CR-171 748

NASA
National Aeronautics and
Space Administration

Lyndon B. Johnson Space Center
Houston, Texas 77058

JSC-14710
JUL 30 1980

SOFTWARE DEVELOPMENT GUIDELINES

CPD 902

Job Order 53-449

Prepared By

Lockheed Electronics Company, Inc.
Systems and Services Division
Houston, Texas

Contract NAS 9-15800

For

ENGINEERING ANALYSIS DIVISION

ENGINEERING AND DEVELOPMENT DIRECTORATE

January 1979

LEC-13182

(NASA-CR-171748) SOFTWARE DEVELOPMENT GUIDELINES (Lockheed Electronics Co.) 48 p HC A03/MF A01 CSCL 09B

N84-18919

Unclas
G3/61 12354

ORIGINAL PAGE IS
OF POOR QUALITY

JSC-14710

SOFTWARE DEVELOPMENT GUIDELINES

Job Order 53-449

PREPARED BY

Nicholas Kovalevsky
Nicholas Kovalevsky
Job Order Manager
Dynamic Systems Department

[illegible]
J. M. Underwood
Technical Monitor, ADAP-I

APPROVED BY

LEC

P. H. Horsley
P. H. Horsley, Supervisor
Data Management Section

[illegible]
W. J. Reicks, Manager
Applied Mechanics Department

NASA

[illegible]
L. O. Hayman, Chief
Aerodynamics Branch

Prepared By

Lockheed Electronics Company, Inc.

For

Engineering Analysis Division

NATIONAL AERONAUTICS AND SPACE ADMINISTRATION
LYNDON B. JOHNSON SPACE CENTER
HOUSTON, TEXAS

January 1979

LEC-13182

ORIGINAL PAGE IS
OF POOR QUALITY

CONTENTS

Section		Page
1.	INTRODUCTION.	1
2.	DESIGN CONSIDERATIONS	2
2.1	ANALYSIS	2
2.2	MODULARIZATION	2
2.3	FLOWCHARTING	3
2.3.1	SYMBOLIC.	3
2.3.2	PROGRAM DESIGN LANGUAGE	3
2.4	EXISTING PROGRAMS AND SUBROUTINES.	7
2.5	COMPATIBILITY.	9
2.5.1	LOGICAL UNIT ASSIGNMENTS.	9
2.5.2	USE OF FLAGS.	9
2.5.3	CONSISTENCY AMONG VARIABLES	9
2.5.4	MACHINE DEPENDENT SOFTWARE.	10
2.5.5	USE OF SPECIAL COMPILER FEATURES. . . .	10
2.6	INPUT AND OUTPUT DATA.	10
2.6.1	GENERAL	10
2.6.2	GROUPING BY CASES	10
2.6.3	IMMEDIATE OUTPUT.	10
2.7	ADAPTABILITY TO CHECKOUT	11
2.7.1	CHECKOUT METHOD	11
2.7.2	USAGE OF WRITE STATEMENTS	11
2.8	GENERAL-PURPOSE SUBROUTINES.	11

ORIGINAL PAGE IS
OF POOR QUALITY

Section Page

3. CODING. 12

3.1 COMMENTS 12

3.1.1 GENERAL 12

3.1.2 HEADER COMMENTS 12

3.1.3 PROGRAM MODIFICATIONS 16

3.1.4 SUBROUTINE COMMENTS 16

3.1.5 DISTRIBUTION OF COMMENTS. 16

3.1.6 ERROR RECOVERY. 16

3.1.7 ARRAY DIMENSIONS. 16

3.1.8 PRINTING STYLE. 16

3.1.9 PROGRAM MODULARIZATION COMMENTS 17

3.2 INITIALIZATION 17

3.3 STATEMENT ORDERING AND NUMBERING 17

3.3.1 NON-EXECUTABLE. 17

3.3.2 FORMAT STATEMENTS 19

3.3.3 STATEMENT NUMBERING 19

3.4 SPECIFICATION STATEMENTS, COMMON STORAGE . . . 19

3.4.1 SPECIFYING VARIABLE TYPE. 19

3.4.2 COMMON BLOCKS 20

3.5 VARIABLE NAMES 20

3.5.1 GENERAL 20

3.5.2 CONSTANTS 21

3.5.3 NAMING CONVENTION 21

Section Page

3.6 ARRAYS 21

3.6.1 COMBINING 21

3.6.2 SUBSCRIPTS. 22

3.6.3 USAGE IN SUBPROGRAMS. 22

3.7 ARITHMETIC EXPRESSIONS AND STATEMENTS. 22

3.7.1 UNAMBIGUOUS USAGE 22

3.7.2 TEST FOR IMPROPER CONDITIONS. 23

3.7.3 COMPOUND EXPRESSIONS. 23

3.7.4 USAGE OF SQRT 24

3.7.5 PREFERRED CONSTRACTIONS 25

3.8 CONTROL STATEMENTS 26

3.8.1 CALCULATIONS IN A LOOP. 26

3.8.2 COMPUTED GO TO'S. 26

3.8.3 ASSIGN STATEMENTS 26

3.8.4 DO LOOPS. 27

3.8.5 CALL STATEMENTS 27

3.9 INPUT/OUTPUT 28

3.9.1 RECORD FORMAT 28

3.9.2 PLACEMENT OF I/O OPERATIONS 29

3.9.3 DEFAULT VALUES. 29

3.9.4 OUTPUT FORM 30

3.9.5 ERROR MESSAGES. 30

3.9.6 INTERMEDIATE OUTPUT 31

3.9.7 CARD READING. 31

ORIGINAL PAGE IS
OF POOR QUALITY

Section Page

3.10 SUBROUTINES. 32

3.10.1 GENERAL 32

3.10.2 CALLING ARGUMENTS 32

3.10.3 ERROR CODES 33

3.10.4 RETURN STATEMENTS 33

3.10.5 ARRAYS. 33

3.16.6 COMMON BLOCKS 34

4. CHECKOUT AIDS 35

4.1 INTERMEDIATE RESULTS 35

4.1.1 PROGRAM FLOW. 35

4.1.2 DATA STRUCTURES 36

4.1.3 VALIDITY OF RESULTS 36

4.2 DESK CHECKING. 36

4.2.1 GENERAL 36

4.2.2 PROGRAM LOGIC CHECK LIST. 37

4.2.3 DECK STRUCTURE CHECK LIST 37

4.3 CHECKOUT DATA. 38

4.3.1 GENERAL 38

4.3.2 VERIFICATION OF INPUT 38

4.4 DUMPS. 39

4.4.1 GENERAL 39

4.4.2 CORE DUMPS. 39

4.4.3 TECHNIQUE 39

4.4.4 INSTRUCTION DUMPS 40

Section Page

4.5 STORAGE MAPS 40

4.5.1 GENERAL 40

4.6 DIAGNOSTICS. 40

4.7 PROGRAM TIMING 40

FIGURES

Page

Figure 1. Structured flowchart symbols 4

Figure 2. A PDL example. 8

Figure 3. Routine general structure. 13

Figure 4. Header comments. 14

Figure 5. Specification statements 18

ORIGINAL PAGE IS
OF POOR QUALITY

1. INTRODUCTION

PROGRAMMING	- The art of creating logical computer programs.
PROGRAMMER	- A person who prepares problem solving procedures through functionally designed and logically coded routines for the computer to execute and who typically also debugs those routines.

