NASA Technical Memorandum 86407

GUIDELINES For Developing Structured FORTRAN Programs

March 1985

Central Scientific Computing Complex

Document N-38

NASA

National Aeronautics and Space Administration

Langley Research Center

Hampton, Virginia 23665

NASA Technical Memorandum 86407

GUIDELINES FOR DEVELOPING STRUCTURED FORTRAN PROGRAMS

B. M. EARNEST

MARCH 1985

NASA
National Aeronautics and
Space Administration

Langley Research Center
Hampton Virginia 23665

Guidelines for Developing Structured
FORTRAN Programs

March 1985

1. INTRODUCTION

This document describes computer programming and coding standards which represent guidelines for the uniform writing of FORTRAN 77 programs at NASA Langley. Wherever possible, these guidelines should be adopted as the required standards for program development. For example, these guidelines are to be followed in the design of wind tunnel data reduction programs utilizing the Langley Central Scientific Computing Center.

Not all of the capabilities and options of FORTRAN 77 are included herein; see the list of references for manuals describing other capabilities.

This document supports contemporary software engineering techniques; however, it does so without defining them with academic precision. For example, the term structured programming may mean to some readers that only the prime constructs DO-WHILE, IF-THEN-ELSE, and SEQUENCE are used when writing a program. To others it means the step-wise refinement process necessary to decompose a specification into more manageable partitions prior to coding. Both viewpoints are correct and thus the term used throughout this document denotes a label for a collection of techniques that can be systematically applied to produce programs.

Reference 2 was used extensively in the development of these guidelines.

2. SOFTWARE DEVELOPMENT PHILOSOPHY

Before any programming begins there should be careful thought and planning. The high cost of software and maintenance of software makes it imperative that the program design and documentation be easily followed and understood. The following ideas drawn from several sources will aid in program design.

2.1 Need for Design

A complete and accurate design is a critical requirement for a large programming system. Given a complete system specification, in which all the requirements of the system are defined, the responsibility for a well-designed program lies with the programmer and the program manager. Documents described in FIPS Publication 38 (reference 6) would provide a complete system specification.

Programmers should use a top down design approach where all the requirements of the system are completely specified and designed before actual coding begins. Before designing or coding of a program ensure that the capability to perform its task does not already exist. This will eliminate duplication of effort.

2.2 Development Procedure

For each functional area and the modules (individual unit or subroutine) within, the following procedures should be used:

1. Programmer designs the module from requirements received.
2. Programmer writes the module, including the prologue, in English text (structured design) for review.
3. Colleagues read the module for correctness, understanding, and readability.
4. Iterate steps 1-3 until satisfactory.
5. Colleagues attend a walk-thru of designs of an entire functional area.
6. Iterate steps 1-5 until satisfactory.
7. Begin coding modules or functional areas.
8. Colleagues read code for correctness (see Appendix E).
9. Test each functional area beginning with the executive structure (use stubs for code not yet completed).
10. Initial operational capability achieved.

2.3 Modularity

Modularity is a concept that allows systematic development of programs as a set of interrelated individual units (called modules) which can be tested separately and later linked together to form a complete program.

With a modular approach, the program design stage becomes the most critical function. A top-down examination of the overall system must be resolved before coding begins.

The following attributes of modular programs should be maintained:

2.3.1 Functional Separation

Individual algorithms should be kept functionally separate for ease of production and maintenance.

1. Each algorithm should be in a separate module.
2. Keep all input statements for a file in one module.
3. Keep all output statements for a file in one module.
4. All error handling should be in one module.
5. All general purpose code should be isolated in separate modules.

2.3.2 Top Entry/Bottom Exit

Each module shall have one entry point and one exit point and utilize structured programming techniques.

2.3.3 Size Limitations

Each module should be kept as small as feasible. Modules, excluding prologue and comments, should not exceed 100 lines of code.

2.3.4 Data Transfer

The preferable way to transfer data from routine to routine in order to maintain autonomy is the use of argument lists (calling sequence). Where the argument lists are long use FORTRAN labelled common storage.

2.3.5 Environment

A labeled common block can be used to contain the parameters which define the environment required for a group of modules (e.g. working array sizes, error tolerance).

2.4 Control

A person must be assigned responsibility for each program or set of programs and someone must also be assigned responsibility for the overall system of programs.

Control over programs and associated files shall be implemented through use of a general purpose symbolic file maintenance program such as Control Data Corporation programs UPDATE or MODIFY.

Assignment of responsibilities for control of programs insures adherence to the following concepts:

2.4.1 Program Integrity

The integrity of programs must be maintained through control. The person responsible for each program, or set of programs, should document all modifications. This information should be made available to people using the program.

2.4.2 Version Capability

Even though having many versions of a program which do not differ much is not justified, there are legitimate cases where more than one version of a program is necessary.

With a modular approach, a given configuration can be built from the basic modules by compiling and loading only the modules required. By loading only the modules required and eliminating multiple program paths the storage necessary for the unneeded modules is saved.

When all versions of a module are required in memory simultaneously, multiple program paths will be used as opposed to maintaining several separate versions of the entire program.

2.5 Machine Independence

It is important that computer programs be written so that with minimum effort they may be transported to and executed on systems other than the one for which they were written. Features of the compiler which do not conform to the ANSI Standard should be avoided. For example, use of comment cards with a $ in column one and seven character names should not be allowed. The use of non-ANSI features is, however, preferable to the use of assembly language.

The use of any features which do not conform to the ANSI Standard must be approved by the person responsible for the system of programs. When these features are required and approved, they should be documented as non-ANSI features in the program document and internally in the code via comments.

2.6 User Considerations

In order for a program to have value it must be useful, therefore, it must be written with the user in mind. Complex input and output make it extremely difficult for the user to run the program.

Input should be well-defined, simple, and allow the user as much freedom as possible. Default conditions should be used so that minimum user input is

required and data which is frequently used should be stored on a file accessible to the user rather than requiring that it be supplied by the user every time.

Output should be clearly labeled and readable and be structured so that the user may suppress output that is not needed.

2.7 Configuration Management

For a large programming system it is important that complete control of the configuration and changes to the configuration be maintained and recorded. The ability to recreate previous versions of the system must be available.

Software configuration management of large systems requires a symbolic file maintenance program. With such a program, changes to and control of the system are achievable.

Control Data Corporation (CDC) supplies two symbolic file maintenance programs (UPDATE and MODIFY). These programs have all the facilities necessary for configuration control on programs installed on CYBER 170 Series computer systems.

