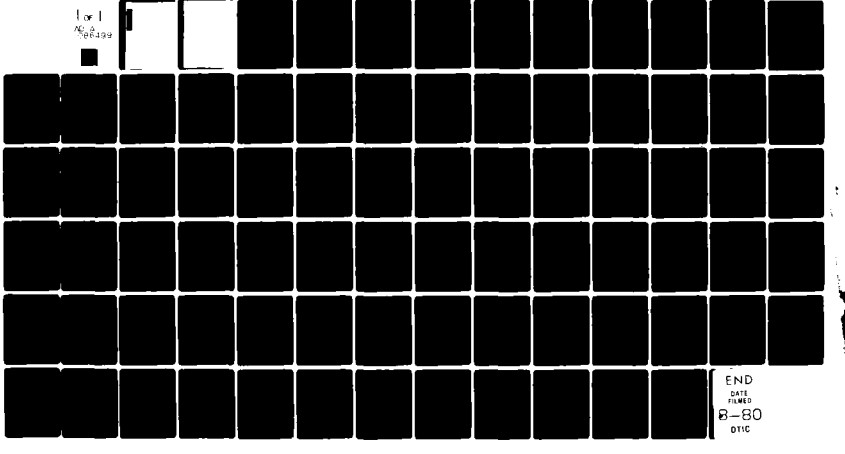
AD-A086 499 NAVAL SURFACE WEAPONS CENTER DAHLGREN VA F/G 9/2
COMPUTER PROGRAMMING AND CODING STANDARDS FOR THE FORTRAN AND S--ETC(U)
APR 80 R T BEVAN, J H REYNOLDS
UNCLASSIFIED NSWC/TR-3878 NL



UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: TR-3878	2. GOVT ACCESSION NO.: AD-A086 499
3. RECIPIENT'S CATALOG NUMBER:	
4. TITLE (and Subtitle): Computer Programming and Coding Standards for the FORTRAN AND SIMSCRIPT II.5 Programming Languages,	5. TYPE OF REPORT & PERIOD COVERED: Final
	6. PERFORMING ORG. REPORT NUMBER:
7. AUTHOR(s): Robert T. Bevan, John H. Reynolds	8. CONTRACT OR GRANT NUMBER(s):
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Naval Surface Weapons Center (K71), Dahlgren, Virginia 22448	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: N0003079WR92417, 9K50TR001
11. CONTROLLING OFFICE NAME AND ADDRESS: Naval Surface Weapons Center (K71), Dahlgren, Virginia 22448	12. REPORT DATE: Apr 1980
	13. NUMBER OF PAGES: 73
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)	15. SECURITY CLASS. (of this report): UNCLASSIFIED
	15a. DECLASSIFICATION/ DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Standards, FORTRAN, SIMSCRIPT II.5, Computer Programming Standards, Coding Standards, Software Engineering, Structured Programming, Software Development

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The computer programming and coding standards included in this report are meant to provide a general purpose set of standards for use at the Naval Surface Weapons Center (NSWC).

Contemporary studies of the software development process have shown definitively that an "ad-hoc" attempt at developing large and complex programming systems yields software that is too expensive, too difficult to maintain,

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-LF-014-6601

UNCLASSIFIED 411 561
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. inefficient, and, often, incorrect. Worse, it is very difficult to assure oneself that it is correct.

This report deals with standards that make the development of programs a more precise process. This does not mean to imply that use of a rigorous set of programming standards is a panacea for computer programming. On the contrary, use of standards merely reduces the number of errors introduced. Using standards such as those documented in this report will be seen to require more effort on the part of the programming staff than heretofore. This effort is not in vain, however, since the additional effort consists of tasks that were simply not done in the approaches of the past. Additionally, standards greatly improve the reliability, maintainability, and transportability of the resultant programs.

One final point should be made: use of the most rigorous set of programming standards does not restrict the creative process. It simply defines a consistent, organized, understandable procedure that simplifies the mundane tasks that are associated with the programming discipline, thereby freeing the programmer to design (think).

FOREWORD

This document was written in the Ballistic Sciences Branch (K71), Computer Programming Division, of the Strategic Systems Department at the Naval Surface Weapons Center (NSWC), Dahlgren, Virginia.

In 1975, K71 was requested to develop two large-scale digital computer simulations for the Fleet Ballistic Missile (FBM) Geoballistics Division (K50) of the Strategic Systems Department. Due to the complexity and magnitude of the software required, a rigorous approach to software management was followed which included the development and use of a set of programming and coding standards.

This report documents the general purpose version of those programming and coding standards.

This document was reviewed by Mr. Ira V. West, Head of the Ballistic Sciences Branch of the Computer Programming Division, and Mr. Walter P. Warner, Head of the Computer Programming Division.

Released by:

R. T. RYLAND, JR., Head
Strategic Systems Department

ACKNOWLEDGEMENT

Although the basic computer programming and coding standards of this report were developed to support specific Fleet Ballistic Missile (FBM) Weapon System simulations, many suggestions for improvement have been made and incorporated.

The personnel of the Ballistic Sciences Branch (K71) of the Computer Programming Division (K70) were instrumental in proving the worth of these standards when applied to large-scale system simulation development. Their suggestions were valuable and appreciated.

The Operation Sciences Branch (K72) of the Computer Programming Division was the first group to apply these standards to projects unrelated to those for which they were developed. That effort has further demonstrated the value of programming standards, and personnel from K72 made many worthwhile suggestions for improvements which were most appreciated. For example, the "code reading" procedure of Appendix F was developed in the Operation Sciences Branch and documented by Georgene B. Burton of K72.

The Physical Sciences Branch (K73) of the Computer Programming Division has also reviewed and recommended changes to this document to make it more general purpose. Those changes are included.

In order that these standards evolve into a tool that can be applied even more broadly, continuing suggestions from personnel actually applying these standards are essential and welcome. Of special importance are those areas of the standards that others cannot directly use for various reasons. Experience dictates that many times those reasons indicate that a change is in order that will benefit others.

With that in mind, the authors invite and solicit further recommendations for improving the worth of this document.

CONTENTS

CONTENTS (Cont'd)

CONTENTS (Cont'd)

I. INTRODUCTION

This document applies to computer programs that may be coded using either of two programming languages, FORTRAN and SIMSCRIPT II.5.

Both languages are implemented on the CDC 6700/CYBER 74 Computer Systems at the Naval Surface Weapons Center, Dahlgren, Virginia. The CDC version of FORTRAN is called "FORTRAN Extended." Some of the examples in this document assume the syntax of "FORTRAN Extended."

Throughout this report, several terms are used which may mean different things to different readers. In this report, the following meanings apply:

PROGRAMMING PROJECT MANAGER - The highest level technical person who is responsible for the correct implementation of a system of programs

MODULE - A procedure in the system of programs (synonymous with subprogram, routine, subroutine, or function)

SUBSYSTEM or FUNCTIONAL AREA - A collection of modules

EXECUTIVE - The highest level module which controls the flow through all lower level modules

MODEL - The mathematical representation of a subsystem or functional area as encoded

WALK-THRU - A design review attended by several members of the development team

STUBS - Dummy routines which "stand in" for actual modules and are used to allow testing of entire configurations before all actual modules are coded

Although this document supports contemporary software engineering techniques, 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 DOWHILE, IFTHENELSE, and SEQUENCE are used when writing a program. To others, it connotes the stepwise refinement process necessary to decompose a specification into more manageable partitions prior to coding. Both viewpoints are correct and thus the term as used throughout this report denotes a label for a collection of techniques that can be systematically applied to produce rigorous programmed solutions to problems.

The following concepts shall be used when writing programs to be included as part of any program to which these standards apply (*actual requirements are in light italics*).

II. DESIGN

A. 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 by the sponsor, the responsibility for a well-designed program lies with the programmer.

The projects to which this document applies shall use a "top-down" design approach, wherein all the requirements of the system are completely specified and designed before actual coding begins.

As a tool in defining and documenting the design, a *Program Design Language (PDL) shall be used.* A PDL is a structured, pseudo-programming language that allows one to specify design information at a high level (lots of English text), an intermediate level (some text, some structured programming constructs), or a very low level (close to a one-to-one correspondence to actual code).

The PDL to be used is that of Caine, Farber & Gordon, Inc. (see Reference 1). This PDL is similar to several others currently in use and was chosen because it also has a processor that indents and aligns the PDL statements for a neat, readable document of a given design.

B. Design Procedure

For each functional area and the modules within, the following procedure shall be used:

1. Programmer designs module.
2. Programmer writes PDL.
3. Colleague(s) reads PDL for correctness, understanding, and readability.
4. Iterate steps 1-3 until satisfactory.
5. Colleagues attend "walk-thru" of designs of an entire functional area.
6. Iterate steps 1-5 until satisfactory.
7. Begin coding of PDLs.
8. Colleague(s) reads code to insure correctness (see Appendix F).
9. Testing of functional areas begins within the executive structure (using "stubs" for code not yet completed).
10. Initial Operational Capability (IOC) achieved.

III. 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.

Given a modular approach, the "program design" stage prior to coding becomes the most critical function. A "top-down" examination of the overall system must be taken so that the interfaces among modules are resolved before coding at lower levels begins.

The following attributes of modular programs shall be maintained:

A. Functional Separation of Algorithms

1. Each algorithm in a separate module
2. All input statements for a subsystem in one module
3. All output statements for a subsystem in one module
4. All error-handling code in one module
5. All general purpose code isolated in separate modules

B. Top Entry/Bottom Exit

Each subprogram shall have one entry point (top) and one exit point (bottom) and utilize "structured programming" techniques. (Note: *for FORTRAN, the "pseudo-structures" of Appendix D must be used;* the SIMSCRIPT II.5 programming language supports structured programming.)

C. Size Limitation

Each subprogram shall be maintained as small as feasible (this is a natural result of functional separation of algorithms and needs to be artifically imposed only rarely). *Routines exceeding two pages of code must be approved by the Programming Project Manager* {these two pages do not include PROLOGUE comments (see Section VII, Documentation) or Specification and Declarative statements}.

D. Data Transfer

Use of argument lists ("call lines") is the preferable way to transfer data from routine to routine in order to maintain autonomy. However, there frequently exist cases where argument lists become prohibitively large; in these cases the use of FORTRAN COMMON or SIMSCRIPT "global" storage may be required.

IV. CONTROL

A person responsible for each program (or set of programs), known as a "control point," must exist. There shall be a control point for each program in addition to a "master" control point for the overall system of programs.

Control over programs and associated files shall be implemented through the use of a general purpose symbolic file maintenance program. For example, on Control Data Corporation (CDC) 6000 Series Computer Systems, the UPDATE program (see Reference 2).

The purpose of each control point is to insure adherence to the following concepts:

A. Program Integrity

Program integrity must be maintained through control. Unless one person is completely aware of modifications to a given program (or set of programs), program confidence is destroyed and versions proliferate.

B. Duplication of Effort

Duplication of effort must be eliminated by insuring that a capability to perform a task does not already exist before the task is implemented.

C. Version Capability

Although many versions of given programs do not differ enough to be justified, there are legitimate cases where more than one version of the object code is necessary.

Given a modular approach, one can "build" a given configuration (version) from the basic modules. For example, perhaps three different algorithms are required (for different applications) to perform a given task. Since all algorithms are isolated and compatible (interfaces resolved), *only the one algorithm necessary for a given configuration need be loaded into the computer.* By loading (and not having "option branching") the storage necessary for the "code" of the unneeded algorithms is saved. Also, as with "option branching," there is no penalty in execution time.

When all versions of a routine are required in memory simultaneously, "option branching" shall be used as opposed to maintaining several separate versions of the entire program.

V. MACHINE INDEPENDENCE

It is important that computer programs be written so that they may be executed on systems other than the one they were written for with a minimum of effort.