The purpose of this document is to provide engineers, programmers and managers with software development procedures which may be applied in the development of computer software systems. The intent of the procedures presented is to promote quality and uniformity of FORTRAN programs and thereby lessen the time and cost of program development, maintenance, and modification, and to increase program efficiency and reliability.

The key to program reliability is to design, develop, and manage software with a formalized methodology which can be used by computer scientists and applications engineers to describe concepts, perform data analysis, and evaluate systems with visual, conversational, and descriptive data prints or data displays.

The first step in defining and developing a system (be it a large software program or just a few small routines) with a formal methodology is to apply a formalized set of rules and enforce those rules, especially on a large project which is subject to change of personnel or task definition. This document presents a set of rules which may be applied by a FORTRAN programmer/engineer to aid him in writing efficient, reliable, easy to change, and system compatible programs.

ORIGINAL PAGE IS
OF POOR QUALITY

2. DESIGN CONSIDERATIONS

2.1 ANALYSIS

The first step in solving a scientific problem is to analyze the problem. Then a functional design to solve the problem can be made. Of primary importance are the logical flow of the program, data tables, equations and definition of variables to be programmed, where and how the program is to be executed, the program's input/output, and other special considerations and/or constraints.

The logical flow should be a simple sequence of descriptive block steps, including equations, with side comments concerning future program expansions and possible constraints. These descriptive blocks should verbally describe the functions to be performed and should never include programming language.

2.2 MODULARIZATION

After the problem has been analyzed and a functional design developed, the next important step before coding the program is to define all the possible routines or modules. Each module should be a function of the level of execution required. This will reduce program complexity, improve program clarity, and permit easier modifications and program checkout, easier production program maintenance and easier building of a new advanced product.

The following are guidelines that the coder should follow:

1. Each module should be well documented internally by the use of header and in-line comments.

2. Use as many levels of modularity as needed to simplify program control flow.

3. Organize modules logically to make the program easier to understand and modify.

4. Allow room for expansion without destroying simplicity of sequential flow.

5. Each module's variables and arrays should be well defined and the source of each given.

6. Use separate module for data input/output.

7. Use separate module for each specialized function; I.E. bit manipulation function or frequently called mathematical function.

ORIGINAL PAGE IS
OF POOR QUALITY

8. Modules should not be larger than 100 lines of executable code.

2.3 FLOWCHARTING

The functional logic flow of the program may be in the form of a structured flow using structured logic symbols, or a functional level program design language, PDL.

2.3.1 SYMBOLIC

In general, a flowchart gives a pictorial representation of logic within the program and its routines. The flowchart should be readily understandable to the extents that other programmers/engineers could code the routines without lengthy deciphering.

The following are the suggested conventions:

- Use flowchart symbols which have been defined as standard for the project(s) problem. For structured flowchart symbols refer to figure 1

- Use page number references to indicate logical connections

- Include all subroutine and executive references

- Use programming language for equations and logic

- Use structured program flow; that is, the main program or module flow is always top/down on the left side of the paper and the intermediate flow is from left to right and top/down

2.3.2 PROGRAM DESIGN LANGUAGE

The program design language PDL is a tool for designing programs in detail prior to coding. Its purpose is to enable one to express the logic of a program in an English-like language.

Figure 2 illustrates the PDL usage.

ORIGINAL PAGE IS
OF POOR QUALITY

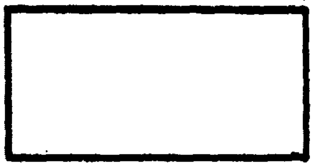
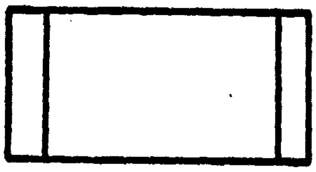
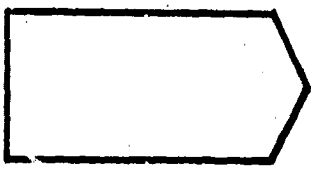
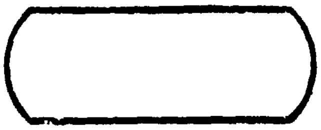
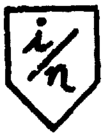
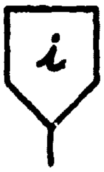
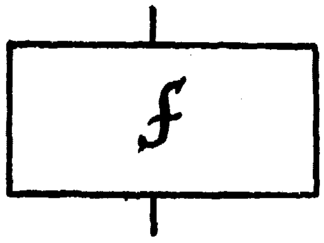
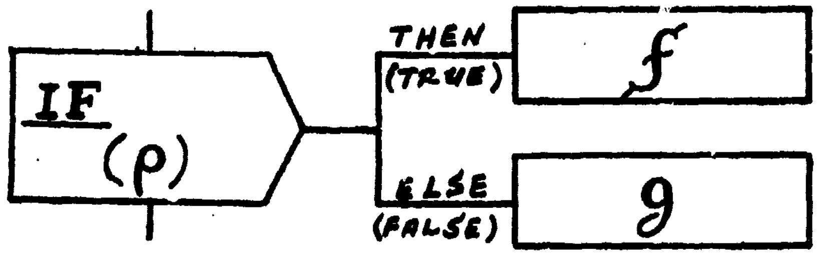
Symbol	Usage
	PROCESSING A group of instructions which perform a processing function of the routine.
	EXTERNAL PROCESSING Identifies the exit to and return from an external function or subroutine with its calling arguments which performs a processing function of the routine.
	DECISION/DO The decision function used to document points in the routine where a branch to alternate paths is possible based upon variable conditions or the DO loop specification statement.
	TERMINAL INTERRUPT The beginning, end, or point of interruption in a routine.
	OFF PAGE FLOW TRANSFER A connector used to show that flow transfers to another part of the flow chart. The symbols i and n indicate a transfer to entry number i on page number n of the flow chart.
	ENTRY INDICATOR An indicator that shows an entry into the logic flow from another part of the flow. The entries on each page are sequentially numbered 1, 2, ...

Figure 1. - Structured flowchart symbols

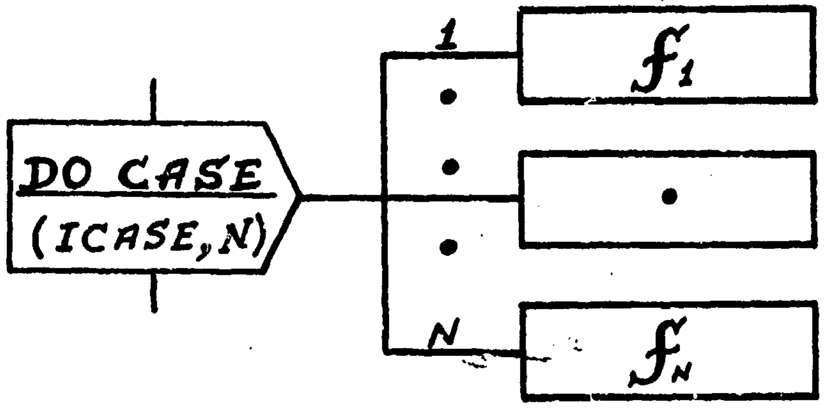
Symbol **Usage**



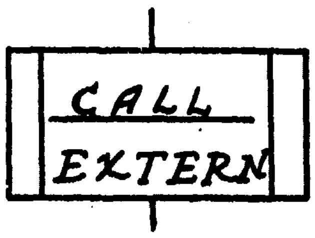
The basic unit of a structured flow chart is the segment. A segment is a module that has a single entrance and a single exit. This segment accomplished the processing identified within.



