MAN1674

FORTRAN IV
User Guide

Revision D
June 1976

PRIME
Computer, Inc.
145 Pennsylvania Ave.
Framingham, Mass. 01701

Copyright 1976 by

Prime Computer, Incorporated

145 Pennsylvania Avenue

Framingham, Massachusetts 01701

Performance characteristics are
subject to change without notice.

CONTENTS

	Page
SECTION 1 INTRODUCTION	
FORTRAN VERSIONS	1-1
REFERENCE DOCUMENTS	1-1
SCOPE OF MANUAL	1-2
COMPILING AND RUN TIME FEATURES	1-3
FORTRAN LIBRARY SUBROUTINES & FUNCTIONS	1-3
FORTRAN COMPILER SUBROUTINES	1-3
INDICATION AND CONTROL SUBROUTINES	1-4
INPUT/OUTPUT CONTROL SYSTEM (IOCS)	1-4
FORTRAN MATH LIBRARY (MATHLB)	1-4
PRIME FORTRAN IV FEATURES	1-4
SECTION 2 SOURCE PROGRAM FORMAT	
BASIC TERMINOLOGY	2-1
CHARACTER SET	2-2
PROGRAM FORM	2-4
SECTION 3 ASSIGNMENT STATEMENTS	
GENERAL PRINCIPLES	3-1
CONSTANTS IN A FORTRAN STATEMENT	3-2
SECTION 4 CONTROL STATEMENTS	
UNCONDITIONAL GO TO STATEMENT	4-1
COMPUTED GO TO STATEMENT	4-2
ASSIGNED GO TO STATEMENT	4-2
ASSIGN STATEMENT	4-2
ARITHMETIC IF STATEMENT	4-3
LOGICAL IF STATEMENT	4-4
DO STATEMENT	4-4
CONTINUE STATEMENT	4-7
STOP STATEMENT	4-8
PAUSE STATEMENT	4-8
END STATEMENT	4-8
SECTION 5 SPECIFICATION STATEMENTS	
DATA TYPE MODE SPECIFICATION STATEMENTS	5-1

CONTENTS (Cont)

STORAGE SPECIFICATION STATEMENTS 5-3
COMPILATION AND RUN TIME CONTROL STATEMENTS 5-11
LISTING CONTROL STATEMENTS 5-12

SECTION 6 I/O AND FORMAT CONTROL 6-1

GENERAL PRINCIPLES 6-1
READ AND WRITE STATEMENTS 6-2
FORMATTED RECORDS 6-8
PRINT & PRINTER CONTROL 6-33
END AND ERROR RETURNS 6-35
B FORMAT STATEMENT 6-35
UNFORMATTED (BINARY) RECORDS 6-38
DEVICE CONTROL STATEMENTS 6-39
ENCODE/DECODE STATEMENTS 6-39

SECTION 7 FUNCTIONS AND SUBPROGRAMS

GENERAL OVERVIEW 7-1
LIBRARY FUNCTIONS 7-2
INTRINSIC FUNCTIONS 7-3
STATEMENT FUNCTIONS 7-7
FUNCTION SUBPROGRAMS 7-8
SUBROUTINE SUBPROGRAMS 7-11
BLOCK DATA SUBPROGRAMS 7-17
LIBRARY SUBROUTINES 7-18
SENSE LIGHT/SWITCH SUBPROGRAMS 7-18
LINKING FORTRAN AND ASSEMBLY LANGUAGE PROGRAMS 7-18

SECTION 8 PROGRAMMING TECHNIQUES

MAXIMUMS 8-1
ATTACHING TO ANOTHER USER FILE DIRECTORY (UFD) 8-1
CLOSING AND OPENING FILES 8-2
RECORD LENGTH OPTION 8-6

APPENDIX A COMPILER ERROR MESSAGES A-1

APPENDIX B RUN-TIME ERROR MESSAGES B-1

APPENDIX C LIST OF STATEMENTS C-1

APPENDIX D PROGRAM EXAMPLES D-1

APPENDIX E PRIMOS SUBROUTINES SUMMARY E-1

APPENDIX F SUMMARY OF IOCS SUBROUTINES F-1

CONTENTS (Cont)

APPENDIX G FORTRAN LIBRARY SUBROUTINES G-1

APPENDIX H FORTRAN MATH LIB SUMMARY H-1

APPENDIX I FORTRAN COMPILER SUBROUTINES I-1

APPENDIX J INDICATOR/CONTROLS J-1

APPENDIX K SUMMARY OF SORT ROUTINES K-1

ILLUSTRATIONS

ILLUSTRATIONS

Figure No.	Title	Page
3-1	Computer Internal Word Formats for Constants and Variables	3-5, 3-6
6-1	Format of External Input to Type D,E,F, or G Field Descriptors	6-24
7-1	FORTRAN/Assembly Language Argument Transfer (Without F$AT)	7-20

TABLES

Table No.	Title	Page
1-1	Reference Documents	1-2
3-1	Operator Priority	3-18
3-2	Rules for Assignment of B to A	3-20
6-1	Characteristics of Formatted External Records	6-2
6-2	Logical Device and their IOCS Numbers	6-4
6-3	Summary of Output Field Descriptors	6-11
6-4	Summary of Input Field Descriptors	6-13
6-5	Interpretation of Gw.d Descriptors	6-23

FOREWORD

This handbook has been prepared as a handy reference guide to the FORTRAN IV language as implemented by the Prime FORTRAN Compiler. The handbook is organized for quick look-up of syntax conventions, data formats and the effects of statement execution. New material is usually presented in terms of elements that have already been defined. However, the handbook is not intended as a basic primer on FORTRAN programming. Novice FORTRAN programmers may find valuable supplementary information in the many excellent FORTRAN textbooks currently in print. A few samples are:

E. I. Organick: A FORTRAN IV Primer, Addition-Wesley Publishing Company (1967)

D. A. McCracken: A Guide to FORTRAN Programming, Wiley

The following document is the definitive reference for FORTRAN IV language conventions:

American National Standards Institute: USAS X3.9 - 1966 (USA Standard Fortran)

Revision A updates the handbook for compatibility with Prime FORTRAN compilers supplied on Revision 3 master disks and paper tapes. Changes are identified by bars in the margin of the page.

Revision B adds Appendix F which describes enhancements introduced on Revision 4 and 5 master disks and paper tapes. Altered compiler A-register settings are shown in a revised Figure 8-1.

Revision C adds Appendix G which describes enhancements introduced on Revision 6 and 7 master disks.

Revision D incorporates changes to REV 10 of the master disk and the incorporation of appendices F and G.

SECTION 1

INTRODUCTION

FORTRAN VERSIONS

Three versions of the FORTRAN Compiler are available: 1) a large version (LFTN) (more than 16K PRIMOS system), 2) a small version (SFTN) (a 16K PRIMOS system), 3) the PRIME 400 FORTRAN Compiler (VFTN). A program compiled by one version can be compiled by another. However, the large version performs additional functions not available in the small version. These include:

- Text error messages
- Extended code optimization
- In-line Desectorization option
- Improved symbolic listing
- Undeclared variable check option
- Cross-reference listing

VFTN is a modification of LFTN that includes the ability to generate code in the 400´s segmented addressing mode (64V). The hardware and software features available at Rev. 10 allow FORTRAN programs nearly two megabytes long (15 segments of 128K bytes) to be executed under PRIMOS IV. VFTN retains the ability to produce non-segmented code identical to LFTN that runs on the entire family of Prime computers. VFTN will execute on all PRIME computers with over 16k memory, but code generated in 64V mode will execute on the PRIME 400 only.

NOTE:

After installation, the FORTRAN compiler version becomes known to the user as FTN. However, a reference to each version will be made in this manual when required to distinguish the capability of one version over another.

REFERENCE DOCUMENTS

Table 1-1 lists the publications that are recommended to accompany this handbook.

Table 1-1. Reference Documents

Publication	Prime Document No.
Prime CPU System Reference Manual	1671
PRIMOS II & III Interactive User Guide	2602
PRIMOS II & III Computer Room User Guide	2603
PRIMOS II & III File System User Guide	2604
Prime Program Development Software Guide	1879
Prime CPU RTOS Reference Manual	1856
Prime Software Library User Guide	1880
USA Standard FORTRAN (USAS X3.9-1966) American National Standards Institute	---

The procedures for loading the compiler, using it to compile source programs, and loading and running object programs, are provided in the Program Development Software Guide (MAN 1879).

SCOPE OF MANUAL

This manual is a detailed reference manual for the Prime FORTRAN IV Compiler. It is organized in eight sections for ease of reference.

Section 1 introduces the features and special capabilities of Prime FORTRAN IV.

Section 2 describes the format of source programs prepared for processing by the Compiler.

Section 3 contains reference information on assignment statements and includes definitions of constants, variables, arrays, expressions, and data formats.

Section 4 defines the control statements (GO TO, DO, etc.,) that guide the sequence of program execution and provide for conditional program

branching, etc.

Section 5 describes the non-executable specifications statements which supply information to the compiler concerning data mode typing, storage allocation, data initializing, and run time TRACE.

Section 6 covers FORTRAN input/output and format control: the READ and WRITE statements that transfer data between the processor and external devices, ENCODE/DECODE, and formatting of input or output character s strings. Device control statements (BACKSPACE, etc.) are also covered.

Section 7 discusses statement functions, the intrinsic functions (provided as a special feature of this Compiler), FUNCTION and SUBROUTINE subprograms, and the available library subroutines.

Section 8 contains programming hints and miscellaneous usage details.

The Appendices provide concise reference information:

> Appendix A defines the FORTRAN compiler error messages; Appendix B defines the RUN-TIME error messages; Appendix C summarizes the FORTRAN statements; Appendix D provides program examples; Appendix E summarizes PRIMOS subroutines; Appendix F summarizes IOCS subroutines; Appendix G summarizes FORTRAN Library subroutines; Appendix H summarizes the FORTRAN Math Library subroutines; Appendix I summarizes the FORTRAN compiler subroutines; Appendix J summarizes indicator and control subroutines, and Appendix K summarizes SORT routines.

This manual is concluded by a computer-generated subject index which offers a quick cross reference to any subject covered in this manual.

COMPILING AND RUN TIME FEATURES

Refer to a summary in Appendix B and the compiler section of Prime's Program Development Software User Guide, MAN 1879 for general information pertinent to compiling, debugging, and running of FORTRAN programs.

FORTRAN LIBRARY SUBROUTINES & FUNCTIONS

Refer to Appendix G for a summary and to Prime's Software Library User Guide MAN 1880 for Prime's Floating Point Arithmetic subroutines.

FORTRAN COMPILER SUBROUTINES

Refer to Appendix I for a summary and to Prime's Software Library User Guide, MAN 1880 for a detailed description of Prime's FORTRAN compiler

subroutines.

INDICATION AND CONTROL SUBROUTINES

Refer to Appendix I for a summary and to Prime's Software Library User Guide, MAN 1880 for details of Prime's Indication and Control Subroutines.

INPUT/OUTPUT CONTROL SYSTEM (IOCS)

IOCS is comprised of subroutines that perform input/output between the Prime computer and the disks, terminals and peripheral devices within the system configuration. IOCS is used by programs that use FORTRAN READ and WRITE statements. The device numbers used in these statements correspond to IOCS logical device. Refer to Section 6 of Prime's Software Library User Guide, MAN 1880 or Appendix F of this manual for a summary of IOCS subroutines.

FORTRAN MATH LIBRARY (MATHLB)

MATHLB provides a set of subroutines to perform commonly used applications such as routines to perform matrix opertions, solve systems of simultaneous linear equations, and generate permutations and combinations of elements. Refer to Appendix H of this manual.

PRIME FORTRAN IV FEATURES

Prime's FORTRAN IV Compiler processes source programs prepared in USA Standard FORTRAN, as defined in American National Standard ANSI X3.9-1966. In addition, Prime FORTRAN has several powerful extensions which improve the language's usefulness in writing high-level programs such as disk or real time operating systems.

The one-pass compiler is compatible with PRIMOS II, PRIMOS III, PRIMOS IV and Real Time Operating System, and is able to run in a stand-alone environment as well. The compiler produces highly optimized code and is supported by an extensive array of mathematical functions and subroutines.

