Most existing higher order programming languages strive toward these goals as secondary objectives,with program composition as the first priority. Certainly any new programming language must be easy to use but composition is only the "front-end" of a long process to develop reliable space software. In perspective, stronger emphasis must be placed on software control techniques and accompanying documentation.

Figure 4-1 illustrates the development and operation cycle of software. The important point is that software, once generated, exists for a far more significant period of time during its operational life. Consequently, the language should emphasize readability and clarity for maintaining the software rather than emphasizing ease in program preparation.

The printed appearance of a language greatly influences its usefulness as a communications medium. Most of today's higher order languages, being fundamentally compatible with card punches or data terminals, are written in single line format. The wide appeal of FORTRAN is testimony that the single line format can be popular, perhaps universal, and that programmers or other persons with a need to know, find great utility in its applications to scientific programming. But the single line is a constraint and requires special symbols and conventions in order to provide the spectrum of natural mathematical functions. For example, in FORTRAN, exponentiation requires a double asterisk and a matrix or vector must be represented by its subscripted components, the subscripts being contained within parenthesis - all on the single line. While any language can become familiar, a preponderence of special rules and symbols reduces the effectiveness of the language for control, specification and documentation.

Figure 4-1 Chronology of Software Development



Some Observations:

1. The writing of code is closely tied to the specifications.
2. The time required for computer preparation is small compared to the program life.
3. A lengthy period of debug and modification must be provided.
4. Period of program usage extends many times that of program generation.
5. Many more people will use a program than generated it.

INTERMETRICS INCORPORATED • 380 GREEN STREET • CAMBRIDGE, MASSACHUSETTS 02139 • (617) 868-1840

A multi-line, or two-dimensional, format can bring a programming language closer to being natural in expression and in mathematical form. The MAC language of MIT's Draper Laboratory is an excellent example. Equations, whether scalar or vector-matrix, look like equations and promote understanding among programmers and technical managers. Subscript and superscript lines, in which exponents and subscripts appear in standard mathematical notation, provide an opportunity to make data forms self-evident. For example

$$d\bar{v}/dt = -\bar{g} + K_3^2 \overset{*}{M} \bar{c}$$

is obviously a differential equation involving the vector data types $\bar{v}$, $\bar{g}$, $\bar{c}$; the matrix data type M; and the square of the subscripted scalar K_3.

The appeal of the two-dimensional format is not simply esthetic; the conventional mathematical notation for input and output will be decisive factors in promoting software reliability. Coded operations will become visible to the managers and supervisors who have ultimate project responsibility. The language can provide the basis of communication for a broad spectrum of contributing engineers, scientists and technicians. For these reasons a requirement for a two-dimensional language format will be placed on the advanced manned space programming language. The following items are also considered to be part of this requirement:

a. The language must not require the design and production of a special terminal device.

b. The language must provide for an optional single-line input stream for present day data terminal usage.

c. The language format must consist of at least three lines (More lines may find applicability in the future.)

It should be noted that future technology may bring optical recognition devices which could simplify the task of generating computer input.

4.1.2 Character Sets. In conjunction with the determination of language format, the necessary character set must be evaluated, since together they provide the full notation of the language. The character set influences punctuation, naming-rules and choice

of operators. Character sets are obviously constrained by available hardware. There are two basic coding schemes in use on computer equipment today: EBCDIC and ASCII. EBCDIC (Extended Binary Coded Decimal Interchange Code) is used primarily in IBM equipment; ASCII (American Standard Code for Information Interchange) is being used in most other computer and communications systems, and on most available terminal devices. Figure 4-2 presents the characters in both sets and relates them to some of the common input and output devices. Note that the first six lines of characters form a common set. In order to make maximum use of existing equipment this set will form the lower bound on the totality of acceptable characters for an advanced space programming language. This will mean that all language syntax, input stream and printed output must be accomplished with this common set. This statement may be modified with respect to printed output to the extent that printers usually provide more standard characters than do input devices. It must be emphasized that the language may utilize characters outside the common set for any purpose, e.g. syntax, but that alternate forms must be available within the set.

