1 4P

Report No. 72-0005
Contract NAS8-26990

(NASA-CR-123571) FLIGHT PROGRAM LANGUAGE REQUIREMENTS. VOLUME 1: EXECUTIVE SUMMARY Final Report (M&S Computing, Inc.) 7 Mar. 1972 29 p CSCL 09B

N72-22179

Unclas
G3/08 27279

FLIGHT PROGRAM LANGUAGE REQUIREMENTS

VOLUME I

EXECUTIVE SUMMARY

CR-123571

March 7, 1972

Prepared for:

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

M&S COMPUTING, INC.

CAT. 08

PREFACE

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

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

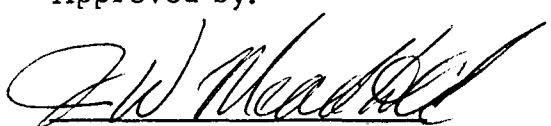
This report is the final report of the study and is provided in three volumes. This first volume is the Executive Summary and provides an overview of the effort and results. Subsequent volumes provide more detailed information.

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

Participating personnel were:

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

Approved by:



J. W. Meadlock

TABLE OF CONTENTS

VOLUME I

VOLUME II

TABLE OF CONTENTS
(continued)

TABLE OF CONTENTS
(continued)

(BLANK)

1. INTRODUCTION

This report describes the activities and results of a study for the "Definition of MSFC Flight Program Language Requirements" performed by M&S Computing under Contract NAS8-26990. The final product of the study is a set of detailed requirements for a language capable of supporting onboard application programming for the Marshall Space Flight Center's anticipated future activities in the decade of 1975-85. These requirements are based in part, on the evaluation of existing flight programming language designs to determine the applicability of these designs to flight programming activities anticipated by MSFC.

An integral part of the study was the coding of benchmark problems in the selected programming languages. These benchmarks are in the form of program kernels selected from existing MSFC flight programs. This approach was taken to insure that the results of the study would reflect state of the art language capabilities, as well as to determine whether an existing language design should be selected for adaptation to MSFC's applications.

1.1 Study Plan

The study was performed in three distinct, sequential phases:

- o Evaluation Method Definition Phase, during which the languages to be evaluated were selected, and the techniques and criteria by which they were to be evaluated were established in detail.
- o Language Evaluation Phase, during which four selected flight programming languages were evaluated through review of language specifications and coding of benchmark problems.
- o Requirements Definition Phase, during which detailed programming language requirements were synthesized.

The major steps performed in each of these tasks are described in the following paragraphs.

1.1.1 Phase I - Evaluation Method Definition

The following major tasks were performed during the first phase of the study:

- Selection of Languages
- Selection of Evaluation Criteria
- Selection of Flight Program Kernels

Four current flight programming language designs were selected for evaluation. These languages represent the latest pertinent language developments sponsored by several government agencies. The languages and their sponsoring agencies are listed below:

- Space Programming Language (SPL)
  U. S. Air Force
- HAL
  NASA Manned Spacecraft Center
- Computer Language for Aeronautics and Space Programming (CLASP)
  NASA Electronics Research Center
- Compiler Monitor System -2 (CMS-2)
  U. S. Navy

The development background of these languages is highlighted in Section 2 of this report.

The evaluation criteria are based primarily on those items of direct interest to the users of the language. That is, the people who are designing, coding, reading, verifying, and maintaining flight programs are given highest consideration. This approach tends to emphasize the flight program and the flight program development cycle rather than the language processor and other support software. While the impacts on support software are not ignored by the evaluation criteria, M&S Computing feels that the projected extensive use of the flight language dictates an approach which heavily favors the user. The resulting evaluation criteria are described in Paragraph 1.3 of this report.

To support the benchmark coding task to be accomplished during Phase II, it was necessary to select existing program designs containing functions critical to flight programming. It was not intended that each selected program kernel be coded in every detail or be translated literally. The main purpose was to select program kernels which included as many functions as possible. Eleven kernels were

selected, ten from the Saturn V Flight Program and one from the Skylab Apollo Telescope Mount Flight Program. These kernels and the flight program characteristics they represent are described in Appendix A of this report.

