NOAA Technical Memorandum NOS C&GS 1

SELECTION OF A GENERAL PURPOSE LANGUAGE AND A REAL-TIME LANGUAGE SUITABLE FOR SHIPBOARD DATA SYSTEM III

Rockville, Md.
January 1984

U.S. DEPARTMENT OF COMMERCE / National Oceanic and Atmospheric Administration / National Ocean Service

NOAA Technical Memorandum NOS C&GS 1

SELECTION OF A GENERAL PURPOSE LANGUAGE AND A REAL-TIME LANGUAGE SUITABLE FOR SHIPBOARD DATA SYSTEM III

Steven R. Barnum
Nautical Charting Division

Rockville, Md.
January 1984

UNITED STATES
DEPARTMENT OF COMMERCE
Malcolm Baldrige, Secretary

National Oceanic and
Atmospheric Administration
John V. Byrne, Administrator

National Ocean Service
Paul M. Wolff,
Assistant Administrator

Mention of a commercial company or product does not constitute an endorsement by the National Oceanic and Atmospheric Administration (NOAA), National Ocean Service (NOS). Use for publicity or advertising purposes of information from this publication concerning proprietary products or the tests of such products is not authorized.

CONTENTS

TABLES

SELECTION OF A GENERAL-PURPOSE LANGUAGE AND A REAL-TIME LANGUAGE SUITABLE FOR SHIPBOARD DATA SYSTEM III

Steven R. Barnum
Nautical Charting Division
Charting and Geodetic Services
National Ocean Service, NOAA
Rockville, Md.

ABSTRACT: All major general-purpose languages and all major real-time languages were surveyed and filtered based on the requirements of Shipboard Data System III. The languages which passed the filter were then evaluated in depth. At this time FORTRAN 77 is judged to be the best overall choice for general-purpose applications software. FORTRAN 77 with the real-time extensions (FORTRAN ISA 61.1) is judged to be the best choice for real-time applications software. When Ada becomes widely available, it will be the best choice for both the general applications language and the real-time language.

INTRODUCTION

Identification and selection of the most suitable programing language for Shipboard Data System III (SDS III) could save the National Ocean Service (NOS) a significant amount of money during the implementation phase and over the operational life of the system. The growing cost of ship operations, programers, and software maintenance to support hydrographic surveys demands a language that will increase the reliability of software, reduce software development time, and reduce the cost of software life-cycle maintenance. The purpose of this study is to identify the most suitable language for SDS III based on the language's technical advantages and disadvantages. The following three criteria were used to evaluate the languages: 1) development time and cost, 2) system effectiveness, and 3) software life-cycle maintenance cost. To judge the languages using the above management criteria, technical features were chosen which reflected those criteria.

The SDS III project development staff is using the structured design techniques of Yourdon and Constantine (1975) and De Marco (1978) to develop physical and logical models of the Data Acquisition Subsystem (DAS) and Data Processing Subsystem (DPS)

of SDS III. Structured design techniques are tools used to manage the design of large software projects. Software, which is designed using structured techniques, is easier to develop, verify, and maintain. The appropriate programing language for SDS III should support structured programing in order to gain the maximum benefits from the resultant structured model.

Many authorities believe that when a programer solves a problem, the programer thinks in terms of the programing language. If the language is ill-suited to the application, the programer will have a harder time solving the problem. A gross example would be to use a language designed for report generation to construct a space shuttle navigation program. A language tuned for the application provides the programer with the proper tools to build the program. The possible results of using an inappropriate language can be a delayed project, cost overruns, and increased costs of life-cycle software maintenance.

SDS III PERFORMANCE REQUIREMENTS

SDS III will perform more onsite (shipboard and field site) data verification, formatting, quality control, and report generation than is presently done. The effect of these actions will be to decrease the period of time between the date the data is collected and the date the data appears in various NOS published products. To accomplish this, SDS III has been divided into two parts: the Data Acquisition Subsystem (DAS) and the Data Processing Subsystem (DPS). Each part will have different operational requirements. The DAS will perform the data acquisition in real time, while the DPS will perform post processing of the data collected by the DAS. Limited data verification and error detection will be performed by the DAS, while full data verification, smoothing, and other data manipulation will be performed by the DPS. Most hydrography will be performed by survey parties that consist of a ship carrying up to four 29-foot launches. Each ship will have a DPS. Each launch, as well as the ship, will have a DAS. Both the DAS and the DPS will support the same general-purpose language.

To specify which languages to use, the requirements of the system must be known. The requirements are taken from the functional specification document (NOAA 1983). For a complete description of the operational and technical requirements, refer to the Operational Requirements Baseline (EG & G 1983) and the Functional Specification Document (NOAA 1983).

Data Acquisition Capabilities

The DAS must be capable of the following functions for which NOS-developed application programs will be necessary.

1) Acquire data from NOS-furnished sensor subsystems and a contractor-supplied system control device.

2) Record all raw data on magnetic media for later processing by the DPS.

3) Monitor data validity and notify the vessel operator of all malfunctions.

4) Filter raw navigation data and compute current vessel position in real time.

5) Provide the vessel helmsman and the DAS operator with steering guidance (distance and direction from planned track line), current raw data values (depth and navigation), bearing and distance to a point of reference, and current position, course, and speed.

6) Generate graphic displays of acquired data and historical nautical chart data.

Data Processing Capabilities

The DPS must be capable of the following functions for which NOS-developed applications software will be necessary.

1) Manage all project data for the survey party.

2) Transfer data to and from the DAS, Marine Center, NOS headquarters, and other computer systems.

3) Scan the raw data to assure quality and produce printed reports.

4) Interactively edit raw data.

5) Apply correctors to the raw data.

6) Identify data of hydrographic significance.

7) Interactively edit raw data correctors.

8) Perform geodetic and other utility computations.

9) Digitize graphic and handwritten data.

10) Generate printed data and text reports.

11) Generate graphic displays of acquired data and historical nautical chart data.

12) Perform historical and contemporary data comparisons.

METHODOLOGY

This study involved a literature search for articles on all major general-purpose and real-time languages, interviews with the managers of various Automatic Data Processing (ADP) groups

within NOS, and the calculation of a figure of merit to determine a quantitative measure of suitability for each language evaluated (Anderson and Schumate 1982). The language with the highest figure-of-merit was defined as the most suitable for SDS III applications.

Even though a language is standardized, variations of the language's implementation are sometimes very large. Since the SDS III computer has not been selected, no evaluations were possible for various versions of compilers. Thus, the evaluation of execution time, generated program size, and compile time efficiency could not be done. To determine the best compiler, the study of these parameters should be done when the computer is selected.