The compiler may specifically use characters outside the set for improving readability of the output (e.g. lower case letters, brackets). The mapping among the defined input set, the set for compiler usage and the available output set will be the subject of design.

4.2 Data and Its Description

4.2.1 Data Types. The scope of allowable data types is an important aspect of language design. Data types are the representative forms of information which are processed by computer programs. They include for example: numerical quantities (scalars, vectors, etc.), strings of characters, boolean variables, and collections or combinations of data. The required data types depend to a large extent on the intended language applicability, whether scientific, text, business, etc. Furthermore, data definitions have a direct implication on the necessary operational features of a language.

Figure 4-3 presents the results of an analysis to determine the relative importance of a number of data types with respect to the functional requirements for manned spaceflight software. The requirements were first assigned figures of merit in the following way: for each requirement the percentages of code were estimated for a complex manned space mission. The documenting aspect of the language is more important than code composition. The number of users reflects the extent of problems in control, communication and organization and suggests the need for compre-

Listing of Characters	Codes			Teletype Terminals		Key Punch	Prin-ter	IBM Terminal
	6 Bit ASCII	7 Bit ASCII	8 Bit EBCDIC	KSR33	KSR37	IBM 029	IBM 1403N1-TN	IBM 2741
A - Z	X	X	X	X	X	X	X	X
0 - 9	X	X	X	X	X	X	X	X
+ - =	X	X	X	X	X	X	X	X
" ' . , ; : ! ?	X	X	X	X	X	X	X	X
< > ( ) / _	X	X	X	X	X	X	X	X
* # $ % & @	X	X	X	X	X	X	X	X
Space	X	X	X	X	X	X	X	X
^ [ ] \	X	X		X	X			
a - z		X	X		X		X	X
` ~ { } ¦		X			X			
¢ ¬ |			X			X	X	X
Blank						X		
□ ┐ ┌ ┘ └ ▯ ≥ ≤ +							X	
≠ ± ° ( ) + - 0-9 ***							X	
{ } [ ] · _							X	
Delete	X	X	X					
Backspace					X		X	

*** superscripts

Figure 4-2 Character Sets

Figure of Merit (Based on # of users and # of lines of code generated)	.267 Guidance, etc.	.133 Control & Stab	.150 Communication & Display	.150 Clock & Monitcr Subsystem Mode	.167 Data Management	.083 Supervisory	.050 Compiler	= 1 Exclus. Weight	Range	Total Score	Order
Scalar	3	3	3	3	3	3	3	1	1	3.000	1
Vector	3	3	0	0	0	0	0	.286	.285	1.197	3
Matrix	3	3	0	0	0	0	0	.286	.285	1.197	4
Boolean	1	2	1	2	1	2	2	0	.524	1.414	5
Logical Bit Strings	0	1	2	2	1	1	1	0	.381	1.032	7
Arrays	2	2	3	1	3	3	3	.572	.810	2.315	2
Structures	0	0	1	0	2	1	1	0	.238	0.615	10
Characters	0	0	3	0	3	3	3	.572	.57	1.347	6
Pointers	0	0	2	0	2	1	1	0	.287	0.765	8
Labels	1	1	0	1	0	1	0	0	.190	0.632	9

Weight: 0 = not necessary
1 = useful
2 = highly desirable
3 = necessary

Figure 4-3 Data Types Versus Functional Software

hensive documentation. Accordingly, the percentages of numbers of users were weighed at 2:1 over volume of code. The figures of merit were then normalized to equal a sum of one.

For each functional requirement the utility of various data types was estimated quantitatively (Weight: 0 to 3). Several scores were then computed and are shown in the figure. The "total score" represents the average over all seven categories of the individual weights multiplied by the figures of merit. The "order" is based on these averages and conveys the general utility of each type of space application. An interesting observation is that arrays of data are as necessary as the mathematical data types.

