

AD-A173 393

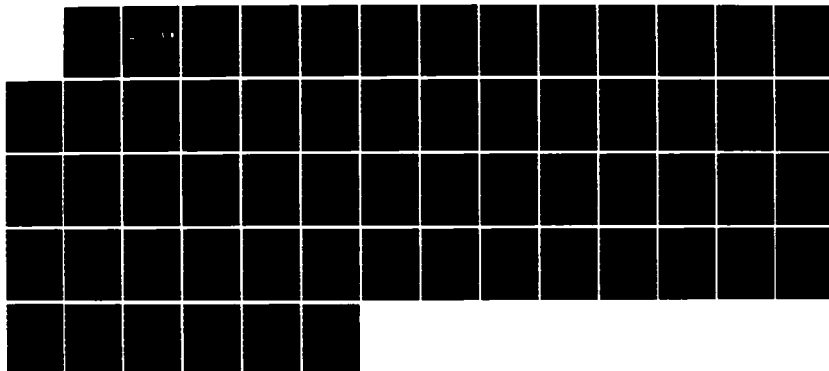
FORTRAN-77 COMPUTER PROGRAM STRUCTURE AND INTERNAL
DOCUMENTATION STANDARD..(U) AIR FORCE WEAPONS LAB
KIRTLAND AFB NM J F JANNI ET AL. JUN 86 AFML-TR-85-26

1/1

UNCLASSIFIED

F/G 9/2

NL



2

AD-A173 393



FORTAN-77 COMPUTER PROGRAM STRUCTURE AND INTERNAL DOCUMENTATION STANDARDS FOR SCIENTIFIC APPLICATIONS

Dr J.F. Janni - Principal Author

Maj R. Berry - Contributors

Mr J. Burgio

Dr G. Cable

Mr R. Conley, Jr

Capt H. Happ, III

Ms D. Janni

Capt L. Lutz

Mr H. Murphy

Mr N. Philliber

Mr G. Radke, Jr

Capt J. Spear

June 1986

Final Report

Approved for public release, distribution unlimited.

DTIC
ELECTE
OCT 23 1986
S D

DTIC FILE COPY

AIR FORCE WEAPONS LABORATORY
Air Force Systems Command
Kirtland Air Force Base, NM 87117-6008

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

AD-4173 393

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified			1b. RESTRICTIVE MARKINGS		
2a. SECURITY CLASSIFICATION AUTHORITY			3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited.		
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE					
4. PERFORMING ORGANIZATION REPORT NUMBER(S) AFWL-TR-85-26			5. MONITORING ORGANIZATION REPORT NUMBER(S)		
6a. NAME OF PERFORMING ORGANIZATION Air Force Weapons Laboratory		6b. OFFICE SYMBOL (If applicable) NTCT		7a. NAME OF MONITORING ORGANIZATION	
6c. ADDRESS (City, State, and ZIP Code) Kirtland Air Force Base, NM 87117-6008			7b. ADDRESS (City, State, and ZIP Code)		
8a. NAME OF FUNDING/SPONSORING ORGANIZATION		8b. OFFICE SYMBOL (If applicable)		9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER	
8c. ADDRESS (City, State, and ZIP Code)			10. SOURCE OF FUNDING NUMBERS		
PROGRAM ELEMENT NO. 62601F		PROJECT NO. 8809		TASK NO. 19	WORK UNIT ACCESSION NO. 01
11. TITLE (Include Security Classification) FORTRAN-77 COMPUTER PROGRAM STRUCTURE AND INTERNAL DOCUMENTATION STANDARDS FOR SCIENTIFIC APPLICATIONS					
12. PERSONAL AUTHOR(S) Janni, J.F.; Berry, R.; Burgio, J.; Cable, G.; Conley, R.; Happ, H.; Janni, D.; Lutz, L.; Murphy, H.; Philliber, N.; Radke, G.; Spear, J.					
13a. TYPE OF REPORT Final		13b. TIME COVERED FROM 84-01-01 TO 85-12-31		14. DATE OF REPORT (Year, Month, Day) 1986, Jun	
15. PAGE COUNT 60					
16. SUPPLEMENTARY NOTATION The contributors are listed in alphabetical order.					
17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)		
FIELD 09	GROUP 02	SUB-GROUP	Standards Program Structure Internal Documentation FORTRAN Structured Programming Modular Programming (over)		
19. ABSTRACT (Continue on reverse if necessary and identify by block number) An improved approach to programming has been developed in the last decade which produces reliable, efficient computer programs using fewer labor hours and far fewer maintenance hours than other coding approaches. This approach uses a disciplined style and is usually referred to as structured programming. This standard applies the concepts of structured programming to FORTRAN-77 (ANSI X3.9-1978) and contains procedures which result in better, more reliable computer programs. It is based on many actual experiences, careful research, and documented studies. This standard classifies coding practices into five categories: mandatory, recommended, permitted, discouraged, and forbidden. The objectives of this standard in directing the use of disciplined programming practices are: (over)					
20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS			21. ABSTRACT SECURITY CLASSIFICATION Unclassified		
22a. NAME OF RESPONSIBLE INDIVIDUAL Dr Joseph F. Janni			22b. TELEPHONE (Include Area Code) (505) 846-0861		22c. OFFICE SYMBOL AFSTC/CA

DD FORM 1473, 84 MAR

83 APR edition may be used until exhausted.
All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE

UNCLASSIFIED

18. SUBJECT TERMS (Continued)

Programming Practices

19. ABSTRACT (Continued)

- (a) To apply an architectural and syntactical method to the FORTRAN-77 language that greatly reduces the probability of errors.
- (b) To produce code that is modified easily, rapidly, and reliably by applying the principles of modular, structured, and machine interchangeable FORTRAN-77.
- (c) To improve code clarity, simplicity, robustness, and reliability.
- (d) To prohibit convoluted logic. *and*
- (e) To produce well-documented code.

This final report was prepared by the Air Force Weapons Laboratory, Kirtland Air Force Base, New Mexico, under job order 88091901. Dr Joseph F. Janni (NTCT) was the Laboratory Project Officer-in-Charge.

When Government drawings, specifications, or other data are used for any purpose other than in connection with a definitely Government-related procurement, the United States Government incurs no responsibility or any obligation whatsoever. The fact that the Government may have formulated or in any way supplied the said drawings, specifications, or other data, is not to be regarded by implication, or otherwise in any manner construed, as licensing the holder, or any other person or corporation; or as conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

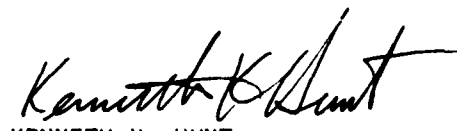
This report has been authored by employees of the United States Government. Accordingly, the United States Government retains a nonexclusive, royalty-free license to publish or reproduce the material contained herein, or allow others to do so, for the United States Government purposes.

This report has been reviewed by the Public Affairs Office and is releasable to the National Technical Information Service (NTIS). At NTIS, it will be available to the general public, including foreign nationals.

If your address has changed, if you wish to be removed from our mailing list, or if your organization no longer employs the addressee, please notify AFWL/NTCT, Kirtland AFB, NM 87117-6008 to help us maintain a current mailing list.

This report has been reviewed and is approved for publication.


JOSEPH F. JANNI
Project Officer


KENNETH K. HUNT
Maj, USAF
Chief, Technology Branch

FOR THE COMMANDER


MARION F. SCHNEIDER
Lt Col, USAF
Chief, Space/C³/Reentry Systems Div

DO NOT RETURN COPIES OF THIS REPORT UNLESS CONTRACTUAL OBLIGATIONS OR NOTICE ON A SPECIFIC DOCUMENT REQUIRES THAT IT BE RETURNED.

FOREWORD

These standards are dedicated to all the programmers who have to modify someone else's code.

The standards were born out of frustration and developed out of necessity. The situation that initiated their development occurred at 3 o'clock in the morning on a Sunday. We had been working all weekend trying to get answers from large FORTRAN programs designed to calculate a wide range of complex radiation effects. We had worked on the project for several months and had encountered so many problems with these programs that we had been forced into a 24 hour-a-day dedicated effort in order to meet the impending deadline. The codes failed during execution every few hours. As I debugged each code failure, I found the same poor programming that I had been finding repeatedly in prior months. It was on this Sunday in the early morning hours that I decided to produce a set of standards that would resolve these problems and prevent them in the future.

The radiation effects programs were almost impossible to understand. They also proved to be very unreliable. The programs contained convoluted logic full of GO TO statements and confusing structure. Whenever one "bug" was found and patched, the program was resubmitted but another problem would soon occur. These programs evolved over a 10 year period and were written by dozens of programmers at several different facilities. All of the programmers who had written these codes said that they understood and applied good programming practices. Although they may have believed it themselves, only a few actually implemented good programming practices. Most of the programming was a disaster.

It took over a year to develop the standards in this document. In addition to drawing on our many years of combined experience for background material, the modern literature was carefully reviewed. Recent well-documented and well-researched studies have been used extensively. Appropriate literature citations are provided for those who wish to verify the published research that supports this document. These standards are now solving the overwhelming majority of our programming problems. I believe they can do the same for others.

These standards are, of necessity, mechanical in nature. In some cases, recommendations are made but specific programming decisions are left to the programmer. In many cases the programming approach is mandated. Adherence to this standard is a necessary but not sufficient condition to produce large high-quality FORTRAN programs. These standards must be used with a positive and constructive attitude if they are to be successful.

The team of contributors to this document all share two crucial attributes: (1) they are excellent FORTRAN programmers (many are also proficient in other languages), and (2) they all have had painful experiences with junk code which didn't work reliably and which wasn't documented. The entire team contributed to this document and the members abide by it themselves.

After this document was released for an initial review by other programmers, two inter-

esting trends were observed in their reaction. Programmers who routinely had to modify code provided by someone else were strongly in favor of the standards. Programmers who wrote original codes and who didn't have to use or modify other codes frequently did not like the standards. They claimed too many constraints were placed on their judgement, unduly inhibiting their full capabilities as superior programmers. My sympathies are primarily with the first group. Although I understand the attitude of the second group, I do not agree with it. Large FORTRAN programs must be written in a consistent form that maximizes the probability of success and simultaneously minimizes the introduction of "bugs" through poor programming practices. Undisciplined programming with clever methods and tricky algorithms is contrary to this objective. Undisciplined programming also tends to be imbued with a programmer's idiosyncracies and is very difficult to verify and modify.

The most controversial aspect of this document is the tight control on the utilization and structure of COMMON blocks. Our position on COMMON blocks is well-founded. Another controversial issue is the mandatory application of FORTRAN implicit declarations for variable and PARAMETER names. Again, the programmers who use and debug programs written by others have supported both restrictions; the programmers who write code for their own use or for use by others usually disagree with our position.

This standard discusses three general aspects of programs: structural content, documentation, and cosmetic appearance. The latter characteristics include the format of statement label fields, the columns and form of the comments, indentation rules, and other similar features. They are mandated primarily to produce a uniformity of style and appearance that is highly desirable in large FORTRAN programs written by many individuals who would otherwise use widely different styles.

The standard is intended to be read from start to finish. It is written in a direct style for easy reading, but as a result is not organized as a reference manual. For this reason, a cross-reference index is included to help locate specific topics.

In order to keep this document as short as possible, the rationale for most of the individual features of this standard is usually omitted. Exceptions are made for particularly controversial features or for specific issues meriting a textual explanation. When possible, citations to references are provided for those readers desiring supporting information.

Implementation of these standards will produce less expensive programs over the long term because the resulting coding will be more reliable, will contain fewer bugs, will be easier to modify, and will require far fewer maintenance hours than alternate FORTRAN coding approaches. Over the short term, adherence to these standards will not substantially increase the development time of a new program because the improved reliability, organization, modularity, and understandability will offset the extra effort required for documentation, testing, and peer review.

Joseph F. Janni

Albuquerque, New Mexico
December 9, 1985

ACKNOWLEDGEMENTS

The suggestions of Capt. Raymond Leong and Ms. Cherise Jarrett were very helpful in the development of this document. The extraordinary patience of Ms. Antoinette Aguilar in flawlessly typing and correcting this document is sincerely appreciated. The thorough editing by Ms. Carol Thompson of the AFWL Technical Reports Branch contributed greatly to the final publication of this standard. The standard is a by-product of computer code development in heavy ion transport that was funded by Dr. Arthur Guenther, the chief scientist of the Air Force Weapons Laboratory, under ILIR8215. His support is gratefully acknowledged.