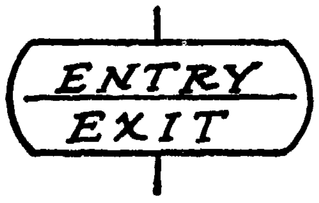
The IF THEN ELSE symbol with the else clause.



The CASE symbol.



A segment that is an external reference to another routine.



The terminal interrupt - the beginning, end, or point of interruption in a program.

Figure 1. - Continued

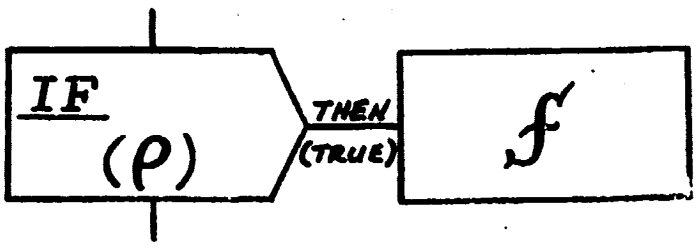
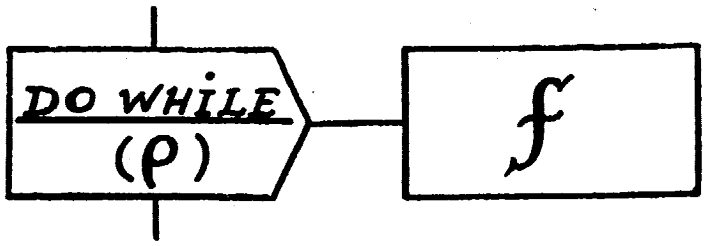
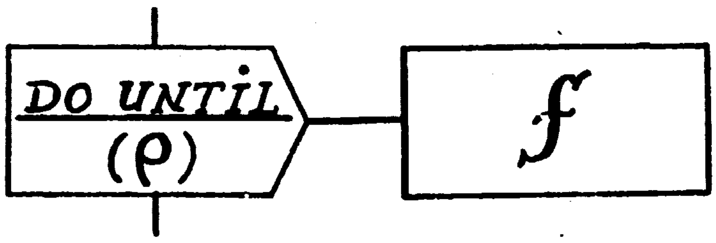
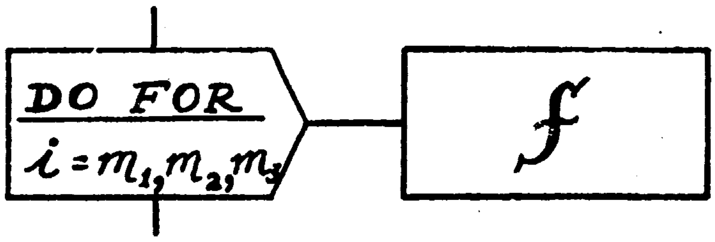
Symbol	Usage
	The IF THEN ELSE symbol with a null else clause.
	The DO WHILE symbol.
	The DO UNTIL symbol
	The DO FOR symbol.

Figure 1. - Concluded

The PDL has the following advantages:

- It states logic in an easy-to-read fashion
- It permits concentration on logic; it frees the designer/programmer with the low-level details of coding
- It can be converted easily to executable code; this is accomplished by step-wise refining the English-like statements until they become statements of a higher level language
- It contributes to the readability aspect of a structured walkthrough for the nonprogrammers
- It can be used to teach structured programming; in fact it is a method of expressing structured programming logic
- It can be retained as comments at the beginning of a program for documentation purposes
- It can be kept on the file in the text mode, updated using the editor, and listed

The main disadvantage to the PDL usage is:

- It does not present the logical flow of the problem in a pictorial form

2.4 EXISTING PROGRAMS AND SUBROUTINES

Before writing a program, search for available programs and subroutines related to your problem. These may do all or part of the job, or may be useful in analysis.

When designing a system, a file should be started which contains all program, subroutine and function names, as well as any entry points within routines. This will avoid future use of a routine name which is already in existence.

ORIGINAL PAGE IS
OF POOR QUALITY

```
INITIATE PROGRAM
GET FIRST TEXT RECORD
DO WHILE MORE TEXT RECORDS
     DO WHILE MORE WORDS
          GET NEXT TEXT WORD
          SEARCH TABLE FOR WORD
          IF WORD FOUND
          THEN INCREMENT WORD'S COUNT
          ELSE WORD NOT IN TABLE
               INSERT WORD INTO THE TABLE
          END IF
          INCREMENT WORD PROCESSED COUNT
     END DO END OF TEXT RECORD
     GET NEXT TEXT RECORD
END DO ALL RECORDS HAVE BEEN PROCESSED
PRINT TABLE
TERMINATE PROGRAM
```

Figure 2. - A PDL example

2.5 COMPATIBILITY

2.5.1 LOGICAL UNIT ASSIGNMENTS

When designing a system, logical unit assignments within programs should be planned and be designated before any programming starts. This way any inconsistencies in future file usage can be eliminated beforehand, and cumbersome and time-consuming releases between executions can be avoided.

2.5.2 USE OF FLAGS

Be consistent in the use of flags on input cards to avoid confusing production personnel setting up the program decks. For example, a zero or blank value could always imply "do not do", and a nonzero values can describe more than one "do" condition (e.g., 0 = do not plot, 1 = plot on linear grid, 2 = plot on log grid; or 0 = do not calibrate data, 1 = calibrate using polynominal expansion, 2 = calibrate using linear interpolation).

2.5.3 CONSISTANCY AMONG VARIABLES

Extend this same consistency to all other variables used in different programs such as start times, stop times and time biases. It is nerve-racking to production personnel, to say the least, to have one program read a start time in integer days, hours, minutes and seconds; a second program read it in interger milli-seconds; and a third program read it in floating-point seconds. The field size for these variables should also be identical in all programs.

When using additive and multiplicative time biases to correct or convert time in a program, their usage should be specified beforehand to avoid future problems. Sometimes one program will add the additive bias and another one will subtract it; sometimes a program will add first and then multiply, and a second program will multiply first and then add. Obviously, these operations will not give the same results.

2.5.4 MACHINE-DEPENDENT SOFTWARE

Keep the machine-dependent portions of a program separate; for example, plan individual modules for I/O operations. This simplifies conversion to other computing systems.

2.5.5 USE OF SPECIAL COMPILER FEATURES

Do not use special features provided by a particular compiler unless it is absolutely necessary. When special features are used, they should, of course, be identified and justified in comments.

2.6 INPUT AND OUTPUT DATA

2.6.1 GENERAL

Design a program so that input data are easy to create and output data easy to read.