Two other scores are listed. The "exclusive weight" is a simple arithmetic average of only the "necessary" (i.e. code = 3) entries. It is a measure of how important a data type is, if it is important at all. Note that scalars, vectors, matrices and characters appear to be mandatory. The "range" is simply the arithmetic average of the weights over all the categories and reflects the scope or range of the data throughout its possible usage.

The following paragraphs contain descriptions of the listed data types:

a. Arithmetic - These are real numeric, scalar, data variables.

b. Vectors - A quantity described by magnitude and direction corresponding to the mathematical definition of a vector.

c. Matrix - An array (or collection) of vectors obeying standard vector-matrix arithmetic.

d. Boolean - This is a data variable that takes only two values, true or false, 1 or 0, or ON or OFF.

e. Bit Strings - May be considered as a collection of one or more binary bits.

f. Arrays - A collection of identical data types.

g. Structures - A structure is a hierarchical organization of data types (or arrays). A structure may contain different data types.

h. Characters - A non-numeric (in the sense of value) data type consisting of letters, numerals, or other symbols.

i. Pointers - This is a data type which contains information about the location of another data type.

j. Labels - This data type defines the variable as a pointer to a statement.

4.2.2 Data Type Notation. Annotated data types are an advantage to the reader and can aid the programmer in debugging his program. A vector variable indicated by an over-bar or a matrix by an over-asterisk enhances the readability, clarity and self-expression of the language. Unfortunately, not all data types possess readily acceptable annotation; however, an attempt should be made to provide unambiguous marks for booleans, arrays, characters, etc. or to provide notation where confusion may exist.

The need to supply identifying marks in the input stream can be tedious for the programmer; therefore the compiler will be required to supply these marks on the printed output whenever the programmer explicitly declares the data type. The reverse may also be true. That is, if the programmer chooses to supply all identifying marks then the explicit declarations may be superfluous. The extent to which the compiler will implement this last statement is the subject of detailed design.

4.2.3 Data Attributes. The data type alone is insufficient to totally define usage within a program. Additional attributes must be assigned at the point of declaration or automatic assumptions will be made by the compiler. These attributes include: dimensions for the data type or organization; e.g. size of matrix or bit string; fixed point, floating point or integer; qualification for arithmetic scalar variables, precision of variables, type of storage required, and initialization values for a variable. The language will be required to contain these,and possibly other attributes to further describe data types.

4.2.4 Identifiers. Data names and statement labels are defined as identifiers. As a general rule, identifiers will be descriptive of the purpose or use of the data types or statement label to enhance readability and understanding. This may require lengthy identifiers, for which the compiler will allow up to 31 characters (similar to PL/1). An identifier should be unique over the scope of program organization to which it applies. The subject of the scope of an identifier will be a part of language design. However, it is required that identifiers for global variables be applicable for all program limits and that a technique be provided for handling similar names in separate subprogram modules.

4.3 Executable Statements

4.3.1 Data Type Operators. Operators fall into three groups: computational, relational and logical.

Based on the defined data types the allowable operators have been arranged in the matrix shown in Figure 4-4. The guideline for allowing an operation is whether the action makes standard mathematical sense or in some other way conveys standard practice. Allowable operations are indicated by a check (√), the word "no" means that the compiler will be required to reject, by error message, the implied usage. Certain entries are circled, and require the following explanations:

a) The cross product is defined only in the context of three-dimensional vectors.

b) A matrix can only be raised to a positive integer exponent or to -1. The positive integer causes repeated multiplication of the matrix itself; -1 is the shorthand notation for the matrix inverse.

c) For structures and arrays all operations are valid on an element by element basis. That is two arrays of boolean variables will respond to the logical AND operation but cannot be mathematically added. A structure of scalars and vectors may be subtracted from another structure of identically defined data types. (Note, for example, that no operations can be performed on entire structures containing both scalars and boolean variables.)