The first step was to set up a filter which would eliminate all the unsuitable languages from consideration. The languages which passed through the filter were then studied more closely. The filter was comprised of requirements from the functional specification document and from a list of desirable language features, produced by the SDS III staff, which should be present in the selected languages.

Quantifying The Language Features

In order to compare one language to another, a method was needed to quantify each language feature relative to its importance. The method used for estimating the relative importance of each language feature was taken from Introduction to Operations Research (Churchman 1957). A fundamental example from this book is quoted to illustrate the process used to assign weights to the various technical features.

> Suppose there are four pieces of wood of unequal length and no device is available for measuring them. Suppose further that we want to determine the relative length (not absolute length) of these four strips. One possible way would be to order the strips from the longest to the shortest and label the longest A, the next B, the next C, and the shortest D. Give A a value of 100% and estimate seperately B, C, and D what percentage of A's length they represent. Suppose we get the following results. B = 60%, C = 30%, and D = 20%. Now we can put B, C, and D end to end and compare A with the this combined length. If our initial estimates were correct, the result B + C + D would be equal to 110% of A. If this comparison reveals a discrepancy, some adjustment to the original estimates would be required. Next we compare A to B + C, and we would expect B + C to equal to 90% of A. This comparison would provide another check on the original estimates. Finally we would compare B to C + D and expect to find B to be 60/(30 + 20) or 120% of (C + D).

This procedure consists of a systematic check on relative judgments by a process of successive comparisons.

The features of each candidate language considered in this study were scored on a scale of 0 to 1. The scores were assigned by the author and were based on the information assimilated from

the references in this publication. The final score (figure of merit) was calculated by summing the products of each technical feature's relative weight and the language's score for that technical feature. The algorithm stated algebraically is

$$\text{Figure of Merit} = \sum_{i=1}^{N} WiSi,$$

where Wi is the weighting factor for the ith technical feature and Si is the score for the ith factor. The figures of merit produced for each language represent the relative suitability of the language to the needs of SDS III.

Selection Criteria

The criteria by which the languages were evaluated are divided into two areas: management and technical. The management criteria served as a guide in selecting the features which were then quantified. The three management criteria by which the programing languages were evaluated are development time and cost, system effectiveness, and life-cycle maintenance. Development time and cost should be minimized whenever it is not at the expense of life-cyle maintainability. This criterion favors technical features that facilitate software development, make projects easier to staff, and minimize additional software tool development (Anderson and Schumate 1982). System effectiveness is concerned with how well the end programs perform in the user environment. Features which promote the security of data, good algorithms, and minimize software bugs increase the reliability and effectiveness of the system. Life-cycle maintenance is concerned with how easy the code is to update and modify over the life of the system. Features which promote readability and modularity decrease life-cycle maintenance costs.

Technical features were then selected which reflect the management criteria discussed above. For the two different applications, data processing (the DPS) and real-time data acquisition (the DAS), the various technical criteria were ranked and weighted according to the needs of the application. For example, the need for systems programing in the general-purpose language is not as great as it would be for the real-time language where access to individual registers and bits is required. The ideal language for SDS III should contain at least all of the features mentioned below.

Data Type

The language should be strongly typed. A strongly typed language has each of the following qualities:

1) Every object has its own unique type.

2) Each type defines a set of values and a set of operations for the object.

3) In every assignment operation, the type of the assigned value and the type of the data object assigned to it must be equivalent.

The programing languages Pascal and Ada are examples of a strongly typed language. A program written in a strongly typed language is better protected against such programing errors as typographical errors which can be caught by the compiler at compile time rather than at run time. A strongly typed language increases program reliability and clarity (Young 1982).

The ability to assign ranges to the data types should also be present in the language. An example would be the declaration of the variable SECONDS, with an associated range of 0 to 59. If a value of 61 or any negative value were assigned to the variable SECONDS, the error would be caught by the compiler at compile time. The language should also permit the use of user-defined types. Data typing could be used in SDS III software to keep the various data objects (depths, ranges, positions) separate and prevent them from becoming accidentally mixed together. The type of operations that can be performed on the objects could also be specified; this prevents the performance of operations which are illegal for that data object.

Data Scope

The language must have the ability to limit the scope of its data. This feature is also a measure of reliability and program clarity. Data scope is concerned with how available data is to the whole program. The language should have a means of controlling the availability of data between program modules. This prevents one module from accidentally modifying data used in another module.

Control Structures

The language must provide a rich set of control constructs (WHILE, IF-THEN-ELSE, DO, CASE, etc.) for representing the commonly occurring types of program structure. The structures must be unambiguous so that the overall structure of the program can be easily understood. They must allow the programer to express algorithms in a straightforward and efficient manner. The control structures should end with a closing keyword (ENDIF,CONTINUE) and there should be only one entry and exit point (a principle of structured programing). The real-time languages should have the necessary constructs to build efficient real-time programs.

Data Structure

The language should have the ability to group sets of logically related data into single units. The language must be able to describe data at different levels in order to design programs in

a top down manner (Young 1982). Some examples of data structures are arrays, records, linked lists, trees, and sets.

Program Support Environment

The language should have a complete set of software tools to support the development and maintenance of software. The support tools include editors, linkers, librarians, project managers, and utility programs. The availability of software tools for a language can have a large influence on the productivity of programers and the quality of the software.

Readability

The ability to easily read and understand the code is required for maintenance and enhancements. The basic quality of readability is the ability to absorb the main concepts of a program by reading the program text only, without having to resort to flowcharts and written descriptions. The language's keywords and program structure affect the readability of a program. The person responsible for maintaining the program is frequently not the one who wrote the program, and the former must be able to understand it in order to modify the code efficiently.

Learnability

The language should be relatively easy to learn, or else the productivity of the programers may be decreased in the beginning of the project.

Simplicity

The language must be easily mastered. If the language is too large to master, the programer may resort to using only a subset of the language without utilizing its full potential. The language must be free of hidden restrictions and conditions. The basic rules of a language are usually easy to learn, but the associated lists of restrictions and conditions can be hard to remember (Young 1982).

Flexibility

The language should be flexible enough to allow the programer to express all the operations in a program without having to resort to machine code.

Systems Programing

Systems programing has to do with the maintenance, control, and supervision of computers. Systems programing requires the ability to handle interrupts and to manipulate words, bits, and registers with minimum use of assembly language. This feature is especially important for the real-time language.

Multitasking

The ability to run concurrent programs is required in both the DAS and DPS. The coding process and the effectiveness of the system can be improved when the language supports multitasking.

Extent of Use

The language should be in wide commercial use. A widely used language that is available on many different computers provides NOS with a large array of computers from which to choose.

Previous Programer Proficiency

A large number of programers already proficient in the selected language will lower development time and cost. If the programers are not familiar with the selected language, then the programers will need training and time to become proficient in the new language.