Therefore, *"non-standard" features of the compiler shall be avoided.* For example, use of such trivial FORTRAN options such as (1) comment cards with a "$" in column one and (2) seven-character names, will not be allowed.

However, there frequently arises the need to execute an algorithm that would require assembly language coding (certainly a non-standard feature) if "extended" features of the language were not used. Use of these features is far more preferable than the use of assembly language; for example, (1) use of FORTRAN I/O for random (direct) access files, and (2) use of FORTRAN ENCODE/DECODE for packing and unpacking data.

Use of any of these features must be approved by the Programming Project Manager.

In addition, *when these features are required and approved, they shall be documented as "non-standard features" in the program document and in the code per se via comments.*

VI. USER-ORIENTATION

It should be remembered that in order for a given program to have value it must be useful. In order to be useful 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 shall be well defined and simple and allow the user as much freedom as possible. For example, *allowing for "default" conditions* so that the user's input is minimized; *storing frequently used data on a file* accessible by the user, rather than the user having to supply it every time.

Output shall be clearly labeled and readable and be structured so that the user may suppress unwanted output.

VII. DOCUMENTATION

One of two problems usually exists regarding computer program documentation: (1) it doesn't exist or (2) it exists but is out of date. (The problem of multiple versions makes keeping documentation up to date very difficult.)

The solution to these problems is concurrency--*documentation must be written concurrently with program development* in order for it to be accurate and available when checkout is complete.

In addition, *when possible, documentation shall be stored in the computer system.* It may then be accessed via computer assisted documentation techniques.

Two types of documentation are required:

A. *In-line*

This documentation is that which appears in the program code itself. Normally neglected as consuming too much time, it is a valuable aid to one attempting to understand the program. It is of greatest benefit only if done concurrently. The overall review process, from preliminary review during the design (PDL) phase through final code reading, is most important in insuring that the documentation is current, understandable, and correct.

The terminology used in in-line documentation (comments) must be that of the system being represented, and not of the program itself. For example, one should say "check for boundary conditions", and not, "check to see if alpha > beta".

Two types of in-line documentation are required:

1. *PROLOGUE*

This comment code appears at the very start of every subprogram. See Section VIII and the sample programs of Appendices A and B for the format of PROLOGUEs.

2. *Interspersed*

This comment code defines the executable code itself. *Each comment shall be aligned with the code constructs and not begin with a "reserved" structured programming keyword* (e.g., DOWHILE), *unless, of course, that keyword is being used as part of a structured construct of Appendix D.*

For example, in FORTRAN,

```
C          CHECK NUMBER OF FRAGMENTS AGAINST MAXIMUM
           IF (.NOT.(NOFRAG .GT. MXFRAG)) GO TO 1000
                CALL ERROR (ICODE)
                .
                .
                .
  1000          CONTINUE
C          ENDIF
```

B. *Formal*

This documentation includes all reports necessary to adequately define the system. Several types of documentation are required (each of which may be separate documents if the system size or application so dictates). The types of documents required are dictated by the Department of Defense "Automated Data System Documentation Standards Manual" (4120.17-M). The names of each type of document and a brief description follow:

1. Functional Description (FD)

The FD reflects the initial definition of a programming project and provides the ultimate users with a clear statement of the operational capability to be developed. It provides a basis for mutual understanding between the development group and the user group and should be written in non-computer-oriented language.

2. Data Requirements Document (RD)

The RD defines the input required of the user, the procedures to be followed to provide this input to the system, the description of the expected output data, the specification of all use of standard data elements, and the data limitation of the system.

3. System/Subsystem Specification (SS)

The SS guides the development of large projects for systems personnel. It provides detailed definitions of the system/subsystem functions, communicates details of the ongoing analysis to the user's operational personnel, and defines in detail the interfaces with other systems and subsystems.

4. Program Specification (PS)

This document describes the program design in sufficient detail to permit program production by the programmer/coder. *Included as a part of this type is the "Program Design Language (PDL)."*

5. Data Base Specification (DS)

The DS must be prepared when many analysts and programmers will be involved in writing programs that will utilize the same data. It describes the storage allocation and data base organization and provides the basic design information necessary for the construction of the system files, tables, dictionaries, and directories.

6. Command/Management Manual (CM)

The CM presents general and specific information on a given computer program system and is directed toward an organization's general management and non-ADP personnel. It discusses how to provide input to the system, respond to requests from the system for information, and make use of output from the system.

7. Computer Operation Manual (OM)

This report provides precise and detailed information on the control requirements and operation procedures necessary to successfully initiate, run, and terminate the subject system.

Included in the OM should be instructions regarding the use of utility programs and procedures ("tools") that might have been developed to exercise the system.

8. Program Maintenance Manual (MM)

The MM presents general and specific information on the computer program. It includes a detailed technical presentation to allow the maintenance programmer to effectively maintain the system.

Utility programs and procedures ("tools") necessary to maintain the system should be documented in the MM.

9. Project Manual (PM)

For small projects, the Command/Management, Computer Operation, and Program Maintenance Manuals may be combined into one document entitled the "Project Manual (PM)."

VIII. GENERAL CODING CONVENTIONS

The following discussion details general coding conventions, recommendations, and constraints.

A. Program Layout

Each subprogram shall be comprised of the following sections:

1. *PROLOGUE Documentation*

The following description defines the content and format for PROLOGUEs in the context of a sample FORTRAN routine called NAME.

```
      SUBROUTINE NAME
   G                      (GPAR1    , GPAR2    , PI
   B                      ,GY1
   Y                      ,YPAR1    , YPAR2      )
C     AUTHOR(S)
C          JOHN H. REYNOLDS
C     CODE READ BY
C          ROBERT T. BEVAN
C     PURPOSE
C          THE PURPOSE OF "NAME" IS TO ILLUSTRATE THE CONTENT AND FORMAT OF
C          PROLOGUES.
C     DESCRIPTION
C          THE FORMAT FOR PROLOGUES THAT IS ILLUSTRATED IS FOR SUBPROGRAMS, AND
C          NOT "MAIN" PROGRAMS.  FOR "MAIN" PROGRAMS, THE FORMAT DIFFERS ONLY
C          IN FOUR WAYS:
C
C               1)  "CALLING SEQUENCE" SHOULD BE REPLACED WITH
C               THE "PROGRAM CARD" DEFINING REQUIRED FILES.
C
```

```
C               2)  A "FILES" CATEGORY IS ADDED UNDER THE
C               KEYWORD COMMUNICATION (SEE BELOW).  EACH FILE
C               SHALL BE DEFINED IN A FORMAT SIMILAR TO THE
C               "LOCAL GLOSSARY"; HOWEVER, THE VALUE IN
C               PARENTHESES SHALL INDICATE WHETHER THE FILE
C               IS AN "INPUT (I)" FILE, AN "OUTPUT (O)" FILE, A
C               "BOTH INPUT AND OUTPUT (B)" FILE, OR A
C               "SCRATCH (S)" FILE.  ACTUAL UNIT NUMBERS, IF
C               USED, SHALL BE ASSOCIATED WITH THE FILE DEFINI-
C               TIONS IN THE "FILES" CATEGORY.
C
C               3)  UNDER "ASSOCIATED SUBPROGRAMS", ONLY THE
C               "CALLS TO" NEED BE INCLUDED.
C
C               4)  THE EXTERNAL FILES CATEGORY IS ADDED.
C
C          PROLOGUE INDENTATION BEGINS IN COLUMN 5, INDENTING EVERY FIVE (5)
C          COLUMNS THEREAFTER.  PARAGRAPHS OF TEXT BEGIN IN COLUMN 10, AND ARE
C          NOT FURTHER INDENTED, BUT ARE SEPARATED FROM EACH OTHER BY BLANK
C          CARD IMAGES (WITH THE APPROPRIATE COMMENT CHARACTER).
C
C          THE "AUTHOR(S)" CATEGORY SHOULD INDICATE NOT ONLY THE NAME OF THE
C          AUTHOR OF THE COMPUTER PROGRAM, BUT ALSO THE NAME OF THE COGNIZANT
C          ANALYST, WHERE APPROPRIATE.
C
C          VARIABLE DEFINITIONS, WHETHER "COMMUNICATION" OR "LOCAL GLOSSARY",
C          BEGIN IN COLUMN 20, ALLOWING 6 CHARACTERS FOR FORTRAN NAMES,
C          FOLLOWED BY ('TYPE'), BLANK=BLANK, AND THE DEFINITION ITSELF
C          ALIGNED IN COLUMN 32.  'TYPE' IS EITHER 'A' FOR ALPHA, 'R' FOR
C          REAL, 'I' FOR INTEGER, 'X' FOR EXTERNAL ROUTINES, 'L' FOR LOGICAL,
C          'N' FOR "NO TYPE", OR '*' FOR "SAME AS THE TYPE DEFINED IN THE
C          CALLING ROUTINE".
C
C          FOR SIMSCRIPT VARIABLE DEFINITIONS, BEGIN THE NAME IN COLUMN 20
C          (AS IN FORTRAN), AND KEEP THE DEFINITIONS ALIGNED BASED ON THE
C          "WORST CASE" NAME IN A GIVEN ROUTINE.
C
C          IF A PARAMETER IS BOTH "GIVEN" AND "YIELDED", IT SHOULD BE DEFINED
C          UNDER THE "BOTH" SECTION, AND SHOULD APPEAR IN THE PARAMETER LIST
C          AFTER THE "GIVEN" AND BEFORE THE  "YIELDED" ARGUMENTS.
C
C          PARAMETERS SHOULD NOT BE ALPHABETIZED WHEN DEFINED, BUT SHOULD
C          APPEAR IN THE SAME ORDER THAT THEY APPEAR IN THE ACTUAL CALL LINE.
C          (FOR FORTRAN FUNCTION SUBPROGRAMS, THE FUNCTION NAME ITSELF SHOULD
C          APPEAR AS THE FIRST "YIELDED" ARGUMENT.)
C
C          FOR PARAMETER LIST DEFINITIONS OF ARGUMENTS THAT ARE ACTUALLY
C          "GLOBAL" AT A HIGHER LEVEL, USE "GLOBAL" AS THE DEFINITION (SEE
C          EXAMPLE BELOW).
C
```

```
C          FOR "GLOBAL" VARIABLES (I.E., THOSE STORED IN COMMON, FOR FORTRAN),
C          ONLY THE NAME OF THE VARIABLE NEED BE SHOWN, AND NOT THE NAME WITH
C          TYPE AND DEFINITION.  ALSO, MORE THAN ONE VARIABLE NAME MAY APPEAR
C          PER CARD IMAGE.
C                                          .
C          THE DEFINITION OF THE ENTIRE SET OF GLOBAL VARIABLES SHOULD BE KEPT
C          IN A GLOSSARY OF VARIABLES THAT INCLUDES THE NAME, TYPE, UNITS,
C          DEFINITION AND THE LOCATION OF THE DATA (E.G., IN FORTRAN, THE
C          NAME OF THE FORTRAN "COMMON").
C
C          "LOCAL" DEFINITIONS SHALL BE ALPHABETIZED AND NEED NOT INCLUDE EVERY
C          LOCAL VARIABLE; FOR EXAMPLE, LOOP COUNTERS, TEMPORARY STORAGE, AND
C          THE LIKE NEED NOT BE INCLUDED.
C
C          UNDER THE ERRORS CATEGORY, AT LEAST THE ERROR NUMBER AND THE NAME OF
C          THE ROUTINE THAT SETS THE ERROR SHOULD APPEAR.  IT MAY BE DESIRABLE TO
C          ALSO INCLUDE THE ACTUAL TEXT OF THE ERROR MESSAGE.
C
C          ALL CATEGORIES SHALL ALWAYS BE ENTERED FOR THE TYPE OF ROUTINE TO
C          WHICH THEY APPLY (I.E., "MAIN" PROGRAM CATEGORIES ARE NOT REQUIRED IN
C          "SUBPROGRAMS").  WHEN THERE IS NO INFORMATION TO BE ENTERED, (NONE)
C          SHALL BE ENTERED.  WHEN THE ENTRY HAS NOT YET BEEN DETERMINED, THE
C          ENTRY "TO BE DETERMINED" SHALL BE USED.
C     RESTRICTIONS
C          THE  RESTRICTIONS  TO  USING  THE  ROUTINE  SHOULD  APPEAR  IN  THIS
C          SECTION, PREFERABLY IN A LIST FORM.
C     COMMUNICATION
C          FILES ("MAIN" PROGRAMS ONLY)
C                    TEXFIL(I) = TEXT INPUT FILE
C                    ERRFIL(O) = ERROR MESSAGE FILE
C          PARAMETER LIST
C              GIVEN ARGUMENTS
C                    GPAR1 (R) = GIVEN PARAMETER # 1 DEFINITION
C                    GPAR2 (R) = GIVEN PARAMETER # 2 DEFINITION
C                    PI    (R) = GLOBAL
C              BOTH
C                    GY1   (R) = GIVEN/YIELDED (BOTH) PARAMETER
C                                DEFINITION
C              YIELDED ARGUMENTS
C                    YPAR1 (I) = YIELDED PARAMETER # 1 DEFINITION
C                    YPAR2 (A) = YIELDED PARAMETER # 2 DEFINITION
C          GLOBAL DATA
C               GIVEN
C                    NCPW
C               BOTH
C                    (NONE)
C               YIELDED
C                    (NONE)
```

```
C    LOCAL GLOSSARY
C                   CNTER (I) = COUNTER, EXIT ON CNTER = 5
C                   MASS  (R) = WEIGHT OF BACH'S FATHER
C                   NAME1 (A) = NAME OF BEETHOVEN'S FATHER IN
C                               A-FORMAT
C    ERRORS
C          22 (NAME) - "TEXT OF ERROR MESSAGE"
C    ASSOCIATED SUBPROGRAMS
C          CALLED BY (N.A. FOR "MAIN" PROGRAMS)
C               CNAME  = NAME OF CALLING PROGRAM
C          CALLS TO
C               (NONE)
C    EXTERNAL FILES ("MAIN" PROGRAMS ONLY)
C          THIS CATEGORY SHOULD INCLUDE THE NAMES OF ALL ADDITIONAL FILES RE-
C          QUIRED FOR EXECUTION, E.G., SYSTEM LIBRARIES AND OTHER USER PROGRAM
C          LIBRARIES.
C    REFERENCE(S)
C          DOCUMENTATION FROM WHICH THE CODE WAS DERIVED
C    LANGUAGE
C          FORTRAN
C
```