For a more specific set of standards for configuration management see references 9 and 13.

3. DOCUMENTATION

Two problems usually exist regarding computer program documentation: (1) it doesn't exist, or (2) it exists but is out-of-date. Documentation must be written concurrently with program development in order for it to be accurate and available when checkout is complete.

Two types of documentation are required: Comments in the source code (in-line) and reports (formal) which provide user guidance as well as program maintenance information.

3.1 Inline

This documentation is embedded within the source language of the program. Although this is neglected many times because of consuming too much time, it is a valuable aid to those who need to understand a program without studying the program itself. It is of greatest benefit if done concurrently with program development. The overall review process, from preliminary prologue review during the design phase through final code reading, is important in insuring that the documentation is current, understandable, and correct.

The two types of inline documentation are prologue and interspersed comments.

3.1.1 Prologue

This comment code appears at the beginning of every module or routine. See Chapter 4 and appendices B and C for explanation and examples.

3.1.2 Interspersed

This comment code defines executable code itself. For readability of these comment lines, it is suggested that a "C" or "*" be placed in column one and the comment be indented as appropriate with the code. See Appendix A for a description and example of inline documentation using the DOCMENT program developed by Control Data Corporation. Other examples are shown in Appendix C.

Comments should be written with a sense of style and with a feeling for what is going to help the reader understand the program. The goal is to anticipate the questions that a reader will have and answer them in advance.

3.2 Formal

This documentation includes all reports necessary to adequately define a system. Federal Information Processing Standards Publication 38 (reference 6) gives the guidelines for formal documentation of computer programs and automated data systems. It also gives guidelines to determine the level of documentation required for a particular program or system. As a minimum, formal documentation should include a program design document, a users manual, and a program maintenance manual. A call structure map is a useful item to include in the program maintenance manual of complex programs.

4. GENERAL CODING CONVENTIONS

Every program has to be tested to see if it performs as expected and also will need to be maintained and revised. The following general coding recommendations and constraints will aid in testing and maintenance of a program or subroutine.

4.1 Program Layout

Each subroutine or module in a program shall be comprised of a subroutine statement, a prologue, specification and data statements, code, and format statements. A brief discussion of each is given below.

4.1.1 Subroutine statement

The subroutine statement may be continued over several lines so that input parameters, output parameters and parameters used as both input and output appear on separate lines of the listing.

Example:

```
      SUBROUTINE ABC
     I              (IN1  ,IN2  ,...
     B               BOTH1,BOTH2,...
     O               OUT1 ,OUT2 ,...)
```

4.1.2 Prologue

The prologue is used to give vital information about the module. This information should include the purpose of the module, the programmer, program language, and any other information which will aid in the use of the module. The minimum information in the prologue would be Title, Name, Source Language, Purpose, Author, and Date. A more detailed explanation of the items to be included in the prologue and an example are given in appendices B and C.

```
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
C  Title:                                                              C
C  Name:                                                               C
C  Source Language:                                                    C
C  Purpose:                                                            C
C  Author:                                                             C
C  Date:                                                               C
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
```

4.1.3 Specifications and DATA Statements

1. PARAMETER
   The PARAMETER statement gives a name to a constant. The name may then appear anywhere a constant is permitted or required.

2. DIMENSION
   The DIMENSION statement defines symbolic names as array names and specifies the bounds of each array.

3. COMMON Data
   COMMON blocks and variables within them should be alphabetized. Labeled COMMON can be defined in a BLOCKDATA subroutine.

4. TYPE statements
   This is a way to explicitly specify the type of variable names (e.g., REAL, DOUBLE PRECISION). This can be used in place of a DIMENSION statement.

5. DATA statements
   The DATA statement is used for local variables only.

6. Statement Function definitions
   A statement function is a user-defined procedure. It is a nonexecutable, single-statement computation that applies only to the program unit containing the definition.

4.1.4 Code

The philosophy of structured programming should be used in designing and coding the modules. The use of sequence, IF-THEN-ELSE, DO-UNTIL, CASE, DO-WHILE constructs shall form the basis for all programs. To maintain the idea of modularity, routines should not exceed 100 FORTRAN statements (excluding Prologue and Comments).

4.1.5 FORMAT statements

These statements should normally be at the end of a module. If the FORMAT is only used in one place in the module, it could be listed next to the I/O statement that uses it.

4.2 Readability

An important aspect of the verification of a design or program lies in the code review. Without readable programs this exercise is futile.

A programmer must put forth some effort to create and maintain readability within the framework of FORTRAN. The difficulties can be blamed on inconsistent subroutine layouts, names that fail to accurately reflect their semantic roles, lack of structure illumination (via alignment and indentation), random assignment of statement numbers, and so forth.

An excellent article by D. D. McCracken and G. M. Weinburg on writing readable FORTRAN programs is found in the October 1972 issue of Datamation (reference 8). For other information on programming style see reference 7. The following suggestions will aid in writing readable programs.

4.2.1 Naming Conventions

Names of variables and modules should serve the purpose of helping to identify specific entities. The names should convey the meaning of the variable.

Almost without exception, confusion arises when abbreviations are chosen that result in an acronym or a word that makes the reader think of a different,

unrelated entity. Avoid acronyms which may be confused with common words. Also avoid using zero (0) as a character in a name.

In addition to the use of meaningful names, purely mathematical functions should be defined by using familiar notations to lend understanding. Thus, Z = F(X,Y) is much more suggestive than Y = X(Z,F).

The following table illustrates the concept of maintaining psychological distance between entity names. That is, names that look, sound, or are spelled alike, or have similar meaning are not distant psychologically.

Name for One Entity	Name for Another	Psychological Distance
BKRPNT	BRKPNT	Invisible (keypunch error)
MOVLT	MOVLF	Almost none
CODE	KODE	Small
OMEGA	DELTA	Large
ROOT	DISCRM	Large and informative

4.2.2 Formating to Generate Readable Listings

The term "pretty printing" is used to express the idea of indenting and spacing source code so that the listing displays the logical structure of the code and aligns the variables appearing in the declarative and common lists.

Example 1:
The following FORTRAN COMMON does not generate much eye-appeal and is also prone to error when changes are made:

```
 COMMON/NAMES/CP,CL,CD,
1MACH,TEMP,AOA,
2VEL,ACCEL
```

By selecting the nominal width that will accommodate the largest name and choosing a vertical alignment of both commas and names, the result is much better.

```
 COMMON /NAMES/ CP    , CL    , CD    ,
1               MACH  , TEMP  , AOA   ,
2               VEL   , ACCEL
```

Example 2:
To show the logical structure of code requires the alignment of statements that are logically grouped and the indentation of statement groups that are a part of larger logical units. Vertical spacing of the logical units themselves can be accomplished with blank comment statements or blank lines.