2.6.2 GROUPING BY CASES

When data can be distinctly grouped for separate cases, provide a means of flusing data for the current case and going on to the next. This way, an irrecoverable error in processing one case does not necessarily preclude processing others.

2.6.3 INTERMEDIATE OUTPUT

Make available to the user an option for obtaining selected intermediate output. An input code can easily be used to indicate which intermediate results, if any, are desired.

ORIGINAL PAGE IS
OF POOR QUALITY

2.7 ADAPTABILITY TO CHECKOUT

2.7.1 CHECKOUT METHOD

Plan your checkout method while designing a program. Organize the program so checkout data are easy to prepare. Make up a block diagram and preliminary checkout data before coding. Use the checkout data and block diagram in "desk checking" the program.

2.7.2 USAGE OF WRITE STATEMENTS

Organize the program so that WRITE statements, causing meaningful printouts at several points in the program, can be inserted for checkout. These are explained in detail in section 4.1.

2.8 GENERAL-PURPOSE SUBROUTINES

The primary influence on the design of a general-purpose subroutine (i.e., a subroutine reasonably expected to be used in two or more unrelated programs) should be correct results within the required range of accuracy.

Minimization of storage and execution time should be considered next.

ORIGINAL PAGE IS
OF POOR QUALITY

3. CODING

3.1 COMMENTS

3.1.1 GENERAL

Make your program self-explanatory by including meaningful comments throughout. Since most programs outlive their authors' responsibility for them, and because no computer is permanent, your program will probably be modified according to new machine software, or performance requirements. If these comments are properly prepared, they will provide sufficient documentation for most routines and aide in conversion and modifications.

The comments needed to document a subroutine fall into the following classes:

- Routine header comments at the top of the routine
- Comments at various places in the code to describe the logic flow in the routine. Depending on the complexity of the program, the number of necessary comments varies, but usually the ratio of comments to statements should be at least 1:5
- Special comments in large routines to segment the code into logical blocks

The general structure of the program or subprogram is given in figure 3.

3.1.2 HEADER COMMENTS

Identify the program or subprogram in a comment at the beginning of the listing. Comments should follow this card to provide a program abstract answering such questions as: What does the program do? It is confined to any particular application? Is it a special version? Why was it written, by whom, and when? It is derived from or directly related to another program? Are any relevant references published? See figure 4 for the structure of these comments.

<u>ROUTINE ORGANIZATION</u>

```
C*******************************************************************************

                                 header comments

C
C*******************************************************************************
C
C

                           non-executable statements

C
C-------------------------------------------------------------------------------
C

                             executable statements

      CALL EXIT OR RETURN
C
C-------------------------------------------------------------------------------
C

                                format statements

C
C
        END
```

Figure 3. - Routine general structure

```
C**************************HEADER COMMENTS************************
C
C
C     PROGRAM NAME ( or SUBPROGRAM NAME)
C
C      The name of the program (or subprogram) should be here
C
C     LAST UPDATE:  date of the last major revision
C
C     AUTHOR                               NASA MONITOR
C
C         Author's name                    Technical Monitor's name
C         Co. Division                     NASA Division
C         Company name                     NASA JOHNSON SPACE CENTER
C         Date originated
C
C     PURPOSE
C
C         The purpose of the program should be defined here in
C         several sentences. ----------------------------------
C         ------------------------------------------------------
C
C     INPUT VARIABLES
C
C         This section defines the variables INPUT to the sub-
C         program whose value the subprogram does not change.
C         This includes all variables passed by the calling
C         routine to the subprogram through both calling arguments
C         and COMMON blocks.  A sample format follows:
C
C         VARIABLE  COMMON BLK        DESCRIPTION
C         --------  ----------        -----------
C
C         VARBL1     BLOCK1   DETERMINES HOW MANY TIMES ---------
C                              --------------------------
C                              --------------------------
C         XPOS       NONE     POSITION OF THE X VECTOR
C
C     OUTPUT VARIABLES
C
C         This section defines the variables OUTPUT by the sub-
C         routine to whose value the subroutine DOES NOT USE but
C         DOES CHANGE.  This includes variables returned to the
C         calling routine both in the calling arguments and in
C         COMMON blocks.  The format should be similar to that
C         of the INPUT VARIABLES section.
C
```

Figure 4. - header comments

```
C     INPUT/OUTPUT VARIABLES
C
C         This section defines those variables which are used for
C         BOTH INPUT AND OUTPUT, i.e., a variable whose value is
C         INPUT to the subprogram AND whose value the subprogram
C         CHANGES.  The format should be similar to the INPUT
C         VARIABLES section.
C
C     PROGRAM VARIABLES
C
C         This section defines the primary variables that are
C         neither input nor output variables.  The format should
C         be similar to that of the INPUT VARIABLES section, except
C         that a "COMMON BLK" column is not needed.  All internal
C         "flag" should be defined here, and what each of the
C         various codes mean.
C
C     SUBPROGRAMS REQUIRED
C
C         This section briefly defines all subprograms which the
C         subject routine requires.  A one or two sentence should
C         be used to state the basic function (purpose) of each
C         subprogram.  A sample format follows:
C
C         MATMUL - SUBROUTINE THAT PERFORMS MATRIX MULTIPLICATION
C
C         ASSIGN - GENERAL PURPOSE SUBROUTINE WHICH ISSUES A
C                  COMMAND TO THE OPERATING SYSTEM TO @ASG A
C                  SPECIFIC FILE
C
C     REMARKS
C
C         Any special considerations, requirements, restrictions,
C         etc., should be mentioned here
C
C***************************HEADER COMMENTS************************
```

Figure 4. - Concluded

3.1.3 PROGRAM MODIFICATIONS

Program modification should be noted by the date and number of modification (1, 2, 3, ..) and the name of the programmer making it.

3.1.4 SUBROUTINE COMMENTS

For a subroutine, comments describing the calling sequence should follow the identification information. Identify each argument as input, input/output, or output; and explain its purpose, type, dimension, etc. The different values that an indicator (such as an error code) can assume should be defined, for both input and output. Also, all variables used in common blocks should be similarly identified.

3.1.5 DISTRIBUTION OF COMMENTS

Distribute comments describing and summarizing the computation appropriately throughout the listing. These should correspond in terminology to the program block diagram. Clever, but possibly obscure, coding should be explained in detail; for example, if the function J=2*I-I/3-1 is used in a loop, where I takes on the values 1 through 6, the following might be written:

```
C    AS    I = 1,2,3,4,5,6    J = 1,3,4,6,8,9
```

3.1.6 ERROR RECOVERY

Explain error recovery procedures in comments. This information is important to those who maintain or modify the program.

3.1.7 ARRAY DIMENSIONS

Explan in comments any reasons for peculiar array dimensions; e.g., storage limitations or use by other routines.