The benchmark kernels were selected to reflect three types of flight program functions:

- o The more complex program functions.
- o Those functions which were common to many modules.
- o Functions which were unique but performed a particularly significant operation within the flight program.

1.1.2 Phase II - Language Evaluation

The following three major tasks were performed during the second phase of the study:

- o Coding of Program Kernels.
- o Review of Language Specifications.
- o Review of Evaluation Criteria and Program Kernel Selections.

Selected program kernels were coded in each of the languages evaluated. It was initially intended that the sequence of applying the languages be different for different kernels, so that no single language would bear the brunt of solving the unique "first-time" problems for all the kernels. However, this approach was unmanageable because it forced the programmer to switch from one language to another on an almost daily basis. This consumed time and created errors through trying to recall and look up such things as which language uses:

- o A period instead of a colon
- o TO instead of UNTIL
- o GO TO separated by a blank instead of GOTO as one word.

This burden was judged worse than the "first-time" coding problems, so the languages were used one at a time in the following sequence:

- SPL
- CLASP
- HAL
- CMS-2

After accounting for an initial analysis burden on the SPL effort, the language learning times and kernel coding times did not vary significantly from language to language. The CLASP learning and coding times were somewhat lower than for the other languages, but this is undoubtedly because CLASP coding followed immediately after SPL coding. Consequently, these statistics reflect the fact that CLASP is almost a proper subset of SPL, rather than supporting a hypothesis that CLASP is easier to learn and use. By comparison of the source card listings in Appendix B, it is apparent that none of the languages requires significantly more actual writing than another. Since the evaluation was concerned with the language design, rather than a particular implementation of the design, no attempt was made to process the source cards through a compiler. The details of the kernel coding exercise are described in Section 8 of this report.

The kernel coding was an effective application of the language, but only to current flight program characteristics. To insure that potential differences between current and future applications were considered, the specifications for the four languages were reviewed and analyzed in detail. This analysis was largely independent of the coding of the program kernels. The language specifications were inconsistent in organization and depth of description from one language to another. Therefore, part of the specification review effort required generation of language summary descriptions organized around a single language characteristics outline. These summary descriptions provided a common base of information for analysis and comparative evaluation of the languages. These language characteristics summaries are found in Section 7 of this report.

During the Phase II effort, the evaluation criteria and program kernels were reviewed to determine if modification of the criteria or additional kernels could increase the effectiveness of the evaluation. No revisions were found necessary.

1.1.3 Phase III - Language Requirements Definition

The single objective of this phase was to define in detail the specific requirements for a programming language for future onboard

applications. This was accomplished through extensive presentation and review of the Phase II results and through a final review of the most recent developments in future space programs.

An additional effort was performed during this phase to help clarify the significant characteristics which make a "flight programming" language unique from a more general purpose programming language. It was decided to select a current general purpose high level language and identify those capabilities of the four flight programming languages which were significantly different from the general purpose language. IBM's PL/I was selected for this comparison, because it is the most powerful general purpose language available, it is in widespread use, and it contributed to the designs of three of the flight programming languages reviewed.

1.2 Conclusions

The primary results of this study are, of course, the flight programming language requirements, which are introduced in Section 3 and detailed in Section 4 of this report. Some additional conclusions which are not immediately apparent from the language requirements or from the evaluation discussions deserve special emphasis.

The first conclusion is that any one of the reviewed languages can, if properly implemented, provide the user a significant improvement over assembly language for aerospace programming. In that respect, there are many more similarities than differences among these languages. While the comparative evaluation naturally concentrates on exposing and analyzing differences, the reader should bear in mind that these differences are on top of a broad base of similarity and value. While this conclusion is no surprise, the point cannot be over emphasized. A high level language is an integral part of a cost-effective approach to future flight programming, and recent development efforts reflect a common opinion about many of the capabilities which should be provided in such a language.

The second conclusion summarizes the language evaluation effort. A quantitative rating of the four flight programming languages revealed that SPL and HAL represent a significantly increased capability and applicability over CLASP and CMS-2. However, there is no clear indication that either SPL or HAL is more appropriate than the other. This situation arises primarily because of a fundamental difference in the approaches taken by the two language designs. The SPL designers attempted to maximize the power of the language by

retaining many of the popular programming techniques which have traditionally been used by assembly language programmers. The HAL designers attempted to maximize the reliability of the source language programming, partly by avoiding some of the popular assembly language programming techniques which historically have contributed to the generation of, or retarded the detection of, program errors. This dichotomy between power of expression and reliability of the higher level language program has become a key issue in the establishment of flight programming language requirements.