The following structured FORTRAN example illustrates these concepts:

```
      NPTSM1 = NPTS-1

C     ...PERFORM BUBBLE SORT ON TABLE
      DO 1550 I= 1,NPTSM1
        IPLUS1 = I+1

C       ...PERFORM SWAP IF NECESSARY
        DO 1500 J= IPLUS1,NTPS
          IF( TABLE(I) .GT. TABLE(J) ) THEN
              TEMP    = TABLE(I)
              TABLE(I)= TABLE(J)
              TABLE(J)= TEMP
          ENDIF
 1500   CONTINUE
 1550 CONTINUE
```

A highly modular structure is a requirement in software development. The individual routines must be small (both logically and physically), well-designed code blocks dedicated to one specific task or function.

4.2.3 Statement Label Usage

Semantically, most programmers envision labels as providing a unique identifier to different versions of a specific construct (e.g., FORMAT statements) or the capability to specify alternate paths within a routine.

An overabundance of labels in the latter category usually implies a poorly designed routine. More than likely, the routine is not a reflection of a specific task or function: It represents a collection of tasks. This, in turn, usually implies that the reader's eye will behave like a pogo stick when attempting to follow the code. On the other hand, a relatively label-free routine implies, at first glance, that one may read from top-to-bottom.

Labels can also play an additional role. Their appearance can serve to clarify program purpose and structure. Although the numbers used to represent FORTRAN labels do not convey information, categories of related entities (e.g., input formats, output formats) can be isolated by dedicating a range of numbers with a constant increment to represent a particular category. FORTRAN statement numbers should always be assigned and appear in ascending order within a module.

No matter how well designed a program may be, changes are inevitable. Changes involving the insertion of additional labeled statements should not upset the ascending order rule. It does mean that the programmer define the magnitude of the initial increment between statement numbers large enough (e.g., 50) to accommodate the insertion of new labeled statements at a reduced increment (e.g., 20). The value of the original (and reduced) increments should always be a multiple of the same number (e.g., 10).

By definition, the terms modularity and structured programming imply that programmers are developing and maintaining small, relatively label-free routines. Thus, any fear of running out of label numbers during the life of the routine is unjustified.

4.3 Linkage/Communication

Overall software development should be approached as a tool-building process. Where possible, individual modules should be written as if they are to be placed on a computer system support library for a project.

Modules that do too much or rely on COMMON instead of parameter lists for communication have little or no value as reusable packages.

4.3.1 Subroutine versus Functions

A function should be used only for its returned value. It should behave the same as in a purely mathmatical environment. Many programmers misuse functions by failing to consider the impact on the program environment. Thus, if F(X) + F(X) does not always equal 2*F(X) because X is altered, then an unwanted, difficult to detect side effect has been introduced.

Subroutines differ from functions in that they can alter formal parameters or global variables. Subroutines are not immune to side effects if heavy reliance is made on hidden globals, as opposed to the more visible parameter lists, for data communication. For example, a routine that redefines a global may produce unexpected results.

4.3.2 Parameter Lists

Where possible, total communication with a routine should be confined to the calling sequence. This is in keeping with the tool building concept of program development. A routine becomes much more attractive to another user if he can pass his own environment entirely through a calling sequence.

Exceptions will arise, especially in the highest level routines where large numbers of variables must be made available to lower level routines. Carefully chosen COMMON blocks should then be used. If a routine needs so much information as to make parameter passing impractical or impossible, then it could be that the routine is doing too much. A routine could be so highly specialized that its use as a general tool is very remote. A mix of named COMMON and formal parameters should be used.

4.3.3 COMMON Variables

Variables in labeled COMMON blocks will be used to isolate truly global variables, i.e., those variables needed by more than one routine. Such blocks should be formed by grouping related variables.

5. SPECIFIC FORTRAN CODING CONVENTIONS AND CONSTRAINTS

The coding conventions and constraints given below shall be used for well-structured FORTRAN routines. The most important is adherence to the simulated structured FORTRAN standards illustrated in Appendix B and the CDC FORTRAN Version 5 Reference Manual (Reference 4).

Arbitrary non-ANSI programming practices should not be used unless that practice can be shown to be required to accomplish a given task. If a task cannot be done in ANSI FORTRAN, extensions of the compiler (shown in Reference 4) are preferable to assembly language programming. Non-ANSI practices that are approved must be documented in the prologue and interspersed comments.

5.1 Alphanumeric Data

Use type CHARACTER as described in Reference 4.

5.2 Assignment Statements

Do not use multiple assignment statements (e.g., X=Y=Z=0).

5.3 Documentation

Every module shall have a prologue of the form illustrated in Appendix A and described in the general programming conventions (Chapter 4).

Interspersed comments shall define every branch point and block of code and appear before the code they define. Comments shall be inserted with the original code. These comments should be aligned with the code and not begin with any of the keywords of structured programming (e.g., IF).

5.4 DO Loops

1. Do not use DO loops with an iteration count of less than 3 unless the upper limit is variable or a duplication of a large block (30-50 lines) of code would result.

2. End all DO loops with the CONTINUE statement.

3. Do not end nested DO loops on the same label.

4. Do not calculate constants or variables which do not change within the range of the DO.

5.5 Flow Control Statements

1. Do not use assigned GO TO statements.

2. Do not use a GO TO to any statement preceding the GO TO statement (i.e., branching up in the routine). Exceptions are the simulated structured FORTRAN conventions of Appendix B.

3. Use only simple integers as the index of a computed GO TO. CDC FORTRAN 5 does not give a message if the index is beyond the range so

a check for a valid index should be made or there should be an error handling statement after each computed GO TO.

4. Use the more structured IF-THEN-ELSE rather than the arithmetic IF.

5.6 General Programming Suggestions

1. Continuation characters shall be the integers from one to nine (1-9), and then the letters A-J for a maximum of 19 continuation statements.

2. Use only simple integers as indices or subscripts.

3. Do not use machine dependent techniques (e.g., bit manipulation, masking, shifting, etc.) unless there is no other way to do the job. When required, these techniques shall be clearly identified in the program via comments and approved by the program manager.