A summary of operators follows:

OPERATIONS	NOTATION (Preliminary Definition)
I. Computational General Mathematical Operations	
Addition	+
Subtraction	-
Multiplication	blank
Division	/
Exponentiation	in exponent field (e.g. A^2)
II. LOGICAL Boolean Albebraic Operations	
and	AND
or	OR
not	NOT

Note: The use of & and | for AND and OR are alternate forms as are ¬ and ^ for NOT.

Figure 4-4 Allowable Operators as Functions of Data Types

Data Type	add	sub	mpy	div	DOT	CROSS	EXP	AND	OR	NOT	Concatenation	Relational
Scalar	✓	✓	✓	✓	no	no	✓	no	no	no	no	✓
Vector	✓	✓	✓	no	✓	(a)	no	no	no	no	no	no
Matrix	✓	✓	✓	no	no	no	(b)	no	no	no	no	no
Boolean	no	no	no	no	no	no	no	✓	✓	✓	no	no
Logical (bit strings)	no	no	no	no	no	no	no	✓	✓	✓	no	no
Characters	no	no	no	no	no	no	no	no	no	no	✓	(c)
Structures and Arrays	where applicable											

III. Relational Operations

Equal to	=
Not equal to	¬= or ^ =
Greater than	>
Less than	<
Greater than or equal to	> =
Less than or equal to	< =

IV. Special Operations on Particular Data Types

Dot Product for Vectors Only		•
Cross Product for Vectors Only		*
Matrix Transpose	Matrix Operations	*T M
Matrix Inverse	Matrix Operations	*-1 M
Concatenate for Character Strings		|| or CAT or ¦¦

4.3.2 Other Statements. A comprehensive set of executable statements similar to those available in other languages is required in the manned space programming language. The design of these statements must provide features for

- assignments
- control transfer
- conditional statements
- loop control
- error response
- I/O (See Sec. 4.4)

As of this writing, there are no known special requirements for unique executable statements.

4.4 Operating System Interfaces

The language and compiler must ultimately be machine dependent at least in effecting input/output operations and real time control functions. These are now described.

4.4.1 Input/Output.

4.4.1.1 Specialized I/O. The software for a manned spaceflight computer involves an extensive set of input and output operations; e.g.:

a. Display of information to the crew: e.g., dynamic data, control options, and attitude information. (Typically, these functions have involved special purpose display devices; in the future they will be more general, most likely 2-dimensional CRT-type displays.)

b. Response to crew requests from a number of consoles.

c. Generation of input and output commands to various onboard subsystems (e.g. the reaction control system, main engine, inertial platform, etc.).

d. Receipt and transmission of telemetry data (uplink and downlink).

Input and output operations can be quite specific to the (on-board) operating system. Tailored system subroutines are normally written for control, operation and support. A key question in designing the language is whether specialized input and output operations be included in the syntax of the language. If special I/O functions are not included, the programmer can call or use system subroutines to accomplish them (e.g. CALL DISPLAYR - -). In this way, the language is not cluttered with a set of commands which may not satisfy specific I/O requirements for all applications. On the other hand, having I/O commands as part of the language enables the compiler to enforce rules concerning I/O operations which may be particularly useful in a flight computer application. It also can provide more readability in the program listing and facilitate code modification.

Another approach is to employ macros. Macros can be defined for a specific flight computer implementation of the compiler. The programmer can use the macro to define his input or output requests. The compiler will recognize the keyword(s), interpret the request, validate it, and then substitute the proper code to execute system subroutines. Essentially a "substitutional" macro format should be as close to natural language as possible to enhance readability.

In summary, it will be assumed that specialized flight computer I/O operations are not required as an integral part of the general language; they will be specifically designed for the particular flight system and made available for the programmer by control transfers or macros.

4.4.1.2 General I/O. Since the language will be used as a general programming language, both independent of and as a complement to flight software development, a set of general I/O operations must be available.