The third conclusion is that, considering all the useful features of all four flight programming languages, there is surprisingly little capability provided which is not provided in some form through a commercially available language such as PL/I. As indicated in Section 2, all of the evaluated languages except CMS-2 grew out of comparative language studies which included PL/I as a candidate. These studies each found PL/I to have many applicable features but to be too powerful and too difficult to use. Each study had its own reasons to then depart from PL/I to a greater or lesser degree. Their differences in capability from PL/I do not seem to justify their drastic departures from the appearance and formats of a language which has existed for many years and has the advantage of a broad base of users.

1.3 Evaluation Criteria Summary

This introductory discussion summarizes the evaluation criteria which were established during the first phase of the study. These criteria are explained in greater detail in Section 6 of this report, where their application to the languages is discussed. The criteria were developed in five categories, ordered as follows:

- Quality of Expression
- Expression of Environment Interaction
- Process Reliability
- Object Program Ffficiency
- Source Program Readability

The order of importance of the categories is as significant as the categories themselves, and the following discussion explains the reasons for that ordering as it summarizes each category. The first

two categories are concerned with what can be expressed in a language and how well it can be expressed by the programmer. They encompass the most important evaluation criteria, because they reflect how easily and reliably the programmer can describe a desired process. They determine what programming techniques can be invoked by the programmer in describing his problem. This description activity is the first step in the program development process to be directly influenced by the programming language. Furthermore, the product of this activity is the source language program which is the object of the other evaluation criteria. If effective programming techniques cannot be invoked or if the description of their use is not well expressed, then every evaluation category is adversely affected. For example, if the programmer must resort to "creative tricks" to overcome a lack of straightforward expression capability in the language, then:

- Reliability tends to be decreased by the departure from "time-tested" techniques and constructions.

- Reliability is decreased for program maintenance activities, because the creativity of the technique tends to obscure its intent and, therefore, the implications of modifying it. The obscurity increases with time and experiences discrete step increases as new personnel are assigned to maintenance of the program.

- Efficiency of the object program may be degraded because the compiler optimization techniques were geared to more conventional use of the language features employed.

- Readability of source program is degraded by the same obscurity which reduces the reliability of program maintenance.

This overall influence of language expression is what causes the expression categories to be so highly ranked.

The Quality of Expression category covers all the activities internal to a program module. These activities include most data declarations and manipulations, decision making, iterations, and other execution controls. Environment Interaction has been evaluated in a separate category from the rest of the expression criteria because it interfaces with hardware and system software which impose external constraints on the language and the programmer. Each program module has a distinct environment with which its execution

interacts. The major functions that a language needs to provide to express this interaction are in four areas:

- o Communication with Vehicle Systems/Subsystems
- o Synchronization with the Vehicle Mission
- o Intertask Communication with Other Program Modules
- o Mass Storage Access

Process Reliability is the next most important category after the expression categories. It is concerned with the production of error-free source language statements when they are written, as well as the ability to efficiently detect and correct errors which do occur. The reliability of the program development process is reflected in the reliability of its final product, the object program. A higher level language's greatest contribution toward a reliable object program is in the generation of more reliable code in the beginning. That is, most of the language features which contribute to effective expression also naturally contribute to reliable code. However, program reliability is at least confirmed, and usually established, during the verification step of the process. There are specific features within a language which can assist the verification step directly. Verification, in this context, includes all activities directed toward finding and correcting errors in a program after it has been coded. The Process Reliability category does not attempt to reflect the reliability of a specific compiler or other language processor. It is assumed that the programmer's source language is accurately translated into object language. The area of concern is in preventing and/or detecting errors made by programmers.

Object Program Efficiency reflects the quantities of storage space and execution time required by the object program which results from compilation of a source language program. Computer storage space and processing power have been, traditionally, limited to minimize weight and power requirements. Consequently, the efficiency of the utilization of these computer resources has been a major concern. Existing and near future computer system designs provide a many-fold increase in processing power over previous aerospace computers. Although this relieves the heavy emphasis on efficiency somewhat, it by no means negates it completely. It should be realized that onboard computer applications continue to increase in scope as well as complexity, so it is naive to assume that the available processing power will easily outrun the required processing power.