4. Calculations of constants should be performed only once at the beginning of the program outside any loops.

5. If a constant is a multiple of 10, write it in scientific notation as 1.E6, never as 10**6.

6. Initialize every variable before use.

7. Values of local variables computed in a subroutine are lost upon exit.

8. Avoid temporary or shared storage.

9. Avoid STOP statements within subroutines unless there are appropriate error messages.

10. Do not use mixed mode arithmetic except explicitly (FLOAT,INT). Do not retype variables through assignment.

11. Do not assume that comparison of two or more variables will be identically zero. Do not make equal (.EQ.) tests on floating point data. Use a tolerance test.

12. For loops which iterate to a convergence criterion use a maximum overall counter to avoid getting into a closed loop if the convergence fails.

13. Do not pass constants (literals) as parameters.

14. Use the PARAMETER statement to define literal constants used in a program. One of the most error-prone programming practices is locking literal constants into the code. Keep executable code independent of specification statements. For example, a change to the size of an array should affect only the specification part of the program:

Example:

```
PARAMETER (NROWS=9,NCOLS=9)
DIMENSION MARRAY(NROWS,NCOLS)

CALL MATRIX(NROWS,NCOLS,MARRAY,.......)
```

and not

```
DIMENSION MARRAY(9,9)

CALL MATRIX(9,9,MARRAY,.......)
```

15. Avoid using literals for I/O units or limits on DO loops. Use the PARAMETER statement.

For example, use:

```
PARAMETER (MXMSLS=24, MSLUNT=5)
DIMENSION LEVARM(MXMSLS)
          .
          .
READ(MSLUNT) list
          .
          .
          .
DO 4000 N= 1,MXMSLS
          .
          .
          .
4000 CONTINUE
```

Do not use:

```
DIMENSION LEVARM(24)
          .
          .
READ(5) list
          .
          .
          .
DO 4000 N= 1,24
          .
          .
          .
4000 CONTINUE
```

16. Write only straightforward readable code. Use the construct that is the most applicable to the process/algorithm.

17. Indent code to show the structure, making the indentation increment large enough to allow for later insertions.

18. Parenthesize and space to enhance readability and to avoid ambiguity.

19. Use blanks and grouping to make code more readable. For instance, no blanks between factors, one blank between terms. For example, use A = B*C + D/E instead of A = B*C+D/E

5.7 Input/Output

1. ANSI 'File,Format' reads and writes shall be used. For example, use
   READ(IFILE,FMT) A,B instead of READ FMT, A,B
   or
   WRITE(IFILE,FMT) A,B instead of PRINT FMT, A,B

2. All unit specifiers shall be data-defined, parameter defined, or input, and not literals.

3. Unit numbers 5, 6, and 7 shall be reserved for input, output, and punch, respectively.

5.8 Linkage

1. Pass data in parameter lists to keep routines modular, passing input arguments first, both (input and output) arguments second and output arguments last.

2. There will be only one entry and one exit point.

3. If an array is passed from one subroutine to another, use the asterisk (*) form of adjustable dimension in each of the intermediate routines. When using variable dimensions, the name and dimensions must be passed in the calling sequence.

4. Do not use the same variable name more than once in a calling sequence, since this, in effect, equivalences those locations in the called routine. Insure arrays are initialized or reset with arrays and simple variables with simple variables.

5.9 Naming Conventions (COMMON Blocks)

A definitive naming convention for COMMON blocks should be selected so that different types of data are grouped together. This naming convention should be consistent across the entire program. For example, the convention could be based on program structure (input commons, output commons, real commons, integer commons, etc.) or based on program logic.

5.10 Program Control Labels

1. Begin with 10, 100, or 1000 and then increment by an even increment such that there is room for insertion of additional labels.

2. Keep labels in ascending order.

3. Right justify labels in column 5.

4. Use labels only when needed.

5.11 Specifications and DATA Statements

1. Labeled COMMON Blocks
   A labeled COMMON block shall be the same in every routine in which it appears. (i.e., exactly the same length and variable names).
   Do not mix control variables with data storage in the same common block.
   Initialize COMMON data via BLOCK DATA.
   Dimension arrays within the COMMON statement and not with a DIMENSION or declarative statement. However, variably dimensioned arrays must be dimensioned in a DIMENSION statement.

2. Blank Common
   Do not use without the approval of the program manager.

3. DATA Statements
   Align names or data to improve readability. Define all constants to the maximum precision of the machine on which the program will be executed.

4. DIMENSION Statements
   Use only for local data and variably dimensioned arrays.

5. EQUIVALENCE Statements
   Do not use the EQUIVALENCE statement without the approval of the Program Manager.

6. FORMAT Statements
   For labels use numbers much larger than other statement labels. Increment by 10 and have FORMAT statements at the end of the program. Use the H-format or apostrophe(') delimiter to define text.

5.12 Subprograms

1. Use a function only for its returned value. Do not modify global data or arguments within a function.

2. Do not interchange functions with subroutines.

3. Do not exceed 100 lines of code (not including prologue and comments) for a given routine without approval.

4. Every routine should have only one entry and one exit.

5. When available, compile each module with the option to identify non-ANSI code so it can be removed.

Appendix A

DOCMENT Explanation and Example

DOCMENT is a Control Data Corporation NOS documentation processing program. It was designed to extract documentation embedded in the program source language as comments using the asterisk or C in columns 1 to 5. These special comments define two types of documentation:

1. External documentation which is produced for the general user to show how the program functions and how the program is used.
2. Internal documentation which describes the internal characteristics of the program so that a programmer can see how the program works in order to modify the program as required.

DOCMENT comment statements contain only asterisks and blanks in columns 1-5 and text in columns 6-71. The following rules apply to the format of comment statements for documentation.

1. Comment statements with an asterisk (or C) only in column 1 indicate that this statement is a continuation of internal or external documentation, or it is a comment statement not included in the formal documentation (i.e., it is not processed by DOCMENT).
2. Comment statments with asterisks in columns 1 and 2 indicate internal documentation.
3. Comment statements with asterisks in columns 1, 2, and 3 indicate that this statement and all following statements are internal and external documentation. Documentation ends when another comment statement containing 3 asterisks in encountered.
4. A statement with four asterisks beginning in column 1 indicates that all following statements are internal documentation (whether they are comment statements or not). Documentation ends when another comment statement containing four asterisks is encountered.
5. A statement with asterisks in columns 1 through 5 indicates that this and the following comment statements are internal and external documentation and provide program overview.
6. A statement with an asterisk in column 1 and blanks in columns 2 through 71 is a blank comment statement and is used as a separator to improve readability of documentation.