Object code generated by the compiler is in a binary format suitable for loading by Prime's (LOAD) Linking Loader. Library subroutines are supplied in the same format.

Advantages of Prime FORTRAN

* Uses terms already familiar to the scientist, engineer or mathematician; easy to learn and use.

* Procedure-oriented terms and statements eliminate the need for detailed coding and reduce the chance of coding errors.

* One statement replaces many assembly language instructions; this reduces time and cost normally associated with programming.

* Programs are virtually self-documenting; this permits the work of one user to be referenced, maintained or altered by another with ease.

* Program correction is simplified by error diagnostics automatically inserted into program listing.

* Opens and closes PRIMOS files. This means that the code is written within the FORTRAN program to update PRIMOS files.

* Assembly Language Prime library subroutines can be called by a FORTRAN IV program.

Prime FORTRAN Extensions

* Intrinsic Functions: XOR, AND NOT, IABS, SHFT. (In many cases, compiler generates in-line coding instead of calling library subroutine.)

* Octal constants: allowed in forms nOdddd and :dddddd, where n is the number of octal digits, ´O´ or ´:´ indicates an octal constant and dddd is the actual number.

* Quoted Hollerith Strings: can be stated between apostrophes in addtion to the standard format (nH---). Example: ´ABCD´ (same as 4HABCD)

* Global Variable Type: if no argument appears in a REAL, INTEGER, COMPLEX, DOUBLE PRECISION, or LOGICAL mode type statement, all variables not specifically designated as to type will be in that mode.

* Protected Functions and Subroutines: the FUNCTION or SUBROUTINE statements may be preceded by the word PROTECTED; interrupts are then inhibited upon entry to the subroutine and enabled upon return to the main program.

* Special Array LIST and LOC function: enables FORTRAN user to reference absolute memory locations, as an aid in systems programming.

(Note: The LIST feature does not apply to VFTN.)

* Library of Real-Time Functions: as defined by the Workshop of Standardization of Industrial Computer Languages, for operation under RTOS.
* Concordance of symbol usage.
* ENCODE, DECODE statements for format conversion within program unit.
* END= extensions for READ.
* ERR= extensions for DECODE, READ & WRITE.
* Mixed mode expressions permitted.
* Provision for in-line comments.
* Powerful format extensions (B specification).

SECTION 2

SOURCE PROGRAM FORMAT

This section defines many of the basic features of FORTRAN: program and subprogram organization, source statement line formats, the character set, use of spaces, and other general features and restrictions.

Line formats for comment, continuation, initial, and special control lines are also described here. The general format of statement lines is covered, but detailed requirements for the many different types of FORTRAN statements are presented in later sections.

BASIC TERMINOLOGY

This section introduces some basic terminology and the meaning of grammatical forms and particular words.

Program Units

The term program unit refers to either a main program or subprogram.

A program that can be used as a self-contained computing procedure is called an executable program.

An executable program consists of precisely one main program and possibly one or more subprograms.

A main program is a set of statements and comments not containing a FUNCTION, SUBROUTINE, or BLOCK DATA statement.

A subprogram is similar to a main program but is headed by a BLOCK DATA, FUNCTION, or SUBROUTINE statement. A subprogram headed by a BLOCK DATA statement is called a specification subprogram. A subprogram headed by a FUNCTION or SUBROUTINE statement is called a procedure subprogram. (See Section 7).

Any program unit except a specificaton subprogram may reference an external procedure.

An external procedure that is defined by FORTRAN statements is called a procedure subprogram. External procedures also may be defined by other means. An external procedure may be an external function or an external subroutine. An external function headed by a FUNCTION statement is called a function subprogram. An external

subroutine headed by a SUBROUTINE statement is called a subroutine subprogram.

Any program unit consists of statements and comments. A statement is divided into physical sections called lines, the first of which is called an initial line and the rest of which are called continuation lines. However, not all lines contain statements. There is a type of line called a comment that is not a statement and merely provides information for documentary purposes.

The statements in FORTRAN fall into two broad classes--executable and nonexecutable. The executable statements specify the action of the program while the nonexecutable statements describe the use of the program, the characteristics of the operands, editing information, statement functions, or data arrangement.

The syntactic elements of a statement are names and operators. Names are used to reference objects such as data or procedures. Operators, including the imperative verbs, specify action upon named objects. One class of name, the array name, deserves special mention. An array name must have the size of the identified array defined in an array declarator. An array name qualified only by a subscript is used to identify a particular element of the array.

Data names and the arithmetic (or logical) operations may be connected into expressions. Evaluation of such an expression develops a value. This value is derived by performing the specified operations on the named data.

The identifiers used in FORTRAN are names and numbers. Data and Procedures are named. Statements are labeled with numbers and input/output units are numbered.

At various places in this document, there are statements with associated lists of entries. In all cases, the list is assumed to contain at least one entry unless an explicit exception is stated.

Example:

```
SUBROUTINE s (al, a2, . . . an)
```

It is assumed that at least one symbolic name is included in the list within parentheses. A list is a set of identifiable elements each of which is separated from its successor by a comma.

CHARACTER SET

A program unit is written using the following characters: A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z, Ø, 1, 2, 3, 4, 5, 6, 7, 8, 9, and the following special characters:

Character	Name of Character
	Blank
=	Equals
'	Single Quote
:	Colon
+	Plus
-	Minus
*	Asterisk
/	Slash
(	Left Parenthesis
)	Right Parenthesis
,	Comma
.	Decimal Point
$	Dollar Sign

NOTES:

1. The order in which the characters are listed does not imply a collating sequence.

2. Blank (space) characters have no meaning (except in character string constants) in Prime FORTRAN IV programs. However, each blank space counts as a character position.

Digits

A digit is one of the ten characters: Ø, 1, 2, 3, 4, 5, 6, 7, 8, 9. Unless specified otherwise, a string of digits will be interpreted in the decimal base number system when a number system base interpretation is appropriate.

An octal digit is one of the eight characters: Ø, 1, 2, 3, 4, 5, 6, 7.

Letters

A letter is one of the twenty-six characters: A, B, C, D, E, F, G, H, I, J, K, L. M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z.

Alphanumeric Characters

An alphanumeric character is a letter or a digit.

Special Characters

A special character is one of the eleven characters: blank, equals, plus, minus, asterisk, slash, left parenthesis, right parenthesis, comma, decimal point, dollar sign, single quote, and colon.

Blank Character

A blank character has no meaning except within Hollerith constants and may be used freely to improve the intelligibility of the program. However, with formatted inputs or outputs, a blank is considered one character position. (See Section 6).

PROGRAM FORM

Every program unit is constructed of characters grouped into lines and statements. The required ordering of FORTRAN statements is shown in Figure 2-1. TRACE and LIST control statements can be used anywhere in the program.

The program can consist of any combination of statement, comment, and special control lines, provided the last line contains an END statement.

FORTRAN program may be handwritten on coding forms for unit record keypunching or keyed directly into the computer with the aid of the Text Editor. In either case, certain rules regarding column boundaries must be observed. Figure 2-2 illustrates a section of a FORTRAN program as entered at an on-line teleprinter keyboard. This also illustrates each of of the line/control functions given below.

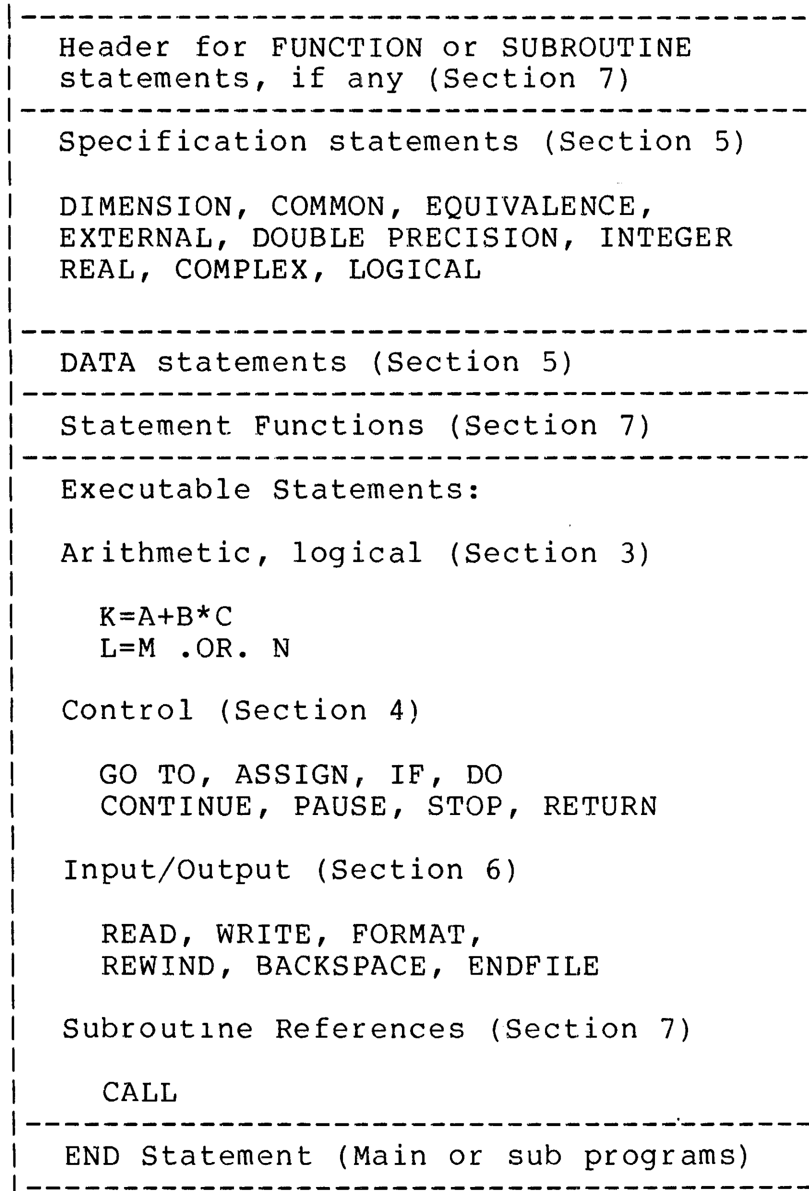


Figure 2-1. Order of Elements in a FORTRAN Source Program

COLUMN

```
         1         2         3         4         5         6         7
123456789012345678901234567890123456789012345678901234567890123456789012

C    PROGRAM
      INTEGER P1,P2,P3
      DIMENSION IAR(200),RAR(100)
      EQUIVALENCE(IAR(1),RAR(1)),(IAR(1),IAR1),(RAR(1),RAR1)
      P1=100
      P2=100
      P3=100
      DO 100 I=1,P1
      DO 100 J=1,P2
      DO 100 K=1,P3
100   CONTINUE
C    FIXED POINT ARITHMETIC TEST
      DO 200 I=1,P1
      DO 200 J=1,200
      IAR(J)=((((P1*10)+(P2/10))+P3)/10000)*20
C    RESULT SHOULD EQUAL 29325
200   CONTINUE
C    FLOATING POINT ARITHMETIC TEST
      DO 300 I=1,P2
      DO 300 L=1,100
      AA=10000.0
      BB=9999.99
      CC=10.32
      RAR(L)=((((AA*10.0)+(AA/10.0))+AA)/BB)*CC
C    RESULT SHOULD EQUAL 114.552
300   CONTINUE
      NOUT=1
      WRITE(NOUT,400)IAR1,RAR1
400   FORMAT(15H FIXED VALUE IS,I5,15H FLOAT VALUE IS F7.3)
      CALL EXIT
      END
```

Figure 2-2. Example of FORTRAN Source Program Input

Lines

A line is a string of up to 72 characters. All characters must be from the FORTRAN character set except where otherwise stated.

The character positions in a line are called columns and are consecutively numbered 1, 2, 3, through 72, from left to right.