Efficiency of resource utilization therefore remains a factor to be considered.

Object Program Efficiency, as a language design consideration, is low in the ordering of evaluation categories. This is not done to encourage inefficiency in object programs; it is done because so few language features are influenced directly by efficiency requirements. Since efficiency is a measure of a resulting object program, it is mainly a result of the compiler implementation rather than the language definition. That is, the same language implemented by different groups or techniques can result in different efficiencies. However, there are a few features which can be incorporated into a language which can help or hinder the compiler in the generation of efficient object code. The evaluation was limited to the domain of these language features, not the techniques by which the compiler was implemented.

Source Program Readability, as considered here, is the relative ease by which a programmer, as well as cognizant personnel not directly involved in program implementation, can relate the completed computer program to the problem to be solved. Clearly, this category is closely related to Quality of Expression, because most of the features which make a language more readable also make it more expressive. It is because of this commonality with expression that readability is ranked last among the evaluation categories.

It was decided at the outset, and verified through the evaluation, that readability needed to be considered in a restricted environment. Specifically, it was assumed that the reader was somewhat familiar with the problem being solved, or the process being described in the program, and that he had received some prior introduction to the language.

(BLANK)

2. LANGUAGE EVALUATION SUMMARY

Before delving into the detailed characteristics of each language and relating them to the evaluation criteria (Section 6), this summary attempts to provide a concise presentation of the evaluation results. Although the application of the evaluation criteria is a qualitative process, the overall results have been quantified for purposes of presentation. Figure 2-1 is a table of ratings of each language in each of the five evaluation categories. A value of five is assigned to the "best" language(s) in each category and the others are assigned lower values as their capabilities dictate. Weights representing the relative importance of the criteria are included. A weighted sum of the ratings for each language then quantifies its overall evaluation.

Note that for weighting purposes, a single category of expression combines Quality of Expression and Expression of Environment Interaction (in the ratio of 70% to 30%), to form half of the total evaluation weight.

The following paragraphs present very briefly the backgrounds of the four languages and compare the highlights of the languages which result in the ratings of Figure 2-1.

2.1 Space Programming Language (SPL)

2.1.1 Background

SPL was designed and implemented by the System Development Corporation (SDC) under the sponsorship of the Air Force Space and Missile Systems Organization. The effort grew out of an SDC study of ALGOL, FORTRAN, JOVIAL, NELIAC, and PL/I to determine their suitabilities to space applications. These five languages were quantitatively rated by SDC within ten different semantic language attribute criteria. PL/I ranked highest in eight of the ten categories and was tied with ALGOL in the ninth. JOVIAL ranked highest in the tenth criterion. From this preliminary screening, JOVIAL and PL/I were selected as the two prime candidates and were analyzed in detail. Neither was found suitable, as indicated by the following quotation from "Conception of SPL" (Reference A9*):

"The primary deficiencies were arithmetic and algebraic

*References are provided in the bibliography included in Volume II.

LANGUAGE RATINGS

Languages Categories	Wt.	SPL	HAL	CLASP	CMS-2
o Expression - Quality of Expression (70%)	0.50	5	4	3	2
- Environment Interaction (30%)		5	5	2	4
o Process Reliability	0.25	3	5	4	4
o Object Program Efficiency	0.15	4	3	5	3
o Source Program Readability	0.10	5	5	4	3
WEIGHTED RATINGS		4.35	4.35	3.50	3.05

Figure 2-1

formula manipulation, direct code, flexible data overlay, input/output, and interrupt processing. PL/I capabilities were found to be the most extensive but were not optimally designed to spaceborne software. As well as being deficient, PL/I had superfluous features and was very complex. JOVIAL's capabilities were nearly all applicable to the space application; nevertheless, it too lacked capabilities as well as possessing data restrictions. It was concluded that a procedure-oriented, application-specific language should be developed for aerospace programming and that it should be identified as SPL (Space Programming Language)."

The SPL design effort was begun in 1967. In consideration of an attempt by the Department of Defense to curtail the proliferation of programming languages, and since the Air Force had adopted JOVIAL for command and control applications, SPL was made a dialect of JOVIAL (SPL/J6).