The DOCMENT control statement explanation is given in the Central Scientific Computing Complex Document N-2a (Volume 1, NOS Version 1 Reference Manual, CDC publication number 60435400N).

A more complete explanation of the documentation standards and procedures is presented in Appendix I of the Central Scientific Computing Complex Document N-6 (Volume 2, NOS Version 1 Reference Manual, CDC publication Number 60445300N).

The following example developed by S. W. Pillow of SDC shows the use of the above documentation procedure and the external and internal documentation extracted by program DOCMENT.

```
      PROGRAM ADD(INPUT,OUTPUT,TAPE5=INPUT,TAPE6=OUTPUT)
*****
*     ADD - INTERACTIVE ADDER PROGRAM
*           THIS PROGRAM ADDS TWO POSITIVE INTEGERS AND WRITES THE
*           SUM TO THE OUTPUT FILE
*
*     S.W. PILLOW SYSTEM DEVELOPMENT CORP.   5/5/83  PHONE:  865-1111
*
*
***   ADD IS A FORTRAN 5 PROGRAM WHICH ADDS TWO USER SUPPLIED POSITIVE
*     INTEGERS AND RETURNS THE SUM WHILE EXECUTING IN AN INTERACTIVE
*     ENVIRONMENT.  INTEGER PAIRS CAN CONTINUE TO BE ENTERED AND SUMMED
*     AS LONG AS THE USER DESIRES.
*
*
*     INPUT
*
*     INTEGER VALUES ARE TYPED IN BY THE USER ON THE CRT (FILE "INPUT"
*     IS ACTUALLY READ)
*          FORMAT - UNFORMATTED (NO COLUMN RESTRAINTS)
*     RESTRICTION - VALUES SHOULD NOT EXCEED 9999999999
*         DEFAULT - NO DEFAULT VALUE FOR EITHER INTEGER
*
*
*     OUTPUT
*
*     ALL MESSAGES AND PROMPTS ARE WRITTEN TO THE CRT SCREEN (FILE
*             "OUTPUT" IS ACTUALLY WRITTEN)
*
*
*     ERROR MESSAGES
*
*     IF EITHER INTEGER IS NEGATIVE:  "NEGATIVE VALUE READ".
*     INPUT WILL BE REQUESTED AGAIN
*
*
*     TERMINATION
*
*     A CARRIAGE RETURN IN RESPONSE TO THE INPUT PROMPT WILL END
*     PROGRAM EXECUTION.  "EOF FOUND ON INPUT - PROGRAM END" WILL
*     BE WRITTEN TO THE SCREEN.
*
*
***


**       PROGRAM FLOW
*
*         PROMPTED INPUT OF THE INTEGER VALUES IS FOLLOWED BY A
*         CHECK FOR NEGATIVE VALUES.  IF BOTH INTEGERS ARE POS-
*         ITIVE THEY ARE SUMMED AND THE RESULT WRITTEN OUT.  IF
*         A NEGATIVE VALUE IS FOUND, NO SUM IS CALCULATED OR
*         WRITTEN FOR THAT INTEGER PAIR, AN ERROR MESSAGE IS
*         WRITTEN, AND NEW INPUT IS REQUESTED.  WHEN EOF IS FOUND
*         ON INPUT (A (CR) FROM THE CRT) THE PROGRAM STOPS.
**
```

FIGURE A1. SOURCE CODE FOR DOCMENT

```
*
*     ...  PROMPTED INPUT
*
  10  WRITE(6,'("ENTER FIRST INTEGER")')
      READ(5,*,END=20) J1
      WRITE(6,'("ENTER SECOND INTEGER")')
      READ(5,*,END=20) J2
*
*     ...IF NO NEGATIVE VALUES SUM INTEGERS AND WRITE RESULT
*
      IF(J1.LT.0.OR.J2.LT.0) THEN
          WRITE(6,'("NEGATIVE VALUE READ")')
      ELSE
**** ...EQUATION USED TO CALCULATE SUM OF INTEGERS J1 AND J2
*
          ISUM = J1 + J2
****
          WRITE(6,'(" SUM = ",'I11')') ISUM
      ENDIF
*
*     ... GO BACK AND GET NEW INPUT
*
      GO TO 10
  20  CONTINUE
*
*     ... WRITE TERMINATION MESSAGE AND STOP
*
      WRITE(6,'(" EOF FOUND ON INPUT - PROGRAM END")')
      STOP
      END
```

FIGURE A1 CON'T. SOURCE CODE FOR DOCMENT

```
PROGRAM ADD(INPUT,OUTPUT,TAPE5=INPUT,TAPE6=OUTPU 83/05/06.  EXTERNAL*

      ADD - INTERACTIVE ADDER PROGRAM
            THIS PROGRAM ADDS TWO POSITIVE INTEGERS AND WRITES THE
            SUM TO THE OUTPUT FILE

      S.W. PILLOW  SYSTEM DEVELOPMENT CORP.  5/5/83  PHONE:  865-1111

      ADD IS A FORTRAN 5 PROGRAM WHICH ADDS TWO USER SUPPLIED POSITIVE
      INTEGERS AND RETURNS THE SUM WHILE EXECUTING IN AN INTERACTIVE
      ENVIRONMENT.  INTEGER PAIRS CAN CONTINUE TO BE ENTERED AND SUMMED
      AS LONG AS THE USER DESIRES.


      INPUT

      INTEGER VALUES ARE TYPED IN BY THE USER ON THE CRT (FILE "INPUT"
      IS ACTUALLY READ)
           FORMAT - UNFORMATTED (NO COLUMN RESTRAINTS)
      RESTRICTION - VALUES SHOULD NOT EXCEED 9999999999
          DEFAULT - NO DEFAULT VALUE FOR EITHER INTEGER


      OUTPUT

      ALL MESSAGES AND PROMPTS ARE WRITTEN TO THE CRT SCREEN (FILE
              "OUTPUT" IS ACTUALLY WRITTEN)


      ERROR MESSAGES

      IF EITHER INTEGER IS NEGATIVE:  "NEGATIVE VALUE READ".
      INPUT WILL BE REQUESTED AGAIN


      TERMINATION

      A CARRIAGE RETURN IN RESPONSE TO THE INPUT PROMPT WILL END
      PROGRAM EXECUTION.  "EOF FOUND ON INPUT - PROGRAM END" WILL
      BE WRITTEN TO THE SCREEN.
```

* Program DOCMENT cuts off the program line and inserts date and the word external or internal in the heading.