Lines are subdivided into field, as shown in Figure 2-3.

```
                          STATEMENTS:          MUST BE BLANK

                          COMMENTS:            ANY CHARACTER

 $=CONTROL SYMBOL         CONTINUATION LINES:  MUST CONTAIN A CHARACTER
 C=COMMENT                                     (ANY EXCEPT BLANK OR ZERO)

                                                                 SEQUENCE
                                         STATEMENT FIELD         NUMBER

 1   2          5    6   7                               72  73        80

   STATEMENT
   NUMBER
```

Figure 2-3. Line Format

Comment Lines

If column 1 contains the letter C, the rest of the line is treated as comments (i.e., is ignored by the compiler except for being reproduced on the listing). The comment text may begin in column 2 and extend through column 72. See example X5 in Appendix D for program example.

Inline Comments

Inline comments are permitted by using the following syntax:

```
/* comment */
```

The comments may not be used within Hollerith strings but may appear anywhere else. The end of a source record terminates the comment.

Statement Line

The statement field of a statement line contains any one of the statement types defined elsewhere in this manual. The statement must begin in column 7 and may extend to column 72. If a statement is longer than one line, it may be continued. (See Continuation Line.)

Statement Number (Label)

Columns 1-5 of a statement line may be used for an optional statement number. Statement numbers consist of 1 to 5 decimal digits positioned anywhere in columns 1-5. Spaces and leading zeroes are ignored. A statement number is requried only if the statement is referenced as a label in a GO TO or similar statement.

Continuation Line

Continuation lines must be blank in columns 1-5 and must contain a character (anything except blank or zero) in column 6. Columns 7-72 are then interpreted as a continuation of the statment on the preceding line. Any number of continuation lines for a given statement are permitted.

Sequence Number

Columns 73-80 may be used for sequential line identification numbers. This field is ignoed by the compiler except for being reproduced on the listing. It may be left blank.

Spaces

Spaces may be used freely between operators, constants, variables, etc., to improve legibility. Spaces have no meaning except within Hollerith constants.

SECTION 3

ASSIGNMENT STATEMENTS

GENERAL PRINCIPLES

This section defines the form of the arithmetic and logical assignment statements that are the main calculating tools in a FORTRAN program. These statements, which resemble equations in familiar mathematical notation, apply arithmetic, logical, or relational operators to data values. These include:

CONSTANTS

- Integer
- Logical
- Real
- Double precision
- Complex

VARIABLES

- Subscripted Array
- Array Storage

OPERATORS AND EXPRESSIONS

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Mixed Expressions

In FORTRAN IV, data values take the form of arithmetic or logical constants, variables, or expressions. This section also defines the legal forms and ranges for numeric and logical data values.

Simple Statements

The arithmetic and logical statements take the basic form:

```
A=B
```

where A is a simple variable or array element, B is an arithmetic, logical or relational expression, and the equals sign is the replacement operator.

Arithmetic expressions use arithmetic operators, constants, and numerically defined variables.

In the statement:

```
PI=3.14159
```

a variable PI is assigned the value of a constant, 3.14159, in real format. Note that the equals sign has the meaning "let PI equal --". or "assign PI the value of --".

A subsequent statement:

```
A=PI+1.2
```

would assign a value of the expression PI+1.2 to the variable A.

Logical expressions use relational operators on numerical constants or variables to form a logical truth value (TRUE or FALSE) as a result. These truth values are used to control program direction when used in IF and similar statements. A logical TRUE is stored internally as an integer 1, and FALSE is stored as integer 0.

In the expression:

```
A=PI.EQ.3.14159
```

the logical variable A is set TRUE if the variable PI equals the real constant 3.14159. Relational expressions may also be used in IF statements, as in:

```
IF (PI.EQ.3.14159) GO TO 10
```

Logical expressions also apply the common logical operators (AND, OR, NOT) to logically defined (TRUE or FALSE) expressions. The result is a logical TRUE or FALSE.

In the statement:

```
A=P.OR.Q
```

the variable A is assigned the truth value corresponding to the logical OR of logical variables P and Q.

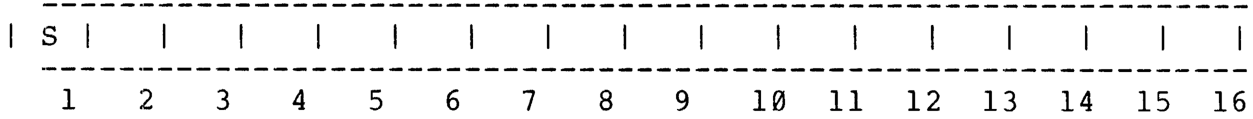
Constant formats, rules for assigning variables, and rules for forming arithmetic, logical, and relational expressions are defined in detail in the following paragraphs.

CONSTANTS IN A FORTRAN STATEMENT

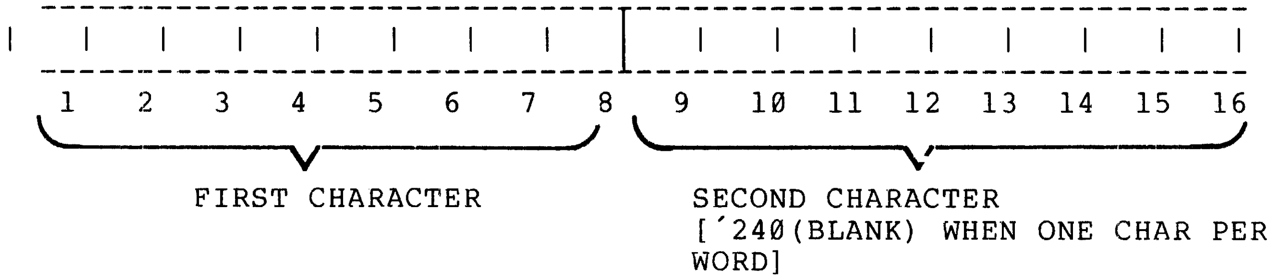
Constants maintain the same numerical, logical or ASCII value during the execution of a program. Numerical constants may be expressed in

six different data type modes: integer (decimal or octal), logical, ASCII, real, double precision, and complex. Computer word formats for these data types are shown in Figure 3-1.

15 MAGNITUDE BITS

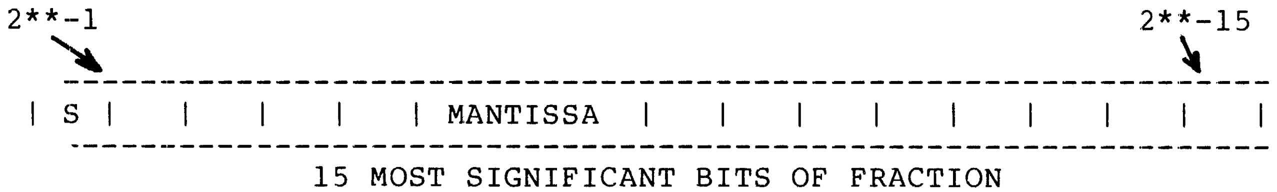


A. INTEGER & LOGICAL

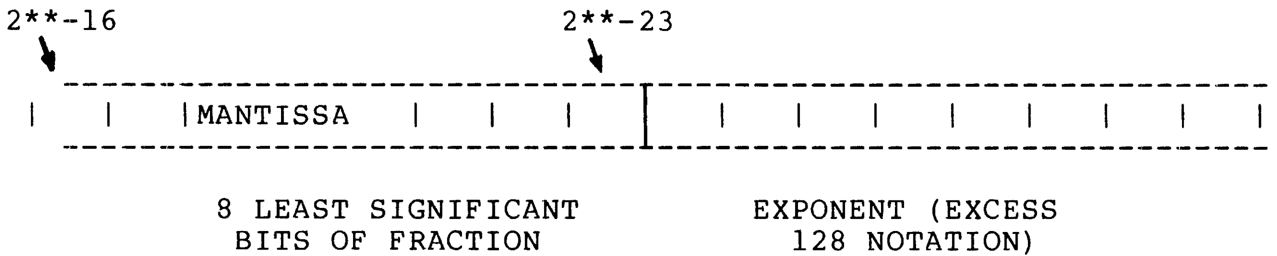


B. ASCII

WORD 1 (MANTISSA)



WORD 2 (MANTISSA + EXP)



C. REAL

Figure 3-1. Computer Internal Word Formats for Constants and Variables

```
                          EA (MANTISSA)

2**-1                                                        2**-15
   ------------------------------------------------------------
 | S |- -|- - - - - - - - MANTISSA- -|- -|- -|- -|- -|- -|- -|- -|
   ------------------------------------------------------------
   1    2   3   4   5   6   7   8   9   10  11  12  13  14  15  16
      Binary Point

                         EA+1 (MANTISSA)

2**-16                                                       2**-31
   ------------------------------------------------------------
 |- -|- -|- -|- -|- -|- -|-MANTISSA- - - - - - - - - - - - - - - |
   ------------------------------------------------------------
   1    2   3   4   5   6   7   8   9   10  11  12  13  14  15  16

                         EA+2 (MANTISSA)

2**-32                                                       2**-47
   ------------------------------------------------------------
 |- -|- -|- -|- -|- -|- -|-MANTISSA- - - - - - - - - - - - - - - |
   ------------------------------------------------------------
   1    2   3   4   5   6   7   8   9   10  11  12  13  14  15  16

             EA+3 EXPONENT  (TWO´S COMPLEMENT)

2**15                                                        2**0
   ------------------------------------------------------------
 |- -|- -|- -|- -|- -|EXPONENT (EXCESS 128) -|- -|- -|- -|- -|- -|
   ------------------------------------------------------------
   1    2   3   4   5   6   7   8   9   10  11  12  13  14  15  16
```

D. DOUBLE PRECISION

Figure 3-1. Computer Internal Word Formats for Constants and Variables (Continued)

Integer Data Types

nd mayr data types occupy one internal computer word (Figure 3-1A) a positive,agnitude from -32,768 to +32,767 (decimal). It may assume negative, and zero values. It may only assume integer values.

Decimal integers are represented by up to 5 decimal digits preceded by an optional + or - sign. No decimal point is allowed. Examples:

```
5     29     -13579   Ø       +ØØØ      -Ø34
```

Octal integers are specified by the form:

```
:dddddd
```

where: dddddd is the actual number; or,

```
nOdddddd
```

where n is the number of octal digits, O is the letter O, and dddddd is up to six octal digits ranging from Ø to 177777. Examples:

```
6O177777     1O1     5O1ØØØØ
```

For program examples, see X1 and X14 in Appendix D.

Logical Data Types

Logical truth (data types) occupy one internal computer word. In source programs, truth values are represented by the notation .TRUE. or .FALSE.. The compiler represents the notation .TRUE. to a memory location containing an integer 1 and represents the notation .FALSE. to a location containing an integer zero. the notation .FALSE. to an integer zero.

Examples:

```
I = .TRUE.  LOG = .FALSE.    IF I .EQ. LOG GO TO 5Ø
```

(Logical expressions are described in more detail later.)

ASCII Data Type

An integer may be assigned the character codes of one or two ASCII characters, represented in either a Hollerith or ´quoted´ format. The Hollerith format is the same as used for entering Hollerith data in Format statements:

```
nHcccccc. . . .
```

where n is the number of characters and each c is one of the ASCII printing characters. The 'quoted' format permits a string of characters to be enclosed in single quotation marks, with no character count or H designator required:

    1HX   2HXY   6H-3.6E2   'X'   'XY'   '-3.6E2'

This will be discussed in more detail in Section 6.

ASCII data may be stored into any variable type. The number of words associated with the type determines the number of I/O characters. ASCII constants of one to two characters are assigned type integer; 3-4 are real and 5-8 are double precision. Over eight characters are allowed in only subroutine calls or data statements.

If only one character is assigned, it is left justified and a space character (240 octal) is placed in the right half of the integer word. (See Figure 3-1B.)

Real Data Types

Real data types occupy two computer words (Figure 3-1C) using the single- precision floating-point data format. It may assume positive, negative and zero values.

Real numbers are represented in source programs by a series of decimal d ndex Real numbers digits including a mandatory decimal point. Any number of digits may be used but only the most significant 7 digits are retained. A sign is optional; if omitted, the quantity is assumed to be positive. An exponent, in the form E+nn, is also optional. Real values can range in magnitude from approximately 10(-38) to 10(+38). Examples.