SPL was designed to satisfy the requirements of three separable programming areas associated with the development of spaceborne software:

- Mission planning and formulation
- Support programming
- Flight programming

When the original SPL specification was criticized for being too extensive for a specific application, it was decided to redefine SPL as a single language with application oriented subsets. Currently, there are five subsets which are supposedly upward compatible. (However, CLASP is clearly not compatible with SPL.) The five subsets and their intended ranges of application are:

- MARK I — CLASP
- MARK II — Fixed-point aerospace computer
- MARK III — Floating-point aerospace computer
- MARK IV — Ground-based support computer
- MARK V — Ground-based multi-processor

MARK IV is the subset which was evaluated in this study as SPL. (CLASP is discussed separately in Paragraph 2.3.)

2.1.2 Evaluation

SPL is clearly the richest and most powerful of the languages reviewed. Its capabilities are provided through high level, user-oriented features such as decision tables and matrix operations, as well as computer oriented features which allow the programmer to "get at the bits" of the machine when he wishes to. The computer oriented features include, for example, a full capability for indirect addressing, the ability to name and directly manipulate any of the program controllable registers in the computer, and the ability to customize the packing of data in a table by specifying word number and starting bit number for each field. SPL has clearly attempted to forestall, at least in many areas, the common programmer complaint ". . .but assembly language lets me do it".

To a large degree SPL has succeeded in that goal, but at a significant cost to Process Reliability. Several of the techniques available are generally error prone and are capable of introducing very subtle errors into the coding whose detection and diagnosis can be very expensive. One of the biggest shortcomings of SPL in this respect is that the flexibility and freedom given the programmer allow him to generate coding errors which elude detection through static checking by the compiler. Opportunities for this static checking are denied in two ways. First, programming techniques like indirect addressing are provided in such a general form that the compiler cannot determine,for example, that an intended indirect data access is actually (and erroneously) modifying executable instructions, or that an indirect branch is transferring control into a data array. The other languages impose greater restrictions on such capabilities, thereby making a broader class of errors detectable.

The second way static checking is denied by SPL is through programmer convenience features. Through implicit data conversions and implicit data declarations, for example, the compiler assumes an intent on the programmer's part, and generates object code accordingly. Other languages require more explicit statements from the programmer and can, therefore, detect inconsistencies which result in errors.

Environment Interaction is provided in SPL through a powerful input/output capability for conventional peripheral equipment, and

through an ability to control interrupts and react to interrupts and other external devices. Real time synchronization and communication among program modules are also provided. HAL's approach is more user-oriented by providing features which interface through an executive, so they can and should be used directly by the application programmer. SPL provides a more direct interface with the computer hardware, and is more oriented toward coding an executive than interfacing with it. While the approaches are different, the capabilities are considered nearly equivalent.

Program Efficiency in SPL is more under the direct control of the programmer than in any of the other languages. A variety of individual techniques are available to conserve memory for specific data organizations and to reduce execution time for particular functions. SPL is ranked below CLASP in this category primarily because CLASP provides generalized optimization directives in addition to limited specialized techniques.

Readability in SPL is high, primarily because of the higher level of description of operations through decision tables and matrix and vector operations. Status variables and constants also provide a mnemonic capability which gives condition flags and simple state counters much more meaning to the readers.

2.2 HAL

2.2.1 Background

HAL is a programming language developed by Intermetrics, Inc., for manned spaceflight computer applications. It was designed under contract to the NASA Manned Spacecraft Center for application to both on-board and support software. The design reflects an effort to accomplish three major objectives:

- Increased readability
- Increased reliability
- Real time control

Specific features within the language contribute to the fulfillment of each of these goals. Note that these three goals correspond closely to the three evaluation categories in which HAL received its highest ratings.

HAL's design followed a study by Intermetrics of eight existing languages: CLASP, SPL, JOVIAL, ALGOL, FORTRAN, PL/I, MAC, and APL. The greatest influence on the design was PL/I, as evidenced by the following quotation from the final report (Reference A7): of the development effort:

> ". . . The basic syntactical structure of HAL is straight from PL/I, which in turn owes much of its form to ALGOL. The conditionals, DO groups, DECLARE statements, and STRUCTURES are almost pure PL/I with word changes here and there. On the other hand, PL/I is a very rich and full language and has many, many features that are not needed nor applicable. Moreover, there have been efficient implementation of PL/I subsets, which is what HAL superficially resembles."