I/O routines represent an interface between two languages - the language in which the program is written and that in which the data is written. I/O is not really an inherent part of a language, but a means of communication between values in the computer and their external counterparts, e.g., certain graphic symbols and their specific layout on a printed page. What passes across the interface is "graphic" and "control" information. To achieve their interface function, I/O routines make certain minimal demands on the language. What must be designed are generalized input and output commands that are easy to use, flexible, and machine independent.

It should be recognized that there are two fundamentally different kinds of I/O; (1) file directed, and (2) stream directed. File directed data is oriented towards secondary storage devices such as tapes or discs. File data has the distinguishing characteristic that it is recorded in such a manner that it may be retrieved later and still maintain its logical structure and integrity. It may be updated and/or new information may be inserted, and the result refiled. To facilitate the retrieval process, a directory is usually maintained. General routines for file handling (READ, WRITE, OPEN, CLOSE, etc.) must be provided.

The other form of I/O is stream directed. In general, there may be several different input or output streams. The desired I/O statements must be generalized routines to either input (READ) or output (WRITE). There are two main options. One is to output (or input) numbers with the proper conversion to (from) their equivalent graphic format. The other handles textual or alphanumeric data with no conversion. Thus, numbers on a card can be input either in a numerical form or as a string of symbols.

4.4.2 Real Time Control. Flight software involves real time control programming. In the process of coding, a programmer must be able to interface with the computers' operating system in order to perform control, synchronization, and tasking. More specifically, he must:

a. Initiate jobs conditionally upon event occurrence or elapsed time or unconditionally.

b. Terminate jobs conditionally or unconditionally.

c. Temporarily suspend tasks for a time period, or await the occurrence of an event.

d. Synchronize parallel and multiple tasks.

e. Request the allocation of system resources on either an exclusive or shared basis, such as LOCK and UNLOCK for global data or external devices.

f. Define control actions to be taken on particular events such as error conditions or non-nominal conditions.

Clearly, many of these functions will be an integral part of flight programs. The question of whether these functions should be included in the language syntax is similar to that addressed in the consideration of I/O control. Real time control can be machine and operating system dependent and built-in statements will detract from the general applicability of the language. Again the use of substitutional macros may be the best solution.

A careful analysis should be made to define and select the best method for providing real time control functions. A dominating factor in this selection will be the desire to insure that the features be generalized and incorporated as part of the syntax of the language. A preliminary review of the Real Time Operating System (RTOS) for the 360/75 in Houston, indicates that orienting a language syntax towards its features could impose undesirable requirements on future flight computer operating systems. For the initial version of the compiler, however, only a limited set of real time control functions will be implemented; only those functions that can easily co-exist with RTOS without modifications to RTOS.

4.5 Program Organization

The development of software for a flight computer system requires that programs be modular, i.e., they can be broken down into logical subsections which can be developed and tested individually. The following is a list of general requirements with regard to program organization:

a. Interfaces and interactions between procedures must be carefully defined and checked by the compiler.

b. Reentrant and relocatable procedures are required. Recursive procedures and subroutines are not.

c. A set of built-in common mathematical functions will be provided. The capability to return values other than single scalar items; e.g., vectors, matrices, is a requirement.

d. The compiler must provide for interfacing with basic machine language subroutines.

5.0 PRELIMINARY COMPILER REQUIREMENTS

5.1 Compiler Technical Features

5.1.1 Characteristics of the Code Generator. All developmental versions of the code generator should produce relocatable code. In addition, when generating code that is to be link-edited, the compiler shall possess the capability for the production of reentrant coding when needed (or alternatively it should always produce reentrant code). It is expected that these attributes will be required for the flight computer as well as the 360/75.

It is not expected that recursive code will be generated; the need for recursion does not seem to justify the cost of increased execution time. This appears to be especially true for an airborne computer system where time may be "of the essence". It seems virtually impossible to construct an example of a numerical problem (excluding symbol manipulation) where recursion could not be replaced by iteration with an attendant gain in efficiency. Moreover, if at a later date the need for recursion becomes clearly demonstrated, it would still be possible to add this capability. Of course, for a computer that has special hardware designed into its architecture to facilitate recursion, such as stacks and descriptors, then recursive procedures would be implemented.