FIGURE A2. EXTERNAL DOCUMENTATION OF PROGRAM

```
PROGRAM ADD(INPUT,OUTPUT,TAPE5=INPUT,TAPE6,OUTPU  83/05/06.  INTERNAL*

      ADD - INTERACTIVE ADDER PROGRAM
            THIS PROGRAM ADDS TWO POSITIVE INTEGERS AND WRITES THE
            SUM TO THE OUTPUT FILE

      S.W. PILLOW  SYSTEM DEVELOPMENT COPR.  5/5/83  PHONE:  865-1111

      ADD IS A FORTRAN 5 PROGRAM WHICH ADDS TWO USER SUPPLIED POSITIVE
      INTEGERS AND RETURNS THE SUM WHILE EXECUTING IN AN INTERACTIVE
      ENVIRONMENT.  INTEGER PAIRS CAN CONTINUE TO BE ENTERED AND SUMMED
      AS LONG AS THE USER DESIRES.

      INPUT

      INTEGER VALUES ARE TYPED IN BY THE USER ON THE CRT (FILE "INPUT"
      IS ACTUALLY READ)
           FORMAT - UNFORMATTED (NO COLUMN RESTRAINTS)
      RESTRICTION - VALUES SHOULD NOT EXCEED 9999999999
          DEFAULT - NO DEFAULT VALUE FOR EITHER INTEGER

      OUTPUT

      ALL MESSAGES AND PROMPTS ARE WRITTEN TO THE CRT SCREEN (FILE
              "OUTPUT" IS ACTUALLY WRITTEN)

      ERROR MESSAGES

      IF EITHER INTEGER IS NEGATIVE:  "NEGATIVE VALUE READ".
      INPUT WILL BE REQUESTED AGAIN

      TERMINATION

      A CARRIAGE RETURN IN RESPONSE TO THE INPUT PROMPT WILL END
      PROGRAM EXECUTION.  "EOF FOUND ON INPUT - PROGRAM END" WILL
      BE WRITTEN TO THE SCREEN.

      PROGRAM FLOW

          PROMPTED INPUT OF THE INTEGER VALUES IS FOLLOWED BY A
          CHECK FOR NEGATIVE VALUES.  IF BOTH INTEGERS ARE POS-
          ITIVE THEY ARE SUMMED AND THE RESULT WRITTEN OUT.  IF
          A NEGATIVE VALUE IS FOUND, NO SUM IS CALCULATED OR
          WRITTEN FOR THAT INTEGER PAIR, AN ERROR MESSAGE IS
          WRITTEN, AND NEW INPUT IS REQUESTED.  WHEN EOF IS FOUND
          ON INPUT (A (CR) FROM THE CRT) THE PROGRAM STOPS.

      ...EQUATION USED TO CALCULATE SUM OF INTEGERS J1 AND J2
          ISUM = J1 + J2
```

* Program DOCMENT cuts off the program line and inserts date and the word external or internal in the heading.

FIGURE A3. INTERNAL DOCUMENATION OF PROGRAM

Appendix B

Prologue Description and Example

The following description defines the content and format for suggested prologues:

```
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
C                                                                        C
C TITLE:                                                                 C
C NAME:                                                                  C
C SOURCE LANGUAGE:                                                       C
C                                                                        C
C PURPOSE:                                                               C
C                                                                        C
C PARAMETERS:                                                            C
C   INPUT:                                                               C
C   BOTH INPUT/OUTPUT:                                                   C
C   OUTPUT:                                                              C
C   KEY LOCAL VARIABLES:                                                 C
C   COMMON VARIABLES REFERENCED:                                         C
C   COMMON VARIABLES DEFINED:                                            C
C FILES:                                                                 C
C SUBROUTINES CALLED:                                                    C
C ROUTINE CALLED BY:                                                     C
C DEPENDENCIES AND RESTRICTIONS:                                         C
C                                                                        C
C COMMENTS:                                                              C
C                                                                        C
C AUTHOR:                                                                C
C DATE:                                                                  C
C CHANGES:                                                               C
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
```

The first and last lines of the prologue will contain a C or an asterisk in columns 1 to 72. All others will have a C or an asterisk in columns 1 and 72.

The prologue will contain at least the following information:

1. Title - a short title which gives subject of the module.
2. Name - a 1-6 character name which identifies the module. NAME appears in the subroutine statement.
3. Source Language - the language in which the module is written (for example, FORTRAN 5, COMPASS).
4. Purpose - a brief description of the purpose of the module and a list of its major components.
5. Author - the programmer who wrote the module.

6. Date - the date on which the module was implemented.

The following items may be included in the prologue if applicable:

1. Parameters - include input variables, output variables, variables that are both input and output, key local variables and variables in COMMON blocks. For all parameters include the following as applicable:

   a. List the name, units of measure (where applicable) and a functional description of each parameter.
   b. If the parameter is a name constant, list its value.
   c. If the parameter is a status flag or indicator, list the values the parameter can assume and a description of the state or condition associated with each.
   d. If the parameter is an multi-dimensional array, state what each dimension represents. (For example, row =, column =).
   e. For parameters in COMMON blocks list only the names used in the module. Fully document all common block parameters in the executive or in a block data routine.

2. Files - List the format, source, use (input or output), unit identification, and functional description of each file.

3. Subroutines called - list of subroutines which this routine uses.

4. Routine called by - list of routines which use this subroutine.

5. Dependencies and Restrictions - list special or unusual features which restrict or limit the performance of the module, special or unusual features which are not ANSI Standard, and identify all non-ANSI and CDC unique statements, functions, routines, or operations used in a module. Identify the code with inline comments.