2. Specification and DATA Statements (FORTRAN only)

   A) COMMON data (with appropriate types)
   COMMON blocks shall be alphabetized.

   B) Local DIMENSIONs (with appropriate types)

   C) DATA statements
   (Only local data shall be defined in DATA statements; COMMON data is to be defined in BLOCK DATA Subprograms.)

   D) Statement Functions

   E) FORMATs

3. Coding

B. Structure

The use of "structured" programming techniques shall be used in designing and coding the models. Use of SEQUENCE, IFTHENELSE, and DOWHILE constructs shall form the basis for all programs.

For routines coded in SIMSCRIPT II.5, adherence to these constructs will not be difficult, since the language itself supports them.

However, for routines coded in FORTRAN, implementation is more difficult because FORTRAN does not support "structured" constructs. Even so, *all program designs (via PDL) and the programs themselves (via "structured" FORTRAN) shall be structured.*

The conventions of Appendix D, "Simulated FORTRAN Structured Constructs," shall be used to make FORTRAN code conform to the standard constructs.

C. Readability

The primary intent of this section is to impress upon programmers that readability of the design of a program and the program itself is paramount. One wouldn't think of asking a child to begin writing English until he or she could read English. Yet we still teach programmers to write programs before they are adept at reading them.

An important aspect of the verification of a design or program lies in reading each other's code. Without readable programs this exercise is futile. (See Appendix F for hints when reading "structured" FORTRAN code as implemented with the Control Data Corporation FORTRAN Extended compiler.)

Two contrasting languages, FORTRAN and SIMSCRIPT II.5, may be used in developing models. The characteristics of SIMSCRIPT (English-like constructs, lengthy mnemonics, descriptive statement labels, free form, etc.) permit the "natural" development of a readable program.

However, a programmer must put forth some effort to obtain readability within the framework of FORTRAN. Historically, a most discouraging task has been to familiarize (or re-familiarize) oneself with the content of a FORTRAN program. The difficulties can be blamed on inconsistent subroutine layouts, mnemonics that fail to accurately reflect their semantic roles, lack of structure illumination (via alignment and indentation), random assignment of statement numbers, and so forth.

The comments and suggestions that follow are taken, in part, from Henry F. Ledgard's thoughts for developing FORTRAN readability (Reference 3).

1. Mnemonic Usage

A mnemonic should serve the purpose of assisting the human memory in identifying specific entities. Thus, the mnemonic itself should convey a meaning. On the other hand, an acronym as a program symbol does not identify an entity directly. Therefore, *it is important that acronyms be avoided*, especially those that can be interpreted as common words.

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

For example, the legal SIMSCRIPT II.5 variable "WRITE.OR.READ.MEMORY" might be shortened by a FORTRAN programmer to "WORM". To some, this may suggest a "worm gear," to others it may suggest fishing bait. Since the variable in question has something to do with "accessing memory," perhaps a better choice of mnemonic would be "ACCMEM".

Confusion can also arise when programmers choose to rename real-world variables that begin with letters that "interfere" with FORTRAN TYPE defaults for variable names. It is infrequent that one can preserve the true spelling of a physical quantity as a FORTRAN mnemonic. Thus, MASS might be explicitly declared as a "real" variable, instead of renaming it as, say, RMASS. Although this particular choice may suggest "real mass", to others it may suggest "re-entry mass".

This document requires that one of the following three options be selected regarding naming of FORTRAN variables:

a) Explicitly declare the TYPE of only those variables that "interfere" with FORTRAN TYPE defaults (see the previous discussion), or

b) Explicitly declare the TYPE of all variables in the program, or

c) Require that the FORTRAN convention for TYPE defaults be followed (i.e., names beginning with the letters 'A' through 'H' and 'O' through 'Z' are real; those beginning with the letters 'I' through 'N' are integer).

   This absence of explicit TYPE statements permits, if the need arises, the programmer to implicitly change all REAL variables, for example, to DOUBLE PRECISION. This eliminates the risk of an explicit declaration overriding any implicit declarations, as could happen in a) and b) above.

In addition to the use of mnemonics, 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, taken from Reference 3, illustrates the concept of maintaining a proper "psychological distance" between entity names. That is, names that look, sound, or are spelled alike, or have similiar meanings are not "distant."

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

Caution: *The preceding comments do not imply that the mathematical notation used in the specification should be pre-empted.* Because of the programmer-analyst interface, one should try to maintain compatibility with the formulation whenever possible. While this statement may appear to be in contradiction to the preceding comments, it is not when one considers that the analyst formulating the mathematics for the program is working from an entirely different notation. Giving complete naming responsibility to a programmer, even when the names chosen may be entirely mnemonic, may turn out to be counter productive.

2. Generating Readable Listings

The term "pretty printing" is introduced in Ledgard's book to express the idea of punching and spacing source code so that the listing itself: 1) displays an eye-appealing alignment of variables appearing in declarative and COMMON lists, and, 2) amplifies the logical structure of the code.

Examples:

The following FORTRAN COMMON does not generate much eye appeal and is also prone to error when changes are made due to its "overcrowding" of mnemonics with each other and with the continuation symbol:

```
     COMMON/NAMES/JACK,JILL,ROBERT,
    1GEORGE,NANCY,SAM,
    2IDA,JOHN
```

If one decides ahead of time that the "worst case" mnemonic field width will be the nominal width and that vertical alignment of both commas and mnemonics will be maintained, the result is:

```
     COMMON/NAMES / JACK    , JILL     , ROBERT    , GEORGE
    1             , NANCY   , SAM      , IDA
    2             , JOHN
```

To display the logical structure of code requires the alignment of statements that are logically grouped together 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 cards (with comment character, and not entirely blank card images).

Comments should be aligned with the code structures and never begin with a "reserved" structured programming keyword (e.g., IF).

The following "structured" FORTRAN example illustrates these concepts:

```
          NPTSM1 = NPTS - 1
C
C         PERFORM BUBBLE SORT ON TABLE
C
          DO 1550 I = 1, NPTSM1
               IPLUS1 = I + 1
C              PERFORM SWAP IF NECESSARY
               DO 1500 J= IPLUS1, NPTS
                    IF (.NOT.(TABLE(I) .GT. TABLE(J))) GO TO 1400
                         TEMP     = TABLE(I)
                         TABLE(I) = TABLE(J)
                         TABLE(J) = TEMP
 1400                    CONTINUE
C                    ENDIF
 1500                CONTINUE
C                ENDDO
 1550            CONTINUE
C         ENDDO
```

A highly modular structure is a requirement in software development. Thus, the individual routines must be small (both logically and physically), well-designed pieces of code dedicated to one specific task or function. Thus, the probability for change is much lower.

It must be conceded that, for very large routines undergoing continuous change, the addition of new code would cause rapid deterioration of the original structure. Without automated "re-indenters" the only recourse would be to re-punch existing statements to maintain the structure.

Even so, minimal changes can be accommodated (and indentation maintained in total) by choosing the original indentation increment large enough (e.g., 5 columns) to allow a smaller increment (e.g., 1 column) for the added statements.

3. Statement Label Usage

From a semantic point of view, most programmers envision labels as providing a) a unique identity to different versions of a specific construct (e.g., FORMAT statements), and b) 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. The label syntax (text strings) of SIMSCRIPT enhances readability in a natural way. For example, the label 'PRINT.NEXT.TIME.LINE' makes obvious the intent of the code that follows. In addition, it immediately informs the reader what will happen next when it appears as a branch target after a logical test. Of course, purely structured SIMSCRIPT programs will be label-free, i.e., "GO TO-less".

Admittedly the numbers used to represent FORTRAN labels do not convey any information. Nevertheless, 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.*

No matter how well designed a program may be, changes are inevitable. However, changes involving the insertion of additional labeled statements should not upset the "ascending order" rule. This is not to suggest that one be prepared to repunch existing labeled statements. It does suggest that, at the outset, 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., 25).*

By definition, the terms "modularity" and "structured programming" imply that programmers are developing and maintaining small, label-free routines (except, for FORTRAN, those labels necessary to support the simulated structured constructs of Appendix D). Thus, any fear of "running out" of candidate label numbers during the life of the routine is unjustified.

D. Linkage/Communication

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

Routines that do "too much," or are difficult to communicate with because of a dependency on a particular data environment (e.g., heavy reliance on global/COMMON instead of parameter lists), have little or no value as "off the shelf" packages.

1. Subroutines Versus Functions

Violations of the old programming adage, *"do not use a function when you need a subroutine and vice versa,"* occur too frequently and should be avoided.

A function should be used only for its returned value and nothing more, i.e., it should behave the same as in a purely mathematical 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 (or a global variable) 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. However, subroutines are not immune to "side effects" if heavy reliance is made on "hidden" globals, as opposed to "more visible" parameter lists, for data communication. For example, a routine that redefines a global, which serves as an input global for a subsequent call, may produce unexpected results.

2. Parameter Lists

Where possible, total communication with a routine will be confined to the call line only. 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 call line.

Of course, exceptions will arise, especially in the highest-level routines where large numbers of input variables must be made available to lower level routines.