5.1.2 Compiler Diagnostics. The compiler will issue extensive diagnostics both at compile and run time. At compile time, the compiler will:

a. Detect programming errors and classify them according to severity.

b. Flag suspected errors or strange usages to catch the programmer's attention.

c. Continue compiling if at all feasible in order to point out as many errors as possible during one compilation and also allow incomplete programs to be compiled for error checking.

d. Inhibit execution of the compiled program if the severity of the errors exceeds some threshold level. It will be possible for the programmer to readjust the threshold level, within reason.

At run time the diagnostics will refer back to the source code statements whenever possible. Thus, the messages can be associated with statements that have some meaning to the programmer. If the programmer supplies action statements to be executed upon the detection of error statements, for example, "On Overflow",

then the programmer directed actions will be taken. However, for other error conditions specific and to-the-point diagnostic messages will be issued.

5.1.3 Compiler Documentation. The output of the compiler will include a reformated and annotated listing of the source statements in an easily readable form for documentation purposes. This standard output format will be the standard medium for discussion, exchange of information, and the answer to questions concerning the characteristics and behavior of the program under various conditions. This suggests that the compiler will use any and all of the information at its disposal gathered throughout the compilation process to produce a form of output that is self-documenting. Much work needs to be done to determine the extent to which this should be carried and the form that the output should take. Specific examples of items that the compiler could do include the following.

5.1.3.1 Automatically indent the source code so that it is virtually easy to group logical sections of code (e.g. DO-END blocks).

5.1.3.2 Supply special symbols before, above, next to, or with variables to make the characteristics clear at each usage; for example, to distinguish between matrices, vectors, booleans, and structures.

5.1.3.3 Annotate constants to indicate where uncertainties are introduced due to the truncation caused by computer word length. This is especially important for fixed point constants when the accuracy of the numerical representation is only known to plus or minus a certain value which the compiler could determine and display. Although not as obvious, some form of this mechanism is also important for floating point numbers.

5.1.3.4 Supply information about characteristics of the output code, assumptions made by the compiler, default conditions and attributes, and other similar items. The purpose of this material is to clear-up doubts, answer questions, and resolve ambiguities about the compiler interpretation of input and subsequent actions.

5.2 Environmental Considerations

5.2.1 Compilation by Parts. It should be possible to compile pieces of the program separately by section or procedure, or other logical subunit. These compiler pieces are bound together or linked to form an executable package. This permits the modular construction of flight programs; the modules can be separately

written and tested and then combined with other modules to form larger units. However, this does require a relocatable loader or link editor. The modular assembly system also permits the collection of different programs as needed for system test, vehicle checkout, etc. This compilation by parts facility is necessary for the development of flightworthy software where a large body of programmers and engineers are working cooperatively on a large scale programming task. Initially, the compiler will be directed toward the checkout of the language characteristics and this limited scope may eliminate the need for partial compilation in the first compiler version.

5.2.2 Sharing of Common Data. A fundamental concept that must be incorporated into the language and the compiler is the orderly use of global data, especially since the language is to be used in a real time control environment. It includes the separate definition, description, and maintenance of pools of common data and the introduction of these global data tables in each individual user's procedures in a systematic fashion; it also includes the mechanism for interlocking and sharing of the data in a dynamic multiprogrammed environment. To do so implies that the compiler and the language must provide the interlocking and sharing mechanism and automatically insist on its usage. Thus, a system for defining and maintaining the state of locked variables must be established.

Because of the similarity in nomenclature (read protect, write protect, etc.) the concept of interlocked data is often confused with memory protection mechanisms. Memory protection is a hardware feature designed into many computers that permits areas of memory to be protected from the inadvertent accesses by unauthorized users, e.g., undebugged programs. This is under control of the executive and isolates programs and bounds their effects so that they cannot interfere with other users. Interlocked data, on the other hand, provides the means for many authorized users to share the same data in an orderly fashion.