Accession For	
NTIS	CRA&I ☒
DTIC	TAB ☐
Unannounced ☐	
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

CONTENTS

FOREWORD	iii
ACKNOWLEDGEMENTS	v
1 BACKGROUND	1
2 OBJECTIVES	1
3 CONCEPTS, DEFINITIONS, AND RELATED RESTRICTIONS	2
3.1 PROGRAMMING LANGUAGE	2
3.2 CODE	2
3.3 PROGRAM	2
3.4 SUBPROGRAM	3
3.5 MODULE	4
3.6 STRUCTURED STYLE	5
4 PROGRAM ORGANIZATION	7
4.1 EXECUTIVE CONTROL	7
4.2 INPUT PRACTICES	7
4.3 COMPUTATIONAL SUBPROGRAMS	9
4.4 OUTPUT SUBPROGRAMS	9
4.5 ERROR EXIT SUBPROGRAMS	10
4.6 BLOCK DATA PLACEMENT	10
5 DOCUMENTATION	10
5.1 EXTERNAL DOCUMENTATION	10
5.2 INTERNAL DOCUMENTATION	10
5.2.1 Preamble documentation	11
5.2.1.1 Executive preamble	11
5.2.1.2 Subprogram preamble	12
5.2.1.3 Input subprogram preamble	12
5.2.2 Comment form, style, and placement	13
6 ROBUST PRINCIPLES AND IMPLEMENTATION	14
6.1 CODE STRUCTURE	14
6.2 GENERAL RULES	17
6.3 DATA INITIALIZATION	18
6.4 ARITHMETIC OPERATIONS	19
6.5 GENERAL INPUT/OUTPUT GUIDANCE	21
6.6 DO LOOPS	21

6.7	BRANCHING	22
6.8	VARIABLE NAMES, TYPES, AND USE	22
6.9	COMMUNICATIONS VIA ARGUMENTS	24
6.10	COMMON BLOCKS	25
6.11	ARRAYS	26
6.12	PROGRAM TERMINATION	27
6.13	DEVELOPMENT PRACTICES AND PROGRAM TESTING	27
7	LISTING ORGANIZATION	28
8	SYSTEM DEPENDENT CONSIDERATIONS	29
8.1	PROGRAM EXECUTION PRESET	29
8.2	PROGRAM RESTART	29
8.3	COMPILER OPTIONS	29
9	NONSTANDARD PROGRAMMING	30
10	RESTRICTIONS ON FORTRAN-77	30
	REFERENCES	32
	INDEX	35
	APPENDIX A: MODULES AND COMMON BLOCK RESTRICTIONS	38
	APPENDIX B: EXECUTIVE PREAMBLE EXAMPLE	39
	APPENDIX C: SUBPROGRAM PREAMBLE EXAMPLE	42
	APPENDIX D: INPUT SUBPROGRAM PREAMBLE EXAMPLE	44
	APPENDIX E: PRINTABLE US ASCII CHARACTERS	46
	APPENDIX F: LOOP LEAVE AGAIN CONSTRUCT	48
	APPENDIX G: CASE STRUCTURE AND REPEATED ELSEIF CONSTRUCTS	50

1 BACKGROUND

An improved approach to programming has been developed in the last decade which produces reliable, efficient computer programs using fewer labor hours and far fewer maintenance hours than other coding approaches. This new approach uses a disciplined style and is usually referred to as structured programming. This standard applies the concepts of structured programming to FORTRAN-77 (ANSI X3.9-1978) and contains procedures which result in better, more reliable computer programs (Ref. 1). It is based on many actual experiences, careful research, and documented studies (Refs. 2-31).

This standard classifies coding practices into five categories: mandatory, recommended, permitted, discouraged, and forbidden. A **mandatory** coding practice **must** always be implemented. A **recommended** coding practice **should** usually be used, but the programmer can apply prudent judgement and occasionally deviate in a specific situation. A **permitted** coding practice **may** be used. A **discouraged** coding practice **should not** be used except on rare occasions and only if extraordinary care is taken. A **forbidden** coding practice is **never** permitted. The boldface words in this paragraph clarify the intent of practices discussed in this standard.

2 OBJECTIVES

The objectives of this standard in directing the use of disciplined programming practices are:

- To apply an architectural and syntactical method to the FORTRAN-77 language that greatly reduces the probability of errors. Poor use of the language creates many problems (Ref. 18).
- To produce code that is modified easily, rapidly, and reliably by applying the principles of modular, structured, and machine interchangeable FORTRAN-77 that adheres to top-down design.
- To improve code clarity, simplicity, robustness, and reliability.
- To prohibit convoluted logic.
- To minimize the dependence of one module on the internal details of another. A module should have limited access to the data structures used by other modules.
- To produce well-documented code.
- To avoid the seemingly endless series of patches and repairs whose implementation requires explicit changes in many places throughout the program.

3 CONCEPTS, DEFINITIONS, AND RELATED RESTRICTIONS

3.1 PROGRAMMING LANGUAGE

The programming language is ANSI FORTRAN-77, which is hereby made a part of this standard. If there is a conflict between this standard and ANSI FORTRAN-77, the provisions of this standard apply.*

3.2 CODE

Code is a term for instructions in a computer programming language.

3.3 PROGRAM

A program is an organized set of code tailored to perform specific tasks. The intent of a FORTRAN-77 computer program is to solve a mathematical, logical, physical, technical, or engineering problem. Although programs—including those written to solve apparently simple problems—can be quite complex, the program should have a simple organization and structure. A program should be partitioned into sections of code to perform specific tasks. These sections of code are composed of subprograms and modules consisting of closely related subprograms:

EXECUTIVE (Master Control).

INPUT SECTION.

Subprograms.

COMPUTATIONAL SECTION.

Subprograms and Modules.

OUTPUT SECTION.

Subprograms.

ERROR EXIT SECTION.

Subprograms.

DATA STRUCTURE DEFINITION.

BLOCK DATA Subprograms.

*An extension to the FORTRAN-77 standard is the required use of the INCLUDE statement or its equivalent (Ref. 20). Other limited exceptions are permitted only when specified explicitly by this standard (refer to Section 9, page 30).

3.4 SUBPROGRAM

- A. A subprogram is a SUBROUTINE, FUNCTION, or BLOCK DATA and is limited to a single purpose. A subprogram can reference (call) other subprograms, or it can be referenced by the executive or other subprograms.
 - 1. A directly subordinate subprogram is referenced by an executive or subprogram. Specifically, a subprogram is directly subordinate to an executive (or another subprogram) if it is explicitly referenced by the executive (or other subprogram).
 - 2. An indirectly subordinate subprogram is referenced by a directly subordinate subprogram or another indirectly subordinate subprogram. Specifically, a subprogram is indirectly subordinate to an executive (or another subprogram) if it can be reached from the executive (or other subprogram) only through at least one intervening subprogram.
 - 3. A basic subprogram does not reference any subordinate subprogram.
- B. A subprogram must perform correctly the process claimed for it for all valid combinations of arguments, and it must detect and take defensive action for all invalid arguments.
- C. A subprogram should be written so that a typical programmer can determine that it works correctly by careful inspection, logic verification, and execution tests.
- D. Access to arrays and variables should be limited to those arrays and variables actually needed in the subprogram (e.g., limit COMMON blocks and subprogram arguments to those actually needed) (Refs. 18 and 31).
 - 1. The primary method of providing access to arrays and variables is through argument lists.
 - 2. The secondary method of providing access is through labeled COMMON blocks, whose usage is severely restricted (Ref. 6).
 - 3. COMMON blocks may be either local or global.
 - a. A local COMMON block may be used only within a module. Once a value is set in a COMMON block it may not be changed unless it is a local COMMON block confined to a module, as defined in Section 3.5.
 - b. A global COMMON block may be used anywhere in a program but only within specified constraints.
 - (1) Globally applied COMMON blocks are permitted to carry only unchanging quantities into subprograms. Global COMMON blocks must not transfer variables out of a subprogram that have been modified or altered in that subprogram, except for the initialization of the COMMON block. Data stored in a globally applied COMMON block must not be subsequently modified.

- (2) Globally applied COMMON blocks must be loaded by means of either DATA statements in BLOCK DATA, by reading a data file at the beginning of program execution, or by a one-time calculation.

3.5 MODULE

A module (colloquially referred to as a package of tightly knit subprograms) consists of closely related subprograms which share an execution and interface environment (refer to Appendix A).

- A. A module must have a well-defined objective of limited scope.
- B. A module must manipulate a single conceptually related data structure. This data structure is isolated from the external environment. No subprogram outside of the module has access to, or information about, the data structure.
- C. A module consists of one or more interface subprograms and subordinate subprograms.
- D. A module must have a minimal external interface. Variables and arrays may be passed to and from a module only via arguments of interface subprograms.
- E. Within a module, arrays and variables may be transferred vertically via arguments, or transferred laterally between subprograms of the module via a single labeled COMMON block (containing conceptually related variables). This labeled COMMON block is local and can be used only within the module.
- F. A module must be thoroughly documented.
 1. All subprograms in a module must be identified in their internal documentation as belonging to the module.
 2. Each interface subprogram must be explicitly identified as such in its internal documentation.
 3. Each interface subprogram must contain a master list showing the identities and purposes of every subprogram in the module.
 4. Each interface subprogram must be named in the internal documentation of every subprogram in the module.
- G. Examples of modules include:
 1. An input subprogram and its subordinate subprograms.
 2. The subprograms comprising a consolidated phenomenological model.
 3. A data manipulation subprogram, with subordinate subprograms performing calculations of data on the grid.

- H. Subprograms and modules are fundamentally different; what is central to one may be inappropriate to the other. A subprogram performs a single task, whereas a module collects several tasks together which share the support of a data structure, and which hide the data structure from the external environment. While a single interface is good structure for a subprogram, it does not necessarily promote sound programming for a module. A module should have a separate subprogram interface for each operation on the data structure; each interface subprogram should have an argument list that corresponds to the information needed to perform the task on the data structure.

3.6 STRUCTURED STYLE

One of the fundamental elements of well-written code is the application of a structured style to the complete program, to each subprogram, and to each module (Ref. 31).

A. Structured style applies to the complete program.

1. Good structure is the key to both a well-organized program and a program that can be easily adapted to solve new problems (Ref. 31). Changes to a program should only require modifications to the executive and the addition of new subprograms and modules. It is widely agreed that logically partitioned FORTRAN programs minimize code errors and execution failures (Refs. 4 and 6).
2. The schematic of a program is illustrated in Fig. 1. Each computer program must be organized into separate sections:
 - a. executive,
 - b. input,
 - c. computations,
 - d. output,
 - e. error exit if needed, and
 - f. data structure definition, if needed.
3. This overall structure consolidates related tasks. Modifications and incorporation of new computational models are readily integrated into these sections.

B. Structured style applies to each subprogram.

1. Good structure allows a subprogram to be written independently of other subprograms, and allows subprograms to be modified or replaced quickly, easily, and reliably (Ref. 2).
2. Each subprogram must have a single purpose so that program modifications are usually limited to changes in the executive and to the replacement or addition of individual subprograms.

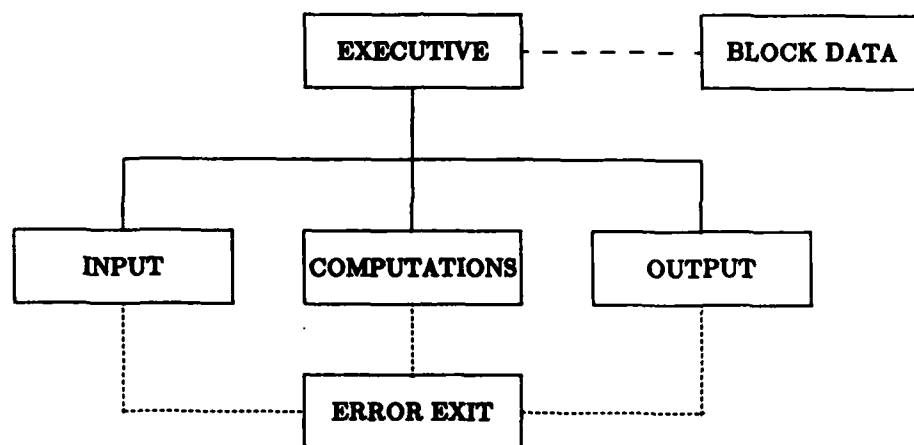


Figure 1. Schematic of a computer program. Error exit subprograms may be called as necessary from any subprogram.