3.1.8 PRINTING STYLE

Use a conspicuous printing style for comments so that they stand out from the rest of the listing. Separate comments from statements by cards that are blank except for the C in column 1 (although the listing looks cleaner without the C, some compilers object to totally blank cards). Comments are futher accented if they are indented, starting in, say, column 15. Short but important comments can be emphasized by inserting a blank between each letter of each word, and four blanks between words; for example,

```
C       I N P U T    E D I T I N G    S E C T I O N
```

Similarly, symbols that are separated from words by two blanks instead of one stand out better in phrases. For example,

```
C       WITH RADIUS  R  AND WITH  H,K  FOR CENTER
```

When spacing comments and statements, consider inclusion of the program listing in the program's formal document.

3.1.9 PROGRAM MODULARIZATION COMMENTS

For program modularization, comments showing the program structure and flow on a higher level are used to divide the program into segments. These comments provide the mechanism to show the structure of the logic.

3.2 INTIALIZATION

Do not expect main storage to be initialized at the beginning of the execution.

Do not assume that a tape is properly positioned. Rewind it before use and, unless it is a scratch tape, unload it afterward.

3.3 STATEMENT ORDERING AND NUMBERING

3.3.1 NON-EXECUTABLE

Place specification statements (e.g., DIMENSION, COMMON) at the beginning of the program. The symbol lists within type, DIMENSION, and COMMON are to be alphabetical. However variables which functionally go together may be grouped, even though they may not be alphabetical. The symbol lists are to be columnized and left justified. This way they are easy to find and do not interrupt following of the logic flow. Further, some compilers object if these statements are not placed first. See figure 5.

NON-EXECUTABLE STATEMENTS

```
C
      Program declaration statements
C
C
      REAL       MACH, ------------
      INTEGER    X    , Y ----------
      other type statements and IMPLICIT statements
C
      DIMENSION AB    (100), ANAME (50), --------------
     1         , VNAME1( 30), X     (25), ------------------
     2         ,
     3
C
      COMMON/COM1   / X      ( 10), ARRAY ( 50), -------------
     1              , ARRAY1(300), ---------------------------
     2
      COMMON /COM2  / ------------------------------
     1              , ------------------------------
     2              , ------------------------------
C
      EQUIVALENCE   (AAA    ( 1), BBB    (10)),  (XXXXXX(20), YY    (50))
     1           ,   (CCCCCC(20), DDDDDD( 1)), ---------------------
     2           ,   ----------------------, ---------------------
C
      DATA
     1
     2
C
C------------------------------------------------------------------
C
```

Figure 5. - Specification statements

The order of the specification statements should be in the following order:

TYPE
DIMENSION
COMMON
EQUIVALENCE
DATA
Statement functions

Minimize the use of TYPE statements, especially TYPE REAL and TYPE INTEGER.

3.3.2 FORMAT STATEMENTS

Place FORMAT statements where they are easy to find. Group them all at the end of the program, except those simple FORMAT statements used by only one I/O statement, which should be placed with that I/O statement. All FORMAT statement numbers are to be five-digit numbers (preferably 2XXXX and larger) and increase sequentially.

3.3.3 STATEMENT NUMBERING

Statement numbers are to increase sequentially from physical beginning to physical end of the executable statements. This permits easy following of transfers. Using separate and distrinct blocks of statement numbers (statement numbers increase by 50, 100, 500 or 1000 depending on amount of code in the block) in different sections of the program to emphasize the structure, to have large enough gaps for future modifications and expansions, and prevent accidental duplication. Statement numbers are to be placed on CONTINUE statements only and right justified with column 5.

3.4 SPECIFICATION STATEMENTS, COMMON STORAGE

3.4.1 SPECIFYING VARIABLE TYPE

Use only one type specification for a variable name. When you must change the type specifications of integer or real variables, rename them in EQUIVALENCE statements, using names beginning with I through N for integers and with other letters for other variables.

3.4.2 COMMON BLOCKS

The structure of each block of common storage, as specified in COMMON and in any other related specification statements (e.g., DIMENSION, INTEGER, EQUIVALENCE), should be the same for the main program and all subroutines using it. All COMMON should be labeled and in alphabetical order, unless blank COMMON is necessary for communication between CHAIN segments. Use several blocks, rather than putting unrelated data into the same block; this incurs no penalty and prevents the confusion of variables specified but not used in a subroutine.

Thus the specification statements for each block of COMMON can be reproduced exactly and a copy inserted into each subroutine using it. For legibility, separate the blocks by blank C cards and use comments to explain the purpose of the blocks, where necessary. Corresponding COMMON and DIMENSION statements should be ordered and spaced the same way.

3.5 VARIABLE NAMES

3.5.1 GENERAL

Unless awkward, use variable names that are meaningful in the context of the problem the program is to solve and that correspond to notation or terminology in the block diagram and program document. This helps make the listing self-explanatory and relates it to the block diagram and document. For example, two arrays, one of positive and the other of negative values, that are denoted in the block diagram and documentation as

$$x_1^{(+)}, x_2^{(+)}, \ldots \text{ and } x_1^{(-)}, x_2^{(-)}, \ldots$$

should be given names like XPOS and XNEG, rather than A and B.

ORIGINAL PAGE IS
OF POOR QUALITY

3.5.2 CONSTANTS

Use variable names for quantities that might be expressed as constants. Examples of such quantities are the number of times a loop is to be performed, the length of a vector, or an I/O device number. Set the values of these quantities during initilization (in a DATA statement if possible); thus they can be redefined, if necessary, in one place at the beginning of the program. Accordingly, refer to I/O files or devices by integer variables rather than by constants. For example, instead of using the constant 3 in several output statements, use a variable such as IOUT that is initialized to 3. Then, if the files or devices are reorganized, the analyst can simply change the definition of IOUT and need not look for all appearances of 3 in this context.

3.5.3 NAMING CONVENTION

In naming variables, use names beginning with I through N for integer variables, and names beginning with A through H and O through Z for other variables. This widely accepted convention reduces confusion. Avoid using variable names similar to FORTRAN verbs. Some compilers might treat them as commands instead of variable names.

3.6 ARRAYS

3.6.1 COMBINING

Do not needlessly combine into one array what could be separate arrays with fewer dimensions. Similarly do not needlessly form a singly-dimensioned array from what could be simple variables. The time and storage required for index manipulation increases as the number of dimensions increases. When the only reason for such combining is to make separate arrays or simple variables adjacent, this can be accomplished by an EQUIVALENCE statement that equates the arrays or simple variables to the elements of an array into which they might have been combined.

ORIGINAL PAGE IS
OF POOR QUALITY

3.6.2 SUBSCRIPTS

Whenever referring to an element of an array, include a subscript for each dimension. Although

```
A(1,1)=0.
```

can sometimes be expressed

```
A=0.
```

not all compilers will accept it, and it may confuse some programmers.

3.6.3 USAGE IN SUBPROGRAMS

An array included in the calling sequence of a subroutine must appear in a DIMENSION statement in the subroutine. Possibly the subroutine does not use the array but passes it on to another subroutine. However, some compilers require that the DIMENSION statement be included in this type of subroutine to ensure that the array is passed by name and not by value. Also, the dimensioning information makes visually apparent in the program listing what are arrays and not simple variables.

A useful convention for singly-dimensioned arrays is that the DIMENSION statement specify a length of 1 if the length of the array is variable (to the subroutine), and the actual length if it is fixed. This convention also applies to the last subscript of a multiply-dimensioned array (the other subscripts must agree exactly with those in the calling program).