FILTERING OUT THE UNSUITABLE LANGUAGES

General-Purpose Languages

The following is a list of features SDS III requires of a general-purpose language. They were used to filter languages under consideration down to a candidate set. The language must:

1) follow a standard,

2) have the ability to call routines written in other languages (FORTRAN, COBOL, assembly),

3) have 14 decimal places of precision,

4) possess the ability to read sequential data files written in assembly and other languages,

5) be able to reach all peripheral devices,

6) have integer formats of 16 and 32 bits,

7) be in widespread use by the computer industry,

8) be clearly suitable for the application, and

9) have the ability to separately compile program modules.

The requirement that the language be standardized ensures that the completed software will be portable from one computer to the next. (This of course relies on how widely a language is implemented on different computers.) The requirement will save the cost of rewriting most of the software if it becomes

necessary to move to a different computer. An example within SDS III is that code written for the DAS should run on the DPS with few changes. The ability to call routines written in other languages is required so that existing software such as geodetic routines written by National Geodetic Survey (NGS) in the programing language PL/1, and Bathymetric Swath Survey System (BSSS) software written in FORTRAN 66 can be utilized. Since accuracy is very important to surveying, 14 decimal places of precision and integer formats of 16 and 32 bits have been specified. The requirements of precision and integer formats have been found to be necessary based on NOS experience. The ability to read files written in assembly and other languages is required so files written by the NOS Automated Information System (AIS) and the DAS real-time language can be read by the DPS. Since SDS III is concerned with the input and output of data, the language must be able to reach all of the input/output devices efficiently. To ensure a full range of candidate computers, the language must be in widespread use. The language must be clearly suitable for the application. A language designed for report generation is clearly not suitable. The language must support separate compilation of program modules to speed the software development process and to increase maintainabilty of the software. Software projects are often broken into subtasks with several programers responsible for different subtasks. If a language supports the separate compilation of program modules, then the programers can compile and test modules independently of the rest of the program. If the language does not support separate compilation of program modules, then all the program modules have to be completed before the program can be compiled and tested.

The following is the initial list of languages under consideration before the filter was applied.

1) Ada

2) ALGOL 68 (ALGOrithmic Language)

3) APL (A Programing Language)

4) Basic (Beginner's All-purpose Symbolic Instruction Code)

5) C

6) COBOL (Common Buisness Oriented Language)

7) FORTH

8) FORTRAN 77 (FORmula TRANslation)

9) HAL/S

10) JOVIAL (Jule's Own Version of the International Algebraic Language)

11) MODULA II (MODULAr programing II)

12) Pascal

13) PL/1 (Programing Language 1)

14) LISP (LISt Processing language)

The requirement that the language be clearly suitable to the the applications of SDS III was applied first. Basic, COBOL, APL, and LISP failed this criterion.

BASIC was conceived at Dartmouth College to teach nonscience majors about computers. BASIC was eliminated because it does not support structured programing and was not designed for scientific applications. COBOL was eliminated because it is a problem-oriented business data processing language and is not well equipped to handle the applications of SDS III which are computationally oriented. APL, a language which is criticized by many as being difficult to learn because of its extremely terse syntax, has not won favor with many people outside the engineering and scientific community. APL is also not well suited to the problem domain of SDS III, and for these reasons it was eliminated. LISP, heavily used in artificial intelligence applications, was deleted from the list because it is not well suited to applications other than list processing and general symbol manipulation. While it may be possible to construct applications software using these languages, the algorithms would probably be implemented in an awkward fashion and the resulting software would be difficult to maintain.

Next, the requirement that the languages be in widespread use was applied. Algol 68, HAL/S, JOVIAL, and MODULA II failed the criterion.

ALGOL 68 is popular language in the European computer industry. Because it has a very small following in the United States, it was eliminated from the list. The language HAL/S, developed by the National Aeronautical and Space Administration (NASA), is used for the space shuttle mission. HAL/S was eliminated because of its limited use outside of NASA. JOVIAL's use is limited primarily to the U.S. Air Force, and it is presently being replaced by the Department of Defense's (DOD) new language Ada. For these two reasons JOVIAL was eliminated. MODULA II is relatively new and is only implemented on smaller computers such as the DEC PDP-11, Sage, IBM personal computer, and Apple computer.

The requirement that the language be standardized was applied to results of the previous operation. FORTH was eliminated because there are no significant standardized versions of the language. The supporters of FORTH are divided into several groups, each having its own version of the language.

Next, the requirement that the language support separate compilation of program modules was applied to the results of the last operation. Pascal was eliminated because it does not support the separate compilation of program modules in its standard form.

The remaining languages, Ada, C, FORTRAN 77, and PL/1 were judged to meet all remaining requirements. An overview of these languages is included the appendix.

Real-Time Languages

The following is a list of features used to filter languages for the real-time application. They must

1) follow a standard,

2) have the ability to call routines written in other languages (FORTRAN, COBOL, assembly),

3) have the ability to randomly access mass storage files,

4) be able to reach all peripheral devices and have the ability to manipulate individual words, bytes, and bits,

5) be in widespread use by the computer industry,

6) be clearly suitable for the application, and

7) provide the ability to separately compile the program modules.

Characteristics 1, 2, 5, 6, and 7 are included in the real-time filter for the same reasons they were included in the general-purpose language filter. The language must have the ability of systems programing. A programer using a language with systems programing ability will not have to resort to the use of assembly language as often as a programer using a language that lacks this ability. The software written for BSSS, a real-time data aquisition and processing system, is composed of approximately 12,000 lines of FORTRAN code and 10,000 lines of assembly language code. Assembly language is machine dependent, and is difficult to develop, maintain, and debug. A language which can significantly reduce the use of assembly language is definitely preferred over a language which does not. These requirements were used to filter all the major real-time languages to a set which could be evaluated.

The following is the initial list of languages under study before applying the filter.

1) Ada

2) C

3) Edison

4) FORTH

5) FORTRAN ISA 61.1 (real-time extension)

6) Micro Concurrent Pascal

7) MODULA (MODULAr programing language)

8) MODULA II (MODULAr programing language II)

9) PEARL (Process and Experiment Automation Real-time Language)

10) DP

11) CSP

12) GYPSY

13) Path Pascal

The requirement that the language have the ability to separately compile program modules was applied first to the initial list. The languages that failed were Edison, Micro Concurrent Pascal, MODULA, DP, CSP, GYPSY, and Path Pascal.

The requirement that the language be in widespread use was applied next to the results of the previous operation. PEARL, a relatively popular real-time language in Europe, was eliminated because it is implemented on very few computers in the United States.