While the resemblance indicated by program listings is rather vague due to HAL's statement format and notation, the resemblance in terms of functions performed and techniques for describing them are judged more than superficial. MAC contributed HAL's two dimensional format and the matrix and vector notations; JOVIAL contributed the compool concept; and the generality of the language was inspired by APL.

2.2.2 Evaluation

In Quality of Expression, HAL was judged slightly less capable than SPL. While it provided the very important matrix and vector operations in the most natural and general form, it lacks such features as decision tables, location pointers, status variables, and multiple entry points to program modules, all of which are judged extremely useful.

Two other issues tend to cloud HAL's quality of expression. The two-dimensional statement format is judged more difficult to write, partly due to its novelty and partly due to required line identifiers in column one and statement delimiters. The very rich character set of HAL provides very expressive operation symbology, but the kernel coding experience revealed that in moving from one line printer to another, limited and different subsets of the character set could be printed. This could be construed as just a readability problem, but if the compiler output is to reflect the input source statement, the restrictions feed all the way back to the coding sheet.

The objective most successfully met by HAL is increased reliability. Some of the more error prone programming capabilities have been excluded, and requirements for explicit statement writing allow the compiler to perform a great deal of status checking on the source code. Program module interfaces are carefully controlled, and effective data protection features are included.

Environment Interaction in HAL has the most user-oriented approach of the four languages. It provides extensive input/output and real time task control features for interfacing with an operating system.

HAL pays the least attention to Object Program Efficiency of the four languages. Features are limited primarily to controls over data packing and temporary allocation of data to a task.

HAL's approach to meeting the increased readability objective was to provide a two-dimensional statement format and a very rich character set and notation. These are judged to be a marginal contribution to readability, being most effective in complicated mathematical expressions. HAL's greater contribution to readability came through such things as the interfaces between program modules, which makes them easier to follow, and the avoidance of confusing constructions, like SPL's nested assignment.

2.3 Computer Language for Aeronautics and Space Programming (CLASP)

2.3.1 Background

CLASP was designed and implemented by Logicon under contract to the NASA Electronics Research Center. It resulted from a Logicon study to define a language for real time aerospace program development. The primary design goal for CLASP development was to produce a language applicable to the aerospace programming problems of the present (1968) and near future. Therefore, the effort was concerned with the programming environment of a relatively small aerospace computer without floating point arithmetic. The result was a very heavy emphasis on generating optimum object code in terms of reducing memory and execution time required, coupled with an ability to easily specify and control the scaling of fixed-point arithmetic operations.

The study investigated SPL and PL/I and found that while both

provided significant advantages, neither was capable of solving certain problems in a direct, straight-forward manner. SPL was selected as a base for developing CLASP partly because SPL was considered to be more nearly applicable to the design goals of CLASP and partly because:

> ". . . the Air Force and System Development Corporation were proceeding with the development of SPL, and it was expected that continuing cooperation among the two government agencies and their contractors might result in further modification of SPL to make it more suitable and at the same time compatible with the corresponding NASA language." (Reference A1)

In an attempt to make CLASP easier to learn and use, less costly to implement, and more efficient, a number of SPL features were specifically excluded. These omissions have significantly reduced CLASP's evaluation ratings relative to SPL.

2.3.2 Evaluation

CLASP's strongest rating among the evaluation categories was in Object Program Efficiency. All of the significant time optimization features of the other languages are provided, and CLASP also has a generalized time optimization directive. Furthermore, the language functions are simple enough to not hinder the generation of efficient object code. The generalized storage optimization directive is also deemed valuable, and if implemented properly should have as effective control over data packing as the explicit data packing attributes of the other languages.

CLASP is also strong in Process Reliability, primarily because individual features are restricted enough that they must be used in a fairly straightforward manner. The elimination of implicit data declarations is considered especially good for reliability. The biggest shortcoming of CLASP in this category is the automatic fixed-point scaling by the compiler which can cause unsolicited and undesired changes in the scaling of variables between compilations.