If this capability is added to the facilities for tasking (the creation of jobs) and the synchronization of these tasks, then the necessary facilities for multiprocessor operations will have been designed into the language and compiler.

5.2.3 Input Interface. The compiler must interface with a symbolic file maintenance system. The source language file handler will provide a great deal of flexibility in the selection of input files to be presented to the compiler. It will have the ability to produce new versions for testing purposes and to backtrack to previous revisions of the source program when the current one proves inadequate.

It is felt that this file handler will not be an integral part of the compiler but will serve as an input interface mechanism for the compiler. A text editor, which operates in concert with the symbolic file handler, although logically a separate program, is assumed to be a highly desirable and useful mechanism for the updating and insertion of text into the input stream. It is also assumed that terminals as well as key punches will be routinely used as input devices.

5.2.4 Output Interfaces. The output interface design is expected to go through several stages of increasing complexity and capability.

5.2.4.1 The first is the most elementary where actual machine language is the output and it is run under a simple monitor that serves as the interface with the operating system.

5.2.4.2 The second step provides for the more complicated output needed for partial compilation and will produce output that can be fed into a relocatable loader or link editor. Thus, separate compiled pieces can be link-edited into one run-time package. At this step in the process, it will be possible to introduce other routines or procedures that were written in assembly language or another language; e.g. Fortran. If Fortran compatibility were desired, it would place additional requirements upon the language and compiler to recognize statements and formats necessary to interface with Fortran subroutines. It is also at this point that library routines are collected and linked together to make a run-time package.

5.2.4.3 The final stage is to output a format necessary to interface with a simulator for a flight computer. This will require modifications to the compiler output as defined by a negotiated simulator interface procedure. For instance, the simulator would undoubtedly require the symbol table to be passed on to it.

At all stages of the output scheme it is expected that data necessary for extensive run-time diagnostics and debugging techniques will be generated and provided. These diagnostics will relate to the source or documentation language as clearly as possible.

5.2.5 The Code Generator. The code generator will be modular so that a new one could be written with relative ease for a different computer, e.g., a flight computer. This means that the portion of the compiler that generates machine dependent code will be isolated and capable of replacement without redoing the rest of the compiler. Although every attempt will be made to

make the language machine independent, it is recognized that several factors do make compilers machine dependent, e.g., word length dependent calculations,and floating point format and precision.

5.2.6 Issue of an Interactive Feature. It is not expected that the compiler will be interactive or incremental. This capability only seems useful in a dedicated time-sharing environment. The demands of aerospace usage indicate a more traditional compiler design.

5.3 Specific Compiler Capabilities

5.3.1 Debugging Aids. The programmer will need debugging aids, which at the very minimum will include a timing facility and a trace option. Timing refers to the real time (with respect to the computer clock) taken to execute a specific or identifiable set of instructions. Tracing can reveal the instruction by instruction operation of the computer including printout of all the special and working register contents. Timing and tracing requests can be designed into the compiler system as compiler directives which generate object code and are executed at run time. Most probably these will not be a requirement for an advanced space programming language. While debugging aids will still be of the utmost significance, they will be part of the monitor-simulator systems, the simulator being an indispensable tool in the checkout of flight software. During simulation, timing and tracing will be done by the simulator. No extra object code is generated and the programmer evaluates the identical code that will be used in an actual program run. Note that unlike the compiler directive, the programmer need not delete his "request" or recompile when he is finished with checkout.

5.3.2 Language Extension.

5.3.2.1 Reversion to Machine Language. There have been many reasons for inserting machine language code into the stream of a higher order language. These reasons usually relate to deficiencies in the higher language or reflect the necessity of "shaping" the higher language to the specific machine. While they may be valid, the accompanying effects are sufficiently undesirable to discourage the use of machine language, and in fact, the ability to "drop into machine language" will not be a requirement for an advanced space language.