    1.   +1.23456   123.E5   -123.E-5   1.23 E 5

    0.   0.0   1234567.E32

Double Precision Data Types

Double precision data types occupy four computer words (Figure 3-1D) using the double-precision floating-point format. The exponent range is 10(-9902) to 10(+9825). Notation used in source programs to represent double precision values is the same as for real values except that up to 14 digits of accuracy are retained, and the letter D identifies the exponent:

    123.456789012D13   0.00000000001D-37   -77.777D2   27.DO

Complex Data Types

Complex data types are represented internally as two consecutive real numbers. In source programs, complex values are specified as two real numbers separated by a comma and enclosed in parentheses. The first value represents the real part of the complex constant, the second value is the imaginary part:

```
(123.456,2.E-4)    (2.E3,.01)    (-79.,2.34E-7)
```

Address Constants

A constant of the form:

```
$n
```

has a value equal to the memory address of the first line of code generated by statement number ′n′. If an address constant is used in an expression the result is integer. These constants are mainly used as alternate return address arguments in subroutine calls.

VARIABLES IN A FORTRAN STATEMENT

A variable is a numerical, logical, or ASCII value that has been assigned an alphanumeric label or "symbolic name". The actual value of a variable may change during program execution under control of assignment statements, READ or ASSIGN statements, or function or subroutine calls.

Variable Name

A variable name consists of one to six alphanumeric characters, the first of which must be alphabetic. Incorrect forms are detected during compilation and cause an error message:

Correct	Incorrect	
A	ALPHABET	(too long)
ARG1	1ARG	(first character is a digit)

Variables may be assigned values in any of the modes specified for constants. The first letter of the variable name causes an implicit mode assignment in the absence of other mode control features. The

implicit mode typing convention is:

First letter of Variable Name	Implicit Mode
I,J,K,L,M, or N	Integer
Other	Real

Implicit mode assignments can be overridden by the mode statements INTEGER, REAL, DOUBLE PRECISION, COMPLEX, and LOGICAL described in Section 5. Note that double precision, complex, and logical variables always must be mentioned in an appropriate mode statement; there is no implicit assignment for these modes.

Subscripted (Array) Variables

The variables described so far are scalar - that is, they represent a single quantity. Variable names may also be subscripted so that one symbolic name can identify a set of data items.

The name part of a subscripted variable follows the same rules as the name of a scalar variable.

Subscripts follow the array name, are enclosed in parentheses, and consist of integer constants, variabales, or expressions of certain limited forms. A given array can have up to seven subscripts separated by commas.

Examples:

DATA (1)	The first item of a one-dimensional array named DATA
TABLE (1,4)	Identifies the data item in row 1, column 4 of a two-dimensional array named TABLE
TABLE (I,J)	Identifies the data item in row I, column J of the array named TABLE
DATA (5*ALPHA+11)	The expression 5*ALPHA+11 is evaluated as an integer expression that specifies one item of the one-dimensional array DATA. ALPHA must be a defined integer.
ARRAY (K,5, 5*ALPHA+11)	Identifies a location in a

three-dimensional array ARRAY
as in row K, column 5, and at
depth 5*ALPHA+11; K and ALPHA must
be defined as integers.

Each array must be assigned a storage area by a DIMENSION or COMMON statement or as part of a mode declaration statement. These topics are discussed in full detail in Section 5.

Rule:

All elements of a given array must be of the same type (i.e., all integer or all complex).

Expressions used in subscripts are limited to the following forms:

```
C*V+K
C*V-K
C*V
V+K
V-K
K
V
```

where C and K are integer constants and V is the name of a variable that is explicitly or implicitly defined as an integer. Expressions and the rules for evaluating them are discussed in more detail in the following paragraphs.

Array Storage Arrangement

In the object program, a two-dimentional array A is stored sequentially in the order A(1,1), A(2,1),...A(m,1; A(1,2), A(2,2),...,A(m,2);..., A(m,n). Note that the first of the subscripts varies most rapidly, and the last varies least rapidly. The same principle applies to the subscripts of dimensional arrays up to seven dimensions.

All arrays are stored forward in storage; i.e., the following array:

```
A(1,1)   A(2,1)   A(3,1)
A(1,2)   A(2,2)   A(3,2)
A(1,3)   A(2,3)   A(3,3)
```

is stored in increasing absolute locations:

Location	Array Element
1	A(1,1)
2	A(2,1)

3	A(3,1)
4	A(1,2)
5	A(2,2)
6	A(3,2)
7	A(1,3)
8	A(2,3)
9	A(3,3)

See Example in Appendix D for program illustration.

OPERATORS AND EXPRESSIONS IN FORTRAN STATEMENTS

Expressions consist of constants, variables (scalar or array), expressions, or function references linked by operators. Arithmetic logical, and relational expressions are distinguished by the types of operator used.

Arithmetic Operators

Arithmetic expressions consist of arithmetic constants, variables, or function references linked by one or more of the following arithmetic operators:

**	Exponentiation
-	Unary minus
* and /	Multiplication and Division
+ and -	Addition and Subtraction
=	Equals or Replacement

The compiler permits mixed mode arithmetic expressions except for operations involving a complex and double-precision item. See program examples X6 and X11 in Appendix D.

Order of Operations Rule:

Operations are performed in the order the operators are listed above (from top to bottom). For operators of equal priority, operations are performed from left to right:

Expression	Result
3+5-7	1
3*5-7	8
3-5*7	-32

Caution is required when integers are used within expressions. FORTRAN performs "greatest integer" arithmetic on integers - remainders after division are truncated. Inadvertent truncation can be avoided by using real values, or by controlling the sequence of evaluation.

Expression	Result	
3*4/2	6	
3/2*4	4	(3/2 is evaluated as the integer 1
3./2.*4.	6.	(Real numbers are evaluated)
3/6*4	0	(3/6 is evaluated as 0)

Parentheses Rule:

Parentheses may be used in much the same way they are used in ordinary arithmetic expressions, to group and clarify operations. In FORTRAN expressions, parentheses may also affect the order in which operations are performed. All operations within a set of parentheses are carried out before the result is processed by operators outside of the parentheses. When parentheses are nested, the innermost expressions are evaluated first:

Expression	Result
3*7-5	16
3*(7-5)	6
(3-5)*7	-14
3./(6.*4.)	.125
2+2*2**2	10
((2+2)*2)**2	64

The examples above use only integer and real numbers for simplicity. FORTRAN arithmetic expressions can use constants of all modes, as well as variables:

2*A/(ALPHA-8)	Integers
2.3*BETA**(B+C-4.7E3)	Real

1.9275D3*(3.27DO-ARG1) Double Precision

(9.2,3.Ø)/3.7 Complex

Multiplication Rule:

Multiplication is never implied; the multiplication operator (*) must be used. The expression AB means A*B in normal algebraic notation, but will be interpreted by FORTRAN as a variable named AB.

Exponentiation Rule:

More than one stage of exponentiation requires parentheses for correct evaluation. While the notation XYZ is valid in algebraic notation, the FORTRAN expression X**Y**Z is illegal. To represent (XY)Z, the FORTRAN expression (X**Y)**Z should be used; X(YZ) is represented correctly by the expression X**(Y**Z).

Two operators cannot follow in succession:

Wrong	Right
X/-Y	X/(-Y) or -X/Y

Relational Operators

The relational operators process two arithmetic arguments to form a logical truth value of .TRUE. or .FALSE. The arguments may be constants, variabales, or expressions of any mode except logical. Mixed mode is permissable except complex and double-precision.

The relational operators are:

.LT.	Less than
.LE.	Less than or equal to
.EQ.	Equal to
.NE.	Not equal to
.GT.	Greater than
.GE.	Greater than or equal to

All have equal priority. The periods are a part of the operator and cannot be omitted.

Examples of relational operator applications:

In Logical Assignment Statements:

A=P.LT.Q	The variable A is set to a logical value of TRUE if arithmetic variable P is less than arithmetic variable Q. Otherwise P is set to FALSE.
B=P.EQ.2.4D-2	Variable B is set to a logical value of TRUE if arithmetic variable P is equal to the specified double-precision constant. Otherwise P is set to FALSE. In this example, P must be in double-precision, real, or integer mode. The result is logical mode.

NOTE

Logical variables (A and B, above) must be declared by a LOGICAL specification statement. See Appendix D, example X1 for program example.

In Logical IF statements:

```
IF (P.LT.Q) GO TO 30

IF (I.EQ.2HAB) GO TO 50
```

(For more information on this usage, see Section 4.

Logical Operators

A logical operator combines one or two operands to form a result that is a logical truth value of .TRUE. or .FALSE. The arguments may be defined by statements using relational logical operators, or by a LOGICAL specification statement. The logical operators are:

Operator	Function
.NOT.	Negates (reverses the logical state of) the following argument, as in:

A=.NOT.P

P	A
F	T

T F

.AND. Generates the logical AND function of two logical arguments, as in:

```
A=ARG1.AND.ARG2
```

ARG1	ARG2	A
F	F	F
F	T	F
T	F	F
T	T	T

.OR. Generates the logical inclusive OR function of two logical arguments, as in:

```
A=ARG1.OR.ARG2
```

ARG1	ARG2	A
F	F	F
F	T	T
T	F	T
T	T	T

Examples of logical operator applications:

In Logical Assignment Statements:

```
A=P .LT. Q .AND. Q .GT. R

B=P .LT. Q .OR. R .EQ. S
```

In Logical IF Statements:

```
IF (P.LT.Q.AND.R.EQ.S) GO TO 15

IF (P.LT.Q) GO TO 20
```

Integer values can be subjected to full-word logical operations using the intrinsic logical functions described in Section 7.

Order of Evaluation

The order in which a complicated expression is evaluated can be determined precisely if a few ground rules are kept in mind. Each torp in the evaluation involves a group consisting of a single opera tnd one or two arguments. The order in which these operator/argumen

derups are evaluated depends primarily on operator priority. The or er,priority for all FORTRAN operators is listed in Table 3-1. Howev position from left to right, and the presence of parentheses, may affect the order of evaluation. For example, in the expression:

```
SQRT(A**(B-C+2.))
```

the SQRT function has highest priority, but its argument consists of and expression, in parentheses, that must be evaluated first. Within the outer parentheses, the exponential operator has priority, but one of its arguments (B-C+2.) is also an expression that must be evaluated. In the latter expression both operators have the same level of priority so they are evaluated from left to right; the expression B-C is evaluated first, producing an intermediate result "R1".

The complete evaluation sequence is:

Step	Argument 1	Operator	Argument 2	Result
1	B	-	C	R1
2	R1	+	2	R2
3	A	**	R2	R3
4	R3	SQRT	--	Final Result

An expression of any complexity can be analyzed in this manner to detect possible error conditions such as use of mixed modes, etc.

Among operators of equal priority, the laws of commutativity and associativity are used to rearrange the order of evaluation.

Table 3-1. Operator Priority

Priority	Operator	Operation
First	FUNCTIONS	Function subroutine
	**	Exponentiation
	* /	Multiply or divide
	+ -	Add or subtract
	.LT.,.LE.,.EQ., .NE.,.GT.,GE.	Relational operators
	.NOT.	Logical negate
	.AND.	Logical AND
Last	.OR	Logical OR

Mixed Mode Expressions

nteger,tic and Relational operators may freely combine operands of i real, double-precision, and complex modes. The restrictions are:

1) No operator can combine a complex and double precision operand.

2) If operands of different modes are combined, the following results:

Mixed Operands	Results
Complex-Real	Complex

Complex-Integer	Complex
DP - Real	DP
DP -Integer	DP
Real-Integer	Real

3) Subscripts of array variables must be in integer mode:

   ARRAY(3,I,J)

4) Arguments supplied to subroutines (for example in CALL statements) must be in the mode required by the subroutine.

Assignment Rules

In arithmetic assignment statements of the form:

```
A = B
```

different data modes may be used on either side of the replacemnt operator, within the limits specified by Table 3-2.

Table 3-2. Rules for Assignment Of B to A

If A Mode is	And B Mode is	The Assignment Rule Is*
Integer	Integer	Assign
Integer	Real	Fix & Assign
Integer	Double Precision	Fix & Assign
Integer	Complex	Fix & Assign Real Part
Real	Integer	Float & Assign
Real	Real	Assign
Real	Double Precision	DP Evaluate & Real Assign
Real	Complex	Assign Real Part
Double Precision	Integer	DP Float & Assign
Double Precision	Real	DP Evaluate & Assign
Double Precision	Double Precision	Assign
Double Precision	Complex	P
Complex	Integer	Float & Assign Real Part Zero Image Part
Complex	Real	Assign Real Part Zero Image Part
Complex	Double Precision	P
Complex	Complex	Assign

*NOTES

(1) P means prohibit combination.