If a routine needs too much information to make parameter passing impractical (or impossible) then it could be that the routine is doing too much. Additional breakdown into smaller modules may be necessary. On the other hand, the routine could be so highly specialized that its usage as a general tool is very remote. Thus, a *mix of globals and formal parameters can be tolerated.*

When defining call lines, SIMSCRIPT forces *all input ("given") parameters to appear first, followed by all output ("yielded") parameters.*

This same practice should also be followed for FORTRAN routines. If a "given" argument also serves as a "yielded" argument (for example, a table sorting routine that replaces the given table with the sorted table), *then the "given/yielded" ("both") argument should appear at the end of the "given" side of the list, i.e., it should be specified as a "both given and yielded" parameter. Of course, this sort of "parameter duality" should be avoided whenever possible.*

3. Global/COMMON Variables

Global (SIMSCRIPT) or COMMON (FORTRAN) variables will be used to isolate "true" global variables, i.e., those variables, needed by more than one routine, that cannot be passed as formal parameters.

IX. SPECIFIC FORTRAN CODING CONVENTIONS AND CONSTRAINTS

The following coding conventions and constraints shall be used for all FORTRAN routines, the most important of which is adherence to the simulated "structured" FORTRAN standards illustrated in Appendix D.

Exceptions to any of the following requirements may be granted only by the Programming Project Manager.

No arbitrary "non-ANSI" programming practices will be allowed (see Reference 4, ANSI Standards) unless that practice can be shown to be required to accomplish a given task. That is, if a task cannot be done in ANSI FORTRAN, "extended" features of the compiler are preferable to assembly language coding.

Non-ANSI practices that are approved must be documented in the code itself. The recommended method is to insert the characters "NON-ANSI" in card columns 72-80.

[CDC's FORTRAN Extended (FTN) compiler is documented in Reference 5.]

A. Alphanumeric Data

1. Declare as integer.

B. Assignment Statements

1. No multiple statements (e.g., no X = Y = Z = 0).

2. No direct storing of Hollerith data (e.g., no STRING=4HABCD).

C. Branching

1. No assigned GO TO statements.

2. No GO TO's to any statement preceding the GO TO statement (e.g., no branching up in the routine). Exceptions to this statement are the simulated "structured" FORTRAN conventions of Appendix D.

3. Use only simple integers as the index of a Computed GO TO; for example,

```
GO TO (1500,2000), CASENO
          and not
GO TO (1500,2000), CASENO(I)
```

D. Code Per Se

1. It is recommended that continuation characters be the integers from one to nine (1-9), and then the letters A-J (giving a maximum of 19 continuation card images). However, any non-blank, non-zero character may be used to indicate continued lines (e.g., the plus-sign).

2. No abbreviation of logical operators (e.g., no .A. for .AND.).

3. Use only simple integers as indices or subscripts.

4. No machine dependent techniques (e.g., bit manipulation, masking, shifting, etc.) are allowed, unless there is no other way of doing the job. When required, these techniques shall be clearly identified in the coding via comments.

5. No "tricky" code, only straightforward, readable code.

6. Use only one statement per card image (e.g., do not use A = B $ C = D).

7. No constant calculations (e.g., no HALF = 1./2.).

8. No "temporary" or "shared" storage.

9. Initialize EVERY variable before use.

10. Do not depend on the values of "local" variables computed on a previous call to a routine (if the program is ever overlayed, the data will be destroyed).

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

12. A maximum of one (1) STOP statement is allowed, and it must reside in a centralized "abort" routine.

13. Parenthesize and separate predicates to enhance readability and avoid ambiguity; for example, use

```
IF (.NOT.((A.GT.B .AND. C.GT.D)  .OR. (E.LT.F))) GO TO 2000
```

and not

```
IF(.NOT.(A.GT.B.AND.C.GT.D.OR.E.LT.F))GOTO2000
```

14. Use blanks to make code more readable; for example, no blanks between factors, one blank between terms, i.e.,

    A = B*C + D/E, and not A=B*C+D/E

15. Align lengthy code to make more readable.

16. No mixed MODE (TYPE) except explicitly (FLOAT, IFIX, or in assignment statement (I = A)).

17. Do not make "equal" (.EQ.) tests on floating point data, use a tolerance (unless exact equality is desired). For example,

    IF (.NOT.(ABS(A-B) .LE. TOLFP)) . . .

18. Use of literal constants in the executable code:

    One of the most error-prone practices that too many programmers use is that of locking literal constants into the code. *These standards specifically prohibit this practice.*

    Typical abuses include using literal constants for: I/O units, limits on DO loops, parameters in call lines, and physical constants in calculations. These abuses are not only error-prone (e.g., using the literal value of π in different places with differing accuracy), but they present an unnecessary maintenance headache (e.g., passing the literal size of an array to a routine and the array size changes so that many lines of code have to be found and changed). Therefore:

    a) Keep the 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 and not the code per se; therefore, use

```
C           DIMENSION MARRAY(NROWS,NCOLS)
            DIMENSION MARRAY(    9,    9)
            DATA NROWS / 9/, NCOLS / 9/
                 .
                 .
                 .
            CALL MATRX
          G             (NROWS, NCOLS, MARRAY, ...
```

    and not

```
      DIMENSION MARRAY (9,9)
           .
           .
           .
      CALL MATRX(9,9,MARRAY, . . . )
```

b) Use literal constants in the code only when they define themselves; for example, if the intent of a line of code is to increment a counter by one, then

```
COUNT = COUNT + 1
```

is a legitimate usage. On the other hand, physical constants, such as π should never appear literally as 3.14159... in the code. They should be defined mnemonically to the maximum machine precision at the highest level and made available to all routines that need them.

c) Do not use literals for I/O units or limits on DO loops. For example, use

```
C          DIMENSION LEVARM(MXMSLS)
           DIMENSION LEVARM(    24)
           REAL      LEVARM
           DATA MXMSLS/24/, MSLUNT/ 5/
                .
                .
                .
           READ (MSLUNT)  list
           DO 4000 N = 1 ,  MXMSLS
                     .
                     .
                     .
 4000           CONTINUE
C          ENDDO
```

and not

```
           DIMENSION LEVARM(24)
           REAL LEVARM
                .
                .
                .
           READ(5) list
```

```
      DO 4000 N=1,24
        .
        .
        .
 4000   CONTINUE
C       ENDDO
```

19. Do not use compound IF statements with interdependent predicates. For example,

    IF (.NOT.(IND .EQ. 9999999999 .AND. T(IND) .EQ. X) ...

    could, depending on the FORTRAN compiler and/or machine memory limitations, cause an invalid subscript reference when IND = 9999999999.

E. DO loops

1. Do not exit and re-enter the range of a DO loop.

2. No DOs with an iteration count less than 3, unless upper limit is variable, or duplication of large block of code would result.

3. End all DOs on CONTINUE, followed by a comment with ENDDO (see Appendix D).

4. No terminating nested DOs on same label.

5. No invariant calculations in the range of DOs.

F. Documentation

1. Every routine shall have a PROLOGUE of the form illustrated in Appendices A and B and described in the General Coding Conventions (Section VIII).

2. Interspersed comments shall define every branch point and block of code, appear before the code they define, and be inserted while coding, not afterwards. These comments shall be aligned with the code and should not begin with any of the "reserved" keywords of structured programming (e.g., IF), unless, of course, those keywords are being used as part of a structured construct of Appendix D.

3. Blank card images used for spacing must have 'C' in column 1.

G. IF Statements

1. No two-branch logical IF statements.

2. No arithmetic IF statements.

H. Input/Output

1. ANSI 'file,format' READs and WRITEs shall be used, i.e.,

READ (IFILE,100) A,B	and not	WRITE(IFILE,300) A,B
READ 100, A,B	and	PRINT 300, A,B

2. It is recommended that unit numbers 5, 6, and 7 be reserved for "system input," "system output," and the "system punch," respectively, since this is a widely used standard. However, based on the decision of the Programming Project Manager, other unit numbers may be selected.

3. All unit specifiers shall be data-defined or input, not literals.

4. Preserve all input, i.e., do not read into a variable and then redefine it during the course of program execution.

I. Naming Conventions (General)

1. No meaningless acronyms. Use 6-character-or-less mnemonics (i.e., names that impart meaning).

2. No use of the zero (0) character in names, unless it is part of a number (e.g., TS560, not TS56Ø).

J. Naming Conventions (COMMON)

A definitive naming convention for COMMON shall be selected, so that different types of data are grouped together. This naming convention should be consistent across the entire program and rigorously enforced. For example, one could set the convention based on program structure (input COMMONs, output COMMONs, real COMMONs, integer COMMONs, etc.) or based on program logic (executive COMMONs, fire control COMMONs, ship COMMONs, navigation COMMONs, missile COMMONs, etc.).

K. Program Control Labels

1. Begin at label 1000, increment by (at least) 50.

2. Keep in ascending order (top-to-bottom).

3. Right-justify in column 5.

4. Add labels according to the General Coding Conventions (see Section VIII).

L. Linkage

1. Pass data in parameter lists to keep routines modular, passing "given" arguments first, "both" (given and yielded) arguments second (if any), and yielded arguments last. For example,

```
        CALL RNAME
G                (ARG1, ARG2, . . .
B                ,ARG7, ARG9, . . .
Y                ,ARG6, ARG8, . . . )
```

2. If a naming conflict exists between a parameter list argument and a variable in COMMON, prefix the argument with a P for parameter.

3. If a naming conflict exists between a local variable and a variable in COMMON, prefix the local variable name with a Q.

4. RETURNS feature not allowed (i.e., multiple points to which one may return from a subprogram).

5. ENTRY feature not allowed (i.e., multiple points at which a given subprogram may be entered).

6. Do not pass expressions as arguments.

7. Do not "dummy out" a call line with the same variable name appearing more than once, since this, in effect, equivalences those locations in the called routine. Also, insure that arrays are "dummied out" with arrays, and simple variables are "dummied out" with simple variables.

8. Arrays that are passed as parameters:

   The DIMENSION of an array that is passed as a parameter is merely an indicator to the FORTRAN compiler that it is an array, i.e., no storage allocation is done within the called routine. However, for multi-dimensioned arrays, some dimension information must be passed and used in the DIMENSION statement within the receiving routine.

   The purpose of this is twofold. First, it permits the FORTRAN compiler to generate proper code for array addressing. Second, maintenance is reduced because dimensioning in the called routine

is not affected when changes in the DIMENSION of an array are made in the calling routine (or higher level routine where the array was initially declared and subsequently passed through several subprogram levels).

The general rule to be followed is that the "intermediate" dimension(s) must be passed while the "final" dimension should be dimensioned in the called routine at 1 or an obviously meaningless value such as 99999 (for a single-dimensioned array the "final" dimension is its only dimension).

Of course, if the called routine needs to use the true value of a final dimension in the code, it can certainly be passed and used, a priori, in the DIMENSION statement as well (i.e., as a VARIABLE DIMENSION).

In the following sections (a-c), the user has the option of using either 1 or 99999 as the final DIMENSION of passed arrays (where 99999 also yields the additional benefit of not causing, on some systems, an informative diagnostic).

a) Single-dimensioned

Unless the array size is well-defined (e.g., a vector), dimension the array at 1, i.e., DIMENSION ARRAY(1).

b) Doubly dimensioned

Pass the number of rows to the routine and dimension the array as DIMENSION ARRAY (NROWS,1).

c) Triply dimensioned

Pass both the number of rows and columns and dimension the array as DIMENSION ARRAY (NROWS,NCOLS,1).

9. If any array is passed from one level through a second level to a third level (or more), it must be dimensioned (as specified in 8. above) in each of the intermediate levels, as well as in the final level (where it is used).

M. Specification and Data Statements

1. Labeled COMMON

a. Use equal-length, one-definition COMMON blocks (i.e., insure that each copy of a labeled COMMON is the same length and contains the same variable names).

b. When initializing COMMON via DATA statements, use only BLOCK DATA subprograms to do so.

c. Where feasible, do not mix control variables with data storage in the same labeled COMMON.

d. Dimension arrays within the COMMON, i.e., not in a DIMENSION or declarative statement.