CLASP is weakest in the areas of expression. This is consistent with the design goals, but unfortunately restricted the language considerably even for the sample coding of the Saturn flight program kernels. The Environment Interaction capability was rated lowest among the languages because there is no input/output capability and

because an entire program organization must be compiled at the same time. Even in the very smallest of aerospace programming tasks, these are very expensive omissions in terms of program development costs. The Quality of Expression rating is also low, based primarily on a shortage of capability to express high level operations and on restrictions on capabilities which are provided.

The Readability rating is low primarily because operations must be described at a detailed level.

2.4 Compiler Monitor System-2 (CMS-2)

2.4.1 Background

CMS-2 was designed by Computer Sciences Corporation for use in a variety of U. S. Navy computer applications. While these applications are not limited specifically to aerospace projects, the language represents the Navy's current efforts for a multi-application, real time programming language. It is included in this evaluation to insure completeness of the effort. The CMS-2 design was based primarily on JOVIAL, FORTRAN, and CS-1, an earlier language used by the Navy.

The CMS-2 language is part of a total system which includes a compiler, a system librarian, loaders and link editors, a flowcharter, and a batch processing monitor. The predecessor to CMS-2 includes the CS-1 compiler and the MS-1 monitor which have been in use by the Navy since the early 1960's. The CMS-2 development effort began in 1966 with a study of future Navy programming requirements. Based on this study and the need to continue support of many then existing tactical data systems, the following CMS-2 design objectives were established:

- To combine the best features of the existing CS-1/MS-1 and other new languages for new and future requirements
- To allow salvage of the maximum value from previously developed CS-1 programs, or facilitate their ready translation
- To provide generation of object code for existing and future computers without changing system tapes
- To include program debugging and testing features needed for quality and efficient performance

2.4.2 Evaluation

CMS-2 is rated higher than CLASP only in the category of Environment Interaction. This higher rating resulted from the fact that, although CMS-2 provides no real time task control features, it has an extensive input/output capability and can include separately compiled program modules in a single program organization. In the area of general Quality of Expression, CMS-2 rates lowest. It provides the least capability for higher level features such as matrix and vector operations and decision tables, and it imposes severe restrictions on the application of concepts it does provide. For example, logical operations apply only to bit-strings of length one.

In the category of Process Reliability, CMS-2 is rated higher than SPL because it does not provide some of the more error prone features of SPL. As in CLASP, the primary source of reliability is through reduced capability rather than any specific positive features.

CMS-2's Program Efficiency is ranked lower than SPL's because it includes only some of SPL's specific features. Storage efficiency is nearly equivalent but the execution time features are lacking. As in CLASP, the simplicity of the features and restriction on their usage may make it easier to generate more efficient object code.

Readability of the language suffers from some of the same shortcomings as its Quality of Expression in that no higher-level operations such as matrix/vector arithmetic are provided. In addition, CMS-2 eliminates the equal sign as a character of the language, which seems exceptionally unnatural for an algorithmic language.

3. GENERAL LANGUAGE REQUIREMENTS

The detailed language requirements, which are presented in Section 4, were driven by the more general requirements reflected in the evaluation criteria. From the wealth of features and capabilities provided by the four languages evaluated, those which proved beneficial to the benchmark problem programming, and were most consistent with the evaluation criteria, were selected as language requirements. A few additional capabilities were incorporated into the language requirements as dictated by either the benchmark programming problems or anticipated future requirements.

During the process of selecting capabilities from existing languages and adding new capabilities, four general requirements were imposed to supplement and emphasize the requirements inherent in the evaluation criteria. These requirements are introduced below and discussed in subsequent paragraphs:

- The scope of the programming activity to which the language addresses itself should be primarily onboard application programming. The need for highly reliable and very efficient onboard programs is better fulfilled by a language which restricts its scope to that area.
- The language should be easily applied to future programs whose requirements are not currently established in detail.
- The language features should be defined incrementally so that, for applications which do not need (or cannot support) the full power of the language, unwanted features can be easily omitted.
- The language design should encourage and enforce the generation of reliable source code, which is subject to thorough automatic testing.