3.7 ARITHMETIC EXPRESSIONS AND STATEMENTS

3.7.1 UNAMBIGUOUS USAGE

Use parentheses to make arithmetic expressions completely unambiguous. The expression A**B**C is computed from right to left by some compilers; from left to right by others. Similarly, the expression I*J/K could mean I*(J/K) or (I*J)/K, and the expression A/B*C could mean C*A/B or A/(B*C).

Do not rely on the order of the evaluation within a single arithmetic expression. For example, instead of the statement Y=F1(X)+F2(X), where F1 and F2 are functions to be taken in that order because one depends on the other, use two statements; i.e., Y=F1(X) followed by Y=Y+F2(X).

3.7.2 TEST FOR IMPROPER CONDITIONS

When undefined operations are possible, such as division by zero or taking the square root of a negative argument, test in advance for improper conditions.

3.7.3 COMPOUND EXPRESSIONS

Replace compound expressions repeated in arithmetic statementts by single variables previously set to the values of the expressions. This not only simplifies the appearance of expressions and statements, but also saves time, storage, and helps to debug the expression. Although some compilers have an optimization feature, this is a good practice to get into. For example, in the statement:

```
    Y = (A*B)/C + COS(A*B)/C - SIN(.5*(A*B)/C))
```

replace the expression (A*B)/C by the variable T; i.e.,

```
    T = (A*B)/C
    Y = T + COS(T-SIN(.5*T))
```

Similarly, simplify expressions algebraically before coding them. This applies to constants as well as variables. For example, for the circumference of a circle in inches, given the radius in feet, write

```
    C=24.0*PI*R,  not  C=2.0*PI*R*12.0
```

ORIGINAL PAGE IS
OF POOR QUALITY

3.7.4 USAGE OF SQRT

When practical, use the square root function instead of exponentiation or other more difficult operations. Generally, the SQRT subprogram is executed faster, is more accurate, and uses less storage. Also it is more likely to be already in core than any other elementary function generator. For example, use

```
         SQRT(X)    not   X**0.5
       X*SQRT(X)    not   X**1.5
    SQRT(SQRT(X))   not   X**0.25
```

Further, where S = SIN(X), T = COS(2.0*X), and U=SINH(X), use

```
    SQRT (1.-S*S)       not   COS  (S)
    SQRT (.5*(1.+T))    not   COS (.5*ACOS(T))
    SQRT (1.+U*U)       not   COSH (ASINH(U))
```

In general, replace complicated operations by simpler operations when possible. For example, to compare the distance between the points (Xj, Yj) and (Xi, Yi) with a prescribed tolerance T, use

```
    IF ((XI-XJ)**2+(YI-YJ)**2-T*T)   N1, N2, N3
```

rather than

```
    IF (SQRT((XI-XJ)**2+(YI-YJ)**2)-T)   N1, N2, N3
```

Given a set of N points whose coordinates are stored consecutively in the singly-dimensioned arrays X and Y, to find the distance between the origin and the point farthest from it, use

```
        D=0.
        DO 100 I=I,N
        D = AMAX(D,X(I)**2+Y(I)**2)
    100 CONTINUE
        D = SQRT (D)
```

rather than

```
          D = 0.
          DO 100 I = 1,N
          D = AMAX (D,SQRT(X(1)**2+(1)**2))
      100 CONTINUE
```

The first method saves N-1 square root calculations.

3.7.5 PREFERRED CONSTRUCTIONS

To speed execution or to conserve storage, use the following preferred constructions (most of these apply to integer as well as to real variables):

To express a power of 10, use E notation, not exponentiation. For example, the expression 20.5E6 causes the compiler to generate a constant, but the expression 20.5*10.**6 requires a calculation during execution.

Mixed mode expressions and replacements are wasteful, even when allowed by the compiler; use

```
           A+2.0   not   A+2
and        A=2.0   not   A=2
```

Addition is always faster than multiplication; use

```
            A+A   not   2.0*A
```

In a loop, multiplication by the reciprocal is faster than division; use

```
           DO 100 I=1,N   not      DO 100 I-1,N
           A = 0.5*A               A = A/2.0
       100 CONTINUE            100 CONTINUE
```

For exponents that are whole numbers, use fixed-point notation. A real exponent requires the general approximation algorithm of exponentiation whereas an integer exponent requires only repeated multiplication or a simpler exponentiation algorithm. For example, the

```
   A**2  (or A*A)   not   A**2.0
```

ORIGINAL PAGE IS
OF POOR QUALITY

3.8 CONTROL STATEMENTS

3.8.1 CALCULATIONS IN A LOOP

Minimize the calculations performed in a loop, and avoid unnecessary subscripting. For example,

```
   DO 100 I = 1,N
    Z(I) = U*V*X(I)+Y(J)
100 CONTINUE
```

is not as efficient as

```
    T  = U*V
    YJ = Y(J)
    DO 100 I = 1,N
    Z(I) = T*X(I) = YJ
100 CONTINUE
```

3.8.2 COMPUTED GO TO'S

The control variable of a computed GO TO statement should be checked in advance if it is read from input data, received through a calling sequence, or calculated from other than perfectly controlled variables. All labels within computed GO TO statement should be sequential in ascending order if possible.

3.8.3 ASSIGN STATEMENTS

The ASSIGN statement and the assigned GO TO statement will not be used. This will prohibit jumps both forward and backward in the code. Transferring all over the routine makes it difficult to follow the logic of the routine and routine complexity grows with additions and changes during the checkout.

3.8.4 DO LOOPS

Usually, the indexing parameter of a DO-loop has a range of permissible positive values, and zero is an unlikely but possible value. Therefore, check the indexing parameters of DO-loops, and of implicit DO-loops in READ and WRITE statements, if there is any change of a zero value. For example,

```
          J = 0
         DO 100 I=1,N
         J = J+I
    100 CONTINUE
```

gives the wrong value for $J = \sum_{i=1}^{N} i$ when N = 0, whereas

```
          J = 0
          IF(N.LE.0) GO TO 200
          DO 100 I = 1, N
          J = J + 1
     100 CONTINUE
     200 ...
```

work for all values of N.

3.8.5 CALL STATEMENTS

Avoid literal arguments in CALL statements. If a subroutine changes the value of an argument passed as a literal constant, subsequent use of that constant by the calling program is invalid. For example, if the following occurred,

```
CALLING PROGRAM          SUBROUTINE SUB (J)
        •                        •
        •                        •
        •                        •
CALLING SUB(3)           J = 2
        •                        •
        •                        •
        •                        •
I = 3                    RETURN
```

every subsequent use of the literal constant 3 in the calling program will actually use a value of 2. In the example 2, not 3, will be stored in I.

ORIGINAL PAGE IS
OF POOR QUALITY

3.9 INPUT/OUTPUT

3.9.1 RECORD FORMAT

When a widely used record format is appropriate or nearly so, do not invent a new one.

Minimize the number of formats for input data. Generally, the fewer forms in which data must be prepared, the less susceptible it is to error and the less storage the program requires.