2. Blank COMMON

Not allowed without approval of Programming Project Manager.

3. Type Statements

Keep adjacent to associated COMMON, DIMENSION, or local definitions.

4. DIMENSION Statements

a. Use for local data or parameters only.

b. For variable dimensioning, each "intermediate" dimension must be a variable, i.e., declarations of the type "DIMENSION ARRAY (M,3,N)" are prohibited. The "final" dimension can be a variable or a constant (as defined in L.8 above).

5. DATA statements

a. No implied loops for arrays.

b. Align names, data for readability.

c. In every routine that requires the routine name as Hollerith data, data-define the routine name as 6 Hollerith characters, storing in the variable name NAMRTN. For example, for a routine named ROUNAM

```
DATA NAMRTN /6HROUNAM/
```

The purpose of NAMRTN is to make consistent the need for the routine name in Hollerith, e.g., to pass it to a "trace" routine.

d. Data-define all constants to the maximum precision of the machine on which the program will be executed.

6. FORMAT Statements

   a. Labels - use increments of 10, and

      Use 100 - 290 for input,
      Use 300 - 490 for output,
      Use 500 - 690 for those FORMATs that are both input and output

   b. Use H-format or the quote (") delimiter to define text. For routines that are to be executed on many computer systems, the H-format is preferred to reduce the conversion effort.

7. EQUIVALENCE

   Not allowed without approval of Programming Project Manager.

N. Subprograms

1. Use a FUNCTION only for its returned value, i.e., do not modify global data or arguments within a function.

2. Do not use a FUNCTION (SUBROUTINE) where a SUBROUTINE (FUNCTION) is needed.

3. Do not exceed two pages of code (does not include PROLOGUE comments or Specification and Declarative statements).

4. Every routine must have one entry (top) and one exit (bottom). The one exit should be RETURN at the end of the routine.

5. When available, compile routines with "non-ANSI diagnostics" turned on, so that non-ANSI code can be reckoned with.

6. Adhere strictly to the structured programming constructs of Appendix D. Sloppy implementation of these very important conventions defeats the purpose of an attempt to "structure" FORTRAN code!

X. SPECIFIC SIMSCRIPT II.5 CODING CONVENTIONS AND CONSTRAINTS

SIMSCRIPT II.5 is a versatile programming language that allows free-form syntax and free ordering of data structure definitions. However, to enhance the readability and consistency of SIMSCRIPT programs, *the following procedures and coding restrictions are required.*

A. The PREAMBLE

1. The order of definitions within a PREAMBLE shall be:

   a) PERMANENT ENTITIES
   b) TEMPORARY ENTITIES
   c) System variables
   d) Sets
   e) EVENT NOTICES
   f) Real variables
   g) Integer variables
   h) Alpha variables
   i) Arrays
   j) Data collection directives
   k) Declaration of all FORTRAN routines
   l) Declaration of all "releasable" SIMSCRIPT routines
   m) Declaration of all "monitored" SIMSCRIPT routines
   n) DEFINE TO MEAN declaratives

2. The mode of attributes of entities shall be defined after their specification.

3. Background mode shall be declared as INTEGER, variable type as RECURSIVE, and DIMENSION as zero.

4. Last column is 72.

5. Pack variables if it is efficient to do so. Field packing shall be used instead of bit packing whenever possible since less code is generated.

6. Use DEFINE TO MEAN statements to add more meaning to the body of the program. For example,

   ```
   DEFINE ON TO MEAN 1
   DEFINE OFF TO MEAN 0
   ```

7. Include only the set attribute and routines required.

B. Coding Standards

1. The new "structured IF" (a compiler option) shall be used.

2. Do not use numerical statement labels.

3. Use WRITE statements instead of PRINT.

4. Use free-form input whenever possible.

5. Local recursive arrays shall be released before leaving a routine.

6. The limits of FOR phrases are re-evaluated for each iteration. Thus, if the limits are invariant for the life of the loop, evaluate those limits prior to execution of the loop.

7. The JUMP AHEAD and JUMP BACK constructs are not permitted.

8. Since the SIMSCRIPT DO UNTIL test is made at the beginning of the DO, the loop may not be executed "at least once." The programmer of a structured PDL DO UNTIL should then avoid use of the SIMSCRIPT DO UNTIL, and rather use the SIMSCRIPT DO WHILE, where applicable.

C. Indenting Source Code

1. Code only one SIMSCRIPT statement per line.

2. Statements shall start in the following card columns: 1, 10, 15, 20, 25, 30, 35, 40, 45, 50. The heading PREAMBLE and all ROUTINE declarations shall begin in column 1. The terminator for these sections shall also be coded starting in column 1. The GIVEN and YIELDING argument list shall each be on a separate line (starting in column 10). All other statements must begin in column 10 or beyond. (Statement labels are discussed in item 16 below.)

3. If a statement cannot be contained on one line, use a second or third line, but indent according to rule 2 above.

4. Never break a line within a key word or name.

5. Code for SIMSCRIPT programs shall not go beyond column 72.

6. To help clarify the association of a terminator with its section, each terminator shall contain a comment giving the section header name to which the terminator belongs. For example,

```
PREAMBLE
   .
   .
   .
END      ''PREAMBLE
```

7. In the PREAMBLE, the subsection headers PERMANENT ENTITIES, TEMPORARY ENTITIES, and EVENT NOTICES shall begin in column 10.

8. Within each of the elements listed in rule 7, the major definition statement shall start in column 15 with the definition of attributes of entities (or parameters of events) starting in column 20. For example,

```
PERMANENT ENTITIES
     EVERY MISSILE HAS
          A FIRST.STAGE,
          A SECOND.STAGE,
          AND BELONGS TO A LAUNCHER
```

9. Definition statements for attributes shall begin in column 15.

10. Definitions of sets, routines, simple variables, DEFINE TO MEAN statements, and data collection directives shall begin in column 10.

11. The first statement in a routine or event (other than the routine specification itself) shall begin in column 10.

12. FOR loops should be coded in such a way that the statements controlled by the FOR are indented one level. Complex FOR statements requiring more than one line shall be indented if control clauses are present. For example,

```
FOR I = 1 TO N.MISSILE
     WITH MI.AWAY(I) > 0,
     DO
       .
       .
       .
LOOP ''  I
```

In the example, note that the DO clause is indented while the LOOP statement is brought out to the same level as the FOR statement, and that it contains a comment which helps to connect the loop to the control variable.

13. FOR loops which search for the first case should be coded as follows:

```
FOR EACH MISSILE,
     WITH MI.AWAY(I) > 0,
FIND THE FIRST CASE
IF FOUND
     .
     .
     .
ELSE
     .
     .
     .
ALWAYS
```

In the example, the clauses FIND THE FIRST CASE and IF FOUND are on the same level as the FOR, the contents of the FOR are indented, and the ELSE and ALWAYS statements are placed at the same level as the FOR.

14. Nested FOR loops should be indented for each new nesting level. For example,

```
FOR I = 1 TO N
     DO
     FOR J = 1 TO L
          DO
           .
           .
           .
     LOOP '' J
LOOP ''I
```

15. IF statements shall be coded with the logic treated as one statement, indenting as specified earlier, if the statement is too long for one card. The statements which comprise the "true" portion of the IF shall be indented one level as well as the "false" portion of the IF. For example,

```
IF A > B AND C >= 0
     LET A = A + 1
     LET C = C - 1
ELSE
     LET B = B + 1
     LET D = D - 1
ALWAYS
      .
      .
      .
```

16. Statement labels shall occupy a separate card. To make them stand out, labels shall always begin in column one. For example,

```
'PREPARE'
        LET MI.PREP(MISSILE) = TIME.V
          .
          .
          .
```

17. Comments shall precede blocks of code that perform a unified function. To help make these comments stand out, a blank comment card shall be included before and after the comment. The first

word of the comment shall line up with the block of code it identifies. For example,

```
''
''      LOAD TACTICAL PROGRAMS
''

        CREATE A J3.LIST CALLED I
          .
          .
          .
```

18. Often it is desirable to comment on a specific statement to expand on its meaning. To specify a single convention for such comments is difficult since statements may begin well into a card image and perhaps terminate on some following card. However, where possible, the comments should be coded starting in column forty-five (45). For example,

```
CALL RUNGE.KUTTA          ''INTEGRATE TO NEXT STEP
```

D. Debugging Aids

1. During the checkout phase, programs shall be compiled with the T option (line numbers printed in TRACEBACK). When individual routines are checked out, they shall be recompiled without the T option to reduce core.

2. A SNAP.R routine shall be coded which prints current values or relevant data structures when a program error is detected. The routine shall be deleted upon completion of checkout to conserve storage.

3. Always include a TRACE statement just prior to program termination so that the Dynamic Storage Map can be studied for possible storage "leakage".

4. In addition to the T option, the E option shall be used. For example,

```
SIMII5(OPT=TE)
```

5. Take advantage of the OLD PREAMBLE or VERY OLD PREAMBLE feature whenever possible.

XI. CONFIGURATION MANAGEMENT

In a large programming system *it is extremely important that complete control of the configuration and changes to the configuration be maintained and recorded.* In addition, *the ability to recreate previous versions of the system must be available.*

Software configuration management of large systems requires a "Symbolic File Maintenance Program." In many cases such a program is available from the computer system vendor (e.g., Control Data Corporation, IBM). This "tool" is so important that it is well worth the effort to write one if it is not available.

Using such a program, changes to and control of the programming system are achievable and a "Software Configuration Control Board" can function much more effectively with it.

The Control Data Corporation (CDC) supplies a symbolic file maintenance program, named UPDATE. This program has all the facilities necessary for complete configuration control on programs installed on CDC 6000 Series Computer Systems. An example of the recommended format for UPDATE (see Reference 2) program libraries is defined in Appendix C.

REFERENCES

1. Caine, Stephen H., Gordon, E. Kent, "PDL, A Tool for Software Design," Caine, Farber, and Gordon, Pasadena, California, 1975.

2. Control Data Corporation, "Update Reference Manual," CDC Publication Number 60342500, Sunnyvale, California, May 1978.

3. Ledgard, Henry F., "Programming Proverbs for FORTRAN Programmers," Hayden Publishing Co., Inc., Rochelle Park, New Jersey, 1975.

4. American National Standards Institute (ANSI), FORTRAN X3.9, 1966.

5. Control Data Corporation, "FORTRAN Extended Reference Manual," CDC Publication Number 60305601, Sunnyvale, California, January 1979.

6. Goyette, Peter J., "INPUTP (General Purpose Input Processor) User Guide," Naval Surface Weapons Center Technical Report TR-3880, Dahlgren, Virginia, January 1979.

7. Control Data Corporation, "Scope Reference Manual," CDC Publication Number 60307200, Sunnyvale, California, January 1979.