The requirement that the language be standardized was applied next to the results of the last operation. FORTH was eliminated because it is not standardized to any extent.

The requirement that the language be able to call routines written in other languages was applied next. Modula II is only capable of calling routines written in MODULA II, assembly, and Pascal; therefore, MODULA II was eliminated.

The remaining languages, Ada, C, and FORTRAN ISA 61.1 passed all remaining tests. An overview of each language is provided in the appendix.

RATING THE GENERAL-PURPOSE LANGUAGES

Weights, Scores, and Figures of Merit

The following procedure was used to assign weights to the technical features. The technical features were initially ranked according to their estimated importance. In order to provide an arbitrary reference for comparison, one feature, the so-called standard outcome (Fs), was selected at random from the set. The features were randomly subdivided into equal-sized groups of no more than four. The standard outcome was added to each group. The features of each group were assigned initial values of importance (Vi) in relation to the standard outcome (Vs = 1) based on the author's perception of SDS III requirements. The features of each group were then evaluated against each other by comparing the feature with the largest Vi, say V1, against the

combined importance of the remainder of the set: F1 versus (F2 AND F3 AND F4 AND Fs). If F1 was found to be more important than (F2 AND F3 and F4 and Fs), then V1 was adjusted so that V1 > V2 + V3 + V4 + Vs. If F1 was not judged more important than the sum of the others in the group, then the values in the group were adjusted, if necessary, to reflect that fact. The top ranking feature of each group was then eliminated from consideration, and the process of comparison was repeated for each group using the next most highly rated feature against the remainder: F2 versus (F3 AND F4 AND Fs). The process was iterated one more time in similar fashion.

The values obtained for all features were then compared to the original order of importance. If the computed values conflicted with the original ranking, and the original order was judged incorrect, the order of importance was modified to reflect the computed values and the procedure was repeated. If the order was judged to be correct, then the values were adjusted in their respective groups. This process was repeated until consistent results were obtained. In order to obtain a set of weights for the calculation of a figure of merit, the values were normalized to sum to a value of 1,000 by dividing each value by the sum of all values and multiplying by 1,000.

The features were weighted in the following order.

FEATURE	WEIGHT
1) Data type	164.6
2) Control structures	146.3
3) Program support environment	134.1
4) Readability	109.7
5) Data structure	91.5
6) Data scope	73.2
7) Extent of use	61.0
8) Learnability	48.8
9) Previous programer experience	42.7
10) Simplicity	42.7
11) Flexibility	36.5
12) Systems programing	30.5
13) Multitasking	18.2

Each language feature was then assigned a score in the range of 0 to 1 according to how well it fulfilled the ideal language requirements discussed under Selection Criteria. The scores were

references and the bibliography. Table 1 shows the individual scores and figures of merit.

Results

Table 1 indicates Ada with 710 points is the clear winner. This is not surprising since Ada was designed from the beginning to embrace all three management criteria. Second, at 546 points, is FORTRAN 77 which scored very strongly in program support environment, extent of use, and previous programer proficiency. Third, at 483 points, is PL/1 which scored slightly better than FORTRAN 77 in the areas of data typing, control structures, and data structure. PL/1 lost points in other areas such as program support environment, simplicity, and readability. Last, with 472 points, was C which had mostly average scores except for data type (score of 0.3) and systems programing (score of 0.9).

The process of weighting and scoring the features was iterated several times to confirm the results. Ada consistently came out on top by at least a margin of 125 points. The other languages remained relatively close together, usually within 75 points of each other. Some place changes occurred among the three according to how the weights were applied, but the most common ranking was 1) Ada, 2) FORTRAN 77, 3) PL/1, and 4) C.

RATING THE REAL-TIME LANGUAGES

Weights, Scores, and Figures of Merit

A new set of weights for technical features was composed for real-time requirements. The weighting process was executed in the same manner as for the general-purpose languages. The features were weighted in the following order.

FEATURE	WEIGHT
1) Systems programing	158.8
2) Data type	123.5
3) Control structures	117.6
4) Program support environment	111.8
5) Previous programer experience	88.2
6) Multitasking	70.5
7) Data structure	58.8
8) Extent of use	58.8
9) Data scope	52.9
10) Learnability	47.7

10) Learnability.....47.7

11) Readability.....47.7

12) Flexibility.....41.2

13) Simplicity.....23.5

The languages were scored on a scale of 0 to 1 according to how well they fulfilled the ideal language requirements discussed under Selection Criteria. Table 2 shows the individual scores and figures of merit.

Results

Ada was the superior language, scoring 685 points. Again, this was not surprising because Ada was designed to be a manageable real-time language. FORTRAN 77 with the ISA real-time extensions was rated second with a figure of merit of 566 points; it scored highly in the areas of extent of use, program support environment, and previous programer experience. The language C placed third with a figure of merit of 468 points. It lost some points because of its poor data typing ability, lack of real-time constructs, and lack of previous programer experience.

The process of weighting and scoring the features was repeated several times to verify the results. Ada consistently came out ahead, followed by FORTRAN, and C.

RECOMENDATIONS

From the results of the rating processes, Ada is the most suitable language for both the general-purpose and the real-time applications software. However, as stated in the Introduction, Ada is not readily available. It was included in the study to keep this report from becoming outdated immediately. It is expected that many more compilers and software tools will become available for Ada as the date set by the DOD approaches (DOD has specified that all advanced work will use Ada by January 1984 and all full-scale engineering work will use Ada by July 1984). Programing for the SDS III project is expected to begin somewhere in the summer of 1984. There is certainly not much time for the industry to test the capabilities of the language, but Ada is also said to be the most carefully designed and implemented language to date. The U.S. Air Force and the Intellimac Corporation have already begun using Ada to take advantage of the long-range benefits of portable, effective, and maintainable software. Soon Ada should be the language of choice for SDS III applications software.

If, however, Ada is not ready and available, then FORTRAN 77 is the next best choice for general-purpose applications software. Care must be taken with FORTRAN because there are many features present in the language that go against the philosophies of structured programing (arithmetic IF, GOTO, and COMMON). There are also many features which can keep the software from being

portable. An excellent article by Larmouth (1981) describes the features within FORTRAN that should be avoided to keep the software portable. FORTRAN 77 with the standard real-time extension (FORTRAN ISA 61.1) is recommended as the real-time applications language if Ada is not ready. When FORTRAN is used to develop the real-time software, the same precautions must be made to avoid the nonportable FORTRAN 77 features.

To produce effective, maintainable software, a set of rules for controlling software development should be created and enforced. The book, Managing The Structured Techniques by Yourdon (1975), presents some ways of managing software development. Some examples of what the rules should include are:

1) The use of COMMON, arithmetic IF statements, and the GOTO statement should be restricted from use (the GOTO statement may be necessary in some instances, but should be used only as a last resort).