3.1 Scope of Language

One of the major conclusions drawn from this study is that many of the features which make a language powerful, expressive, and easy to use, are the same features which tend to allow errors to be introduced into programs. Another conclusion is that considerable code optimization can and should be performed by the compiler in processing a higher level language for onboard programs. Because the

onboard programs generally must be very reliable and highly efficient, it is undesirable to burden the flight programming language with capabilities directed toward ground applications where reliability and efficiency are less critical. Similarly, it is overly restrictive to perform the ground applications programming with the reduced capability and stringent usage rules of the flight programming language.

For ground applications a more general, more powerful, language with less stringent usage rules should be employed, but there should be maximum compatibility between the flight language and the general purpose language. Largely for that reason, the requirements of Section 4 specify very little language syntax. They concentrate on capabilities the language must provide rather than details of how the statements must appear.

PL/I, as the most powerful general purpose language available and having a broad base of current users, is the prime candidate for a general purpose support language. However, the level of compatibility with PL/I is a separate requirement, independent of establishing the level of capability of the language.

Automatic checkout procedures represent a large source of future ground and spaceborne programming. However, much of this activity is omitted from the scope of the flight programming language. There will be some checkout related functions performed as an integral part of the operational flight program, and the flight programming language is expected to serve these. However, the specialized test sequences written solely for automatic checkout should be programmed in a language designed specifically for checkout.

3.2 Applicability to Future Programs

The following programs are representative of the programs to which a higher level language might be applied in the near future:

- Space Shuttle
- Space Station
- High Energy Astronomical Observatory

Review of computer processing requirements for these systems indicates that larger and more sophisticated onboard programmed functions are being proposed for future missions. Vehicle autonomy is dictating less dependence on ground support and, therefore, increased

activity and sophistication particularly in:

- o Guidance and Navigation
- o Attitude Reference and Control
- o Experiment Data Processing
- o Subsystem Redundancy Management
- o Checkout
- o Man-machine Interface

However, there is no significant indication of specific standard implementation techniques which point to new language operation requirements. Experiment data processing will require noise filtering and data compression methods; guidance and navigation functions will require more coordinate transformations and numerical integration; checkout and redundancy management will involve extensive decision making functions. These activities reinforce the need (already clearly established by the benchmark problem coding) for matrix and vector arithmetic and decision tables. However, beyond these functions and limit tests for checkout, no specific data manipulation techniques were sufficiently in demand to justify incorporation into the language.

Instead, the need is for a mechanism whereby the language user, when he does identify a generally applicable and specific technique, may define it into the language by himself. This capability is provided, to some degree, through a requirement for macro statement definitions in the language. Macro statement definitions will allow the programmer to create a new language statement with which he can describe the operation he wishes to perform and identify the data items he wishes to manipulate.

This need is also served to some degree by the compile time identifier which allows the language user to identify a frequently used phrase or portion of a statement with a single name. That name can then be used in program statements to represent, descriptively, a useful construction or combination of constructions in the language. This technique provides the language user with the flexibility to define some conceptual features in the form that is most useful to his activities, rather than relying on a rigid form imposed by the language design. Integers, which are simply fixed point numbers with no fractional part, and Boolean variables, which are nothing more than single-bit

bit-strings, are examples of such features. Because of the power and flexibility of this technique, the detailed language requirements of Section 4 have omitted any language features which are easily defined by the user as combinations of required features through compile time identifiers.

3.3 Incremental Definition of Features

The language requirements are directed toward the maximum capability expected to be useful and cost-effective. There are, however, future missions which will not require, nor even be able to use, some of the capabilities provided. Real time task scheduling and input/output operations are examples where the language might describe more capability than a particular onboard system will support. The requirements have been selected, and the language should be designed so that if language features cannot be supported they can be easily dropped from the language syntax without degrading the syntax of other statements or modifying their meanings.

3.4 Process Reliability Requirements

Process reliability is one of the evaluation criteria categories, and this paragraph is not intended to modify or add to any of those requirements. It simply emphasizes the importance that reliability had in selecting language requirements. Many of the language requirements are in the form of restrictions on the application of a capability, or redundant information which must be supplied in a statement, or a specific ordering of statements. These requirements enforce standardization of techniques, which reduces generation of errors, makes them more easily detected by the compiler or the human reader, makes program modules easier to modify reliably, and simplifies the transfer of responsibility for a program among different programming personnel.