3. The inputs and outputs of each subprogram must be well-defined with clear and specific interfaces (Refs. 17-19).

C. Structured style applies to each module.

1. A module should be written independently of other modules, and should be written so that it can be modified or replaced quickly, easily, and reliably (Ref. 2).
2. Each module must have a primary purpose so that program modifications are usually limited to changes in the executive and to the replacement or addition of modules.
3. The inputs and outputs of each module must be well-defined with clear and specific interfaces through one or more interface subprograms (Refs. 17-19).
4. Data structures within a module must be isolated from the external environment.

4 PROGRAM ORGANIZATION

4.1 EXECUTIVE CONTROL

- A. The executive is the program controller. The executive is a logic and flow director only.
- B. It writes the program name, version number, version date, current time, and date to the primary output text file.
- C. It directs the reading of the input, directs the computations, writes appropriate periodic progress messages, and directs the writing of the final output.
- D. The primary purpose of the executive is to orchestrate the logic. All nontrivial computations are done in subordinate subprograms. The executive is limited to administrative computations necessary for orchestration.

4.2 INPUT PRACTICES

- A. The reading of user-specified data from any source must be controlled by an input subprogram or module.
 1. It may read all the user-defined data directly or be an input master subprogram that directs other subordinate subprograms to read or process input data.
 2. Input data must be read only in input subprograms, not anywhere else in the program.

- B. Keyword-driven input is encouraged (Ref. 21). (Keyword-driven input is similar to table-driven (Ref. 19) and name-directed (Ref. 21) input. This input form is substantively different from NAMELIST, which is forbidden.
1. Keyword-driven input increases readability of input files, and minimizes order-of-input errors (Ref. 21).
 2. The objectives for keyword-driven input are (Ref. 21):
 - Clarity:* minimize user confusion,
 - Conciseness:* minimize useless verbiage,
 - Organization:* maximize proximity of related information,
 - Flexibility:* minimize artificial limitations, and
 - Ease of Use/Learning:* minimize inconvenience or time to learn.
- C. Fixed-field input is acceptable. Human engineering of all formatted input data files is critically important. Fields of five or multiples of five must be used. Such formatting improves readability.
- D. Completely free-field input is discouraged except to implement keyword-driven input. Free-field means the input stream has no predefined column positions where information is to be placed.
- E. Specification of input options with numbers is forbidden; use mnemonic keywords.
- F. Nonsensical or out-of-range values are forbidden as input options or control flags. Example: Do not use the negative component of -10 as a control option for a variable that must be nonnegative to be correct.
- G. Each user-defined input record must be written to an appropriate output file (echo printed) immediately following the READ statement for that record. This echo print must identify the record and all the data fields on the record (Refs. 7 and 19). A string of markers above or beneath an output line or page should be used to assist in determining the columns of each field.
- H. Default values should be assigned whenever possible by the input subprograms to user-defined input variables when a value is not specified by the user (Ref. 7). Where it is reasonable to assign default values, blank fields must be used to set them. Default values must be identified and written out to the output file with the echo print immediately following the READ statement for that record.
- I. Input values which are critical (have no reasonable default) must be explicitly checked as specified in the following paragraph. Such values must be identified in the internal and external documentation. Nonentry of a critical value must be treated as bad input data.

- J. All input data must be checked for unreasonableness, inconsistencies, and out-of-range values (Refs. 7, 19, 27, and 31). As a minimum, exhaustive error checking of all input data against the entire permissible range of values is mandatory. If bad input data are discovered, descriptive error messages must be written in sufficient detail to locate and identify the problem.
- K. If one or more fatal input errors are detected, scanning of input data should continue as far as possible, issuing error messages as problems are detected. After scanning of input data for errors has proceeded as far as practical, execution must stop or be aborted.
- L. Entering the same datum more than once is forbidden (e.g., the density of aluminum must not be entered more than once).
- M. Conceptually similar input data should be grouped together.
- N. Changes of engineering units (e.g., inches to centimeters) should be made in input subprograms, and should be done as soon as practical after associated READ statements.
- O. The reading of input should be terminated by an end-of-file or a marker (Refs. 7 and 19). Termination by predetermined count is discouraged (Ref. 19).
- P. The END option in a READ statement must be used to preclude an abort due to a premature end-of-file on the input data stream (Ref. 19). The ERR option in a READ statement must be used to detect bad data (but this check alone is not sufficient input validation). The IOSTAT= read option may be used in lieu of the END= and ERR= options in cases where it is desirable to do error checking after the read, rather than to execute a simple branch.

4.3 COMPUTATIONAL SUBPROGRAMS

- A. The computational subprograms (FUNCTIONs and SUBROUTINEs) must perform all nontrivial calculations.
- B. Each computational subprogram must either initialize or define at first use all variables internal to the subprogram (Ref. 7).
- C. Temporary scratch files may be written and used as necessary in any computational subprogram.

4.4 OUTPUT SUBPROGRAMS

- A. All final, summary output must be written in output SUBROUTINEs. Intermediate printing in computational subprograms is permitted only when necessary for diagnostic output.

- B. Interim, diagnostic output may be written as needed as specified in Section 4.1, Standards B and C on page 7, and Section 4.2, Standard G on page 8.
- C. All output text and graphical data must be clearly defined and explicitly labeled. The output pages should each provide sufficient explanation of their content. When necessary for full understanding, output must be preceded by a page which explains in detail the output appearing on the following page or pages. Engineering units must be associated with all output numbers and graph axes.
- D. A scale factor of 1 must be used with all printed output which uses the "Ew.d" format field descriptor (i.e., the digit printed to the left of the decimal point must be nonzero as in "1.2345E+12," rather than "0.1234E+13"). Note that a scale factor applies to all subsequent fields in the FORMAT statement. If a "1PEw.d" precedes a "Fw.d" specification, the "F" specification must be changed to "0PFw.d" to obtain the desired "F" specification output.

4.5 ERROR EXIT SUBPROGRAMS

- A. Error exit subprograms are not required, but may be used to perform error analysis, error reporting, and final cleanup prior to abnormal program termination.
- B. When used, error exit subprograms must immediately precede any BLOCK DATA subprograms, or must be the last subprograms if BLOCK DATA subprograms are not used.

4.6 BLOCK DATA PLACEMENT

When used, BLOCK DATA must be at the end of a module or program. All BLOCK DATA must be named.

5 DOCUMENTATION

5.1 EXTERNAL DOCUMENTATION

Each program must be fully documented. Documentation equivalent to the *American National Standard Guidelines for the Documentation of Digital Computer Programs* is recommended (Ref. 12). Documentation must include: (1) the computer program abstract; (2) application information (user's manual); (3) problem or function definition; (4) program design information; and (5) sample problems (refer to Section 6.13, Standard G on page 28).

5.2 INTERNAL DOCUMENTATION

The following internal documentation requirements apply to the executive and all subprograms. In addition to external documentation, complete and detailed comments must

be interspersed within the code itself (Refs. 14 and 31). Subprograms using nonANSI FORTRAN-77 must be specifically identified in their preamble (refer to Section 9, page 30).

5.2.1 Preamble documentation – The beginning of the executive and each subprogram must have a standard comment section that fully describes the coding and specifies all information that passes into or out of it (Ref. 19). This preamble begins with the first line below the name and ends with the line just prior to the first noncomment statement (normally a PARAMETER statement). The following general rules apply to all preambles:

- A. Correct grammatical style and punctuation will be used throughout the preamble.
- B. Abbreviations will be limited to those defined within the preamble except for the standard units abbreviations.
- C. Every line of the preamble will have a 'C' in column 1, except blank comment lines.
- D. The major headings Purpose, Input, Output, etc., will start in column five and all additional indentations are moved to the right at multiples of 5 spaces.

The order and format of the preamble is specified to insure uniformity of content and appearance. The content and order of the information in the preamble changes slightly for the program executive, an ordinary subprogram, or an input subprogram.

5.2.1.1 Executive preamble – The following information in the sequence specified will appear in each program executive:

- A. The description of the program. Included in the description will be a statement of purpose, an outline of the method used, a description of the known limitations of the program, and a brief summary of the input and output data.
- B. The version number and its date.
- C. The name, organization, address and phone number of the programmer.
- D. A list of all files used by the program. The list will include a brief description of the file type, structure, and contents.
- E. A list and one line description of each subprogram called by the executive.
- F. An alphabetized list with full descriptions (including units) of all local variables used in the executive.
- G. A list of references used in the program.

See Appendix B for an example of an executive preamble.

5.2.1.2 Subprogram preamble – The following information in the sequence specified will appear in each subprogram:

- A. A statement of subprogram purpose.
- B. The version number and its date. Revisions to the initial version must include a brief summary of each change, who modified the subprogram, and the date of modification.
- C. The name, organization, address, and phone number of the programmer.
- D. A list of files used and a brief description of the file type, structure, and contents.
- E. A list and one line description of each subprogram required by the subprogram.
- F. An alphabetized list and full descriptions (including units) of all local variables.
- G. A list of references used to develop the subprogram.
- H. An alphabetized list and full descriptions (including units) of all input variables from all sources.
- I. An alphabetized list and full descriptions (including units) of all output variables.
- J. If a commercial software or system utility is used, a statement warning the user of its use is placed here. Identify the I/O units used by the package or utility, if known.
- K. An alphabetized list and full description of all special constants.

Optional information that can be included in the preamble is a statement of method which would be placed immediately after the purpose. Notes may be placed anywhere if they are needed to highlight some unusual feature or provide additional information. See Appendix C for an example of a subprogram preamble.

5.2.1.3 Input subprogram preamble – Input subprograms have some documentation requirements in addition to the requirements in Section 5.2.1.2. The following information must appear in the places specified:

- A. A description of each input record must be given after the output variable list. The information must include the variable name, the columns or field location, the format and full description. The description must include the input engineering units, range of allowable values, and program default values, if any.
- B. If more than one record is read by the subprogram, then the number and types of records that can be read must be summarized. This information will be placed just following the description of the input records.

See Appendix D for a brief example of an input subprogram preamble.

5.2.2 Comment form, style, and placement – Commenting uniformity eases the tasks of reading and understanding the code.

- A. Comments must provide additional information not easily found in the code itself (Ref. 7). Describe the intent of a segment of code; do not merely restate the code (Ref. 19).
- B. Comments containing information (i.e., other than blank separator lines) must equal at least twenty percent of the total number of executable statements in the subprogram. Each subprogram must meet this requirement. This minimum percentage is intended to insure that the internal code documentation is adequate to explain the variables, clarify the logic, and summarize what the code is trying to accomplish.
- C. More than 15 consecutive FORTRAN executable statements are forbidden without at least one informative comment. Meaningful sections of code should be shorter than 10 to 15 executable statements (Ref. 19).
- D. Comments must be written simultaneously with the code, not after coding has been completed. When coding is changed, the comments must be modified simultaneously. Simultaneous commenting has been shown to produce more complete and accurate internal documentation (Refs. 9, 23, and 28).
- E. Comments must always precede, not follow, the code being described. All comments must appear between the PROGRAM, BLOCK DATA, FUNCTION or SUBROUTINE statement, and the associated END statement.
- F. Comments which appear before the executable portion of each subprogram must begin in column 5. Comments which follow the preamble and are interspersed with executable code must be indented; these indented comments must begin in column 20 (Ref. 19). Further indenting is permitted occasionally when it improves clarity.
- G. To improve visual clarity, blank lines must precede and follow a block of one or more informative comments.
- H. Comments must not be bordered in any way by lines or columns of characters. Drawing boxes around comments is forbidden.
- I. Inserting comments between the continuation lines of nonexecutable or executable statements is forbidden.
- J. Identical comments which describe the contents of a COMMON block must immediately precede that block every time it appears. Each COMMON block variable must be described in order of appearance. (Refer to Section 6.10, Standard H on page 26.)
- K. Identical comments which describe the contents of a PARAMETER statement must immediately precede that statement every time it appears.