APPENDIX A

SAMPLE FORTRAN SUBPROGRAM

Appendix A - Sample FORTRAN Subprogram

```
      SUBROUTINE EXAMPL
     G                    (NITEMS
     B                    ,BASKET
     Y                    ,NSOCKS)
C    AUTHOR(S)
C         JOHN H. REYNOLDS
C    CODE READ BY
C         ROBERT T. BEVAN
C    PURPOSE
C         THIS ROUTINE DOES NOT DO ANY MEANINGFUL COMPUTATIONS.  HOWEVER, IT
C         DOES REPRESENT AN ATTEMPT TO ILLUSTRATE PREVIOUSLY DISCUSSED CONCEPTS
C         CONSIDERED NECESSARY TO OBTAIN A UNIFORM LAYOUT FOR FORTRAN ROUTINES.
C    DESCRIPTION
C         IN ADDITION TO THE PROLOGUE, THE ROUTINE ILLUSTRATES ORDERING OF
C         DECLARATIVES, NUMBERING CONVENTIONS FOR FORMATS, NAMING OF LABELED
C         COMMONS AND SO FORTH.  IT ALSO ILLUSTRATES A SUGGESTED FORMAT FOR
C         DEFINING GIVEN (G), BOTH GIVEN AND YIELDED (B), AND YIELDED (Y)
C         PARAMETERS IN THE ACTUAL PARAMETER LIST (SEE ABOVE).
C
C         THE CALL LINE MAY APPEAR TO HAVE MEANING.  IT IS INCLUDED HERE ONLY
C         TO GIVE THE REMAINDER OF THIS PROLOGUE SOME DEGREE OF SUBSTANCE WHILE
C         AT THE SAME TIME ILLUSTRATING HOW TO DOCUMENT A PARAMETER THAT SERVES
C         AS A "GIVEN" AND "YIELDED" ARGUMENT.
C
C         THUS, WE CAN ASSIGN SOME MEANING TO THE PARAMETERS BY ASSUMING THAT
C         THIS ROUTINE COUNTS AND REMOVES ALL SOCKS FROM A BASKET OF LAUNDRY.
C    RESTRICTIONS
C         (NONE)
C    COMMUNICATION
C         PARAMETER LIST
C              GIVEN ARGUMENTS
C                   NITEMS(I) = NUMBER OF LAUNDRY ITEMS
C              BOTH
C                   BASKET(A) = ARRAY OF LAUNDRY WHICH IS YIELDED AS A COMPRESSED
C                               ARRAY
C              YIELDED ARGUMENTS
C                   NSOCKS(I) = NUMBER OF INDIVIDUAL SOCKS FOUND
C         GLOBAL DATA
C              GIVEN
C                   TOTSOK
C              BOTH
C                   (NONE)
C              YIELDED
C                   (NONE)
C    LOCAL GLOSSARY
```

```
C                   DIMSKS(I) = DIMENSION OF SKSTAT
C                   SKSTAT(A) = ARRAY OF KEYWORDS DESCRIBING SOCK CONDITIONS
C    ERRORS
C         103 (EXAMPL) - "TEXT OF ERROR MESSAGE NUMBER 103"
C    ASSOCIATED SUBPROGRAMS
C         CALLED BY
C              WASHER
C         CALLS TO
C              ERROR
C    REFERENCE(S)
C         "COMPUTER PROGRAMMING/CODING STANDARDS"
C    LANGUAGE
C         FORTRAN
C
C               /SOCKS / TOTAL SOCK COUNT
         COMMON /SOCKS / TOTSOK
         INTEGER         TOTSOK
C
C        DIMENSION BASKET(NITEMS), SKSTAT(DIMSKS)
         DIMENSION BASKET(NITEMS), SKSTAT(     5)
         INTEGER   BASKET        , SKSTAT       , DIMSKS
         INTEGER   A     ,C     ,D     ,ERRNO
         DATA SKSTAT/  10HDIRTY      , 10HSMELLY   , 10HCLEAN
     1               ,  10HRIPE      , 10HHOLES    /
         DATA NAMRTN/6HEXAMPL/
C
C        INPUT FORMATS
C
   100   FORMAT (         )
   110   FORMAT (         )
   120   FORMAT (         )
C
C        OUTPUT FORMATS
C
   300   FORMAT (         )
   310   FORMAT (         )
   320   FORMAT (         )
   330   FORMAT (         )
C
```

```
C
C          ****************** START OF EXECUTABLE CODE ******************
C                    .
                     .
                     .
           NSOCKS = 0
                     .
                     .
                     .
C
C          THE FOLLOWING CODE SHOWS THE INDENTATION OF
C          THE "DOWHILE" AND "IF" STATEMENTS IN A STRUCTURED FORM.
C
           I = 1
           ERRNO = 0
C          DO WHILE I ≤ NUMBER OF SOCKS AND ERROR NUMBER = 0
                GO TO 1500
 1200           A = B
                .
                .
                .
C               COMMENT DESCRIBING NEXT "IF" TEST
                IF (.NOT.(A .GT. TOTSOK)) GO TO 1300
                     ERRNO = 103
                     CALL ERROR
     G                         (ERRNO )
                     GO TO 1450
C               ELSE (COMMENT DESCRIBING NEXT "IF" TEST)
 1300                IF (.NOT.(C .EQ. D)) GO TO 1400
                          DO 1350 J = . . .
                               BASKET(J) = BASKET(I)
 1350                          CONTINUE
C                         ENDDO
 1400                     CONTINUE
C                    ENDIF
 1450                CONTINUE
C               ENDIF
                I = I + 1
 1500           IF (I .LE. NITEMS .AND. ERRNO .EQ. 0) GO TO 1200
C          ENDDO
           .
           .
           .
           RETURN
           END
```

APPENDIX B

SAMPLE SIMSCRIPT II.5 SUBPROGRAM

Appendix B - Sample SIMSCRIPT II.5 Subprogram

```
ROUTINE MATRIX,MULTIPLY
        GIVEN FIRST.MATRIX, SECOND.MATRIX
        YIELDING PRODUCT.MATRIX, ERROR.FLAG
''  AUTHOR(S)
''       PETER J. GOYETTE
''  CODE READ BY
''       DONALD J. LEMOINE
''  PURPOSE
''       THE PURPOSE OF THIS ROUTINE IS TO MULTIPLY TWO GIVEN
''       MATRICES (FIRST.MATRIX * SECOND.MATRIX).
''  DESCRIPTION
''       AN ERROR CHECK IS INITIALLY PERFORMED TO BE SURE THAT THE INNER
''       DIMENSIONS OF THE GIVEN MATRICES ARE EQUAL.  THEN, THE ROUTINE
''       RESERVES THE SPACE NEEDED FOR THE PRODUCT MATRIX AND MULTIPLIES
''       THE TWO MATRICES.
''  RESTRICTIONS
''       (NONE)
''  COMMUNICATION
''       PARAMETER LIST
''            GIVEN ARGUMENTS
''                 FIRST.MATRIX   (R) = 1ST FACTOR IN MULTIPLICATION
''                 SECOND.MATRIX  (R) = 2ND FACTOR IN MULTIPLICATION
''            YIELDED ARGUMENTS
''                 PRODUCT.MATRIX (R) = MATRIX RESULTING FROM THE MULTIPLICATION
''                 ERROR.FLAG     (I) = AN ERROR CONDITION IS DENOTED BY A VALUE
''                                      > 0
''            BOTH ARGUMENTS
''                 (NONE)
''       GLOBAL DATA
''            GIVEN
''                 (NONE)
''            YIELDED
''                 (NONE)
''            BOTH
''                 (NONE)
''  LOCAL GLOSSARY
''                 COLUMN (I) = NUMBER OF COLUMNS IN PRODUCT MATRIX
''                 INNER  (I) = INNER DIMENSION OF GIVEN MATRICES
''                 ROW    (I) = NUMBER OF ROWS IN PRODUCT MATRIX
''  ERRORS
''       10 (MATRIX.MULTIPLY)
''  ASSOCIATED SUBPROGRAMS
''       CALLED BY
''            POLYFIT
```

```
''        CALLS TO
''             DIM.F
''             ERROR
''   REFERENCE(S)
''        RALSTON'S NUMERICAL ANALYSIS
''   LANGUAGE
''        CDC SIMSCRIPT II.5
''
          DEFINE PRODUCT.MATRIX, FIRST.MATRIX  , SECOND.MATRIX
               AS 2-DIM REAL ARRAYS
          DEFINE ERROR.FLAG, INNER     , ROW       , COLUMN
                 I         , J         , K
               AS INTEGER VARIABLES

''        CHECK FOR COLUMN DIMENSION OF FIRST MATRIX BEING
''        EQUAL TO ROW DIMENSION OF SECOND MATRIX

          LET INNER = DIM.F(FIRST.MATRIX(1,*))
          IF INNER NOT EQUAL TO DIM.F(SECOND.MATRIX(*,*))
               CALL ERROR(10)
               LET ERROR.FLAG = 1
          ELSE

''             DEFINE DIMENSIONS OF THE PRODUCT MATRIX

               LET ROW = DIM.F(FIRST.MATRIX(*,*))
               LET COLUMN = DIM.F(SECOND.MATRIX(1,*))
               RESERVE PRODUCT.MATRIX(*,*) AS ROW BY COLUMN

''             MULTIPLY MATRICES

               FOR I = 1 TO ROW
                    DO
                    FOR J = 1 TO COLUMN
                         DO
                         FOR K = 1 TO INNER
                              DO
                              LET PRODUCT.MATRIX(I,J) = FIRST.MATRIX(I,K)
                                   * SECOND.MATRIX(K,J)
                                   + PRODUCT.MATRIX(I,J)
                         LOOP  ''K
                    LOOP  ''J
               LOOP  ''I
          ALWAYS
          RETURN
END       '' MATRIX.MULTIPLY
```

APPENDIX C

RECOMMENDED STRUCTURE AND FORMAT FOR CDC UPDATE PROGRAM LIBRARIES

Appendix C
Recommended Structure and Format For CDC UPDATE Program Libraries

The format for Control Data Corporation (CDC) 6000 Series UPDATE program libraries illustrated in this appendix is that used by NSWC (K70 Division) Fleet Ballistic Missile Weapon System Simulations. Although the authors strongly recommend this format (which is supported by a comprehensive set of catalogued job control procedures and utility programs) it is shown in this report merely as an example.

UPDATE is the name of the symbolic file maintenance program for the Control Data Corporation (CDC) 6000 Series of Computer Systems. A facility of this type, which maintains a chronological history of a given file and allows one to revert to previous states of the file, is a requirement for software configuration management.

This section assumes that the reader has, at least, a basic knowledge of UPDATE and directs the reader to Reference C-1, the CDC UPDATE Reference Manual.

The program library format illustrates the use of the *DECK and *COMDECK features of UPDATE for a FORTRAN model file. Each subprogram exists in an *DECK, each subprogram PROLOGUE in an *COMDECK, and each FORTRAN COMMON in an *COMDECK (where an asterisk is the UPDATE master control character).

The UPDATE program library (P.L.) is a multi-record file delimited by UPDATE end-of-records (*WEORs). (The *WEOR commands on the P.L. tell UPDATE to write an end-of-record on the COMPILE file if the *DECK containing the *WEOR is modified; actual end-of-records do not exist on the UPDATE P.L.)

> The reason for this record structure is simply to keep the information associated with a given model in one easily-accessible location. Here then, the model GLOSSARY, ERROR MESSAGES, FORTRAN code, DOCUMENT, Design, and Input Initialization are stored on one file. In this way the probability of maintaining current documentation increases dramatically.
>
> {Additionally, from a purely functional point of view, storing all model-related information on one P.L. allows one to access common information from either the FORTRAN code section, the DOCUMENT section, or the design (PDL) section of the file (via the *COMDECK feature of UPDATE).}

It is critical that this format be maintained for all models since the catalogued job control procedures and utility programs developed assume this structure. This allows the programmer to modify and access each of the sections of the P.L.

In order that the burden of creating the first cut at a new model P.L. be relieved, a catalogued job control procedure is available to create the initial version of the file. This procedure creates a file containing all "global" *COMDECKS needed, or, optionally, only those "global" *COMDECKS needed to support the basic file structure.

Descriptions of the four records on the P.L. follow:

The first record: all *COMDECKS are stored at the beginning of each P.L. to prevent potential errors (calling an *COMDECK before it has been encountered). All FORTRAN *DECKS are also stored in the first record. This record is terminated by a special *DECK (*DECK FOREOR) containing nothing but an UPDATE end-of-record (*WEOR).

The second record: only the model DOCUMENT *DECK exists in this record. This DOC *DECK defines the name of the model and the names of all subprograms in the model with associated brief descriptions. This *DECK allows for the automatic generation of a document containing the model GLOSSARY, ERROR MESSAGES, and each subprogram PROLOGUE (i.e., the *COMDECKS in which the PROLOGUEs are stored are accessed in the DOC *DECK as well as in the FORTRAN and PDL *DECKS).