(2) Assign means transmit the resulting value, without change, to the entity.

(3) Real Assign means transmit to the entity as much precision of the most significant part of the resulting value as a real datum can contain.

(4) DP Evaluate means evaluate the expression, then DP Float.

(5) Fix means truncate any fractional part of the result and transform that value to the form of an integer datum.

(6) Float means transform the value to the form of a real datum.

(7) DP Float means transform the value to the form of a double precision datum.

SECTION 4

CONTROL STATEMENTS

The statements of a FORTRAN program are usually executed in the order in which they are listed in the source program. Control statements make it possible to alter the sequence of execution, branch conditionally to different statements depending on the result of computations, form execution loops, call and return from subroutines, and stop program execution. The control statements described in this section are:

GO TO (unconditional, computed, and assigned)

IF (arithmetic and logical)

DO

CONTINUE

STOP

PAUSE

END

Also described in this section is the ASSIGN statement which is used only in conjunction with assigned GO TO statements. The CALL and RETURN statements are described in Section 7, together with other information on subroutine references.

Statement labels used as arguments in control statements must be assigned to executable statements in the same program unit as the control statement itself.

UNCONDITIONAL GO TO STATEMENT

An unconditional GO TO statement is of the form:

```
GO TO k
```

where k is a statement label.

This statement causes the statement identified by the statement label to be executed next.

Examples: See example in Appendix D.

COMPUTED GO TO STATEMENT

A computed GO TO statement is of the form:

```
GO TO (k(1), k(2), ... , k(n)), i
```

where the k´s are statement labels and i is an integer expression reference.

This statement causes the statement identified by the statement label k(j) to be executed next, where j is the value of i at the time of the execution. The result is a conditional branch to one of the k destinations, depending on the value of i. If i is less than 1 or greater than n, then control will be transferred to the next sequential statement.

Example:

```
GO TO (15,25,7),J
```

If J is elsewhere assigned the value 2, control will be transferred to statement 25. For a program example, see X1Ø in Appendix D.

ASSIGNED GO TO STATEMENT

An assigned GO TO statement is of the form:

```
GO TO i, (k1,k2, ... , kn)
```

NOTE: The parenthesized statement label list is optional.

where i is an integer variable reference, and the k´s are statement labels. More than one statement reference is optional.

At the time an assigned GO TO statement is executed, the current value of i must have been ASSIGNed to a statement label. The statement identified by that label is executed next.

Example:

```
ASSIGN 2Ø TO ADCON

GO TO ADCON, (7,2Ø,1ØØ)
```

ASSIGN STATEMENT

A GO TO assignment statement is of the form:

```
ASSIGN k TO i
```

where k is a statement label and i is an integer variable name. After execution of such a statement, subsequent execution of any assigned GO TO statement using that integer variable will cause the statement identified by the assigned statement label to be executed next, provided the variable has not been redefined.

Example:

```
     ASSIGN 320 TO I

20   GO TO I,(100,310,320,409)
320  A = B + C
     ASSIGN 100 TO I
     GO TO 20
100  Y = A*X
 .
 .
 .
```

Once it has been mentioned in an ASSIGN statement, an integer variable may not be referenced in any statement other than an assigned GO TO statement (or as a statement number parameter in a subroutine call) until it has been redefined.

ARITHMETIC IF STATEMENT

An arithmetic IF statement is of the form:

```
IF (e) K(1),k(2),k(3)
```

where e is any arithmetic expression of integer, real, or double precision type, and the k´s are statement labels.

The arithmetic IF is a three-way branch conditional upon the value of expression e:

Value of e	Statement Executed Next
<0 (negative)	k(1)
=0	k(2)
>0 (positive)	k(3)

Example:

```
IF (TIME) 20,25,30
```

Here, if TIME is elsewhere assigned the value 2.5, control is

transferred to statement 30. Other examples of the format:

```
IF (A+B-C) 20,25,30

IF (A+1.133) 7,50,100
```

Additional examples are Example X2 and X9, Appendix D.

LOGICAL IF STATEMENT

A logical IF statement is of the form:

```
IF (e) S
```

where e is a logical expression and S is any executable statement except a DO statement or another logical IF statement.

The logical expression e is evaluated. If e has the value .TRUE., statement S is executed. Otherwise, control passes to the next statement. Examples:

```
IF (P.OR.Q) C=P1*D

IF (X.GT.Y) CALL XFER (A,B)

IF (K.LT.L) GO TO 10

IF (INPT.EQ.´X´) BFR=INPT
```

See Program example X1 in Appendix D.

DO STATEMENT

A DO statement is of one of the forms:

```
DO n i = m(1), m(2), m(3)

        or

DO n i = m(1), m(2)
```

where:

(1) n is the label of an executable statement following the DO statement in the same program unit. This statement, called the terminal statement of the associated DO, must not be a GO TO of any form, arithmetic IF, RETURN, STOP, PAUSE, or Do statement, nor a logical IF containing any of these forms.

(2) i is an integer variable called the index.

(3) m(1), m(2), and m(3) are the initial, limit, and increment values of the index, respectively. They are each either an integer constant or integer variable reference. If the second form of the DO statement is used, m(3) is assumed to be 1. At time of execution of the DO statement, m(3) must be greater than zero.

A DO statement causes looping or repeated execution of a series of statements. The range associated with DO statement consists of all executable statements following the DO, to (and including) the terminal statement.

During execution of a DO statement, i is set equal to m(1), and all executable statements within the range of the DO are executed at least once. After each execution, the value of i is increased by m(3) and if i is equal to or less than m(2), the same group of statements is executed again. This process repeats until i is greater than m(2) (the limit). After the last execution, control passes to the statement following the terminal statement and the index variable is left at an undefined value. This is called the normal exit from the range. A control statement within the range may also cause exit. Examples:

```
DO 15 I = 1, 10, 1
```

This loop is executed 10 times. I is 10 during the last execution.

```
DO 15 I = 3, 10, 2
```

This loop is executed 4 times. I is 9 during the last execution.

During execution, the index variable is available for use in arithmetic statements to control the results of computation. For example:

```
DO 50 I = 1, 9

50 A(I) = I*3.14159
```

sets up a table of integer multiples of PI.

Nested DO Loops

A DO loop may include other DO loops which in turn may contain other DO loops. The main rule for nesting is that the terminating statement of an inner loop must occur before the terminating statement of the next higher loop. An exception is that a single terminator can terminate two or more loops. (See statement 5 in the second example.) DO loops can be nested to any depth.

First example of Nested DO´s

```
    DO 2Ø K=1,14,2
    A(K,1) = Ø.Ø
    DO 15  +=4,31.3
 15 A(K,1)=A(K,1)+K,L)
    A(K,2)=A(K,1)/3.14
 2Ø A(K,3)=A(K,1)/5Ø3.7
```

Second Example of Nested DO Loops:

```
PERMITTED                          NOT PERMITTED

    DO 5,I=1,5                         DO 1,I=1,2O,2

    DO 1,J=2,1O,2                      DO 2,J=1,5

1   CONTINUE                       1   CONTINUE

    DO 4,K=1,5                         DO 3,K=2,2Ø,2

    DO 2,L=1,1Ø,2                  2   CONTINUE

2   CONTINUE                       3   CONTINUE

    DO 3,M=2,2Ø,3

3   CONTINUE

4   CONTINUE

    DO 5,N=1,3

5   CONTINUE
```

Extended Range

A DO is said to have an extended range if:

(1) A GO TO statement or arithmetic IF statement within the range of the innermost DO of a nest can cause control to pass out of the next, and;

(2) A GO TO statement or arithmetic IF statement not within the
st nest will cause control to return into the range of the innermo
DO of the first nest.

If both of these conditions apply, the extended range is defined as all statements that may be executed from the time control leaves the nest to the time control returns to the nest.

Restrictions

A GO TO statement or an arithmetic IF statement may not cause control to pass into the range of a DO unless it is being executed as part of the extended range of the DO.

The extended range of a DO may not contain another DO that has an extended range.

When a procedure reference occurs in the range of a DO, that procedure is considered to be temporarily within that range during the execution of that reference.

The control variable, initial parameter, terminal parameter, and incrementation parameter of a DO may not be redefined during the execution of the range or extended range of that DO.

A statement label that is the terminal statement of more than one DO statement may not be used in any GO TO or arithmetic IF statement except one within the range of the DO most deeply contained within that terminal statement.

CONTINUE STATEMENT

A CONTINUE statement is of the form:

```
       CONTINUE
```

This statement terminates the current execution of a DO loop. If no DO loop is in effect, control transfers to the next executable statement. CONTINUE may be used as a labelled entry point, for example as the target of a conditional GO TO statement. Example:

```
            If A.EQ.B GO TO 12Ø
       .
       .
       .
       12Ø  CONTINUE
            A = Z
            (etc.)
```

For additional program examples, see X1 and X14 in Appendix D.

STOP STATEMENT

A STOP statement is of the form:

STOP [n]

where ´n´ is an optional string of one to five decimal digits. This statement transfers control to the subroutine F$HT, which types the letters ****ST (followed by the octal equivalent of ´n´ if specified) on the user terminal, and returns to the operating system (or halts the CPU when paper tape). After a START command, the message is printed again.

PAUSE STATEMENT

A PAUSE statement is of the form:

PAUSE [n]

where n is an optional decimal constant. This statement transfers control to the F$HT subroutine, which types the message ****PA, followed by the octal equivalent of the specified number. This feature can be used to stop the program temporarily and allow the operator to change tape, set sense switches, etc. It is customary for n to identify the PAUSE statement that caused the halt.

After a START command, the program continues operation at the first executable statement following the PAUSE statement.

END STATEMENT

This statement must be placed at the end of every subprogram:

END

SECTION 5

SPECIFICATION STATEMENTS

Specification statements are non-executable statements which supply information to the compiler. For convenience, they are divided into the following functional categories:

<u>Data Type Mode Specification Statements</u>

INTEGER
REAL
DOUBLE PRECISION
COMPLEX
LOGICAL

<u>Storage Specification Statements</u>

DIMENSION
EQUIVALENCE
COMMON

<u>External Procedure Specificaton Statements</u>

EXTERNAL

<u>Data Defining Statements</u>

DATA

<u>Compilation and Run Time Control Statements</u>

TRACE
LIST
NOLIST
FULL LIST
$INSERT

DATA TYPE MODE SPECIFICATION STATEMENTS

These statements are used to override the implicit mode assignments controlled by the first letter of a variable name. Thus, variables with names that begin with I, J, K, L, M, N,can be defined as REAL

or DOUBLE PRECISION, and so on.

A mode specification statement is of the form:

```
Mode v1, v2, ... vn
```

where Mode is INTEGER, REAL, DOUBLE PRECISION, COMPLEX, or LOGICAL, and each v is a variable name, an array name, a function name, or an array declarator.

A mode statement is used to override or confirm the implicit mode assignment specified by the first letter of the name (I, J, K, L, M, N, for integer), to declare entities to be double precision, complex, or logical modes, and to supply array dimension information.

A global mode assignment may be made by a mode statement without a list of names. All variables that do not appear in another mode specification statement are defined as the global mode.

NOTE:
The global mode assignment does not affect variables, arrays, or function name occuring before a global mode statement.

INTEGER statements are used to declare integer mode for variables, arrays, or functions.

```
INTEGER A5, MATRIX, B37
```

REAL statements may be used to assign real (single precision floating point) mode to variables, arrays, or functions.

```
REAL J22, NEXT, MATRIX, IFX
```

DOUBLE PRECISION statements assign variables, arrays, or functions to the double precision floating point mode:

```
DOUBLE PRECISION AMT, INT, STAT25
```

COMPLEX statements assign variables, arrays, or functions to complex floating point mode:

LOGICAL statements assign variables, arrays or functions to logical mode:

```
LOGICAL P, Q, R, XOR, IMPL
```

Alternate Methods of Declaring Arrays

The array is specified in the same manner as in the DIMENSION statement, by following the array name with the maximum size of each dimension within parenthesis. Examples:

```
INTEGER X(10),Y,Z, (5,3,2)

REAL K1, K2(4,3)

DOUBLE PRECISION D(10)

COMPLEX C1, C2(3,7)

LOGICAL L(4,4,4)

COMMON A1, B(6)
```

STORAGE SPECIFICATION STATEMENTS

As previously stated in Secton 3, arrays can have up to seven subscripts separated by commas. Each array must be assigned a storage area by a DIMENSION or COMMON statement, or as part of a mode declaration statement.