2) Every variable should be explicitly declared, and all constants should be declared to be of type parameter.

3) All values passed into a subroutine should be explicitly declared.

4) A section should be provided at the head of each routine that explains what the routine does.

5) Each variable, constant, array, etc., should be accompanied by a definition as to what it represents.

These are just some of the rules that must be implemented to ensure that the resulting FORTRAN code will be portable, effective, and easy to maintain.

ACKNOWLEDGMENTS

I wish to express thanks to Gary Guenther, David Enabnit, and the entire Applied Technology Group for their support in reviewing this paper.

REFERENCES

Anderson, G. E., and Schumate, K. C., 1982: Selecting a Programming Language, Compiler, and Support Environment. Computer, August.

Churchman, C. West., Ackoff, Russell L., and Arnoff, leonard E., 1957: Introduction to Operations Research. John Wiley & Sons, New York.

De Marco, Tom., 1978: Structured Analysis and System Specifacation. Prentice Hall, Englewoods Cliff, NJ.

Eg & G Washington Analytical Services, 1983: Operational Requirements Baseline for Shipboard Data System III, Rockville, MD., May 27.

Kernighan, B. W., and Ritchie, D. M., 1978: The C Programming Language, Prentice Hall, Englewoods Cliff, NJ.

Larmouth, J., 1981: Fortran 77 Portability. Software Practice and Experience, 11.

NOAA., 1983: Functional Specification Document. Nautical Chart Branch, NOS. Request For Proposals, NA-84-RFP-00002.

Yourdon, Edward., 1976: Managing The Structured Techniques. Prentice Hall, Englewood Cliffs, NJ.

Yourdon, Edward., and Constantine, Larry L., 1975: Structured Design: A Discipline of Computer Design. Prentice Hall, Englewoods Cliff, NJ.

Young, Stephen J., 1982: Real-Time Languages: Design and Development. Ellis Harwood, Halstead Press (John Wiley & Sons), New York.

BIBLIOGRAPHY

Abbott, R. J., and Moorhead, D. K., 1982: Software Requirements and Specifications: A Survey of Needs and Languages. The Journal of Systems and Software, 2.

Alford, Mark W., 1977: A Requirements Engineering Methodology for Real-Time Requirements. IEEE Trans. on Software Engineering, VOL. SF-3, NO. 1.

Babich, Wayne., Simpson, Richard., and Thall, Richard., 1981: The Ada Language System. Computer, June.

Barron, D. W., 1981: PASCAL - The Language and its Implementation. John Wiley & Sons, Chicester.

Boom, H. J., and De Jong, E., 1980: A Critical Comparison of Several Programming Language Implementations. Software - Practice and Experience, 10.

Brender, Ronald F., and Nassi, Issac R., 1981: What is Ada?. Computer, June.

Brooks, Frederick P., 1975: The Mythical Man-Month: Essays on Software Engineering. Addison-Wesley.

Bulman, David M., 1982: Is Ada the Answer?. The Yourdon Report, 6-6, 7-1.

Carlson, WilliamE., 1981: Ada: A Promising Beginning. Computer, June.

Carlson, William E., 1980: Introducing Ada. ACM Proceedings.

Cherry, George W., 1982: Developing Software With Ada. Seminar notebook, The U. S. Professional Development Institute, 1982.

Dickson, Christine E., 1980: FORTRAN-80 Mixes With Other Languages to Strengthen its Real-Time Powers. Electronic Design, January 4.

Dickson, Christine E., 1979: Modernized FORTRAN Combines Powerful I/O With A Structure Tailored for Engineering. Electronic Design, September 13.

Dickson, Christine E., 1979: Solve Number Crunching Problems With Modernized FORTRAN Programs. Electronic Design, November 22.

Dijkstra, E. W., and Hoare, C. A. R., 1972: Structured Programming. Academic, New York.

Evanczuck, Stephen., 1983: Real-Time Operating Systems. Electronics, March 24.

Fawcette, James E. (Editor), 1982: Ada Goes To Work. Defense Electronics, June.

Feldman, Jerome., 1979: Programming Languages. Scientific American, December.

Fever, Alan., and Gehanni, Marain H., 1982: A Comparison of the Programming Languages C and Pascal. ACM Computing Surveys, 14, 1, March.

Freitas, Robert A., and Carlson, Patricia A., (editors), 1983: Computer Science: Key To A Space Program Renaissance. Institutional Needs, Status, and Recomendations, 1.

Fulton, Cynthia., and Whiffen, Richard., 1980: High-Level Languages Take on Most of Real-Time System Software. Electronics, December 4.

Geller, Dennis., 1983: Coding in Two Languages Boosts Program Reliability. Electronics Design, March 31.

Ghezzi, Carlo., and Jazayeri, Mehdi., 1983: Languages: Which One For You?. Software News, April.

Ghezzi, Carlo., and Jazayeri, Mehdi., 1983: Languages: Which Ones (and how many) For You?. Software News, May.

Ghezzi, Carlo., and Jazayeri, Mehdi., 1983: Languages: Which Ones (and how many) For You. Software News, July.

Gilbreath, Jim., 1981: A High-Level Language Benchmark. BYTE, September.

Gligor, Virgil D., 1983: Basic Technologies for Real-Time System. Final Report, NOAA Contract No. NA-82-SAC-00648, International Software Systems Inc., College Park, MD.

Hall, Edwin., 1981: Extended Pascal Adds Real-Time Multitasking to LSI-11 Family. Electronic Design, November 26.

Hancock, Les., and Kreiger, Morris., 1982: The C Primer. McGraw-Hill, New York.

Hanson, D. K., 1981: Is Block Structure Necessary ?. Software - Practice and Experience, August.

Hartrich, Ford., 1981: FORTRAN Can Beat Pascal in Control Applications. Electronic Design, July 23.

Hecht, Herbert., 1981: Final Report: A Survey Of Software Tool Usage. Computer Science and Technology, NBS publication 500-82, Washington D. C.

Hinden, Harvey J., and Wendy J., 1983: Real-Time Systems. Electronic Design, January 6.

Hogan, Thom., 1982: Discover FORTH: Learning and Programming The Forth Language, Mcgraw-Hill, Berkeley, CA.

Houghton, Raymond C., 1983: Software Development Tools: A Profile. Computer, May.

Hunt, James W., 1982: Programming Languages. Computer, April.

Jensen., K., and Wirth, N., 1975: Pascal User Manual and Report. Springer Verlag, New York, New York.

Johnson, R. Colin., 1981: Special Report: Ada, The Ultimate Language. Electronics, February 10.