Access to this "document generating" facility is provided via a catalogued job control procedure and utility program. The utility program creates an input file for the "Computer Assisted Documentation" (CAD) Program, through which the actual document is generated. This record is also terminated by a special *DECK (*DECK DOCEOR) containing nothing but an UPDATE end-of-record (*WEOR).

The third record: contained in this record is the model design document in the form of a Program Design Language (PDL) from which a design document can be generated using the PDL Processor of Reference C-2. This record is terminated by a special *DECK (*DECK PDLEOR) containing nothing but an UPDATE end-of-record (*WEOR).

The fourth record: the initialization *DECK for the general purpose Input Processor (see Reference C-3) is contained in this record. The IPI (Input Processor Initialization) *DECK is stored on the P.L. so that all model FORTRAN COMMONs, through which the data is passed to the computational program, are available to the Input Processor (i.e., the COMMON *COMDECKS can be called). Also contained in the IPI *DECK are keywords and variables associated with the model default data environment.

Schematically, it looks like the following:

```
All COMMON *DECKS (*COMDECKS)
All FORTRAN *DECKS
     *WEOR
DOCUMENT *DECK
     *WEOR
Program Design Language (PDL) *DECK
     *WEOR
Input Processor Initialization (IPI) *DECK
```

The uniqueness for UPDATE identifiers (i.e., *DECK names and *COMDECK names) is maintained by suffixing each of several generic IDENTs with the model name (which is also a parameter to the catalogued job control procedures).

Each model (MDL) has two identifications. The first is the 6-character (or less) name that is used in the permanent file naming convention and for UPDATE IDENTs. The second is the 3-character (or less) abbreviation of the 6-character name that is used in the "model" columns in the GLOSSARY.

For example,

6 Char Name	3 Char Name	Model Description
PPIV	PPV	Platform Positioning/Initial Velocity Model
NAVSHP	NS	Navigation/Ship Model

This convention applies only to the following IDENTs {where underscore(_) means concatenation}:

Generic Name	Model Name	'IDENT' Meaning
GLO	GLO_MDL	MDL GLOssary *COMDECK
DEF	DEF_MDL	MDL Glossary DEFinitions (only) *COMDECK
ERR	ERR_MDL	MDL ERRor Messages *COMDECK
DOC	DOC_MDL	MDL DOCument *DECK (which defines all routine names and briefly describes them)

Generic Name	Model Name	'IDENT' Meaning
MDL/D	MDL/D	MDL Design *DECK containing PDL text
IPI	IPI_MDL	MDL Input Processor Initialization *DECK containing the "template" of information necessary for variables that are read in by the Input Processor (e.g., class names, class variable mnemonics, *CALLS to COMMON blocks containing those variable mnemonics, . . .)

The following illustrates the structure of a sample FORTRAN file:

```
*COMDECK GLOBALS (an empty *COMDECK, used merely as an aid to
                  reading the UPDATE directory)

        {This section contains all "global" *COMDECKS, any of
        which may be required by any subsystem "model." It is
        supplied to each of the model program libraries by the
        MASTER "control point," and is NEVER to be updated by a
        given subsystem model control point.

        Included in this section are: "global" FORTRAN COMMON
        *COMDECKS, and several "boilerplate" *COMDECKS used by
        the DOCUMENT *DECK (see *DECK DOC_MDL below) that exists
        on all subsystem model UPDATE program libraries. Exam-
        ples of these "boilerplate" *COMDECKS are: TITLE1,
        TITLE2, SECI, SECII, SECIII, GLOHEADER, ERRHEADER, and
        FORSTART.}

*COMDECK ENDGLOBAL (an empty *COMDECK, used merely as an aid to
                    reading the UPDATE directory)

*COMDECK DEF_MDL (must occur on P.L. before *COMDECK GLO_MDL)
C VARNAM (A) FT/SEC     DESCRIPTION OF VARIABLE NAME, CONTINUED      A0001 NS
C                       ON NEXT LINES IF NECESSARY.  ALL GLOSSARIES
C                       SHALL BE ALPHABETIZED BY NAME.

        {The remainder of the "model" GLOSSARY definitions
        appear here as part of the DEF_MDL (definitions of
        model) *COMDECK in alphabetical order by variable name.
```

The header delineating the columns for the glossary is stored in the global *COMDECK GLOHEADER (see above "global" *COMDECKS). The column definitions for the GLOSSARY are:

Columns	Description
1	C, the FORTRAN comment character
3-8	The variable name (<= 6 characters)
10-12	The variable type {(I) for integers, (R) for reals, (A) for alphanumerics}
14-21	The variable units
23-67	The variable definition
69-74	The name of the FORTRAN COMMON block in which the variable is stored
76-78	The 3-character model abbreviation in which the data originated}

```
*COMDECK GLO_MDL
*CALL GLOHEADER
*CALL DEF_MDL

*COMDECK ERR_MDL
*CALL ERRHEADER
C     1       SAMPLE ERROR MESSAGE # 1          FATAL
C     2       SAMPLE ERROR MESSAGE # 2          NON-FATAL
C     3       SAMPLE ERROR MESSAGE # 3          FATAL

*COMDECK FTN1/P (1st PROLOGUE)
*COMDECK FTN2/P (2nd PROLOGUE)
*COMDECK FTN3/P (3rd PROLOGUE)
    .
    .
    .

*COMDECK COMMON1 (1st FORTRAN COMMON)
*COMDECK COMMON2 (2nd FORTRAN COMMON)
*COMDECK COMMON3 (3rd FORTRAN COMMON)
    .
    .
    .

*DECK FTN1         (1st FORTRAN routine)
*CALL FTN1/P       (1st FORTRAN routine PROLOGUE call)
*CALL COMMONi      (some FORTRAN COMMON *COMDECK call)
```

```
*CALL FORSTART
*DECK FTN2          (2nd FORTRAN routine)
*CALL FTN2/P        (2nd FORTRAN routine PROLOGUE call)
*CALL COMMONi       (some FORTRAN COMMON *COMDECK call)
*CALL FORSTART
*DECK FTN3          (3rd FORTRAN routine)
*CALL FTN3/P        (3rd FORTRAN routine PROLOGUE call)
*CALL COMMONi       (some FORTRAN COMMON *COMDECK call)
*CALL FORSTART
    .
    .
    .
*DECK FOREOR (FORTRAN end-of-record)
*WEOR

*DECK DOC_MDL
```

{Note: In the DOC_MDL *DECK, the model name (with 1 or more lines of description on separate card images) is first defined, starting in column 2 or beyond and not exceeding column 50. This is followed by each routine name with up to 8 lines of description each.

The routine names should occur as follows: highest level controlling routine first, remainder of routines in alphabetical order.

Each routine name (<= 6 characters) must begin in column one, followed by the brief description of the routine on separate card images, each description line beginning in column 2 or beyond and none exceeding column 50. A maximum of 8 lines of description are allowed. These routine name descriptions are used to generate the table of contents for the final CAD document; therefore, they are normally only 1 or 2 lines long.}

```
 Model Name (<= 6 characters)
 Model Description (as many lines as needed, <= 50 columns)
NAME1
 Description of NAME1
NAME2
 Description of NAME2
 Description of NAME2 continued
    .
    .
    .
LNAME
 Description of last name
```

```
*DECK DOCEOR (DOCUMENT end-of-record)
*WEOR

*DECK MDL/D
    .
    . (PDL for this MDL)
    .
*DECK PDLEOR (PDL end-of-record)
*WEOR

*DECK IPI_MDL
    .
    . (IPI for this MDL)
    .
```

REFERENCES

C-1. Control Data Corporation, "Update Reference Manual," CDC Publication Number 60342500, Sunnyvale, California, May 1978.

C-2. Caine, Stephen H., Gordon, E. Kent, "PDL, A Tool for Software Design," Caine, Farber, and Gordon, Pasadena, California, 1975.

C-3. Goyette, Peter J., "INPUTP (General Purpose Input Processor) User Guide," Naval Surface Weapons Center Technical Report TR-3880, Dahlgren, Virginia, January 1979.

***** CASE *****

```
C          CASE ENTRY
                GO TO (s1, s2, s3, s4), CASENO
C
C               CASE 1
   s1                (code for case 1)
                     GO TO s5
C
C               CASE 2
   s2                (code for case 2)
                     GO TO s5
C
C               CASE 3
   s3                (code for case 3)
                     GO TO s5
C
C               CASE 4
   s4                (code for case 4)
C
   s5                CONTINUE
C          ENDCASE
```

***** FORTRAN DO *****

```
           DO s1 I=m1,m2,m3
                (code)
   s1           CONTINUE
C          ENDDO
```

********** SUPPLEMENTAL Constructs **********

***** DOUNTIL *****

```
C          DOUNTIL  (p)
   s1           (code)
                IF (.NOT.(p)) GO TO s1
C          ENDDO
```

***** DOWHILE-DO *****

```
C          DO
   s1           CONTINUE
                (code)
C          WHILE(p)
                IF (.NOT.(p)) GO TO s2
                (code)
                GO TO s1
   s2           CONTINUE
C          ENDDO
```

***** IF-ELSEIF-ELSE *****

```
           IF (.NOT.(p)) GO TO s1
                (code)
                GO TO sn
C          ELSEIF
   s1           IF (.NOT.(q)) GO TO s2
                (code)
                GO TO sn
C          ELSEIF
   s2           IF (.NOT.(r)) GO TO s3
                (code)
                GO TO sn
                    .
                    .
                    .
C          ELSEIF
s(m-1)          IF (.NOT.(s) GO TO sm
                (code)
                GO TO sn
C          ELSE
   sm           (code)
   sn           CONTINUE
C          ENDIF
```

APPENDIX D

SIMULATED FORTRAN STRUCTURED CONSTRUCTS

Appendix D - Simulated FORTRAN Structured Constructs

The following structured programming constructs are the only ones allowed when writing "pseudo-structured" FORTRAN code. Of all the standards documented in this report, the use of these conventions is the most important and is critical to the success of a readable, structured set of FORTRAN code. *They must be implemented exactly as defined.*

Note that this appendix contains three sections, the first containing the PREFERRED constructs, the second containing SUPPLEMENTAL constructs that, at times, make the implementation of structured coding easier in FORTRAN, and the third containing an example of actual DOWHILE use.

1) The PREFERRED constructs:

These are the basic constructs and are the only ones necessary to describe any set of logic. They consist of the PRIME control constructs, the CASE statement, and the FORTRAN DO statement.

2) The SUPPLEMENTAL constructs:

These constructs may be used only when approved by the Programming Project Manager. Waivers may be granted when it can be shown that use of the PREFERRED constructs is overly cumbersome, inefficient, or difficult to implement.

For example, implementation of a structured CASE statement can be cumbersome when the CASE number is not readily available. Computation of the CASE number via the PREFERRED IF-ELSE can cause excessive indentation leading to line overflow when there are many cases. For this reason, the IF-ELSEIF construct was added as an alternative to the CASE statement. It was not added as an arbitrary option to the IF-ELSE.