For example, many programs could use a single input format, such as 7E10.0. Data could be converted to fixed-point, if necessary, after it is read. This would make keypunching easier and errors less likely because all numeric input could be punched with decimal points and, more importantly, could be left justified in the fields. Even when standardizing the input increases the number of cards, the benefits of convenience and fewer errors outweigh the cost of additional cards and of processing them.

Avoid writing short records on tape. For a given amount of data, the fewer the number of records, the less likely are read/write errors, the greater is the read/write speed, and the smaller is the amount of tape used. Also, short records can cause tape positioning problems. Avoid tape records of fewer than about 80 characters; they are likely to cause read errors.

If only a few characters are to be written, repeat them enough times (or insert dummy characters) to form a record of at least 80 characters. When the record is subsequently read, the READ statement would, of course, be the same as if the redundant or dummy characters were not there.

ORIGINAL PAGE IS
OF POOR QUALITY

For example, instead of

```
        WRITE (J,1) A,B,C,D
            .     .     .
            .     .     .
            .     .     .
        READ (J,1) A,B,C,D
          .     .     .
          .     .     .
          .     .     .
```

use

```
        WRITE (J,1) A,B,C, (D,I=1, 11)
          .     .     .
          .     .     .
          .     .     .
        READ (J,1) A,B,C,D
```

When writing multiple-file tapes, it is a good practice to have an End-of-File (EØF) mark after each file, and two after the last file on the tape. Thus, a programmer does not need to know how many files there are on the tape in order to process the whole tape. Also, using this convention, it is quite simple to position the tape to the desired file by skipping files.

3.9.2 PLACEMENT OF I/O OPERATIONS

Isolate input and output operations, except perhaps for the permanent input and output files, in subroutines. This allows easier relocation of scratch files from tape to disk, or modification of a plotting tape for new plotting hardware, software, or performance requirements.

3.9.3 DEFAULT VALUES

On card or terminal input, it is a good practice to have default values within the program for some input variables. Thus, by leaving the field blank, the program automatically presets the variable to some commonly-used value.

ORIGINAL PAGE IS
OF POOR QUALITY

This is a good convenience for the user. For example, blank start-stop times on an input card could mean to process all data by having the program set the times to the smallest and largest possible values; a blank multiplicative time bias would cause the program to set the bias to 1. Care must be exercised, however, when reading blank fields which will read in as negative zero on numeric variables. Since zero is a possible data value, a further check for negative zero may have to be made.

All input data read in on control cards should be printed out, just as it was read in but clearly labeled. This allows for quick identification of keypunch errors should the program error off or give the wrong results.

3.9.4 OUTPUT FORM

Output from production programs should be oriented to the user. Clearly identify output as to the name of the calculation, the name and number of the program producing it, and the date. Label printed output and, if the printout is expected to end up on document paper, limit its length and width to the dimensions of the document. This eliminates the need for photo-reduction. For example, a printout confined to a rectangle about 7-1/2 inches wide by 10 inches long could be trimmed and bound as 8-1/2 by 11 inch material. Number and date the pages of a printout when the application calls for it. For the date, use an existing general-purpose subroutine.

3.9.5 ERROR MESSAGES

Provide for labeled printout when errors occur to explain the reasons for the errors. Make the explanation meaningful to the user as well as to the programmer. This frees the user of the necessity of looking up the meanings of error codes in separate documents. These printouts also should explain what counters, etc., are crucial for locating the source of the errors. General-purpose subroutines should not, of course, write such messages. All error messages produced by the program should be clearly identified as such (e.g. "*** PROG XYZ ERROR")

3.9.6 INTERMEDIATE OUTPUT

Make available to the user an option for obtaining selected intermediate output. An input code can easily be used to indicate which intermediate results, if any, are desired.

3.9.7 CARD READING

Explicitly control the reading of large numbers of cards. A control integer specifying the number of cards in a set can easily be wrong due to miscounting. If the integer is too big, the program may read to the end of the data and be terminated; if it is too small, the next input statement will read the wrong cards. One alternative is to punch a flag in the last card so the program can recognize it. For example

```
    N=0
100 CONTINUE
    N=N+1
    READ(5,110) (X(N,I), I=1,7),K
110 FORMAT (7E10.0,I2)
    IF (K.EQ.0) GO TO 100
```

is preferable to

```
    READ(5,110)N,((X(J,I),I=1,7),J=1,N)
110 FORMAT (I10/(7E10.0))
```

and obviates manual card counting and the associated error possibility.

Another alternative, when a particular field is non-zero on all cards in the set, is to insert a blank card behind the last card of the set and read it as follows:

```
    N=0
100 CONTINUE
    N=N+1
    READ (5,110) (X(N,I), I=1,7)
110 FORMAT (7E10.0)
    IF(X(N,1).NE.0) GO TO 100
    N=N-1
```

This way, a card need not be punched with a flag that might later have to be removed when the set is enlarged.

ORIGINAL PAGE IS
OF POOR QUALITY

3.10 SUBROUTINES

The term "subroutine" as used here means either SUBROUTINE or FUNCTION SUBPROGRAM.

3.10.1 GENERAL

Code a group of logically related instructions as a subroutine, rather than as in-line coding if it:

- Is entered from several different places in the program.

- is potentially of general-purpose value.

- Is less stable than other parts of the program; or

- Is simply of appropriate size to be a separate module.

Subroutines concretely express the concept of modular programming.

3.10.2 CALLING ARGUMENTS

For ease of interpretation, group the arguments of a calling sequence in this order: input, input/output, output, error code.

- An input argument is one whose value the subroutine uses but does not change.

- An input-output argument is one whose value the subroutine uses and subsequently changes.

- An output argument is one whose value the subroutine does not use but does change.

- The error code argument is the means of transmitting diagnostic information to the calling program, such as whether the subroutine executed normally or abnormally; it is a special case of an output argument.

ORIGINAL PAGE IS
OF POOR QUALITY

3.10.3 ERROR CODES

An error code returned by a subroutine should be zero for normal execution and a non-zero value otherwise. The more specifically it can describe to the calling program the nature of a malfunction or improper condition in the input data.

A general-purpose subroutine should not write diagnostic messages or cause other input/output operations unless that is its principal function. Error codes should be returned through the calling sequence. The user of the subroutine then is not restricted as to the words in, position of, and storage for diagnostic messages. Further, he has a change to recover gracefully.

3.10.4 RETURN STATEMENTS

Use only one simple RETURN statement in a general-purpose subroutine, and place it physically as the last executable statement. Connect other places where the logic flow terminates to the RETURN statement by GO TO statements. This eases later insertion of statements that must be executed before any return is made. Note that this method still leaves the various paths to termination distinct so that they can be treated separately when necessary.

3.10.5 ARRAYS

If a subroutine uses a variable-length or large fixed-length array for working storage, transmit it to the subroutine through the calling sequence. This way, the array is in the data region of the calling program and can satisfy other needs for temporary storage. Also, if the array varies in length from case to case in a single program, or from program to program, and is not specified in the calling sequence, the array as defined in the subroutine could be either short and sometimes insufficient or long and sometimes wasteful.

Limit the output of a subroutine to prevent array overflow in the calling program. When an output array from a subroutine is of variable length, the maximum allowable length must be communicated to the subroutine by an argument in the calling sequence.