Lamie, Edward L., 1982: PL/1 Programming: A Structured Approach. Wadsworth Publishing Co., Belmont, CA.

Levanthal, Lance A., 1978: Introduction to Microprocessors: Software, Hardware, Programming. Prentice Hall, Englewoods Cliff, NJ.

Linhart, Jason., 1983: Managing Software Development With C. BYTE, August.

Macdonald, George., (Canadian Hydrographic Service, Burlington, Ontario, Canada) 1983: Automation Today - Scratching The Twenty-One Year Itch. (unpublished manuscript).

Martin, Thomas., 1978: Real-Time Programming Language PEARL - Concept and Characteristics, IEEE Cat. No. CH 1338-3/78/0000-0301.

Martin, Thomas., 1979: "PEARL at the Age of Three." proceedings of the International Conference on Software Engineering. IEEE Cat. No. CH 1479-5/79/000-0100.

Merchant, Michael J., and Sturgal, John R., 1977: Applied FORTRAN Programming With Standard FORTRAN, WATFOR, WATFIV, and Structured WATFIV.

Metzger, Philip W., 1973: Managing A Software Project. Prentice Hall, Englewoods Cliff, NJ.

Mills, Harlan D., 1982: The Calculus of Computer Programming. Allyn & Bacon Inc., Boston, MA.

Madia, Andrew., 1983: To Pascal Interpeter, uC's on-chip ROM is Home. Electronics, July 21.

Montgomery, Charles., 1983: Pascal System Plugs Holes in uC Programming. Electronics Design, July 21.

Perrot, R. H., and Dhillon, D. S., 1981: An Experiment With FORTRAN and Pascal. Software - Practice and Experience, 11.

Samnet, Jean F., 1969: Programming Languages. Prentice Hall, Englewood Cliffs, NJ.

Department of Defense, 1980: Reference Manual for the Ada Programming Language - Proposed Standard Document.

Rifkin, Edward M., And Williams, Steve., 1983: The C Language: Key to Portability. Computer Design, August.

Ripps, David L., 1983: Multitasking OS Manages A Team of Processors. Electronic Design, July 21.

Roberts, Bruce., 1983: The C Language. BYTE, August.

Ross, Douglas T., and Schoman, Kenneth E., 1977: Structured Analysis for Requirements Definition. IEEE Transactions on Software Engineering, Vol. SF-3, No. 1, January.

Ross, Douglas T., 1977: Reflections on Requirements. IEEE Transactions on Software Engineering, Vol. SF-3, No. 1.

Schindler, Max., 1983: Real-Time Languages Speak to Control Applications. Electronic Design, July 21.

Shaw, Mary., Alme, Guy T., Newcomer, Joseph M., Reid, Brian K., and Wuif, Wm. A., 1981: A Comparison of Programming Languages for Software Engineering. Software - Practice and Experience, 11.

Spector, David., (Prime Computer, Framingham, MA.): Ambiguities and Insecurities in Modula II. (unpublished manuscript).

Spencer, Bill., 1983: C's Pointer Mechanism Raises System Throughput. Electronic Design, July 7.

Stenning, Vic., Froggart, Terry., Gilbert, Roger., and Thomas, Ellis., 1980: The Ada Environment: A Perspective.

Stotts, Paul David., (Department of Applied Mathematics and Computer Science, University of Virginia, VA.): A Comparitive Study of Concurrent Programming Languages. 21pp. (unpublished manuscript).

Sumner, Roger T., and Gleaves, R. E., (Volition Systems, P.O. Box 1236, Del Mar, CA. 92014): Modula II - A Solution to Pascal's Problems. (unpublished manuscript).

Thomas, Rebecca., 1983: What is a Software Tool. BYTE, August.

Welsh, J., Sneeringer, M, J., and Hoare, C. A. R., 1977: Ambiguities and Insecurities in Pascal. Software - Practice and Experience, 11, 6.

White, James Wm., 1983: Real-Time FORTRAN. Real-Time, Mellichamp, Van Nostrand.

Whitney, Al., and Conrad, Marvin C., 1983: Call FORTH For Real-Time Control Programming. Computer Design, April 21.

Wills, J., 1980: Computer Languages In Perspective. Electronic Engineering, May.

Zeigler, Stephen., Allegre, Nicole., Johnson, Robert., and Morris, James., 1981: Ada for the Intel 432 Microcomputer. Computer, June.

Zvegintzov, Nicholas., 1983: Nanotrends. Datamation, August.

Table 1--Computations of the figures of merit for the general purpose languages

Figure of Merit = FOM
FOM = Weight * Score

Technical Feature	Weight	Ada Score	FOM	C Score	FOM	FORTRAN 77 Score	FOM	PL/1 Score	FOM
Data type	164.6	.9	148.14	.3	49.38	.2	32.92	.3	49.38
Control structures	146.3	.85	124.355	.6	87.78	.4	58.52	.6	87.78
Program support environment	134.1	.8	107.28	.5	67.05	.95	127.395	.6	80.46
Readability	109.7	.8	87.76	.3	32.91	.7	76.79	.5	54.85
Data structure	91.5	.85	77.775	.7	64.05	.4	36.6	.7	64.05
Data scope	73.2	.9	65.88	.4	29.28	.4	29.28	.6	43.92
Extent of use	61	.1	6.1	.4	24.4	.9	54.9	.4	24.4
Learnability	48.8	.3	14.64	.6	29.28	.9	43.92	.5	24.4
Previous programer exper.	42.7	0	0	0	0	.95	40.565	.2	8.54
Simplicity	42.7	.2	8.54	.6	25.62	.6	25.62	.3	12.81
Flexibility	36.5	.8	29.2	.8	29.2	.4	14.6	.5	18.25
Systems programing	30.5	.8	24.4	.9	27.45	.1	3.05	.4	12.2
Multitasking	18.2	.9	16.38	.3	5.46	.1	1.82	.1	1.82
Total Figures of Merit			710.45		471.86		545.98		482.86

Table 2--Computations of the figures of merit for the real-time languages

Figure of Merit = FOM
FOM = Weight * Score

Technical Feature	Weight	Ada Score	FOM	C Score	FOM	FORTRAN/ISA Score	FOM
Systems programing	158.8	.8	127.04	.9	142.92	.5	79.4
Data type	123.5	.9	111.15	.3	37.05	.2	24.7
Control structures	117.6	.85	99.96	.3	35.28	.6	70.56
Program support environment	111.8	.8	89.44	.5	55.9	.95	106.21
Previous programer exper.	88.2	0	0	0	0	.5	44.1
Multitasking	70.5	.9	63.45	.3	21.15	.6	42.3
Data structure	58.8	.85	49.98	.7	41.16	.4	23.52
Extent of use	58.8	.1	5.88	.4	23.52	.7	41.16
Data scope	52.9	.9	47.61	.4	21.16	.4	21.16
Learnability	47.7	.3	14.31	.6	28.62	.9	42.93
Readability	47.7	.8	38.16	.3	14.31	.7	33.39
Flexibility	41.2	.8	32.96	.8	32.96	.6	24.72
Simplicity	23.52	.2	4.704	.6	14.112	.5	11.76
Total Figures of Merit			684.644		468.142		565.91

APPENDIX

Overview Of The Candidate Languages

This appendix provides a brief overview of each language that met the selection criteria described under METHODOLOGY. In addition, an example program accompanys each discussion. The programs are an implementation of the Sieve of Eratosthenes algorithm, which computes all of the prime numbers from 3 to 16,000. This algorithm, taken from the September 1981 issue of BYTE, can be used as a benchmark for comparing programing languages.