L. Comments must use mixed upper- and lower-case letters.

1. Only printable US ASCII characters (refer to Appendix E) may be used in comments and CHARACTER strings.
2. Correct sentence structure and grammatical style should be used.
3. The imperative form of a sentence may be used.
4. Phrases may be used only when their meaning is absolutely clear (sentences are preferred).

6 ROBUST PRINCIPLES AND IMPLEMENTATION

All subprograms must be written employing robust principles. A robust subprogram does not fail under any circumstances. Defensive coding must always be used because it helps achieve this objective (Ref. 7). Many proven techniques that lessen the probability of a failure are summarized below.

6.1 CODE STRUCTURE

- A. A PROGRAM statement must be the first statement in an executive, and therefore must be the first statement in any program.
- B. Each subprogram must have a single entry located only at the first executable statement.
- C. Each subprogram should have only a single exit located at the end of its executable statements (Ref. 6). Exceptions are permitted only to eliminate a branch to the exit and improve clarity. However, when a subprogram needs statements which are guaranteed to execute whenever the subprogram exits, then a single exit is mandatory.
- D. A subprogram should be short, consistent with the process being performed. Subprograms with more than 100 executable statements are discouraged (Ref. 6). More than 200 executable statements are forbidden. (Comments, FORMATS, and DATA definitions are automatically excluded because they are not executable statements. COMMON blocks and type, PARAMETER, and DIMENSION declarations are not executable and are excluded as well.) Long subprograms tend to be disproportionately more complex (Ref. 17).
- E. Decisions made at one level of a software structure frequently have an effect on other levels (Ref. 6).
 1. If a subprogram references a subordinate subprogram, then the subordinate subprogram is within the span-of-control of the first.

2. If a decision made within the first subprogram directly affects a process within the second, then the second subprogram is within the scope-of-effect of the first (Ref. 6).
 3. The direct span-of-control of a subprogram is itself and all directly subordinate subprograms.
 - a. In Fig. 2, Subprograms D, G, and H are in the direct span-of-control of Subprogram D.
 - b. Subprogram K is within the indirect span-of-control of Subprogram D, and in the direct span-of-control of Subprogram G.
 - c. Subprograms I, J, and K are basic subprograms.
 - d. Subprogram J is a general purpose subprogram which may be called from any hierarchical level.
 - e. Subprograms D, G, H, and K constitute a module, with Subprogram D as the interface.
 - f. Coupling between modules employing local COMMON blocks is forbidden. For example, Subprogram G is within a module; it must not be linked laterally to Subprogram F by a local COMMON block (refer to Appendix A).
 4. Subprograms should be designed so that their indirect span-of-control is minimized. A strictly-linear deeply-descending hierarchy of subprograms is generally poor. A shallow parallel structure is recommended.
 5. A subprogram must not directly reference more than nine different subordinate subprograms; thus, the direct span-of-control must not exceed nine (excluding intrinsic FUNCTIONS and general-purpose system library subprograms) (Ref. 13).
- F. The general flow of any subprogram must be downward from the entry to an exit (Ref. 6), except for the DO, and the equivalent FORTRAN implementation of DO WHILE (Ref. 20), REPEAT UNTIL, and LOOP LEAVE AGAIN (Ref. 22) constructs (see Appendix F).
- G. The action clauses of any control structure (looping constructs, CASE structures,* and IF-THEN-ELSE statements) must be indented to provide better readability (Refs. 6 and 19).
1. The code must be indented to reflect the nesting levels.
 2. Each nest must be indented three columns to the right of those at the previous nesting level.
 3. All code within the same level of nesting must start in the same column.
- H. The END statement must not be used in lieu of a RETURN statement. A RETURN statement is always required.

*Refer to Appendix G for additional information on the CASE structure and repetitive ELSEIF use.

- I. If a statement does not need a label, it is forbidden to have one. A statement needs a label only when it is referenced by another statement.
- J. All specification statements must appear before the first executable statement and must precede any statement FUNCTION definitions.
- K. All statement FUNCTIONS must immediately precede the first executable statement in each subprogram and must be preceded by comments which describe the purpose of the statement FUNCTION, the arguments accepted and the results produced.
- L. All statement labels must begin with at least the number 10 and be left justified in column 2. These numbers must be in ascending order within each subprogram (Ref. 6).
- M. FORMAT statements must be grouped at the end of each subprogram immediately preceding the END statement. FORMAT statement labels must begin with at least 100, must be in ascending order (Ref. 6), and must be larger than the preceding executable statement label.

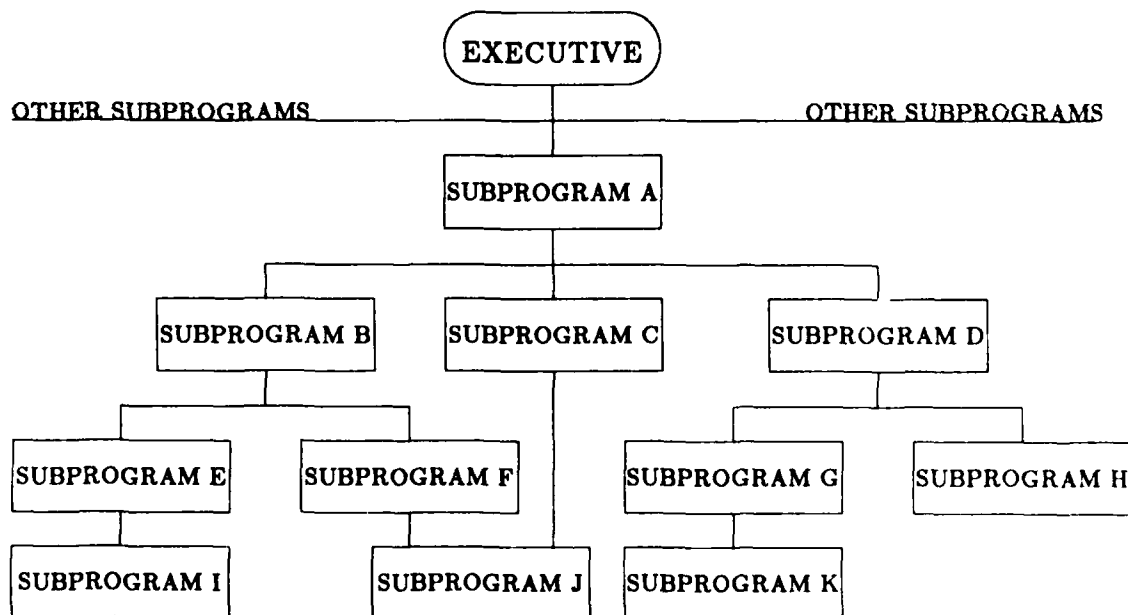


Figure 2. Subprogram hierarchy example (Ref. 6).

N. Continuation lines are permitted.

1. The maximum number of continuation lines for an executable FORTRAN statement is nine (Ref. 6).
2. Continuation lines for executable statements must be sequentially numbered in column 6 with integers sequenced from 1 through 9.
3. Continuation lines for nonexecutable statements must be sequentially numbered in column 6 with integers sequenced from 1 through 9, then with sequential letters of the alphabet beginning with A in column 6 for continuations longer than nine lines.
4. All continuations must have a blank in column 7, with the sole exception of FORMAT statements.

O. Long statements should be organized into short and easily understood sections. Statements with more than five simple sections are generally difficult to understand, and increase the probability of introducing misplaced parentheses and unintended operations (Ref. 19).

6.2 GENERAL RULES

A. Logical correctness, functional reliability, and good architecture are much more important than execution speed.

1. Coding must not be made complex to increase speed. Such efforts frequently introduce errors while also producing coding which is incomprehensible to other programmers (Refs. 7, 9, and 19).
2. Efficiency is usually determined by the algorithm chosen rather than by how compactly it is coded (Ref. 7).

B. Deeply nested statements produce obscure code. Multiple nesting should be used with great care, and only when multiple nesting will produce well-structured code.

1. Triply or deeper nested "IF-THEN-ELSE" statements are discouraged (Ref. 6).
2. Many repetitive ELSEIF clauses with lengthy blocks and inadequate comments produce obscure code and are discouraged.
3. A few well-documented repetitive ELSEIF clauses with short blocks (simulating the Pascal language CASE statement) produce understandable code and are encouraged. Each ELSEIF clause must be preceded by informative comments.

C. Error checking code must be permanently incorporated (Ref. 19). All error messages must be descriptive, grammatically correct, and concisely written.

- D. Each program must be written to minimize requirements on the computer operator such as punched card handling, magnetic tape mounting and dismounting, restarting after programmed PAUSE, and so forth (Ref. 21).
- E. The PAUSE statement may not be used unless it is absolutely essential for the correct operation of the program (Refs. 6 and 21). All such PAUSE statements must be fully documented with complete operating instructions.
- F. All FORTRAN code must be in upper-case letters (Ref. 1), except for CHARACTER data and comments.
- G. CHARACTER data may contain mixed upper and lower case (Ref. 1), and upper and lower case use is recommended for general text output.
- H. Engineering units must be the same throughout the program, and are permitted to be changed only immediately after input or immediately before output. This consistency minimizes the probability of inadvertently using incorrect units in a computation.
- I. Unsatisfied externals are not permitted in executable code. Stub subprograms (dummy subprograms which merely return sample values rather than calculated results) may be used, if necessary, to satisfy externals during code development.

6.3 DATA INITIALIZATION

- A. Floating point and integer constants which have physical, mathematical, or engineering significance (e.g., Avogadro's number "N" and the speed of light "c") should not appear explicitly in the executable code (Ref. 6).
 - 1. These constants should be defined using PARAMETER names and their description provided in comments prior to the PARAMETER statement. Names for these constants must be descriptive.
 - 2. Unitless, simple mathematical floating point values (such as 0.5, 2.0, 3.5, 10.0, 50.0, etc.) usually do not require any additional description and should appear directly in the executable code.
 - 3. Integer values should appear directly in the executable code rather than in a defined constant (refer to Standard C of this section and Standard A of Section 6.5, page 21).
 - 4. Any irrational number (e.g., " π " and the base of natural logarithms "e") must be defined with at least 10 significant digits using a PARAMETER statement. Computer system constraints rather than the programmer should limit precision.
- B. When a scalar variable represents a nonchanging value, the variable must be preset in a PARAMETER statement rather than by executable code (Ref. 7).

1. Other variables should be initialized through the use of executable code (Refs. 7 and 9); however, reasonable exceptions exist (such as the initialization of counters).
 2. Only one version of a PARAMETER statement may be maintained. Each PARAMETER statement and the associated comments used in more than one subprogram must be inserted automatically into each applicable subprogram (Ref. 6). This capability exists on many computer systems using features such as UPDATE, HISTORIAN, or the INCLUDE statement (Ref. 20). PARAMETER statements used only in a single subprogram may be defined explicitly in that subprogram.
- C. PARAMETER names must be used to set all array dimensions. When arrays are in COMMON blocks, explicit PARAMETER statements must be used. All references to array bounds must use the PARAMETER name and not a literal constant. This requirement does not extend to object-time dimensioning. Using the same PARAMETER name to set more than one dimension is forbidden unless the dimensions are logically related and will always be identical.

6.4 ARITHMETIC OPERATIONS

- A. Parenthesize to avoid ambiguity (Refs. 7 and 19). Do not rely on an assumed evaluation order for arithmetic expressions. In all arithmetic expressions, parentheses must be used to define the proper order of evaluation. For example, the following ambiguous statement: $A=B/C/D$ should be either $A=(B/C)/D$ or $A=B/(C/D)$. A better form would replace one of the divisions by a multiplication.
- B. Expressions involving successive exponentiations (such as $A**B**C$) must be explicitly parenthesized to show order of evaluation (Ref. 19). The evaluation order is clarified by parentheses (e.g., $A**(B**C)$).
- C. Whole numbers used as exponents should be integer constants or integer variables (Ref. 6).
- D. Mixed mode arithmetic statements are discouraged (Ref. 6). Mixed mode assignments or arithmetic expressions are allowed only with the explicit use of type conversion FUNCTIONS (e.g., INT(X), FLOAT(N), DBLE(X)) (Ref. 6). Integer exponentiation is not considered mixed mode arithmetic.
- E. When an operation is performed that has a restricted domain, the validity of the operation must be checked before the operation is executed (Ref. 19). Some operations with restricted domains are:
 1. Division – Check every denominator for a zero or near-zero value.
 2. Square Root – Make sure the argument is nonnegative.