ORIGINAL PAGE IS
OF POOR QUALITY

3.10.6 COMMON BLOCKS

Use labeled COMMON for passing arguments to or from special-purpose subroutines whenever possible.

A subroutine must not change the value of an input argument.

General-purpose subroutines should not use blank COMMON storage. One that does limits the calling program in its use of COMMON. Two or more that do are likely to have incompatible requirements for the sizes or names of blocks in COMMON, which necessitate awkward modifications when the subroutines are used together.

4. CHECKOUT AIDS

4.1 INTERMEDIATE RESULTS

4.1.1 PROGRAM FLOW

Place WRITE statements in all major blocks of the program and its subroutines when first coded, so that the progress of a program can be traced from its printed output during debugging. Do not rely on the ordinary (production) output. At least have each special-purpose subroutine print its name as soon as it is entered. It is also useful to print the input variables to a subroutine just before and the output variables just after the CALL statement. These statements should print a clear indication of their position in the program, and any variables printed should be labeled.

In tracing the flow of a program, integer control variables are generally more helpful that are floating-point data, although the floating-point values may be needed to check the numerical algorithm. So it is better initially to code many simple WRITE's of integer vairables, such as indices and counters, matrix dimensions, flags and switches, error codes and computed GO TO variables, than a few massive WRITE's of floating-point arrays.

The WRITE statements used in debugging, and their associated FORMAT statements, may be identified by a word such as TRACE or DEBUG in columns 73-80, so that they are easily removed after checkout. Alternatively, a C can simply be added in column 1 so that the statements can be used again if the program is modified.

ORIGINAL PAGE IS
OF POOR QUALITY

4.1.2 DATA STRUCTURES

Design data structures sensibly so they can be displayed either in dumps or in labeled and well arranged printout. Such printing requires extra coding initially, but this extra code can be included in an error handling subroutine that provides easily read diagnostic information when and only when needed. It also provides a convenient checkout device in program modification, for a CALL to this subroutine can be inserted both before and after the modified program section. This lessens the need to invent intermediate output statements or dump procedures, which usually fail to include all the portions of storage required to diagnose the error.

4.1.3 VALIDITY OF RESULTS

For programs expected to run a long time, provide for frequent checks of the validity of results. When the results seem invalid, and the error is irrecoverable, execution should be terminated.

4.2 DESK CHECKING

4.2.1 GENERAL

Desk checking means manually scrutinizing program logic and deck structure. Mistakes in either can cause an unsuccessful run, so a few minutes of checking is worthwhile.

4.2.2 PROGRAM LOGIC CHECK LIST

- Is there a statement number on the statement immediately following each arithmetic IF statement and each of all kinds of GO TO statements?
- Are there statement numbers for the exist from IF, GO TO, and DO statements?
- Do parentheses balance? Start from the left with 0 and add 1 for each left parenthesis encountered and subtract 1 for each right parenthesis. The count should never become negative. If parentheses balance, the count will end up at 0; however, this does not necessarily indicate correct grouping.
- Does every subscripted variable appear in a specification statement?
- Does any DO-loop end with an IF statement, or GO TO statement?
- Are all referenced FORMAT statements present?
- Is the field length correct for all Hollerith fields?
- Are the number, order, and type of arguments in CALL statements correct?

4.2.3 DECK STRUCTURE CHECK LIST

- For Control cards: is the card necessary, is its position in the deck consistent with its purpose, and is its format correct? Are any control cards missing?
- Are all necessary subroutine decks present?
- Are all necessary data cards present, and does their order agree with what the program expects?
- Is the deck properly identified with your name phone number, location, etc.?

4.3 CHECKOUT DATA

4.3.1 GENERAL

When creating checkout data, remember that anything that can be punched on a card, written in a tape record, etc., will possibly be input to your program sooner or later. A program is never 100 percent checked out, but you are responsible for making checkout as complete as possible. Therefore, prepare checkout data that represent production conditions, including both valid and invalid data, to test diagnostics and recovery features.

Keep test decks and records of test results up-to-date. When new features are added to a program insert representative checkout data. Whenever a program fails and is corrected, include in the test deck the type of data that caused the failure.

When a program is revised or recompiled, check it with the old test deck as well as the new checkout data, if the old deck is still applicable.

4.3.2 VERIFICATION OF INPUT

Know your input. When practical, have checkout data printed out completely in a readable format before using it, so you can check it. (To list out input cards, use an existing general-purpose subroutine.) When the input to a program, particularly a general-purpose routine, consists of a large amount of data, another routine to check the data for consistency, rather than printing it all out, could be helpful. Another technique for handling a large amount of data is to write it in a scratch file and use an existing general-urpose subroutine to transfer it to the output file after execution.

ORIGINAL PAGE IS
OF POOR QUALITY

4.4 DUMPS

4.4.1 GENERAL

If trace printouts are systematically used, there should be only infrequent need for core dumps. Since the latter are more expensive in computer time and less useful as a debugging aid than selective labeled printout, the only dump that should always be provided for is a conditional post-mortem of crucial regions of storage, such as the data region of the main program, in case of abnormal job termination. But, do not rely on a post-mortem dump as a substitute for trace printouts; they will tell only what the program looked like after the crash, not necessarily why it crashed.

4.4.2 CORE DUMPS

Generally core dumps are useful only to more experienced programmers, most of whom will maintain that they cannot work efficiently without them. However, new programmers will not have a good understanding of the computer's workings until they are at least capable of understanding dumps.

4.4.3 TECHNIQUE

The technique of using dumps is best left to the judgment of the individual programmer, but there are a few general principles:

- When a dump is positioned in a loop, be sure to include the relevant control variables, such as the indexing parameter of the loop.

- When preparing a production run, always provide for a full core dump in the event of failure detected by the operating system; this aids immensely in investigating operating system and/or hardware malfunctions. A full core dump in the event of failure detected by the program may or may not be appropriate.

- Carefully select the regions to be included in a dump; but, when in doubt, include too much rather than too little.

ORIGINAL PAGE IS
OF POOR QUALITY

4.4.4 INSTRUCTION DUMPS

Another occasional need for a dump is to examine instruction regions suspected of having been improperly generated by the compiler or of having been mutilated during execution. Since the instruction region cannot be written out by WRITE statements, a dump with mnemonics can be a great help, to those who can interpret it, in isolating these errors.

4.5 STORAGE MAPS

4.5.1 GENERAL

Get a storage map, which shows how the program uses main storage, and use it in checkout. Be sure to watch for:

- Variable names that do not belong there, but appear because of misspelling or other mistakes.

- Arrays treated as functions because they are not specified in DIMENSION statements.

- Proper size of COMMON storage for all routines using it.

Get a loading map, which shows how all of core is allocated to make for easier interpretation of dumps.

4.6 DIAGNOSTICS

When you discover an error in checking out a program, do not resubmit the program until you have checked the diagnostic information of other errors. Often several program errors can be detected from the diagnostics of one checkout run. Examine partial results and incorrect results; even these can be helpful. For example, try to ascertain why deviations from expected or pre-calculated results occurred.

4.7 PROGRAM TIMING

To time a section of a program, use an existing general-purpose timing subroutine when the section is entered and when it exits.