Ada

Ada was developed by the DOD in response to the growing cost of software maintanence. The preliminary work on Ada began in 1975 with a series of competitions in which Honeywell Bull CTI was the final winner. DOD has set out to standardize the language before its official release. This is quite a different route from that taken by other languages. In the past, languages were introduced to the market and would become standardized several years later. DOD has trademarked the name "Ada" so that no vendor may sell a compiler using the Ada name unless the compiler has passed the validation test. This ensures that Ada code will remain portable by keeping nonstandard Ada compilers off the market.

Ada was originally targeted for real-time, embedded systems (systems dedicated to one particular purpose, e.g., fire and guidance control for tanks, missles, and airplanes), but it has grown during development to include all programing environments - business, scientific, and systems programing. As a result, Ada has been critized as being much too large for an individual programer to handle. Ada supports structured programing and provides many checks to help prevent "bad" programing practices. The language is said to make programers code "the Ada way".

Ada provides built-in data types, and the precision of numeric data can be controlled by the programer. Ada is designed to be a very secure language (less prone to programing errors) in that certain operations are declared to be associated with certain data types in the header of the program. For instance, the variables HEIGHT and WIDTH can be declared to be of type INCHES, whereas variable AREA is declared to be of type SQUARE INCHES. Then:

```
function"*"(height:INCHES;width:INCHES)
  return SQUARE INCHES;
```

requires that all multiplication of inches times inches have results in square inches. The compiler will check to make sure that this rule is adhered to. This is in strong contrast to most languages which do not allow user-defined types. Ada provides control structures similar to Pascal, a specialized exit statement to break loops, and the GOTO statement. The language provides a rich set of unit-level control structures such as procedures, function calls, exceptions, and concurrent

activations. Ada, like PL/1, has the ability to recover from execution-time errors. Whenever an error is encountered, an exception is raised. Several such as CONSTRAINT ERROR AND NUMERIC ERROR are predefined. To prevent program execution from stopping when the exception is raised, a "handler" can be attached by the user to recover from the error.

Ada is still in its infant stages. Some authorities say that Ada will become "the" language, and others say that it will find its niche and remain there. Only two compilers at present have been validated with many others in waiting. Almost every major computer manufacturer is working on an Ada compiler or Ada software tools.

The following is an implementation of the Sieve of Eratosthenes in Ada:

```
-- A "Sieve of Eratosthenes" program written in Ada.
--
with TEXT IO;
procedure SIEVE_OF_ERATOSTHENES is
--
-- Declaration of objects
--
SIZE : constant INTEGER := 8_190;
type PRIME ARRAY is array (0 .. SIZE) of BOOLEAN;
COUNT, PRIME : INTEGER_OF_INTEREST;
--
--Procedure body
--
begin
  package INT_IO is new TEXT_IO. INTEGER_IO
  (INTEGER_OF_INTEREST);
  TEXT_IO.PUT_LINE ("10 iterations");
  for ITER in 1 .. 10 loop
    count := 0;
    FLAGS := (0 .. SIZE => TRUE);
    for iter in 1 .. 10 loop
      if Flags(I) then
        PRIME := I + I + 3;
        K := K + PRIME;
        while K <= SIZE loop
          FLAGS(K) := FALSE;
          K := K + PRIME;
        end loop;
        count := count + 1;
        -- INT_IO.PUT (PRIME); -- For debugging
        -- TEXT_IO.NEW_LINE (SPACING => 1); -- For debugging
      end if;
    end loop;
      INT_IO.PUT (COUNT);
      TEXT_IO.PUT_LINE ("primes");
end SIEVE_OF_ERATOSTHENES;
```

C

The programing language C was developed by D.M. Ritchie and B. Kernighan at Bell Labrotories in 1972 on a PDP-11 computer. Though no formal standard exists, C does follow a "de facto" standard as defined in the book, "The C Programming Language" (Ritchie and Kernighan 1978). The language contains both high-level and low-level features and has been termed a mid-level language. It is supported by a full set of development tools under the UNIX operating system.

The largest application of C has been in systems programing (writing compilers, editors, and operating systems) where it is necessary to specify particular addresses or registers. Its popularity has increased in the past few years, especially with software vendors, because of its portability, efficiency, and ease of use. A strong feature of C is its use of primitive elements to construct powerful functions. With this feature it is possible to construct functions which are not present in the language itself. These functions can then be stored in a library as a standard for use by all programers. An example would be functions which would overcome the primitive input/output abilities of C. Instead of giving the programer every special feature, C provides the basic building blocks which can be assembled to perform many varied tasks.

C can specify the precision of integers and real numbers; data types can be aggregated by arrays, structures, and unions; and type conversions are applied freely and automatically. The control structures REPEAT, WHILE, and FOR are provided with the possibility of exit from a loop with the use of the BREAK statement. A limited form of the CASE statement is provided, and the general GOTO statement is available. Functions and procedure calls are the only unit-level control structures. Readability can be a problem because of the existence of many different ways of stating the same concept and the possibility of producing extremely terse code. Unlike Ada, automatic type conversions are allowed in C.

The following is an implementation of the Sieve of Eratosthenes in C:

```
/*Eratosthenes Sieve Prime Number Program in C */
#define true 1
#define false 0
#define size 8190
#define sizepl 8191

          char flags[sizepl];

main() [
        int i, prime, count, iter;

        printf("10 iterations/n");
        for (iter = 1; iter <= 10; iter ++) [
             count = 0;
             for (i = 0; i <= size; i ++)
```

```
            flags[i] = true;
        for (i = 0; i <= size; i ++) [
            if (flags[i]) [
                prime = i + i + 3;
                k = i + prime;
                    while (k <= size) [
                        flags[k] = false;
                        k += prime;
                        ]
                    count = count + 1;
                ]
            ]
        ]
    printf ("/n%d primes", count);
    ]
```

FORTRAN 77

FORTRAN is the oldest language evaluated in this report, and is probably the best known. FORTRAN was developed by IBM in 1954 to satisfy the needs of the scientific community. It has been through several standardizations, the latest one called FORTRAN 77. This latest version incorporates some of the structured design principles developed after its previous standardization in 1966.