These statements provide the compiler with information on the size of arrays, specify common storage areas for use by two or more programs, and identify external subprogram names.

DIMENSION Statement

The DIMENSION statement is used to declare arrays and define their sizes.

A DIMENSION statement is of the form:

```
DIMENSION v1(I1), v2(I2), ..., vn(In)
```

where each v is a variable that is assigned as the name of an array (See Section 3), and each i is a series of one to seven dimensions that define the dimensions of the array. The maximum number of array dimensions is seven. The dimensions are positive non-zero integer constants, or dummy variables, separated by commas. The value of each dimension defines the maximum value for that dimension. The DIMENSION statement thus establishes the name and maximum storage requirement of an array. Example:

```
DIMENSION LST1(100), TABLE(10,10,4), ARRAY(40,10)
```

For program example, see X13 in Appendix D.

An alternate way to define arrays is described under Data Mode Specification Statements.

An array must be re-declared when it is passed to a subprogram. In both declarations, the mode, number of dimensions, and size of each

dimension must agree, but the name of the array need not agree. For example:

```
DIMENSION TABL(100)
.
.
CALL STAT (TABL, PARAM)
- - - - - - - - - - - - - -
SUBROUTINE STAT(JOB, J)
.
.
DIMENSION JOB(100)
```

In a FORTRAN subprogram where the calling program provides (and declares) the array name and all variable subscripts, the subscripts in a DIMENSION statement may be integer variables instead of constants.

This feature adds to the flexibility of general purpose subroutines that process arrays. Rather than specifying an array of fixed size within the subroutine itself, the calling program can declare the size of an array of any size (memory space permitting). Constant dimensions must beused in the calling program, however. Example:

```
DIMENSION MATRIX (25,25)
.
.
CALL EVAL (MATRIX, 25, Z)
- - - - - - - - - - - - - - -
SUBROUTINE EVAL (TYPE, I,B)
.
.
DIMENSION TYPE (I, I)
```

In the subprogram's DIMENSION statement, both the array's name and all variable subscripts must be dummy names.

EQUIVALENCE Statement

The EQUIVALENCE statement permits a program to share memory storage by two or more variables or array elements within a single set of parentheses.

An EQUIVALENCE statement is of the form:

```
EQUIVALENCE (k11,k12,k13...),(k21,k22,k23...)
```

where each k is a variable, subscripted variable, or array name separated by commas. Each element in the list is assigned the same memory storage by the compiler. Subscripts appearing in an equivalence list must be integer, positive, non-zero constants.

An EQUIVALENCE statement equates single variables to each other, entire arrays to each other, elements of an array to single variables, or vice versa.

An element of an array may be expressed in an EQUIVALENCE statement in any of three ways:

1. It may be expressed exactly as in a DIMENSION statement. Assume the element X (1,1) of the two-dimensional array X(5,5) is to be equated to variable Y (3) of the one-dimensional array Y (10) the statement could be:

   ```
   EQUIVALENCE (X(1,1),Y(3))
   ```

   Note that this statement also equates X (2,1) with Y (4), X (3,1) with Y (5) etc.

2. It may be expressed as the equivalent fictitious single-dimensional subscript that indicates the order in which the element is stored in memory. Again assuming element X (1,1) is to be equated to variable Y (3), the statement could be written:

   ```
   EQUIVALENCE (X(1),Y(3))
   ```

   where X (1) specifies that the element X (1,1) is stored in the first location of the storage block reserved for the two-dimensional array X (5,5).

3. The array name may be stated without a subscript; a subscript of 1 is assumed. Thus, elements X (1) and Y (3) could be equated by:

   ```
   EQUIVALENCE (X,Y(3))
   ```

The data type (mode) assigned to each element determines the number of memory cells occupied by each element:

Data Type (Mode)	Number of Memory Words
Integer, Logical	1
Real	2
Double Precision	4
Complex	4

If an INTEGER or LOGICAL data type (mode) variable is made equivalent to a REAL, DOUBLE PRECISION or COMPLEX mode variable, the former variable shares memory storage with the first of the words

required by the latter variable.

Variables and array elements appearing in EQUIVALENCE statements may also appear in COMMON statements. However, the EQUIVALENCE statements must not re-origin a common block, as in:

```
REAL Y(10)
COMMON X(5)                    (Not permitted)
EQUIVALENCE (Y(9),X(2))
```

and must not re-order common, as in:

```
COMMON X,Y
EQUIVALENCE (X,Y)

     or                        (Not permitted)

COMMON X,Y
EQUIVALENCE (X,A)
EQUIVALENCE (A,Y)
```

Dummy arguments for a subprogram cannot be used as elements within an EQUIVALENCE statement contained in that subprogram.

COMMON Statement

A COMMON statement enables a program to share memory storage among two or more program units, and to specify the names of variables and arrays that are to occupy this area.

A COMMON statement is of the form:

```
COMMON/x1/a1/ . . ./xn/an
```

where each a is a nonempty list of variable names, array names, or array characters (no dummy arguments are permitted) and each x is a COMMON block name or is empty.

A COMMON block name may be unspecified (called blank common); if such a block appears first in a COMMON statement, the first two slashes may be omitted.

Names of COMMON blocks must not be identical with the name of a subprogram called on by the program job, or the name of a subroutine in the FORTRAN library. The following example illustrates some acceptable COMMON statements:

```
COMMON P,Q,R (20)

COMMON // P,Q,R (20)
```

```
     COMMON / R1/X,Y,Z

10   COMMON P,Q,R(20)/R1/S,T,U

     COMMON // P,Q,R(20)/R1/S,T,U

    X /R2/V,W(21,4),J

20   COMMON /R1/S,T//P/R1/U//Q,R(20)

C    STATEMENTS 10 AND 20 ARE EFFECTIVELY IDENTICAL
```

Data items are assigned sequentially within a COMMON block in the order of appearance. The loader program assigns all COMMON blocks with the same name to the same area, regardless of the program or subprogram in which they are defined. Blank common data is assigned in such a way that it overlaps the loader program, thereby making the memory area occupied by the leader program available for data storage.

NOTE:
The form // (with no characters except blanks between slashes) may be used to denote blank common.

The number of words that a COMMON block occupies depends on the number of elements, the mode of the elements, and the interrelations between the elements specified by an EQUIVALENCE statement. COMMON blocks that appear with the same block name (or no name) in various programs or subprograms of the same job are not required to have the elements within the block agree in name, mode, or order but the blocks must agree in total words.

Special Common Block, ´LIST´*

As an aid to system-level programming, this compiler defines absolute memory location ´00001 as the origin of a common block named ´LIST.

It is customary to assign an array called LIST into the labeled common area called LIST, such that the first word in this array is location ´00001, the sixth word location ´00006, etc., as in:

```
COMMON/LIST/LIST(60)
```

Effectively, the subscript of array LIST is the actual memory address.

*This feature is not required when compiling in 64V mode with VFTN.

EXTERNAL PROCEDURE SPECIFICATION STATEMENT

The EXTERNAL statement permits the name of an external FUNCTION subprogram (library or user defined) to be passed as an argument in a subroutine call or function reference. An EXTERNAL statement is of the form:

```
EXTERNAL v1, v2, ... , vn
```

where each v is declared to be an external procedure name. If an external procedure name is used as an argument to another external procedure, it must appear in an EXTERNAL statement in the program unit in which it is so used. Only subprogram names used as arguments need to be declared by an EXTERNAL statement. Example:

Main Program:

```
EXTERNAL SIN, COS
   .
   .
A=EVAL(SIN,X)
   .
   .
B=EVAL(COS,Y)
   .
   .
END
```

FUNCTION Subprogram EVAL:

```
FUNCTION EVAL(F,ARG)
   .
   .
EVAL=F(ARG)+F(ARG/2)
RETURN
END
```

In this example, the FUNCTION subroutine EVAL uses a dummy argument F to be replaced by an external function name. In the main program, the first reference to EVAL specifies SIN as the function name. In the second reference, COS is specified.

DATA DEFINING STATEMENTS

The initial state of variables or array elements may be set up at the time of loading by DATA statements.

DATA Statement

The DATA statement sets variables or array elements to initial values during loading of the object program. (The variables are not re- initialized if the program is restarted without reloading.) A DATA initialization statement is of the form:

```
DATA kl/dl/,k2/d2/,......, kn/dn/
```

where k is a list containing names of non-dummy variables or array elements (with constant subscripts) separated by commas. Each d is a corresponding list of constants with optional signs.

The name list and the data list must correspond in order and data type. If the data list consists of a squence of identical constants, the constant need only be written once and preceded by a repeat count (integer constant) and an asterisk. For example:

```
/1.4,3*2.Ø,Ø.Ø
```

is equivalent to:

```
/1.4,2.Ø,2.Ø,2.Ø,Ø.Ø/
```

Acceptable formats for constants used in data lists are:

Data Type	Examples
INTEGER	/11,-4Ø96,1,+8,6*Ø/
REAL	/11.Ø,-4.Ø96E3,89E-2,+6.Ø,6*Ø./
DOUBLE	/11.D,-4.Ø96D3,89D-2,+Ø.6D1,6*Ø.D/
COMPLEX	/(1.Ø,-4.Ø96E3),(89E-2,+6.Ø),6*(Ø.,Ø.)/
LOGICAL	/.TRUE.,.TRUE.,.FALSE./

The first DATA statement in the example below assigns the value Ø.1Ø762 to A1 (4), 1.ØE5 to X, 1 to K, etc. The assignment is done at load time, not at execution time. A DATA statement is not executable. Examples:

```
DIMENSION A1(1Ø),B4(1Ø)

DOUBLE PRECISION D1

LOGICAL L1, L2, L3

COMPLEX C1
```

```
INTEGER I(10)

DATA A1(4),X,K,D1,L1,J1,
1    C1/0.10762,1.0E5,1,1.0D,
2    .TRUE., ´XY´,(4,3,0.0)/

DATA P/23.7/, Q/0.1E-3/,
1    R/0./, J/4095/, L3/.F./,
2    B4(1),B4(2),B4(3),B4(4),
3    B4(5),B4(6),B4(7),B4(8),
4    B4(9),B4(10),/10*0.0/
DATA I/10*0/
```

An ASCII constant may appear in the data list as a string of up to 8 characters (using either the nHxx format or the ´xx´ format). The characters will be stored as ASCII codes, left justified if necessary.

The variable or array element must be of a data type appropriate to the number of characters:

```
DATA I/´AB´/            (Integer)
DATA A/´ABCD´/          (Real)
DATA D/´ABCDEFGH´/      (Double Precision)
DATA C/´ABCDEFGH´/      (Complex)
```

An array or part of an array may be filled with a long Hollerith string, as in:

```
INTEGER I(5)
DATA I/´0123456789´/
DATA I/´01234´,´56789´/
```

The following form is not permitted:

```
DATA I(1),I(2). . I(5)/´0123456789´/
```

dederical and ASCII values can be mixed within the same array, provi enough items are specified to fill the entire array:

```
INTEGER A(7)
DATA A/3,´012345´,32767,´6789´/
```

DEBUGGING AIDS

The following statements determine functions that take place while a program is being compiled or executed. The TRACE function causes Diagnostic printouts of the results of computation to facilitate debugging. Three listing control statements, NOLIST, LIST and FULL LIST are provided.

TRACE Statements

TRACE statements are used as a program debugging function. TRACE allows a listing of each of the current value of each assigned variable.

TRACE statements can be used in two formats, item trace and area trace.

Example:

The following is a partial program and a corresponding TRACE listing format which contains both item trace and area trace.