3. Log or Ln FUNCTIONS – Make sure the argument is greater than zero.
4. Arcsine or Arccosine – Make sure the absolute value of the argument is less than or equal to one.
5. Arctan2 – Make sure that the arguments are not both zero.

However, identical redundant checking is not necessary. If an invalid argument is detected, an error message must be printed out.

- F. Testing whether one computed floating point value is exactly equal to another is very risky (Ref. 7). All floating point tests for equality of computed variables must incorporate a test accuracy tolerance whose value is limited by the unit roundoff of the target computer.
- G. Subtraction of nearly equal floating point numbers should be avoided. Restructure the computations or use double-precision coding in this situation. Quite often the problem can be avoided by finding an alternate expression for the desired quantity. For example, the expression for the floating point quantity

$$A = 1.0 - \text{COS}(X)$$

is given in better computational form by

$$A = 2.0 * (\text{SIN}(0.5 * X)) ** 2$$

The two expressions are identical in a mathematical sense, but if X is close to zero the second expression is better suited for use on a computer. Another example is

$$A = \sqrt{1.0 + X} - \sqrt{1.0 - X}$$

which is very unstable as X approaches zero, but

$$A = \frac{2.0 * X}{\sqrt{1.0 + X} + \sqrt{1.0 - X}}$$

is mathematically equivalent and avoids the numerical problem as X approaches zero. In both examples the preferred expression is more complex in appearance than the original expression. Therefore, the reason for using the more complex expression must be carefully explained in comments.

- H. When using an intrinsic function, use its generic name. Generic names simplify the referencing of intrinsic functions, because the same function name may be used with more than one type of argument. For example, $\text{DSQRT}(X)$ requires the argument X to be a double precision number. If, however, $\text{SQRT}(X)$ is used (the generic form of the square root function) the argument may be either single or double precision and the result will be the same type as the argument.

6.5 GENERAL INPUT/OUTPUT GUIDANCE

- A. Unit numbers for input, output, and scratch files should be designated by a **PARAMETER** constant (Ref. 6); however, an integer variable is permitted if more prudent. An integer constant must not be used.
- B. All files must be explicitly opened and closed.
 - 1. A list of files with a brief description of the exact structure and contents of each file must be included in the comments section at the beginning of the executive.
 - 2. An **OPEN** statement allows the programmer to explicitly define the status of a file when it is opened and to take action if an expected file does not exist. A file should not be opened at the beginning of the program and left open until the end of the program. A file should be closed immediately when no more data are to be read from or written to it.
- C. All input/output statements must be coded using **READ** and **WRITE** statements.
 - 1. All **READ** or **WRITE** statements must contain unit numbers. The unit to be read from or written to must not be expressed as a numeric integer constant, but must use the same **PARAMETER** constant or integer variable which opened that unit.
 - 2. Each **READ** and **WRITE** statement must be internally documented and explained; the documentation must summarize the purpose of the **READ** or **WRITE** statement.
- D. Unformatted input or output is encouraged for very large data files, all scratch files, and restart files (refer to Section 8.2, page 29); it is prohibited for all other files. The **FORMAT** translation that takes place in writing and reading formatted scratch files can needlessly consume significant amounts of computer time. In addition, some accuracy is always lost in the formatting process which is not lost in unformatted input or output.

6.6 DO LOOPS

- A. Each **DO** loop must end solely on a unique **CONTINUE** statement used only for that loop. Unique **CONTINUE** statements must be used as terminal statements for **DO** loops to clearly mark the end of the loop (in a manner identical to the nonstandard **END DO** construct) (Refs. 19 and 20).
- B. The initial and terminal parameters of a **DO** loop must be explicitly checked or coded to assure that array dimensions will not be exceeded if an array is referenced within the loop.

6.7 BRANCHING

- A. Programming logic should be straightforward, orderly, and should use simple logic.
- B. Branching should be held to a minimum (Ref. 31).
 - 1. Particular care must be paid to the necessity and appropriateness of each branching statement.
 - 2. Each branching statement must be internally documented and explained; the documentation must summarize the purpose of the branch.
 - 3. It is sufficient to group a conditional statement with the executable statements it controls and document the combined code segment once. The conditional test and branch statement does not need to be documented separately from the code which it controls.
- C. Branching into any loop or control structure is forbidden.
- D. Backward transfers are restricted to the equivalent FORTRAN implementation of DO WHILE (Ref. 20), and REPEAT UNTIL and LOOP LEAVE AGAIN constructs (Ref. 22) (refer to Appendix F). All other backwards branching is forbidden.
- E. GO TO statements are discouraged; if used, the jump must be downward (Ref. 19), with the exceptions of the DO WHILE, REPEAT UNTIL, and LOOP LEAVE AGAIN constructs. GO TO branching statements usually produce obscure code (Refs. 7, 14, 18, and 19).
- F. Every computed GO TO statement must be followed immediately by an unconditional STOP that locates and identifies the computed GO TO statement and prints the index value for which the computed GO TO failed.
- G. The logical IF should be used instead of the arithmetic IF (Refs. 7, 19, and 21). The logical IF is understood more quickly and easily than the arithmetic IF.
- H. Parenthesize to avoid ambiguity (Refs. 7 and 19). Do not rely on an assumed evaluation order for logical expressions.

6.8 VARIABLE NAMES, TYPES, AND USE

- A. Variable names convey useful information when carefully selected (Refs. 10, 14, and 19).
 - 1. The names of all variables, PARAMETERS, and COMMON blocks should be words or obvious truncations of words that mnemonically relate to the primary purpose of the variable, parameter, and common block. For example, an appropriate variable name for the speed of a moving object could be "SPEED." Inappropriate names would be "XYZ" or "EED" or "SP" because these examples have no obvious and apparent mnemonic relationship to speed.

2. Long (6-character maximum) variable names should be used.
 3. Using the first letter of each word in a phrase to form a variable name is forbidden unless it constitutes an accepted acronym.
 4. Using the numerals 0 (zero) or 1 (one) in variable names is discouraged. These numerals are easily confused with the alphabetic characters "O" and "I" in upper case on many output devices (Refs. 10 and 18).
 5. FORTRAN-77 symbolic names must be different from FORTRAN-77 keywords.
- B. Only one type declaration is allowed for any constant or variable name (Ref. 6).
- C. The FORTRAN-77 default implicit declaration for REAL and INTEGER variable names, PARAMETER names, and nongeneric FUNCTION names must be used.
1. FORTRAN-77 implicitly assigns integer values to such names beginning with I through N; otherwise real values are assigned. Adhering to this convention greatly facilitates debugging and decreases the probability of confusing REAL and INTEGER types.
 2. Meaningful variable names can usually be found within the FORTRAN implicit declaration constraint. Otherwise the leading letter "I" (for integer) or "R" (for real) is recommended as a prefix.
 3. The flexibility of explicitly declaring variable names does not outweigh the subsequent disadvantages. Several modern high-level languages require that all variables be declared explicitly, but this flexibility requires a programmer to remember a large number of variable declarations or repeatedly search out the declaration of each variable in every subprogram. This unnecessarily complicates modification and debugging tasks by seriously hampering immediate understanding of a variable declaration. Constantly looking up variable declarations is usually unrelated to the immediate programming task and has a detrimental effect. Using the FORTRAN implicit declaration makes it much easier to remember those variables that are explicitly declared since they are usually few in number.
 4. Adherence to the FORTRAN-77 implicit declaration is required by this standard. Explicit declaration of variables (and PARAMETERS) within the convention is needlessly redundant and is forbidden. Unlike FORTRAN, most modern procedural languages (including Ada, Jovial-J73, Pascal, and Modula-2) require explicit declaration of entities before they may be used. This mandatory declaration produces reliable code in other languages by allowing the compiler to perform extensive error checking at compile time and catch some programming errors that otherwise are often obscure. These concepts are recognized as useful in other languages, but are not to be applied because of the advantage provided by FORTRAN in this situation. This class of error will be minimized by adherence to Section 6.4, Standard A, page 19; Section 6.9, Standards B

and C, page 24; Section 6.10, Standards D, E and G, page 26; Section 6.13, Standards B, C, D, and G, pages 27 and 28; and Section 8.1, page 29.

D. The integer variable names I, J, K, L, M, and N must be used only as loop indices, counters, or subscripts.

E. A flag is a variable which can have only two possible values (e.g., 0 or 1). It is used to direct the flow of control.

1. INTEGER or REAL flags are discouraged. Use true or false LOGICAL variables with mnemonically meaningful names instead. The mnemonic content of the name should assert the meaning of the "true" logical value.
2. Multiple flags should only be used in a subprogram to clarify and consolidate the logic which controls the computation into clearly defined conditions. The number of flags is an indicator of the number of logical conditions significant to the computation.

F. Temporary variables are discouraged (Ref. 7). They are usually unnecessary and complicate the code (Ref. 19). Most modern optimizing compilers recognize common subexpressions and optimize them automatically.

1. Using a temporary variable to increase execution speed while simultaneously obscuring the coding is forbidden.
2. When temporary variables are used, they should express physical or mathematical relations and should be meaningful quantities having a mnemonic name or a name reflecting standard notation.
3. All temporary variables must be commented.
4. Temporary variables must be assigned before, and in close proximity to, the coding which first uses them.

6.9 COMMUNICATIONS VIA ARGUMENTS

A. A subprogram must always be called with the full set of arguments (Refs. 18 and 19).

B. Variable names must be identical when passed to or from subprograms (Refs. 19 and 31), with the exception of variable names in statement FUNCTIONs and general-purpose or library subprograms. Exempted general-purpose subprograms include but are not limited to interpolation, root-finding, numerical integration, equation-solving, and graphics subprograms.

C. Arguments (also called actual parameters) must be variable names, FUNCTIONs, or simple expressions.

1. Explicit numeric constants must not appear as arguments (Ref. 19). This constraint defends against inadvertently changing the value of a literal constant. Errors in which subprogram values are changed improperly can be detected more easily when numeric constants are names which can be examined by ordinary debugging techniques.
 2. If an expression is used as an argument, it must be simple. A simple argument must not contain more than four arithmetic, logical, relational, or character operators (Ref. 6).
- D. FUNCTION names appearing as arguments must be listed in an EXTERNAL or INTRINSIC statement (Ref. 19).
- E. With one exception, FUNCTIONS must return results only through the normal value of the FUNCTION. A FUNCTION which alters the values of the calling arguments is forbidden; SUBROUTINES must be used for that purpose (Ref. 19). The exception is a FUNCTION used solely to implement keyword-driven input.

6.10 COMMON BLOCKS

- A. COMMON blocks must be carefully defined and strictly controlled to reduce coupling between subprograms (Ref. 6), and to minimize span-of-control/scope-of-effect conflicts (Ref. 18) (refer to Section 6.1, Standard E, page 14).
1. The primary method of providing access to arrays and variables is through argument lists.
 2. The secondary method of providing access is through labeled COMMON blocks, whose scope is severely restricted (Ref. 6).
 3. COMMON blocks may be either local or global.
 - a. A local COMMON block may be used only within a module. Once a value is set in a COMMON block it may not be changed unless it is a local COMMON block confined to a module, as defined in Section 3.5, page 4.
 - b. A global COMMON block may be used anywhere in a program but only within specified constraints.
 - (1) Globally applied COMMON blocks are permitted to carry only unchanging quantities into subprograms. Global COMMON blocks must not transfer variables out of a subprogram that have been modified or altered in that subprogram, except for the initialization of the COMMON block. Data stored in a globally applied COMMON block must not be subsequently modified.
 - (2) Globally applied COMMON blocks must be loaded by means of either DATA statements in BLOCK DATA, by reading a data file at the beginning of program execution, or by a one-time calculation.