The most notable change for FORTRAN 77 is the addition of the IF, THEN, ELSE structure otherwise known as the "block if". This eliminates the need for the arithmetic IF and reduces the need for the use of the GOTO statement with the IF statement. In addition, the ability to handle character strings and to open and close files easily has been added.

FORTRAN has a very large support environment. This is due to FORTRAN being the oldest and most popular language among the scientific community. (Because NOS has been using FORTRAN for its applications in such areas as AIS, BSSS , Marine Center data processing programs, and Sea Beam for many years, there exists within NOS a large pool of programers familiar with FORTRAN.)

FORTRAN is not a secure language, and it allows programers to write confusing programs, especially with the use of the EQUIVALENCE and COMMON features. Automatic conversion of variable types is performed by FORTRAN. Type checking between subroutines is not performed. In keeping with the ANSI standard, all new versions must encompass the old versions. This ensures that existing FORTRAN software will work with the new version. All the language features which are now considered "bad" are kept in the latest version. Existing NOS FORTRAN programs could be transferred to the new SDS III computers if the code did not utilize any language extensions or calls to the operating system. Much of this software, however, uses nonstandard extensions and calls to the operating system. A significant amount of work would be required to revise the software before it could run on a different computer.

FORTRAN can be extended to take on real-time tasks with the use of the extension FORTRAN ISA 61.1. The standards were developed

by a Purdue workshop to standardize a set of real-time subroutine interfaces. The subroutines include the ability to manipulate bits, schedule tasks with the operating system, and provide access to the clock.

The following is an implementation of the Sieve of Eratosthenes written in FORTRAN.

```
* Eratosthenes Sieve Prime Number Program in FORTRAN
      LOGICAL FLAGS (8191)
      INTEGER I, PRIME, K, COUNT, ITER
*
* PERFORM 10 ITERATIONS OF THE SIEVE
*
      DO 92 ITER = 1, 10
        COUNT = 0
*
* INITIALIZE THE ARRAY TO TRUE
*
        DO 10 I = 0, 8190
          FLAGS = .TRUE.
10      CONTINUE
*
* FIND ALL THE PRIME NUMBERS FROM 0 TO 8190
*
        DO 91 I = 0, 8190
          IF FLAGS(I) THEN
            PRIME = I + I + 3
            K = I + PRIME
*
20          IF K .LE.  8190 THEN
              FLAGS(K) = .FALSE.
              K = K + PRIME
              GO TO 20
            ENDIF
          ENDIF
*
          COUNT = COUNT + 1
*
91      CONTINUE
92    CONTINUE
*
      WRITE (1, 200) COUNT
200   FORMAT (1X, I6, ' PRIME')
*
      STOP
      END
```

PL/1

PL/1 was introduced by IBM in 1965 as "the" programing language. It was designed for a wide variety of applications (business, scientific, etc.). PL/1 represents an attempt to incorporate the best features of FORTRAN, COBOL, and Algol 60 into a unique multipurpose language.

IBM put considerable effort into making this language a success much like DOD is doing with Ada, although the situation is slightly different since DOD is the one buying, not selling. Acceptance of PL/1 was not as great as IBM would have liked because few companies were willing to change languages. It has not made any great inroads into the business (COBOL) or scientific (FORTRAN) communities. The full PL/1 language is very large and has been scaled down to a subset known as PL/1 subset G. This subset was standardized in 1981 by ANSI and has been implemented on a number of different computers. IBM of course offers the most implementations of PL/1 on its own computers.

PL/1 provides built-in data types for which a variety of attributes may be specified (base, precision, bit, and picture). Aggregate constructors include structures (record), arrays, and pointers. The language also provides the control structures IF, THEN, ELSE, WHILE, and the GOTO statement. As far as supporting structured programing, PL/1 is one of the better languages available in the industry today.

PL/1 seems to be "holding on" but is not making any great gains in industry acceptance, especially with the advent of Ada. The following is an implementation of the Sieve of Eratosthenes program written in PL/1.

```
/* Eratosthenes Prime Number Program in PL/1*/
prime:
        proc options(main);
        #replace
                size  by 8190,
                false by '0'b,
                true  by '1'b;
        dcl
                flags (0:8191) bit(1),
                (i, prime, k, count, iter) fixed;

        put list('10 iterations');
                do iter = 1 to 10
                count = 0;
                        do i = 0 to size;
                        flags(i) = true;
                        end;
                        do i = 0 to size;
                        if flags(i) then
                                do;
                                prime = i + i + 3;
                                k = i + prime;
                                        do while (k <= size);
                                        flags(k) = false;
                                        k = k + prime;
                                        end;
                                count = count + 1;
                                end;
                        end;
                end;
        put skip list(count, 'primes');
        end prime;
```

NOAA SCIENTIFIC AND TECHNICAL PUBLICATIONS

The National Oceanic and Atmospheric Administration was established as part of the Department of Commerce on October 3, 1970. The mission responsibilities of NOAA are to assess the socioeconomic impact of natural and technological changes in the environment and to monitor and predict the state of the solid Earth, the oceans and their living resources, the atmosphere, and the space environment of the Earth.

The major components of NOAA regularly produce various types of scientific and technical information in the following kinds of publications:

PROFESSIONAL PAPERS—Important definitive research results, major techniques, and special investigations.

CONTRACT AND GRANT REPORTS—Reports prepared by contractors or grantees under NOAA sponsorship.

ATLAS—Presentation of analyzed data generally in the form of maps showing distribution of rainfall, chemical and physical conditions of oceans and atmosphere, distribution of fishes and marine mammals, ionospheric conditions, etc.

TECHNICAL SERVICE PUBLICATIONS—Reports containing data, observations, instructions, etc. A partial listing includes data serials; prediction and outlook periodicals; technical manuals, training papers, planning reports, and information serials; and miscellaneous technical publications.

TECHNICAL REPORTS—Journal quality with extensive details, mathematical developments, or data listings.

TECHNICAL MEMORANDUMS—Reports of preliminary, partial, or negative research or technology results, interim instructions, and the like.

Information on availability of NOAA publications can be obtained from:

PUBLICATION SERVICES BRANCH (E/AI13)
NATIONAL ENVIRONMENTAL SATELLITE, DATA, AND INFORMATION SERVICE
NATIONAL OCEANIC AND ATMOSPHERIC ADMINISTRATION
U.S. DEPARTMENT OF COMMERCE

Washington, DC 20235