PARTIAL PROGRAM

```
   DIMENSION A(3,3)
   TRACE Y,A
   X = 3.24
   Y = X + 1.5
   Z = Y**2
   DO 48 ;=1,3
   A(1,2)= Y/1
48 Y = Y + 1.Ø
   X = Ø.Ø
   K = 2
   TRACE 62
5Ø X = X+1.Ø
   IF(X-3.Ø) 51,53,53
51 K = K*K
   GO TO 5Ø
53 IF(X.LE.Y)  X=X+1ØØ.Ø
62 X = X-1.Ø
   Z = 2 * X
   Y = Ø.Ø
     ...
```

TRACE LISTING

```
Y      = Ø.474ØØØØØØØE Ø1
A      (4) = Ø.237ØØØØØØØE Ø1
Y      = Ø.574ØØØØØØØE Ø1
       (5) = Ø.287ØØØØØØØE Ø1
Y      = Ø.774ØØØØØØØE Ø1
A      (6) = Ø.337ØØØØØØØE Ø1
Y      = Ø.774ØØØØØØØE Ø1
(5Ø)
X      = Ø.1ØØØØØØØØØE Ø1
( )    =-Ø.2ØØØØØØØØØE Ø1
(51)
K      =      4
(5Ø)
K      =      16
(5Ø)
X      = Ø.3ØØØØØØØØØE Ø1
( )    = Ø.ØØØØØØØØØØE ØØ
(53)
( )    =      1
X= Ø.1Ø3ØØØØØØØE Ø3
(62)
X      = Ø.1Ø2ØØØØØØØE Ø3
Y      = Ø.ØØØØØØØØØØE
```

Item Trace

Item trace allows the values of the variables to be listed in the order of execution.In the above example, TRACE Y,A illustrate how the values of Y and A are listed in the TRACE listing.

Item trace is in the form:

```
TRACE V1, V2,...Vn
```

where each V is a variable name or array name. When a program containing such a TRACE statement is compiled, coding is inserted into the object program following every statement where one of the specified variables is redefined, until another TRACE statement that specifies the same variable is encountered. During execution, the TRACE statement causes a printout of the state of modified variable. An item trace statement can be placed anywhere in a program, but trace coding is not inserted until the TRACE statement is compiled. Any number of item trace statements may be included. The following example illustrates the on-off action of item trace:

```
K=1Ø        (No tracing yet)
TRACE K     (Enables tracing of K)
K=15*R      (Result is printed)
TRACE K     (Inhibits tracing of K)
K=ALP       (No printout)
```

TRACE statements take effect in source program physical order, not the logical order of execution.

Area Trace

Area trace allows the values of the variables associated with a statement number to be listed. In the above example, TRACE 62 causes the values of X and Y to be listed starting with the TRACE statement and ending with statement 62.

Area trace is specified by the form:

```
TRACE n
```

where n is any statement number that follows the TRACE statement.

Coding is also inserted after each statement number to cause a printout that enables the programmer to follow the sequence in which statements are executed.

An area trace statement should not be placed within the range of another area trace statement, unless both statements refer to the same statement number.

LISTING CONTROL STATEMENTS

These statements enable the programmer to choose the amount of detail to be present in the listing file for different sections of a program. The statements may appear anywhere in the source program, but they only affect the listing of subsequent source statements. These statements override the A register settings made prior to compilation.

NOLIST - No source listing, No symbolic listing

LIST - Source listing, No symbolic listing

FULL LIST - Source listing, symbolic listing

Inserting Files

The INSERT statement accepts a tree file name specifier which allows the insert of a file located in a different file directory or the same file directory.

Format: $INSERT <treename>

The $INSERT statement must start in column 1 followed by the tree name of the file.

Usage: A file containing COMMON specifications for an executable program, insert statement in each program unit rather than repeating COMMON specification.

SECTION 6

I-O AND FORMAT CONTROL

GENERAL PRINCIPLES

Input/Output and format control statements are used to transfer and control the flow of data between internal storage and an input/output device such as a user terminal, printer, punch, magnetic tape unit or disk storage unit.

Input/Output in FORTRAN IV is accomplished mainly by three types of statements: READ for input, WRITE for output, and FORMAT for input or output format specifications. In addition, device control statements are provided for use with sequential access devices such as magnetic tape transports and ENCODE/DECODE statements convert to and from ASCII data.

The Input/Output statements described in this section are:

READ/WRITE STATEMENTS

- READ
- WRITE
- FORMAT
- PRINT

DEVICE CONTROL STATEMENTS

- REWIND
- BACKSPACE
- ENDFILE

ENCODE/DECODE STATEMENTS

- ENCODE
- DECODE

READ AND WRITE STATEMENTS

Input/Output statements in FORTRAN perform data transfers between storage locations defined in a FORTRAN program and records which are external to the program. On input, a READ statement transfers data from an external device (unit) to storage locations. On output, a WRITE statement transfers data from diverse storage locations to an external device (unit). An I/O list is used to specify which

storage locations are used.

The READ and WRITE statements are identical in format; READ is used for data input, and WRITE is used for data output.

The format is:

```
READ (u,f) list

      or

WRITE (u,f) list
```

where ´u´ is the unit number of the I/O device, ´f´ is the number of a format statement included in the program being compiled, and ´list´ is a list of the variables to supply or receive the data.

List

An I/O list specifies which storage locations are used. An I/O list can contain variable names, array elements, array names or a form called an implied DO.

Unit Number ´u´

The unit number is an integer constant or variable between 1 and 28 that is used by the I/O Control System (IOCS) to refer to a logical I/O device. The actual device used depends on the current setting of the IOCS Logical Unit Table (LUTBL). Default logical device assignments are listed in Table 6-1.

NOTE: Only user terminal, paper tape and Funits 1-16 are supported.

Reading Data Into Arrays

The following example illustrates how data is read into arrays. See Section 3 for storage arrangement and Appendix D for illustrative program examples.

Example:

Write a single statement to read 200 numbers from a paper tape, placing the first 100 numbers in the X array and the next 100 numbers read in the Y array. The paper tape reader will be referenced by the number "2".

Solution:

```
READ(2,1)(X(I),I = 1,100),(Y(I),I = 1,100)
```

Two basic forms of the READ and WRITE statements are:

Form	Purpose
WRITE (u,f) list	formatted WRITE
WRITE (u) list	unformatted WRITE
READ (u,f) list	formatted READ
READ (u) list	unformatted READ

Table 6-1. Logical Device and their IOCS Numbers

FORTRAN Number (Unit No.)	Device
1	user terminal
2	paper tape reader or punch
3	MPC card reader
4	serial line printer
5	Funit 1
6	Funit 2
7	Funit 3
8	Funit 4
9	Funit 5
10	Funit 6
11	Funit 7
12	Funit 8
13	Funit 9
14	Funit 10
15	Funit 11
16	Funit 12
17	Funit 13
18	Funit 14
19	Funit 15
20	Funit 16
21	9-track magnetic tape unit 0
22	9-track magnetic tape unit 1
23	9-track magnetic tape unit 2
24	9-track magnetic tape unit 3
25	7-track magnetic tape unit 0
26	7-track magnetic tape unit 1
27	7-track magnetic tape unit 2
28	7-track magnetic tape unit 3

Reading a Variable List

A variable list consists of one or more variables, subscripted variables, or array names, separated by commas, as in:

```
READ (1,10) A,B,X(B), ARRAY
```

Variables may be of any mode provided each item is matched by an appropriate format specification in the format statement.

The order of items in the list is significant. On output, (WRITE), each item is delivered to the output device in the specified order. On input (READ), the first data item in the input record is entered into the first item on the list, and so on.

In the list of a READ statement, an integer variable used as a subscript may appear as a variable in the same list to the left of the subscripted variable. (See B and X(B) in the above example.)

Example:

Write a statement to read ten numbers and place them in the variables A, B, C, D, E, F, G, H, O, and P, respectively. Use associated statement number 1 in the READ statement.

Solution:

```
READ (1,1)A,B,C,D,E,F,G,H,O,P
```

Implied DO-Loops

This notation provides a concise way to specify sequential input or output of array elements at the same level of subscripting. A starting and ending value for a subscript is specified in much the same way as in a DO loop. Consecutive items of the array at that subscript level are processed as the subscript variable is stepped through its range of values. For example, in:

```
READ (1,50) J, (Y(I),X(I,J),I=1,10)
```

The expression I=1,10 specifies the starting and ending values for the subscript variable, I. The starting and ending values may be integer constants or variables. An increment of +1 is assumed. This statement reads items in the following order:

```
J, Y(1),X(1,J),Y(2),X(2,J), . . . . . .Y(10),X(10,J)
```

The name 'implied do loop' comes from the resemblance to DO statements. Note that the implied do loop statement and all items containing the controlled variable as subscript are nested within

the same outer parentheses. Two or more implied DO loops may appear in a single statement, if they are properly grouped in parentheses:

```
WRITE (1,25) ((ARGH(I,J), I=1,3), J=1,10)
```

This statement outputs items in the following order:

```
ARGH(1,1),ARGH(2,1),ARGH(3,1),ARGH(1,2) . . .etc.
```

Items in the list are processed from left to right, with indexing taking place at each open parenthesis (excluding the parentheses containing subscripts).

While processing arrays, it is possible to select row-column or column-row order by incrementing the subscripts in the proper order.

Processing Entire Arrays

To read or write an entire array, only the array name need be specified; the subscripts can be omitted. For example, to process a matrix of m * n elements named ARRAY, it is only necessary to include the name ARRAY in the input/output list. This causes a transfer of the entire array in its natural order, which is the order that would be achieved by the following list entry: (The innermost entry varies most rapidly.) See example X1, line 0001, in Appendix D of this manual.

```
((ARRAY(I,J),I=1,m),J=1,n)
```

Positioning Data in Formatted Records

READ and WRITE statements specify the input or output device to be used, select a FORMAT statement, and name the variables or arrays containing, or to contain, the transferred data. The variables are named in an input or output list. The specified FORMAT statement must provide a format descriptor for each of the items in the input or output list, in the same order. For example, in:

```
        WRITE (1, 10) ARGH, TEMP, DATO1

10      FORMAT (I5, 3X, F8.5, 20X, E8.2)
```

the WRITE statement sets up the transfer of an output record on unit 1, (typically the Teletype) according to format statement 10. The items in the output list are the variables ARGH, TEMP, and DATO1.

Format statement 10 contains a format descriptor for each of the variables, and in addition provides some vertical and horizontal spacing control as follows:

Assume that,

a. the I/O device is at the beginning of the next record. For a Teletype, that is equivalent to being at column 1 of a new line.

b. the first descriptor in the list, I5, applies to the first variable, ARGH, in the output list of the WRITE statement; I5 treats ARGH as an integer and allows it a field of 5 spaces in the output record. In this case, the field occupies Teletype columns 1 through 5.

c. the 3X descriptor inserts three spaces in columns 6 through 8.

d. the F8.5 descriptor interprets the variable TEMP as a real (single precision floating point) datum and allows it a field 8 characters wide in columns 9-16, with 5 digits to the right of the decimal point.

e. 20 more spaces are inserted by the 20X specification. These occupy columns 17 through 36.

f. DAT01 (also assumed to be a real number) is output in columns 37-44. The E8.2 specification arranges for DAT01 to be output in scaled format (using a decimal exponent), in a field 8 spaces wide and with 2 digits to the right of the decimal point.

The resulting Teletype output line might look like:

```
Column  0         1         2         3         4         5 .
        12345678901234567890123456789012345678901234567890

        32767   12.34567                    1.23E 03
```

For an illustrative program example, see examples X7 and X12 in Appendix D.

The same principles apply for input. To read a line of data from the user terminal, for example, the following READ statement might be used:

```
READ (1,10) ARGH, TEMP, DAT01
```

The same format statement could be used, with the following effect:

Assume that,

a. the input is at the beginning of the next record.

b. the I5 descriptor causes the next five characters to be read, converted to an integer, and stored as variable ARGH (the corresponding item on the READ statement input list).

c. the 3X descriptor skips the next three characters on the input

file.

d. the F8.5 descriptor causes the next 8 characters to be read, converted from mixed-number notation into a real quantity, and stored as variable TEMP.

e. the 2ØX descriptor skips 2Ø more characters on the input file.

f. the E8.2 descriptor causes the next 8 characters to be read, converted from scaled notation to a real quantity, and stored as variable DATØ1.