- B. Undisciplined **COMMON** usage makes a program hard to understand (Ref. 19). Variables should be transferred to subprograms primarily via argument lists, not **COMMON** blocks.
- C. Blank (unlabeled) **COMMON** is forbidden (Ref. 6), with the two rare exceptions in Section 10, Standard G, page 31.
- D. **COMMON** blocks should not introduce extraneous variables into a subprogram (Ref. 18). The fact that a variable exists in a **COMMON** block does not justify including the entire **COMMON** block in a subprogram to access the individual variable.
- E. **COMMON** blocks must be short and contain only information that is conceptually similar or related (Refs. 6 and 31).
- F. All subscripted arrays in **COMMON** blocks must have their dimensions declared in the **COMMON** statement, rather than in another specification statement.
- G. A **COMMON** block must be identical in each subprogram in which it appears (Refs. 18 and 19). This consistency must be assured by using the **INCLUDE** procedure explained below.
- H. Only one version of a **COMMON** block may be maintained. Each **COMMON** block and the associated comments must be inserted automatically into each applicable subprogram (Ref. 6). This capability exists on many computer systems using features such as **UPDATE**, **HISTORIAN**, or the **INCLUDE** statement (Ref. 20).
- I. The **INCLUDE** file defining a **COMMON** block loaded by **BLOCK DATA** must contain an "**EXTERNAL name**" statement, where "name" is the name of the **BLOCK DATA** subprogram. This declaration insures that if the **COMMON** block becomes part of a library, the **BLOCK DATA** subprogram is included in the program linkage if the **COMMON** block is referenced.
- J. Variables in **COMMON** blocks should have at least three-letter names. I, J, K, L, M, and N are forbidden as **COMMON** block variables.

6.11 ARRAYS

- A. Whenever array indices are used outside a region of strict index control (e.g., a **DO LOOP** with a fixed number of loops is a region of strict index control) the indices must be checked to verify that the maximum array bounds have not been exceeded. If the array bounds have been exceeded, the program must stop or abort with an abnormal termination message.
- B. An array should have the same dimensions in all subprograms in which it is included. Changing the number or bounds of array dimensions between a subprogram and its subordinate subprograms is discouraged (Refs. 6 and 18).

- C. When an array and its dimensions are passed to a subordinate subprogram as arguments, the array dimension must be checked prior to the call if the dimension has been obtained from a calculation. It should be validated again in the directly subordinate subprogram.

6.12 PROGRAM TERMINATION

- A. Program terminations must use STOP statements or an abort subprogram.
 - 1. CALL EXIT and END statements will not be used in lieu of STOP statements.
 - 2. There must be only one normal STOP in any program. Normal termination must be in the executive and should say "NORMAL TERMINATION." All other terminations will be abnormal stops or aborts and must provide:
 - a. where the error condition occurred, and
 - b. the variable or condition that caused the termination.
- B. An abort subprogram may be a system-specific subprogram used solely for the purpose of terminating abnormal program execution, such that a controlling job or command procedure is also aborted properly.
- C. Failure of calculations not needed for essential computations, such as those being performed for auxiliary output, should not cause termination of the primary calculation if it can proceed legitimately. However, an error of this type must be diagnosed and an appropriate error message issued.

6.13 DEVELOPMENT PRACTICES AND PROGRAM TESTING

- A. Well-tested, defensively-coded, machine-portable, documented library FUNCTIONS and general-purpose subprograms should be used (Refs. 7 and 31). Do not "reinvent the wheel."
- B. Each subprogram must be meticulously checked by peer review (Refs. 15, 16, 19, 27, 28, and 31).
- C. Each subprogram must be compiled and tested independently during development. Errors can then be traced quickly to specific subprograms (Ref. 3).
- D. Each subprogram must be exhaustively tested by constructing driver programs that provide input and write the output from the subprogram being tested.
 - 1. The driver program must construct sample input data sets for the subprogram being tested such that all paths are taken and all boundaries (geometrical, physical, or numerical) are used (Ref. 27).

2. Software tools such as static analyzers, test-case generators, and coverage analyzers must be used for this purpose (Ref. 25-27, and 29).
 3. Plausible results are encouraging but do not by themselves constitute sufficient evidence of correctness.
- E. Using supporting external and internal documentation, a typical programmer should be able to understand the logic and verify with execution tests that a subprogram works correctly for many representative test cases after evaluating it for less than one day.
- F. After subprogram development, all subprograms in a module should be combined into a single file and compiled as a module. This procedure simplifies subprogram management by reducing the number of files, and allows global compiler optimization.
- G. Each program must be tested (Refs. 27 and 31).
1. Test problems must include a complete set of input data, a complete listing of the output produced by the program when run with the test data, and a compiler listing and cross-reference map of the program itself.
 2. These test problems serve to detect blatantly incorrect programs, but do not assure the general correctness of the program nor can they constitute a complete verification.
 3. At least three different typical test problems must be run.
- H. In order to demonstrate satisfactory machine compatibility performance of a developed program, test problems and the associated results must be furnished so that identical tests can be made on the target computer to demonstrate that the program operates successfully.

7 LISTING ORGANIZATION

The program may be listed in either of two formats. The first format is recommended.

- A. The first listing format must be in the following order: executive, followed by all subprograms (except error exit SUBROUTINEs and BLOCK DATA) in alphabetical order, followed by error exit SUBROUTINEs, followed by BLOCK DATA.
- B. The second listing format uses an order determined by considerations of logical coherence. In this case, the executive must still be listed first, and a list of all subprograms and modules in their order of appearance must be included in the comments at the beginning of the executive. Subprograms of a module should be listed together.

8 SYSTEM DEPENDENT CONSIDERATIONS

8.1 PROGRAM EXECUTION PRESET

Prior to program execution, the computer memory should be preset to a value which represents extreme conditions and which will terminate execution if the value is encountered during a calculation. For example, a preset of negative indefinite should be used on Control Data Corporation (CDC) systems.

- A. Preset values of zero or unity do not meet this criterion and may not be used unless the computer system provides no suitable alternative.
- B. Relying on a machine preset (such as zero) for variable initialization is forbidden. All variables requiring initialization must be preset in the FORTRAN program before they are referenced.

8.2 PROGRAM RESTART

- A. Programs which typically run for many hours on the target machine should have a restart capability. These programs should periodically dump all essential data in some semi-permanent form for restarting the program, should a restart be necessary (refer to Section 6.5, Standard D, page 21).
- B. Restartable programs must be capable of terminating and dumping all essential data using a method which minimizes, and if possible eliminates, interactions with the computer operator.
 - 1. This method may be nonANSI standard and computer system dependent. For example, in the CDC environment, the sense switch is such a method.
 - 2. In programs using extensive computations involving complex cycles or iterations, dumping of restart data may be deferred until the current cycle or iteration is complete.

8.3 COMPILER OPTIONS

- A. When available, program flow error traceback should be activated.
- B. If run-time array-bounds checking is available from the compiler, it must be used but is not a substitute for index checking.

9 NONSTANDARD PROGRAMMING

- A. Computer code not in ANSI FORTRAN-77 may be used only when such coding is essential to the successful operation of the program, and either
 - 1. no means whatever exist to write equivalent code in ANSI FORTRAN-77, or
 - 2. nonstandard FORTRAN-77 code is specifically required by this standard.
- B. Any nonstandard source code except for the INCLUDE (or its equivalent) must be isolated in separate subprograms. The subprogram interfaces must be built so that the nonstandard subprogram can be removed and replaced easily when the program is moved to another processor.
- C. The INCLUDE (Ref. 20) or its equivalent is restricted to PARAMETER statements or COMMON block insertions with associated EXTERNAL statements, and to their associated and preamble documentation.
 - 1. An example of required nonstandard coding is the mandatory use of the INCLUDE (or its equivalent) to standardize COMMON blocks and their associated documentation.
 - 2. On a computer system without this capability, a FORTRAN preprocessor which accomplishes this task using the VAX FORTRAN INCLUDE statement format should be provided to support programs containing PARAMETER statements and COMMON blocks.
 - 3. Compiled listings should be used for code development and debugging, rather than uncompiled source code. This procedure permits the effects of the INCLUDE or its equivalent to be evaluated easily by the programmer.

10 RESTRICTIONS ON FORTRAN-77

- A. ASSIGN statements are forbidden (Ref. 21).
- B. ENTRY statements are forbidden (Ref. 21).
- C. EQUIVALENCE statements are forbidden (Refs. 6, 21, and 24).
 - 1. EQUIVALENCE statements increase coupling between subprograms and increase the possibility of a span-of-control/scope-of-effect conflict (Ref. 18).
 - 2. The primary application of EQUIVALENCE statements was to save memory by allowing multiple uses of the same space, but the storage benefit does not offset the increased risk of error.

- D. RETURN statements which contain an argument are forbidden. This prohibition assures that a subprogram always returns control to the statement immediately following the CALL or FUNCTION reference which invoked that subprogram (Refs. 6 and 21).
- E. PRINT statements are forbidden.
- F. REAL and INTEGER statements are forbidden. Refer to Section 6.8, Standard C on page 23 for supporting rationale. This prohibition is recognized as different from the recommendation in Ref. 21; unfortunately, the trend toward mandatory strong declarations is in conflict with an inherent advantage of the FORTRAN-77 language.
- G. BLANK COMMON is permitted only in two rare situations.
 - 1. Some computer systems have the capability to adjust dynamically a program's main memory usage during execution. Sometimes this capability uses unique system methods in conjunction with BLANK COMMON. This nonstandard approach is discouraged and may be used only when absolutely necessary. This is one of only two times BLANK COMMON may ever be used.
 - 2. The only other situation where BLANK COMMON may be used is when it is imbedded in a commercial software module.
- H. CHARACTER variables, PARAMETERS, and comments should contain only printable US ASCII characters (refer to Appendix E).

REFERENCES

1. American National Standard Programming Language FORTRAN, ANSI X3.9-1978, American National Standards Institute, Inc., NY, 3 April 1978.
2. D. Parnas, "On the Criteria to be Used in Decomposing Systems Into Modules," Communications of the ACM, **15**, p1053, 1972.
3. R. Gauthier, and S. Pont, Designing Systems Programs, Prentice-Hall, Englewood Cliffs, NJ, 1970.
4. R. House, "Comments on Program Specification and Testing," Communications of the ACM, **23**, p324, 1980.
5. R. Smith, "Validation and Verification Study," in Structured Programming Series, Volume XV, produced under USAF contract F30602-74-C-0186, RADC, Griffiss AFB, NY, by the IBM Federal Systems Center, 1975.
6. Radar Set AN/FPS-118, Software Standards and Procedures Manual, produced under USAF contract F19628-82-C-0114, P.D.B. No. 316, Revision O, HQ ESD, Hanscom AFB, MA, by General Electric, Syracuse, NY, 31 March 1983. (This manual is very strict, and in many cases is more stringent than the standards of this report.)
7. B. Kernighan, and P. Plauger, The Elements of Programming Style, Second Edition, McGraw-Hill Book Company, NY, 1978. (This book is an exceptionally fine exposition of good programming practices.)
8. M. Jackson, Principles of Program Design, Academic Press, London, 1975.
9. E. Yourdon, Techniques of Program Structure and Design, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1975. (Pages 292-294 contain an excellent summary of the most common programming bugs.)
10. ANS Standard Recommended Programming Practices to Facilitate the Interchange of Digital Computer Programs, ANS Standard 3-1971, American Nuclear Society, Hinsdale, IL, 1971.
11. Guidelines for Considering User Needs in Computer Program Development, AND-10, National Standards Association, Inc., Washington, DC, 1978.
12. American National Standard Guidelines for the Documentation of Digital Computer Programs, ANSI N413-1974, American National Standards Institute, Inc., NY, 1974.
13. N. Chapin, "Structure Analysis and Structured Design: An Overview," in Systems Analysis and Design, A Foundation for the 1980's, W. Cotterman, et al., eds., North Holland Publishing Company, NY, 1981.