********** PREFERRED Constructs **********

***** IF-ELSE *****

```
          IF (.NOT.(p)) GO TO s1
               ("true" code)
               GO TO s2
C         ELSE
   s1          ("false" code)
   s2          CONTINUE
C         ENDIF
```

***** IF *****

```
          IF (.NOT.(p)) GO TO s1
               ("true" code)
   s1          CONTINUE
C         ENDIF
```

***** DOWHILE *****

```
C         DOWHILE (p)
               GO TO s2
   s1          (code)
   s2          IF (p) GO TO s1
C         ENDDO
```

***** CASE *****

```
C          CASE ENTRY
                GO TO (s1, s2, s3, s4), CASENO
C
C               CASE 1
   s1                (code for case 1)
                     GO TO s5
C
C               CASE 2
   s2                (code for case 2)
                     GO TO s5
C
C               CASE 3
   s3                (code for case 3)
                     GO TO s5
C
C               CASE 4
   s4                (code for case 4)
C
   s5                CONTINUE
C          ENDCASE
```

***** FORTRAN DO *****

```
           DO s1 I=m1,m2,m3
                (code)
   s1           CONTINUE
C          ENDDO
```

********** SUPPLEMENTAL Constructs **********

***** DOUNTIL *****

```
C           DOUNTIL  (p)
   s1            (code)
                 IF (.NOT.(p)) GO TO s1
C           ENDDO
```

***** DOWHILE-DO *****

```
C           DO
   s1            CONTINUE
                 (code)
C           WHILE(p)
                 IF (.NOT.(p)) GO TO s2
                 (code)
                 GO TO s1
   s2            CONTINUE
C           ENDDO
```

***** IF-ELSEIF-ELSE *****

```
            IF (.NOT.(p)) GO TO s1
                 (code)
                 GO TO sn
C           ELSEIF
   s1            IF (.NOT.(q)) GO TO s2
                 (code)
                 GO TO sn
C           ELSEIF
   s2            IF (.NOT.(r)) GO TO s3
                 (code)
                 GO TO sn
                    .
                    .
                    .
C           ELSEIF
s(m-1)           IF (.NOT.(s) GO TO sm
                 (code)
                 GO TO sn
C           ELSE
   sm            (code)
   sn            CONTINUE
C           ENDIF
```

***** IF-ELSEIF *****

```
          IF (.NOT.(p)) GO TO s1
               (code)
               GO TO sn
C         ELSEIF
   s1          IF (.NOT.(q)) GO TO s2
               (code)
               GO TO sn
C         ELSEIF
   s2          IF (.NOT.(r)) GO TO s3
               (code)
               GO TO sn
                    .
                    .
                    .
C         ELSEIF
s(n-1)         IF (.NOT.(s)) GO TO sn
               (code)
   sn          CONTINUE
C         ENDIF
```

***** BEGIN-END BLOCK *****

(used to highlight sections of code by indentation)

```
C         BEGIN
               .
               . (code)
               .
C         END
```

********** Implementation of a **********
********** FORTRAN READ Loop **********

The following samples illustrate the use of a DOWHILE in implementing a FORTRAN READ loop. {Note: These examples illustrate a DOWHILE with a simple predicate (NO EOF). Compound predicates in a READ loop require a second nested DOWHILE for correctness.}

***** DOWHILE (with two READs) *****

```
          READ (FILE) list
C         DOWHILE NO EOF
               GO TO 2000
 1000          .
               . (code)
               .
               READ (FILE) list
 2000          IF (EOF(FILE) .EQ. 0.0) GO TO 1000
C         ENDDO
```

***** DOWHILE (with one READ) *****

```
C         DOWHILE NO EOF
               GO TO 2000
 1000          .
               . (code)
               .
 2000          READ (FILE) list                      {READ and EOF tests
               IF (EOF(FILE) .EQ. 0.0) GO TO 1000     treated as one test}
C         ENDDO
```

```
                    ***** DOWHILE-DO (with one READ) *****

C         DO
 1000           READ (FILE) list
C         WHILE NO EOF
               IF (.NOT.(EOF(FILE) .EQ. 0.0)) GO TO 2000
               .
               . (code)
               .
               GO TO 1000
 2000          CONTINUE
C         ENDDO



               ***** DOWHILE (FORTRAN '77 Syntax, one READ) *****

C         DOWHILE NO EOF
               GO TO 2000
 1000          .
               . (code)
               .
 2000          READ (FILE,eof1b) LIST {where eof1b = EOF label}
               GO TO 1000
eof1b          CONTINUE
C         ENDDO
```

APPENDIX E

PDL CONVENTIONS

Appendix E - PDL Conventions

The following conventions should be followed when using the "Program Design Language" Processor of Reference E-1.

1. *Do not use the ELSEIF, UNDO, or CYCLE keywords. That is, use only "structured" constructs available through the PDL processor.*

2. Use the CALL keyword when accessing SEGMENT names, and follow the SEGMENT name with the actual routine name preceded by a dash. For example:

```
CALL RETRIEVE GRAVITY DATA - RGRAVD
```

3. On SEGMENT cards ($S), follow the name of the SEGMENT with a dash and the name of the actual routine (putting the name in parentheses after the SEGMENT name will keep the routine name from appearing in the PDL Processor table of contents and SEGMENT Reference Tree). For example:

```
$S RETRIEVE GRAVITY DATA - RGRAVD
```

4. Define the data character as a period ($DATACHR.) and use it in all variable names referenced (e.g., TIME.OF.FLIGHT).

5. Define two text segments at the beginning of the PDL, one for "system" global data (i.e., data needed by or from the "system" containing this "model"), and the other for "model" data (i.e., data used across more than one routine in the applicable subsystem model).

Also, for defining PROLOGUE text segments, follow the name with a dash, the routine name, and PROLOGUE in parentheses. For example:

```
$T RETRIEVE GRAVITY DATA - RGRAVD (PROLOGUE)
```

6. In PROLOGUEs, if something has not yet been addressed indicate that is "to be determined"; if it has been addressed and there are "none," enter the character string (NONE).

7. Use the $CDATA control to pick up implicit data items in comment cards.

8. Use parentheses for comments, not periods (since they are reserved for the data character).

REFERENCE

E-1. Caine, Stephen H., Gordon, E. Kent, "PDL, A Tool for Software Design," Caine, Farber, and Gordon, Pasadena, California, 1975.

APPENDIX F

CODE-READING "STRUCTURED" FORTRAN PROGRAMS

Appendix F - Code-Reading "Structured" FORTRAN Programs

The following discussion defines how to "code-read" a FORTRAN program. It documents and consolidates the actual procedures followed. Such a definition should be useful in training personnel so that they know what to do when assigned to "code-read" programs. It will also serve as a definition of the minimum type of checking which can reasonably be expected by someone who is having code read.

The assumption is made that the program was developed according to this document. This implies that a PDL (Program Design Language) document does exist and that the FORTRAN used is the "pseudo-structured" FORTRAN defined in Appendix D. The FORTRAN compilation is assumed to be that generated by Control Data Corporation's FORTRAN Extended compiler. In addition to the FORTRAN compilation and the PDL document, the "reader" should have available the program specification, and file definitions if applicable.

A checklist which summarizes the types of checking done by a "code reader" is included to provide a ready reference for actual "code-reading" assignments.

Check the Symbolic Reference Map for the one-to-one correspondence (i.e., 1 label reference and 1 label definition) between labels and references which should exist if this STANDARDS document has been followed (exceptions are FORMATs and the use of the CASE construct). The "ascending order" rule on labels is easy to check at this point. Also check the map to determine if the compile is "clean"; i.e., no 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. Parameters defined as arrays in the PROLOGUE should be listed as arrays in the map (even if they are just "passing through").

Read the PROLOGUE for understanding and make sure that what it contains is consistent with the FORTRAN code implemented. Try to review it from the standpoint of a potential user. Does the PROLOGUE give enough information for you to make use of the routine? If not, point out what is lacking.

Step through the code comparing it to the PDL insuring that the code reflects what the PDL indicates. If it doesn't, note discrepancies. Although it's not necessary to read "labels" to follow the code, be sure that branches do reference the correct label. 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.

When doing so, insure that executable statements have not been accidentally "commented out" (which can be detected in the Symbolic Reference Map under Statement Labels as labels with only one reference).

Note any comments which are not clearly understood or computations which do not clearly convey their function.

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

Insure that the variables used in the code are the FORTRAN equivalents of the ones referenced in the PDL. Also, 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? Variables containing Hollerith data especially should be checked to make sure that they are declared as INTEGERs.

Check subroutine linkages. Make certain that the number of arguments is consistent. Also 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.

Reference the formulation to actually check equations and logic; use file definitions to check FORMAT statements.

Check the implementation of the "pseudo-structured" FORTRAN constructs. In fact, it is helpful to note any non-adherence to Appendix D.

Although at the code-reading stage one must make a basic assumption that the PDL is correct, one can still question logic. For instance, a test may indicate "less than" and be in agreement with the PDL, but if "less than or equal to" makes more sense to the reader, he should question it.

Sign or initial the code read so you can be questioned by the programmer if your notations are not clearly understood.

As a follow-up activity, check with the programmer who has executed the code which you "read" to see if you missed anything. In this manner your repertoire of "code-reading" expertise can be immensely increased. Then if you would care to share what you learn, please inform the authors so that this discussion may be expanded.

FORTRAN CODE-READING CHECKLIST

1. Symbolic Reference Map

   A. One-to-one correspondence between references and labels
   B. "Ascending order" rule
   C. EXTERNALS
   D. Variables

2. PROLOGUE

3. PDL-versus-code review

4. TYPE (MODE) consistency

5. Subroutine linkage

6. Equations and logic versus formulation

7. FORMATs and lists versus file definitions

8. Comments

9. Machine-dependent or non-ANSI code

10. Adherence to standards

11. Sign-off code

DISTRIBUTION

Defense Technical Information Center
Cameron Station
Alexandria, Virginia 22314 (12)

Defense Printing Service
Washington Navy Yard
Washington, DC 20374

Library of Congress
Washington, DC 20540
ATTN: Gift and Exchange Division (4)

GIDEP Operations Office
Corona, CA 91720

Naval Air Development Center
Warminster, Pennsylvania 18974
ATTN: Technical Library
Code 5033 (H. Stuebing)

Naval Ocean Systems Center
271 Catalina Boulevard
San Diego, California 92152
ATTN: Code 9134 (R. Crabb)

Naval Postgraduate School
Monterey, California 93940
ATTN: Code 52 CL (L. Cox)

Naval Research Laboratory
Washington, DC 20375
ATTN: Code 7503 (K. Heninger)

Naval Ship Research & Development Center
Bethesda, Maryland 20084
ATTN: Technical Library
Code 1828 (M. Culpepper)

Naval Training Equipment Center
Orlando, Florida 32813
ATTN: Technical Library
Code N-74

Naval Underwater Systems Center
Newport, Rhode Island 02840
ATTN: Code 4451 (R. Kasik)

DISTRIBUTION (Cont'd)

Naval Weapons Center
China Lake, California 93555
ATTN: Code 31302 (J. Zenor)

Navy Personnel Research & Development Center
San Diego, California 92152
ATTN: Code P204 (M. Underwood)

Charles Stark Draper Laboratory
68 Albany Street
Cambridge, Massachusetts 02139
ATTN: Mail Station 33
Division 40-B (2)

Compro, Incorporated
24 Marshall Place
Fredericksburg, Virginia 22401

EG&G, Washington Analytical Services Center, Inc.
P.O. Box 552
Dahlgren, Virginia 22448
ATTN: Technical Library (5)

FCDSSA, Dam Neck
Virginia Beach, Virginia 23461
ATTN: Code 00T

General Electric/Ordnance Systems
Electronic Systems Division
100 Plastics Avenue
Pittsfield, Massachusetts 01201
ATTN: Code ASA (2)
Code WCCAE (1)

CSG, Incorporated
51 Main Street
Salem, New Hampshire 03079
ATTN: Mr. Frank Hill

SDC Integrated Services, Incorporated
601 Caroline Street
Fredericksburg, Virginia 22401

Sperry Univac, Incorporated
Dahlgren, Virginia 22448

DISTRIBUTION (Cont'd)

DASC
RADC/ISISI
Griffiss AFB, New York 13441

LOCAL:

E41

F10
F18

K
K02
K10
K20
K30
K31
K32
K50
K51
K53
K54
K55
K56
K60
K64
K70
K71 (100)
K72 (15)
K73 (10)
K74

N20
N22

X210 (6)
X211 (2)