6. COMMENTS - any additional information to clarify the module.

7. Changes - list of changes made to the module. Include date and programmer who made the changes.

Example: Prologue with Subroutine Statement.

```
      SUBROUTINE PRTERR
     I                (ITYCD , ICOL  , MCHK , ISEQ  , ISEQ1)

CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
C                                                                       C
C TITLE:  Print Input Error Messages.                                   C
C                                                                       C
C NAME:  PRTERR                                                         C
C                                                                       C
C SOURCE LANGUAGE:  FORTRAN 5                                           C
C                                                                       C
C PURPOSE:  TO PRINT ERROR MESSAGES ACCORDING TO INPUT TYPE.            C
C                                                                       C
C PARAMETERS:                                                           C
C    INPUT:                                                             C
C      ITYCD - INPUT TYPE (1, 2, OR 3).                                 C
C      ICOL  - INPUT DATA AS 80 CHARACTER STRING BEFORE CALL TO         C
C              FRFLD SUBROUTINE.                                        C
C      MCHK  - SWITCH USED TO PRINT MESSAGE IF THE DATA IS OUT OF       C
C              ORDER.  = 0 IF ALL DATA IS IN ORDER BY SEQUENCE NO.      C
C                      = 1 IF THE DATA IS OUT OF ORDER.                 C
C      ISEQ  - SEQUENCE NUMBER FOR STORING INPUT DATA (TYPE 3).         C
C      ISEQ1 - SEQUENCE NUMBER OF INPUT DATA.                           C
C                                                                       C
C    COMMON VARIABLES REFERENCED:                                       C
C      JOBID - IDENTIFICATION NUMBER FOR THIS SET OF DATA.              C
C      TODAY - DATE FROM COMPUTER.                                      C
C                                                                       C
C SUBROUTINE CALLED BY:                                                 C
C      STORC1 - SUBROUTINE TO STORE DATA TYPE 1.                        C
C      STORC2 - SUBROUTINE TO STORE DATA TYPE 2.                        C
C      STORC3 - SUBROUTINE TO STORE DATA TYPE 3.                        C
C                                                                       C
C AUTHOR:  B. M. EARNEST                                                C
C                                                                       C
C DATE:  11-10-84                                                       C
C                                                                       C
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
```

Appendix C

FORTRAN MODULE EXAMPLE

```
      Subroutine SCAN
     I               (RDATA ,
     B                ND    , XDVAL , XDVAL2, XDMIN, XDMAX)

CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
C                                                                       C
C Title: Accumulate Scanner Data for Daily Statistics Calculations      C
C                                                                       C
C Name:  SCAN                                                           C
C                                                                       C
C Source Language:   FORTRAN 5                                          C
C                                                                       C
C Function or Purpose:  To accumulate scanner data for the daily        C
C                       statistics calculations                         C
C                                                                       C
C Parameters:                                                           C
C    Input:   RDATA   - An array which contains one complete            C
C                       record of data                                  C
C                                                                       C
C    Output:  Five arrays where the first three elements of each        C
C             array contain the accumulated scanner data for each       C
C             type of measurement made.                                 C
C                                                                       C
C             ND      - Array containing the total number of measure-   C
C                       ments made per day.                             C
C             XDVAL   - Array containing the sum of the values          C
C                       accumulated for the day.                        C
C             XDVAL2  - Array containing the sum of the squared         C
C                       values for the day.                             C
C             XDMAX   - The maximum value for the day.                  C
C             XDMIN   - The minimum value for the day                   C
C                                                                       C
C    LOCAL                                                              C
C             DMIN    - Temporary minimum value.                        C
C             DMAX    - Temporary maximum value.                        C
C             K       - Pointer to data in the input array.             C
C                                                                       C
C Subroutines called:                                                   C
C             IFIX    - a FORTRAN intrinsic function.                   C
C                                                                       C
C Restrictions:  Designed for ERBE data Release 1, Module 6.1.4         C
C                                                                       C
C AUTHOR:  J. SATRAN                                                    C
C                                                                       C
C DATE:  2-20-84                                                        C
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
C
C    *INPUT DATA ARRAY
C
```

```
      DIMENSION RDATA(*)
C
C    *OUTPUT DATA ARRAYS
C
      DIMENSION  ND(*)        , XDVAL(*)      ,    XDVAL2(*)     ,
     1           XDMAX(*)     , XDMIN(*)
C
C
      ILP=3
C
C    *ACCUMULATE DAILY STATISTICS
C
C     ...Set initial data location
      K=7
      DO 20 I = 1,ILP
C
        N = IFIX(RDATA(K))
        IF(N.NE.0) then
           ND(I)     = ND(I) + 1
           XDVAL(I)  = XDVAL(I)  + RDATA(K+1)
           XDVAL2(I) = XDVAL2(I) + (RDATA(K+1)**2)
C
C
C             ...Find minimum and maximum values.
              DMIN      = RDATA(K+3)
              DMAX      = RDATA(K+4)
              IF( DMIN.LT.XDMIN(I)) XDMIN(I) = DMIN
              IF( DMAX.GT.XDMAX(I)) XDMAX(I) = DMAX
C
        ENDIF
           K = K+5
C
20      CONTINUE
C
  RETURN
  END
```

Appendix D

Simulated FORTRAN Structured Concepts

The following simulated structured programming concepts should be used when writing FORTRAN code. The use of these conventions help to make a readable, structured set of FORTRAN code. For IF, ELSE, ELSE IF, ENDIF, and CASE structure use BLOCKIF statements of CDC FORTRAN 5 reference manual (Reference 4). For DO, DO while, DO until, CASE and BEGIN use the following:

NOTE: S1 = Statement label or FORTRAN numbered label
p = logical condition

1. FORTRAN DO - use DO statement as described in CDC FORTRAN 5 reference manual (Reference 4).

2. DO WHILE

```
C
C  *Example of DOWHILE(P)
C
         GO TO S2
    S1  (code)
          .
          .
          .
    S2 IF(P) GO TO S1
C
```

3. DO UNTIL

```
C
C  *Example of DO UNTIL(P)
C
    S1  (code)
          .
          .
          .
          .
        IF( .NOT.(P)) GO TO S1
```

4. DO WHILE - DO

```
C
C  *Example of DO WHILE - DO
C
     S1  CONTINUE
         (CODE)
           .
           .
           .
C
C         ***WHILE
          IF( .NOT.(P)) GO TO S2
             (CODE)
                .
                .
                .
          GO TO S1
     S2   CONTINUE
```

5. CASE - use BLOCK IF structure with ELSE IF.

6. BEGIN/END Sequence

```
C
C    Begin
        .
        .
        .
        (Sequence of Code)
        .
        .
        .
C
C    End of sequence
```

Appendix E

Code Reading FORTRAN Programs

One way to assure that a FORTRAN program meets the established requirements is to have someone other than the author read the code. The following discussion defines one way to read through a FORTRAN program to check the code. This will be helpful in training personnel to code read programs and also define the minimum checking which can be expected by the author who has developed a program.

The assumption is made that the program was developed according to the guidelines in this document. This means that a program design document should exist and that the programmer used a structured approach for the program. The reader should have the FORTRAN compilation of the program and the program documentation and have access to the program specifications and file definitions if applicable.

Generally, the code-reader should look for function (is it expressed in plain language?), form (is the style clean, meaningful comments included, initialization and closeout proper?), and economy (is storage consistent, are there redundant operations, is it simple, and will it be easy to modify?).