14. N. Enger, "Classical and Structured Systems Life Cycle Phases and Documentation," in Systems Analysis and Design, A Foundation for the 1980's, W. Cotterman, et al., eds., North Holland Publishing Company, NY, 1981.
15. M. Connor, "Structured Analysis and Design Technique," in Systems Analysis and Design, A Foundation for the 1980's, W. Cotterman, et al., eds., North Holland Publishing Company, NY, 1981.
16. F. Baker, "Software Design or What Stands Between Requirements and Programs?" in Systems Analysis and Design, A Foundation for the 1980's, W. Cotterman, et al., eds., North Holland Publishing Company, NY, 1981.
17. V. Basili and B. Perricon, "Software Errors and Complexity: An Empirical Investigation," Communications of the ACM, **27**, p42, 1984.
18. G. Berns, "Assessing Software Maintainability," Communications of the ACM, **27**, p14, 1984.
19. C. Hughes, C. Pfleeger, and L. Rose, Advanced Programming Techniques—A Second Course in Programming Using FORTRAN, John Wiley & Sons, NY, 1978. (The first three chapters are particularly relevant to good programming style.)
20. Military Standard—FORTRAN, DOD Supplement to American National Standard X3.9-1978, MIL-STD-1753, 9 November 1978. (This supplement to FORTRAN-77 has been approved by the Department of Defense and contains recommended extensions to FORTRAN-77. It has not been made a part of this standard nor is any of it automatically incorporated into this standard.)
21. J. Wagener, "Status of Work Toward Revision of Programming Language FORTRAN," FORTEC Forum SigPlan Technical Committee on FORTRAN, Number 2, Ser. No. 8, June 1984.
22. E. Solonay, J. Bonar, and K. Ehrlich, "Cognitive Strategies and Looping Constructs: An Empirical Study," Communications of the ACM, **26**, p853, 1983.
23. E. Horowitz, and S. Sahni, Fundamentals of Data Structures, Computer Science Press, Inc., Rockville, MD, 1983.
24. M. Metcalf, "Has FORTRAN a Future?," ACM FORTRAN Forum, **3**, No. 3, p21, December 1984.
25. G. Berns, "MAT, A Static Analyzer of FORTRAN PROGRAMS and the Most Common FORTRAN Reliability Problems," Proceedings of the Digital Equipment Computer Users Society, p309, December 1984.
26. L. Gaby, II, ANALYZE, FORTRAN Program Analyzer Users Guide, Computer Sciences Corporation Report H&T-C-7021, Albuquerque, NM, May 1985.

27. S. Saib, "RXVP — Today and Tommorrow," in Software Validation, H. Hausen, ed., North Holland Publishing Company, NY, 1984.
28. A. Ackerman, "Software Inspections and the Industrial Production of Software," in Software Validation, H. Hausen, ed., North Holland Publishing Company, NY, 1984.
29. R. Buck, and J. Dobbins, "Application of Software Inspection Methodology in Design and Code," in Software Validation, H. Hausen, ed., North Holland Publishing Company, NY, 1984.
30. L. Osterweil, "Integrating the Testing, Analysis and Debugging of Programs," in Software Validation, H. Hausen, ed., North Holland Publishing Company, NY, 1984.
31. T. Bowen, G. Wigle, and J. Tsai, "Specification of Software Quality Attributes," RADC-TR-85-37, Vol. I, Vol. II., and Vol. III, RADC, Griffiss AFB, NY, February 1985.

INDEX

- Abbreviations 11
- Abort(s) 9, 26, 27
- Arithmetic Operations 19
- Array(s) 3, 4, 19, 21, 25, 26, 29
- ASSIGN Statements 30

- BLOCK DATA 3, 4, 10, 13, 25, 26, 28
- Branching 14, 22
 - Statements 22

- CALL EXIT Statements 27
- CALL Statements 31
- CASE Structure 15, 17, 50
- CHARACTER Data 14, 18, 31
- Closing Files 21
- Code Structure 1, 2, 5, 11, 13, 14, 15, 17–19, 21–24, 27, 30
- Column(s) 8, 11–13, 15–17
- Comment Documentation 10, 11, 13, 14, 16–21, 24, 26, 28, 31
- COMMON Block 3, 4, 13–15, 19, 22, 25, 26, 30, 31, 38
 - Global 3, 4, 25
 - Local 3, 4, 15, 25
- COMMON Statement 26
- Computational Subprograms 9
- Conditional Statements 22
- Constant(s) 18, 19, 21, 23, 25
 - Special 12
- Continuation 13, 17
- CONTINUE Statements 21
- Control 7, 8, 14, 15, 22, 24–27, 31

- Data Initialization 18
- DATA Statements 4, 25
- Data Structure 1, 4, 5, 7, 38
- Declaration(s) 14, 26, 31
- Declaring Variables 23
- Default 8, 23
- Default Value(s) 12

- Definitions 2, 5, 10, 14, 16
- Development 18, 27, 28, 30
- DIMENSION Statements 14
- Dimension(s) 19, 21, 26, 27
- Documentation 1, 4, 8, 10, 17, 18, 22, 27, 30
- DO Loop Statements 15, 21, 26, 48
- DO WHILE Structure 15, 22, 48

- Efficiency 17
- ELSEIF Statements 15, 17, 50
- END DO Statements 21
- END Statements 9, 13, 15, 16, 27
 - In READ Statements 9
- Engineering Units 9, 10, 12, 18
- ENTRY 30
- EQUIVALENCE Statements 30
- Error Checking 9, 17, 23
- Error Exit 5, 10, 28
- Error(s) 1, 5, 8, 17, 20, 25, 27, 29, 30
 - Fatal 9
- Executive 3, 5, 7, 10, 11, 14, 21, 27, 28
 - Preamble 11
- Exit 5, 10, 14, 15, 27, 28
- Explicit Declarations 23
- External Documentation 8, 10, 28
- EXTERNAL(s) 25, 26, 30

- File(s) 4, 7–9, 11, 12, 21, 25, 26, 28
 - Closing of 21
 - Structure 11, 12
 - Type 11, 12
- Flag(s) 8, 24
- Floating Point 18, 20
- Format 8, 10, 12, 21, 28, 30
 - Listings 28
- FORMAT Statements 10, 14, 16, 17, 21
- FORTRAN 1, 2, 5, 11, 13, 15, 17, 18, 22, 23, 29–31

FUNCTION 3, 9, 13, 15, 16, 19, 20, 23-
25, 27, 31

GO TO Statements 22, 50

HISTORIAN Statements 19, 26

I/O 12, 21

Statements 21

IF Statements 48

Arithmetic 22

Logical 22

IF-THEN-ELSE Statements 15, 17

Implicit Declarations 23

INCLUDE Statements 2, 19, 26, 30

Index 22, 26, 29

Initialization 3, 18, 19, 25, 29

Initialize 9, 19

Input 5, 7-9, 18, 21, 27

Data 7-9, 11, 21, 27, 28

Default values 8

EOF 9

Fixed-field 8

Formatted 8

Free-field 8

Keyword-driven 8, 25

Record 12

Subprogram 4, 7-9

Documentation 12

Preamble 12

Unformatted 21

Unit numbers 21

User Defined 8

Variables 12

INTEGER 23, 24, 31

Integers 23

Constants 19, 21

Exponentiation 19

Values 17, 18, 23

Variables 19, 21, 24

Internal Documentation 4, 8, 10, 13, 21,
22, 28

Keywords 8, 23, 25

Label(s) 3, 4, 10, 16, 25, 26

Length 17

Listings

Compiler 28, 30

Program 28

Local Variables 11, 12

LOOP LEAVE AGAIN Structure 15, 22,
48

Loop(s) 15, 21, 22, 24, 26

Method 11

Modular Programming 1

Module 1-5, 7, 10, 15, 25, 28, 31

Example 4, 15

Structure 4, 7

Naming Variables 22, 24, 26

Nonstandard Programming 30

OPEN Statements 21

Opening Files 21

Output 5, 7, 10, 18, 21, 23, 27, 28

Data 10, 11, 21

Diagnostic 9

Echo Check 8

Echo Print 8

Labeling 8

List 12

Subroutine 9

Unit numbers 21

Variables 12

PARAMETER Statements 13, 18, 19, 22,
30

PAUSE Statements 18

Preamble 11

Documentation 11, 30

Executive 11

Format 11

Order 11

Rules 11

PRINT Statements 31

Program 1

Description 11

Limitations 11
Structure 1, 2, 5, 17, 21
PROGRAM Statements 13, 14
Programmer 11, 12
Purpose 11, 12

READ Statements 8, 9, 21
REAL Statements 23, 24, 31
Real Values 23
References 12
REPEAT UNTIL Structure 15, 22, 48
RETURN Statements 15, 31
Revisions 12
Robust Coding Techniques 1, 14

Scale Factor 10
Scope-of-effect 15, 25, 30
Scratch Files 9, 21
Software, Commercial 12
Span-of-control 14, 15, 25, 30
Specification Statements 16
Speed 17, 24
Stop 9
STOP Statements 22, 27
Structure 5, 14, 15, 22, 38
Structured Style 7
Subprograms 3, 5, 11, 12, 38
 Arguments 24
 as Functions names 25
BLOCK Data 10
Interface 4
Linkage 26
Preamble 12
Structure 5
SUBROUTINE Statements 13
System Utility Programs 12

Temporary Variables 24
Testing, for equality 20
Testing, Program 27
Type Conversion FUNCTIONs 19
Type Declaration 23

UPDATE Statements 19, 26

Variable(s) 3, 4, 8, 9, 13, 18-27, 29, 31
 Names 12
Version Number 11, 12

WRITE Statements 21

APPENDIX A

MODULES AND COMMON BLOCK RESTRICTIONS

A primary consideration of this report is the treatment of structural content. Structural content limits or promotes good programming methodology and governs the ultimate form that programs are allowed to assume. This appendix discusses the problem with regard to COMMON blocks.

A powerful construct in the organization of programs is a modularity intermediate between the subprogram and the entire program. The separation of a large computational process into internally-related subprograms with a minimal external interface is recognized in this report. One condition of separability is the manipulation of a single data structure. Typically the subprograms outside of the module need interface only at the highest level and have no need to know any of the particulars of the data structure or how it is manipulated.

In general, the data structure may involve one or more arrays, indices into the arrays, status values, and other variables. The external environment never needs access to any of the particulars of the data structure in order to use it; all the outside process needs is a functional interface. This leads to grouping the related subprograms, variables, and arrays into a module. As perceived by the external environment, the programming elements which implement the data structure are local to the module and hidden from the external environment. As perceived from within the module, these programming elements are global only within the module and access may be shared only within the module. The external environment has access only through the arguments of the interface subprogram(s) of the module.

High level languages support modules in various ways, but FORTRAN-77 does not provide direct support to a module. Indirect support is available by using a labeled COMMON statement. Its local use in a module can avoid needless vertical coupling via argument lists between the subprograms within a module. However, the wanton misuse of COMMON statements introduces increased coupling between modules and is forbidden.

Quite often COMMON blocks are used to avoid passing variables through intermediate subprograms which do not use them directly in any computations. Artificially passing parameters unrelated to the immediate process being performed is not good programming methodology. The resulting proliferation of parameters can degrade the reliability and readability of a program and increase the coupling of the subprogram to the environment. This situation is frequently caused by bad program architecture. Widespread use of COMMON blocks to pass variables laterally is not a good programming solution. Instead, tightly controlled modules with limited lateral coupling should be used in conjunction with good overall program architecture.

APPENDIX B

EXECUTIVE PREAMBLE EXAMPLE

PROGRAM ORBIT

C Program Description -

C Program ORBIT is an ephemeris program to compute satellite
C orbits. The program is based upon Vinti's (Refs. 1, 2, 3, 4,
C 5) Theory of Accurate Intermediary Orbits. The treatment to
C account for the atmospheric drag perturbations is based upon
C a paper by Watson, et al. (Ref. 6). Vinti found a closed form
C gravitational potential about an axially symmetric planet in
C oblate spheroidal coordinates. This solution accounts for all
C the effects of the second and third zonal harmonics and about
C two-thirds of the fourth harmonic. This potential, which
C simultaneously satisfies Laplace's equation and separates the
C Hamilton-Jacobi equation, succeeds in reducing the problem of
C satellite motion to quadratures. Watson, et al., provided an
C analytical method to account for the drag starting with the
C orbital elements defined by the Vinti theory (in the program
C they are referred to as the Vinti elements). The atmospheric
C model of the thermosphere was developed by Jacchia (Ref. 7).

C V E R S I O N 1 7 MAY 1985

C Questions or comments should be addressed to

C John P. Doe
C XYZ Corporation
C Albuquerque, NM 12345
C Com. Phone (505) 123-4567

C Files Used -

C IUNIT Currently unit number 14. Associated with file name
C INPUT. Coded sequential, input file. Contents are the
C user defined input data.
C MSG Currently unit number 6. Associated with file name
C MESSAGE. Coded sequential, output file. Contents are
C error and warning messages.
C NUNIT Currently unit number 17. Associated with file name
C OUTPUT. Coded sequential, output file. Contents are

C the normal printed output, the orbit specification.