5.3.2.2 Macros. The design of an advanced space programming language must allow for extensibility. That is, all applications cannot be anticipated at design time and provision must be made to cope with developing hardware and mission definition. The "macro" facility can fill this need and will be a requirement for a space language. The macro is simply a programmer-defined expression which the compiler recognizes and implements either by subroutine call or by direct substitution of previously defined language elements or code. However, language extensibility has its pitfalls and a space programming language where reliability and clarity are paramount must be careful to place constraints on macro-defining to the extent that language dialects cannot be generated in the guise of macros. The intent of providing a macro facility will be to allow the programmer to introduce statements like

"Fire Main Engine For 10 Seconds"

"Display on Scope Results of Experiment 12"

or perhaps

"Downlink Engine Data List"

Each of these macros clearly states the desired program action, is addressed to specific mission situations and/or hardware, and may imply an I/O capability. The compiler can be directed to insert the macro definition in the listing following the subject statement to increase readability. In any event, macros will be collected and properly indexed for easy reference.

5.3.3 Space and Time Optimization. For the most part, aerospace computers have been relatively slow machines with limited memory capacity. The challenge has been to accomplish all the mission requirements in the space available. As a result, fixed point machine language code has been predominant and most higher order language/compiler systems have been rejected as being too inefficient. But advanced space computer hardware will be very fast and mass memory techniques promise to alleviate practical space restrictions. In a large space shuttle or station effort, the need for higher order language programming to enhance reliability and communications far outweighs any attendant loss in efficiency. The new spaceborne hardware and the fast compile times achieved by ground computers like the IBM 360/75 obviate the requirement for compiler space and time optimization. Of course, good practice will be followed and unnecessary code eliminated, but primary effort will not be spent in making the object code more concise and faster running.

5.3.4 Compiler Output Options. Program writing and debugging must be aided by an extensive set of compiler printed outputs, the scope of which is influenced somewhat by the necessary complications introduced into the compiler design. Although at this writing a final set has not been developed, the following elements are identified as requirements.

a. Input Listing:

   A straightforward listing of the programmer's input code. References to macros and/or subroutines may cause substitution of definition code, on option.

b. Compiled Listing:

   Source code has been compiled, reformated for readability and annotated where applicable. Annotation will identify data types, macros, subroutines and provide other indicators that are self-documenting.

c. List of Variables:

   An alphabetical list of variables which indicates the data type, the number of references, locations (and page) where each variable was read, and the locations (and page) where the contents of each variable was changed.

d. List of Global Variables:

   Programmers may avail themselves of global (or common) data not defined locally within their programs. These variables are listed and cross-referenced to statement, to make visible software interfacing.

e. Lists of Macros, Subroutines:

   Use of macros and subroutines will be listed and properly cross-referenced.

f. Hardware Interface List:

   The interactions between software and hardware need to be specifically identified to maintain reliability especially in a space application. It is most important to keep track of all computer operations which cause direct hardware response (e.g. engine shut-down) as well as reading of sensor data. This list will include all I/O references, grouped and cross-referenced to statement.

g. Defaults Exercised:

All aspects of language operation will not have to be defined by the programmer. Under certain circumstances, the compiler will assume (default) what the programmer intended. To ease in debugging, all defaults will be collected and listed.

h. Storage Summary Map:

A storage map of the coding as it appears in the object machine will be provided. It will be possible to cross-reference from source code statements to relative storage location. In addition, unused locations will be indicated and summarized.

i. Source Debugging:

Provision will be made to relate operating system diagnostic statements to the source code. It should not be necessary, for example, to interpret a numerical dump in order to locate a matrix-vector product or scalar divide overflow.

j. Input Media Preparation:

It is expected that the program may be compiled and debugged on one machine and then run on another. A requirement will exist for the preparation of system information on tape, cards, disk, etc. for compatible operation.