A checklist is included to provide a ready reference for actual code-reading assignments.

FORTRAN Code Reading

Using the FORTRAN compilation with reference map:

1. Check the symbolic reference map for the one-to-one correspondence between labels and references (exceptions are formats). Check the labels for ascending order.

2. Check the map to determine if there are undefined variables or non-dimensioned arrays appearing as externals. The externals from the FORTRAN map can also be compared to those listed in the prologue.

3. Check the variables in the map to assure that parameters defined as arrays in the prologue are listed as arrays.

4. Read the prologue for understanding. Make sure that its content is consistent with the FORTRAN code implemented. Try to review it from the standpoint of a potential user. Check for enough information in the prologue to make the routine usable. Point out anything that is lacking in the prologue.

5. Step through the code comparing it to the design ensuring that the code reflects what the design indicates. Note any discrepancies.

6. Be sure that each branch does reference the correct label.

7. Mentally execute the code with extreme or unusual values to verify that the code is correct. Avoid the pitfall of relying on the inline comments to convey what the code is doing instead of actually reading the code.

8. Ensure that executable statements have not been accidentally commented out.

9. Note any comments which cannot be clearly understood or computations which do not clearly convey their functions.

10. Make certain that any machine dependent code or non-ANSI FORTRAN has been flagged as such by the programmer as well as by the compiler.

11. Ensure that the variables used in the code are the FORTRAN equivalents of those referenced in the design. Check the types of variables for consistency. For example, is a variable described as an integer actually read as an integer if formats are involved?

12. Ensure that formats and I/O lists are consistent with file definitions. The types of I/O variables should agree with the types of data on files, and the types specified in a format must agree with the file definition. For example, is integer data actually read as an integer? Also, is a binary file being input using a formatted read?

13. Check subroutine linkages. Make certain that the number of arguments is consistent. Check the types of the arguments and whether arguments

should be arrays. Where possible, check whether a COMMON variable passed as an argument is being modified (as a COMMON variable) in the receiving routine.

14. Refer to the design specifications to actually check equations and logic. Use file definitions to check format statements.

15. Check the implementation of structured program constructs.

16. Even if the program follows the design, the reader can still question the program logic. For instance, a test may indicate 'less than' and be in agreement with the design, but if 'less than or equal to' makes more sense to the reader, it should be questioned.

17. Sign or initial the code read so the programmer can ask for clarification of your notations.

18. As a follow-up activity, check with the programmer who has executed the code which you read to see if you missed anything. This will improve your ability as a code-reader.

FORTRAN Code-Reading Checklist

1. Symbolic Reference Map
   a. One-to-one correspondence between references and labels.
   b. Ascending order rule for statement labels
   c. Externals.
   d. Variables.

2. Prologue

3. Design versus code review

4. Comments

5. Machine dependent or Non-ANSI code

6. Type consistency

7. Formats and lists versus file definition

8. Subroutine linkage

9. Equations and logic versus design specification

10. Adherence to coding standards

11. Sign-off when code reading is completed

REFERENCES

(1) Anon.: American National Standards Institute, FORTRAN X3.9 1978.

(2) Bevan, R. T. and Reynolds, J. H.: Computer Programming and Coding Standards for the FORTRAN and SIMSCRIPT II.5 Programming Languages. NSWC/DL Technical Note 3878, Naval Surface Weapons Center, Dahlgren Laboratory, Dahlgren, Virginia May 1979.

(3) Fagan, M. E.: Inspecting Software Design and Code. Datamation, October 1977, pp. 133-144.

(4) FORTRAN Version 5 Reference Manual - CDC Operating Systems: NOS 1, NOS/BE1, SCOPE 2. Publ. No. 60481300G, Control Data Corporation, C.1983. (Document N-4)*

(5) Freedman, D. P. and Weinberg, G. M.: Ethno Technical Review Handbook. Ethnotech, Inc., 1977.

(6) Anon.: Guidelines for Documentation of Computer Programs and Automated Data Systems. U.S. Department of Commerce, NBS, FIPS Publ. 38 1976.

(7) Kernighan, B. W. and Planger, D. J.: The Elements of Programming Style. Second ed., McGraw-Hill Book Co., Inc., 1978.

(8) McCracken, D. D. and Weinberg, G. M.: How to Write a Readable FORTRAN Program. Datamation, October 1972, pp. 73-77.

(9) MODIFY Reference Manual - CDC Operating System: NOS 1. Publ. No. 60450100F, Control Data Corporation, C.1980. (Document N-15)*

(10) Anon.: NASA Langley Research Center, Earth Radiation Budget Experiment, Software Development Standards. ERBE 2-4-2-0-83-6-2, 1983.

(11) NOS Version 1 Reference Manual, Volume 1. Publ. No. 60435400N, Control Data Corporation, C.1981. (Document N-2a)*

(12) NOS Version 1 Reference Manual, Volume 2. Publ. No. 60445300N, Control Data Corporation, C.1981. (Document N-6)*

(13) UPDATE Version 1 Reference Manual - CDC Operating Systems: NOS 1, NOS/BE1, SCOPE 2. Publ. No. 60449900D, Control Data Corporation, C.1981. (Document N-14)*

* Documents in the Langley Central Scientific Computing Complex Documentation Series.

1 Report No	2 Government Accession No	3 Recipient's Catalog No
NASA TM- 86407		

4 Title and Subtitle	5 Report Date
Guidelines for Developing Structured FORTRAN Programs	March 1985
	6 Performing Organization Code: 992-16-05-03
7 Author(s): B. M. Earnest	8 Performing Organization Report No: N-38
	10 Work Unit No
9 Performing Organization Name and Address: NASA Langley Research Center, Hampton, VA 23665	11 Contract or Grant No
	13 Type of Report and Period Covered: Technical Memorandum
12 Sponsoring Agency Name and Address: National Aeronautics and Space Administration, Washington, DC 20546	14 Sponsoring Agency Code

15 Supplementary Notes

16 Abstract

This document is a compilation of computer programing and coding standards which serve as guidelines for the uniform writing of FORTRAN 77 programs at NASA Langley. Not all of the capabilities and options of FORTRAN 77 are included.

17 Key Words (Suggested by Author(s))	18 Distribution Statement
Programing Guidelines, FORTRAN	Unclassified-Unlimited Subject Category 61

19 Security Classif (of this report)	20 Security Classif (of this page)	21 No of Pages	22 Price*
Unclassified	Unclassifed	34	A03

* For sale by the National Technical Information Service, Springfield, Virginia 22161

End of Document