C Subroutines Required -

C DRAG Changes the Vinti elements due to drag effects.
C EPHINP Reads all the program input.
C EPHOUT Writes all the program output.
C INFORM Computes all the orbital point information.
C INITAL Initializes the starting conditions.
C POSITN Computes the satellite position in inertial space.

C Local Variables -

C DRGFLG A logical control flag. If true, atmospheric drag
C calculations are to be included in the ephemeris.
C LAST A logical control flag. If true, the last ephemeris
C point calculation has been completed.
C TIMFLG A logical control flag. If true, the step increment
C between each ephemeris calculation is in equal steps
C of time. Otherwise, the step increment is in equal
C steps of true anomaly angle.

C References -

- C 1. J. P. Vinti, "New Method of Solution for Unretarded
C Satellite Orbits", Journal of Research of the National
C Bureau of Standards B. Mathematics and Mathematical Physics
C Vol. 62B, No. 2, 105-116 (1959).
C 2. J. P. Vinti, "Theory of an Accurate Intermediary Orbit
C for Satellite Astronomy", Journal of Research of the
C National Bureau of Standards B. Mathematics and Mathematical
C Physics, Vol 65B, No. 3, 169-201 (1961).
C 3. J. P. Vinti, "Intermediary Equatorial Orbits of an
C Artificial Satellite", Journal of Research of the National
C Bureau of Standards B. Mathematics and Mathematical Physics
C Vol. 66B, No. 1, 5-13 (1962).
C 4. J. P. Vinti, "Inclusion of the Third Zonal Harmonic in an
C Accurate Reference Orbit of an Artificial Satellite",
C Journal of Research of the National Bureau of Standards B.
C Mathematics and Mathematical Physics, Vol. 70B, No. 1, 17-46
C (1966).
C 5. J. P. Vinti, "Improvement of the Spheroidal Method for
C Artificial Satellites", The Astronomical Journal, Vol. 71,
C No. 1, 25-34 (1969).
C 6. J.S. Watson, G.D. Mistretta, and N.L. Bonavito, "An
C Analytical Method to Account for Drag in the Vinti

- C Satellite Theory'', Celestial Mechanics, Vol. 11, 145-176
C (1975).
C 7. L.G. Jacchia, ''Revised Static Models of the Thermosphere
C Exosphere with Empirical Temperature Profiles'', SAO
C Special Report No. 332, May 1971.

APPENDIX C

SUBPROGRAM PREAMBLE EXAMPLE

SUBROUTINE SEMIAX(ARRAY,AXIS,BALIST,CHANGE,ECCENT,NUMBER,SMALLD)

C Purpose -
C To calculate the change of the semimajor axis of the orbit due
C to drag.

C VERSION 1, 19 OCT 83

C Programmer - John P. Doe
C XYZ Corporation
C Albuquerque, NM 12345
C Phone (505) 123-4567

C Files Used - None

C Subroutines Required -
C FACTOR Calculates the atmospheric fitting factors.
C PACKS Packs ARRAY with atmospheric fitting data.

C Local Variables -
C COEFF3 An array of three integration coefficients that are
C interval dependent, in kilometers.
C COEFF7 An array of seven integration coefficients that
C are constant over the entire integration interval,
C unitless.
C SMALLB The small b in the King-Hele expression for the
C atmospheric density, unitless.

C References -
C 1. J.S. Watson, G.D. Mistretta, and N.L. Bonavito, "An
C Analytic Method to Account for Drag in the Vinti Satellite
C Theory", Journal of Celestial Mechanics, Vol. 11, 145-177
C (1975).
C 2. T. E. Sterne, "An Introduction to Celestial
C Mechanics", Interscience Publishers, Inc., New York, 165
C (1960).

C Input -

C ARRAY An array that contains the integrals of the form
 C $\text{EXP}(\text{TB2} \cdot \text{E}) \cdot \text{COS}(\text{E})^{**\text{I}}$, $\text{I}=0,1,\dots,13$ (unitless).
 C AXIS The semimajor axis of the orbit in kilometers.
 C BALIST The ballistic coefficient of the satellite in
 C kilograms per kilometer**2.
 C ECCENT The eccentricity of the orbit, unitless.
 C NUMBER The number of integration intervals, unitless.
 C SMALLD The small d is the King-Hele expression for the
 C velocity of the satellite relative to the atmosphere
 C (Ref. 2), unitless.

C Output -

C CHANGE The change of the semimajor axis of the orbit due to
 C the atmospheric drag, in kilometers.

C Special Constants -

C CONST The gravitational constant for the earth in
 C kilometers**3/second**2.
 C ROTATE The rotational rate of the earth in radians per
 C second.

APPENDIX D

INPUT SUBPROGRAM PREAMBLE

EXAMPLE

SUBROUTINE READIT(ERROR,HEIGHT,LAT,LONG,NCARD)

C Purpose - To read, verify and write an echo check of the observer's
C location information required for look angle computation.

C VERSION 1, 13 FEB 83

C Programmer - John P. Doe
C XYZ Corporation
C Albuquerque, NM 12345
C Phone (505) 123-4567

C Files Used -
C IUNIT Currently unit number 14. Associated with file name
C INPUT. Coded sequential, input file. Contents are the
C user defined input data.
C NUNIT Currently unit number 17. Associated with file name
C OUTPUT. Coded sequential, output file. Contents are
C the normal printed output, the ephemeris.

C Subroutines Required - None.

C Local Variables - None.

C Input -
C NCARD The number of records that have been read.

C Output -
C ERROR A logical control flag. If true, a nonrecoverable
C error has occurred.
C HEIGHT The height of the observer's location above the
C reference geoid in kilometers.
C LAT The latitude of the observer's location in radians.
C LONG The longitude of the observer's location in radians.

C Record Format -
C VARIABLE CARD

C	NAME	COLS.	FORMAT	VARIABLE DESCRIPTION
C	HEIGHT	1-10	F10.2	The height of the observer in kilometers.
C				Value must be greater than zero.
C	LAT	11-20	F10.2	The latitude of the observer in degrees.
C				Value must be in the range of -90. to 90.
C	LONG	21-30	F10.2	The longitude of the observer in degrees.
C				Value must be less than the absolute
C				value of 360.0 degrees.

APPENDIX E

PRINTABLE US ASCII CHARACTERS

The printable US ASCII characters in Table E-1 are the only characters that may appear in comments or as literal character strings (e.g., ' ? ' , 2H<>). In rare situations, the CHAR intrinsic function may be used to assign CHARACTER variables and PARAMETERS to characters that are not shown in Table E-1. The use of other characters must be isolated in a small number of subprograms and carefully documented as machine-dependent code. These restrictions are motivated by the ideas that a printed program listing should accurately represent the program, and that programs should not depend on a particular collating sequence, but only that the collating sequence has the properties specified in the FORTRAN-77 standard.

TABLE E-1. PRINTABLE US ASCII CHARACTERS

Decimal	Octal	Character	Decimal	Octal	Character	Decimal	Octal	Character
32	40		64	100	Q	96	140	`
33	41	!	65	101	A	97	141	a
34	42	"	66	102	B	98	142	b
35	43	#	67	103	C	99	143	c
36	44	\$	68	104	D	100	144	d
37	45	%	69	105	E	101	145	e
38	46	&	70	106	F	102	146	f
39	47	'	71	107	G	103	147	g
40	50	(	72	110	H	104	150	h
41	51	)	73	111	I	105	151	i
42	52	*	74	112	J	106	152	j
43	53	+	75	113	K	107	153	k
44	54	,	76	114	L	108	154	l
45	55	-	77	115	M	109	155	m
46	56	.	78	116	N	110	156	n
47	57	/	79	117	O	111	157	o
48	60	0	80	120	P	112	160	p
49	61	1	81	121	Q	113	161	q
50	62	2	82	122	R	114	162	r
51	63	3	83	123	S	115	163	s
52	64	4	84	124	T	116	164	t
53	65	5	85	125	U	117	165	u
54	66	6	86	126	V	118	166	v
55	67	7	87	127	W	119	167	w
56	70	8	88	130	X	120	170	x
57	71	9	89	131	Y	121	171	y
58	72	:	90	132	Z	122	172	z
59	73	;	91	133	[	123	173	{
60	74	<	92	134	\	124	174	
61	75	=	93	135	]	125	175	}
62	76	>	94	136	^	126	176	~
63	77	?	95	137	_			

APPENDIX F

LOOP LEAVE AGAIN CONSTRUCT

Several loop constructs other than FORTRAN-77 DO loops are allowed. They are the DO WHILE, REPEAT UNTIL, and LOOP LEAVE AGAIN constructs. The preferred order of implementation of these three constructs is: (1) DO WHILE, (2) REPEAT UNTIL, and (3) LOOP LEAVE AGAIN.

When implemented in FORTRAN-77, the DO WHILE loop takes the form:

```
1 IF ( <condition> ) GO TO 2
```

```
    <statements>
```

```
GO TO 1
```

```
2 <next statement>
```

The REPEAT UNTIL implementation is:

```
1 <statements>
```

```
    IF ( <condition> ) GO TO 1
```

These two constructs include a statement label on the first executable statement of a block, a body of statements, and a terminal branching statement. The two forms are distinguished by the location of the IF statement, which causes an exit from the loop based on the value of a condition. The DO WHILE requires the IF to be the first executable statement of the construct. The REPEAT UNTIL requires the IF to be the last executable statement.

Since the FORTRAN-77 forms of these constructs differ only in the location of the loop exit statement, a more general form can be introduced that includes both forms as special cases:

```
1 <statements>
```

```
    IF ( <condition> ) GO TO 2
```

```
    <statements>
```

```
GO TO 1
```

```
2 <next statement>
```

The generality is obtained by allowing the loop exit statement to fall anywhere within the body of the loop. This generalization is the LOOP LEAVE AGAIN construct. In certain circumstances the LOOP LEAVE AGAIN construct expresses the action of the loop without being error-prone.* This standard acknowledges the general LOOP LEAVE AGAIN construct as acceptable.

*E. Solonay, J. Bonar, and K. Ehrlich, "Cognitive Strategies and Looping Constructs: An Empirical Study," Communication of the ACM, 26, p853, 1983.

APPENDIX G

CASE STRUCTURE AND REPEATED ELSEIF CONSTRUCTS

The CASE structure applies to situations where at most one of a mutually exclusive set of conditions can exist at some point in a computation. The CASE statement does not exist in FORTRAN-77, but a CASE structure can provide a multipath switch to select the course of computation based on the specific value of the condition which is in effect. A CASE structure can be implemented in FORTRAN-77. It can increase the clarity of code because, if used in a consistent way, one can immediately recognize that a decision is being made among a mutually exclusive set of options.

When such a situation presents itself to a FORTRAN-77 programmer, two choices are available. A computed GO TO can be used if the set of options is determined by an integer value in the range $1 \dots N$. The other choice is the repeated ELSEIF construct. The repeated ELSEIF construct is preferred over the computed GO TO statement.

The repeated ELSEIF construct has the following desirable features:

- No GO TO statements are used. GO TO statements usually obscure the code.
- Entry to a repeated ELSEIF is at the top of the construct, flow of control is linear, and exit is at the bottom. Within the flow of control, each block is either executed or bypassed, based upon a condition tested at the start of the block.
- Control exits the repeated ELSEIF structure following the execution of the first code segment which follows a successful test. Frequently the mathematics of a computation favors one condition over the others; placing this condition first optimizes average performance while maintaining sound programming methodology. Other FORTRAN-77 programming methods force best case, average case, and worst case performance to be the same (equal to the worst case); this can be computationally expensive if there are several paths.
- The trapping of unexpected errors, which occur when none of the anticipated conditions hold, is automatic with the use of an ELSE block following the final ELSEIF block. If the repeated ELSEIF construct is not used, then special testing of the error condition is forced into a separate statement. Since such a separate test must be coded explicitly, and must redundantly test the mutual exclusion of all the other tests, there is potential for introducing errors. Any change in a disjunctive test must be reflected in the separate exclusion test, which makes the code more error prone and modification more difficult.

END

12-86

DTIC