FORMAT statements may also include literal text strings to be output as comments or messages. For example, the message TEST could be output by the following statements:

```
        WRITE (1,2Ø)

2Ø      FORMAT (4HTEST)
```

Note that the WRITE statement does not require an output list, since the only item output by the FORMAT statement is the Hollerith string "TEST".

FORMATTED RECORDS

Summary

Format statements do the translation between the external form of data and the way it is stored internally within the processor. FORMAT statements also provide for vertical and horizontal spacing control. Internally, data is stored in one of six formats - integer, real, double precision, complex, logical, or ASCII. These are described in Section 4. The external form depends on the peripheral device and the type of record.

Format statements have the general form:

```
SN  FORMAT (dF1 dF2 dF3....Fn)
```

where SN is a mandatory statement number, each F is a format field description and each d is a delimiter. The first d may be null. The closing parenthesis of the format statement selects the next record.

Delimiters will be discussed next followed by a discussion of the field descriptors.

Delimiters (Slash and Comma)

The delimiters are the slash (/), meaning proceed to the next

record, or the comma, meaning remain within the current record. Two or more slashes may appear in a row to skip several records. A slash at the beginning of the specification can be used for additional vertical spacing.

Examples:

```
            WRITE (1,10) I,J,K

      10    FORMAT (I6/I6/I6)
```

could print:

```
            0         1         2
     Column 12345678901234567890123456789012345. . .
                 1
             32767
            -32767
```

The affect of a statement such as:

```
      FORMAT (I6///I6)
```

depends on whether it is being used for input or output.

During output, two blank records are written; during input, two records are skipped. For output, the statement writes a 5-column integer in the current record, writes two blank records, and writes another 5-column integer at the beginning of the next record. For input, the statement reads a 5-column integer from the current record, skips two records, and reads another 5-column integer from the next record.

Record Length Options

See Section 8 for record length option details.

Format Field Descriptors

Execution of a formatted READ or formatted WRITE statement initiates format control. Each action of format control depends upon format field descriptors. These are of the forms:

```
      srFw.d

      srEw.d

      srGw.d

      srDw.d
```

```
B'<character string>'

rIw

rLw

rAw

nHhhhhhh...
nX
Tn
```

where:

(1) The letters F,E,G,D,I,L,A,H,X, and T indicate the manner of conversion and editing between the internal and external representations and are called the conversion modes.

(2) w and n are nonzero integer constants representing the width of the field in the external character string.

(3) d is an integer constant representing the number of digits in the fractional part of the external character string (except for G conversion code).

(4) r, the repeat count, is an optional nonzero integer constant indicating the number of times to repeat the succeeding basic field descriptor.

(5) s is optional and represents a scale factor designator.

(6) Each h is one of the characters capable of representation by the processor.

For all descriptors, the field width must be specified. For descriptors of the form w.d, the d must be specified even if it is zero. Further, w must be greater than or equal to d.

Format Field Descriptor Summary

Table 6-2 summarizes the output field descriptor and Table 6-3 summarizes the input field descriptor.

Table 6-2. Summary of Output Field Descriptors

FIELD DESCRIPTOR	CONVERSION	***** INTERNAL	EXAMPLES STATEMENT	****** OUTPUT
nIw	INTEGERS NUMERALS Converts internally stored integers to a group of numerals (Ø to 32767). Without decimal point . only negative signs are output.	+12345 -12345	I6 I7	b12345 b-12345
nX	n spaces are written into current record			
Tn	Tabulation settings (note: For output, the 1 in 1976 starts in Column 1Ø)	1976	T1Ø,I4	1976
nAw	INTEGER,REAL or DOUBLE PRECISION variables ASCII	XY	A2	XY
Lw	LOGIC-IF NONZERO-TRUE -IF ZERO-FALSE	+1 Ø	L1 L5	T bbbbF
nHstring	Outputs headings, messages, etc, in a ´string´ of ASCII characters	String	3ØHTEXTb-STRING	TEXTb-STRING
rFw.d	REAL or DOUBLE PRECISION MIXED- The Number of characters specified by w is converted from real or double precision format to a mixed number without an exponent. Where n represents no. of times to be repeated and d represents the number of characters to the right of the decimal point.	1Ø digits 5 places	F1Ø.5	1234.56789
rEw.d	REAL or DOUBLE PRECISION SCALED external no.= internal No. X 1Ø, where the power of 1Ø is the	+123.456 (scale factor)	E1Ø.4	Ø.1235EbØ3

	No. X 1Ø, where the power of 1Ø is the scale factor.			
srGw.d	FIELD WIDTH COMPAR-ISON - The magnitude of the input number is compared with output field width and either an E or F conversion is made according to available space.	.123456 X1Ø(1)	G11.6	1.23456-bbb
srDw.d	DOUBLE PRECISION or COMPLEX SCALED- The number of characters specified by w is converted from internal double precision or complex number to a scaled number.	+123.456-	D15.9	Ø.1234-56789DØ3

Table 6-3. Summary of Input Field Descriptors

FIELD DESCRIPTOR	CONVERSION	******* EXTERNAL	EXAMPLES********** FORMAT	INPUT
nIw	Numerical Integers (Ø to 32767). Number assumed positive when no sign given. The number is truncated accordingly.	bb123	I5	+123
nX	Skip n columns			
Lw	LOGIC-T set to +1 -F set to Ø	T F	L1	+1 Ø
srFw.d	REAL or DBL PREC MIXED. The number of characters specified by w is converted from REAL or DOUBLE PRECI-SION format to a mixed number without an exponent. N is number of characters and d is number of characters after decimal point and w is field width.	123.456789	F1Ø.6	123.456789
srEw.d	REAL or DBL PREC SCALED. The number of characters specified by w is converted to scaled and n.	Ø.12345EØ3	E1Ø.4	+123.456
nAw	REAL, INTEGER or DBL PREC variables to ASCII.	XY	A2	XY

Integer-I Field Descriptor

The I field descriptor is used to process numerical quantities that are represented internally as integers.

The format is:

nIw

where n specifies the number of times the descriptor is to be repeated, and w specifies the width of field (number of character positions to be read or written).

I Output

I format descriptors convert internally stored integers to a group of numerals ranging from Ø to 32767, without a decimal point. Only negative signs are output.

Each number is right justified within the specified field width. Thus, if the specified field is wider than the number of digits to be output, the number is effectively spaced away from the preceding item. This feature can be used to space items on a line so that successive lines form vertical columns of numbers, as illustrated in preceding examples. (See the X descriptor for another method of inserting horizontal spaces.) A + or - sign or blank uses one character position.

If specified field width is less than the number of digits in the number (including the sign, if negative), the number is truncated. To show that truncation has occurred, a positive number is preceded by a dollar sign ($) and a negative number is preceded by an equals sign (=). Examples:

Internal No. (Integer)	I Conversion Descriptor	Resulting Output
+12345	I6	b12345
	I5	12345
	I4	$123
-12345	I7	b-12345
	I6	-12345
	I5	=1234
Ø	I1	Ø
	I5	bbbbØ

NOTE

In these and subsequent examples, the letter 'b' represents blanks (space characters).

See example X1 in Appendix D for program example.

I Input

I format descriptors convert numerical fields in a data record into internally stored integers. The external representation may range from Ø to 32767. If no sign is present, the number is assumed to be positive, a plus sign is interpreted as a blank. A + or - sign or blank uses one character position. Numbers align from right to left. There must be no decimal point. Spaces between numerals or to the right of the number are ignored. If the specified field width has fewer positions than the number, the number is truncated accordingly. Examples:

Characters in Input Record	I Format Descriptor	Converted Integer
bb123	I5	+bb123
b123b		+bb123
123bb		+bb123
12345		+12345
123456		+12345
b+123		+b123
+b123		+b123
b1b23		+b1b23
b-123		-b123
-123b		-b123
-1234		-1234
-12345		-1234

NOTE

Space characters (blanks) do not count as zeros.

Spaces-X Field Descriptor

The X field specification provides another way to insert spaces between entries on a single line.

The format is:

    nX

where ´n´ is an integer constant that specifies number of spaces:

```
          5X   I3  I5  I5     I5              20X      I5

Column    1234567890123456789012345678901234567890123456789Ø1...

                 1  37   99   3278                          24
```

```
          1Ø    FORMAT (5X,I3 3I5,2ØX,I5)
```

On input, n columns are skipped; on output, n spaces are written into the current record.

A negative value for 'n' is permitted. During output, this has the effect of backspacing and overprinting. For example, a real variable with the value 123.ØØØ would be printed as follows:

Descriptor	Characters Output
F6.Ø	123.
F6.Ø,-1X,' '	123

(The F6.Ø format descriptor and ' ' Hollerith string are described later.)

During input, a negative value for 'n' can cause the program to read part of an input record twice. For example, in:

```
     READ (1,1Ø) I,J

  1Ø FORMAT (I5,-5X,5A1)
```

Five characters entered at the Teletype keyboard are both converted to an integer value in variable I and stored as a 5-character ASCII string in integer array J.

(TAB)-T Field Descriptor

The T field descriptor operates like a tabulation (tab) control. The form is:

```
Tn
```

where 'n' is an integer constant that specifies the column where the next format descriptor will take effect.

Example:

```
         0         1         2         3         4         5

Column   12345678901234567890123456789012345678901234567890 12. . .
                   12345         32767                   123



          20 FORMAT (T10,I5, T25,I5, T45,I3)
```

Real, Double Precision or Complex Field Descriptors

Numerical data stored internally in real or double precision format is processed by the E, F, or G field descriptors. Data stored internally in real, double precision or complex format is processed by the D field descriptor. Field specifications using these descriptors have the following form:

nKw.d

where:

1. The letter n is a positive integer representing the number of times the format descriptor is to be repeated. If n is 1, it may be omitted.

2. The letter K specifies the type of conversion to be used: D, E, F, or G.

3. The letter w specifies the width of the field (number of characters).

4. The letter d represents the number of characters to the right of the decimal point.

NOTE

Scale factors will be described in subsequent paragraphs.

Type F conversion is used when a number stored internally in real or double precision format is represented externally as a mixed number

(has a decimal point but no exponent), as in: 123.456, +3., Ø.246, -99.2.

Example:

Write a FORMAT statement to control the reading of three real numbers, according to the following specification:

first number: 1Ø digits, 5 decimal places
second number: 8 digits, 3 decimal places
third number: 12 digits, no decimal places

Solution:

```
1  FORMAT(F1Ø.5,F8.3,F12.Ø)
```

Type E conversion is used when an internal real or double precision number is externally represented in scaled number format, i.e., as a decimal fraction multiplied by a power of 1Ø, as in:

Ø.12345E12

Ø.13524E-3

Type G conversion compares the magnitude of the internal number with the external field width, and performs either a type E or type F conversion according to the space available.

Complex numbers are stored internally as two consecutive real numbers and consequently are handled by two consecutive field descriptors of type F, E, or G.

Type D conversion is used for numbers stored internally in double precision or complex format. The external notation is the same as the scaled format of E output, but uses the letter D to identify the decimal exponent:

Ø.123456789D12

Ø.123456789D-3

Because the output formats of these types differ, rules and examples for each type of output statement will be presented. These types have similar input characteristics, described following the different output forms.

Mixed Number-F Output

A type F field descriptor causes a real or double precision number to be output as a mixed number, with the decimal point and no decimal exponent. Values less than 1 are preceded by a zero (Ø.123, etc.). The sign is output only if it is negative.

If the field width, w, is greater than the number of characters to be output, the number is right justified in the field.

If the field width is smaller than required by the number (including the decimal point and the negative sign, if present), the number is truncated. As in the case of integers, a truncated positive number is preceded by a dollar sign ($), and a truncated negative number is preceded by an equals sign (=). Thus:

TRUNCATION SIGN	REASON FOR TRUNCATION
S	Field width in format specification smaller than positive number inputted.
=	Field width in format specification smaller than negative number inputted.

If the number of decimal places is wider than necessary, the field is filled with zeroes to the right of the number. If the number of decimal places is smaller than necessary, the number is rounded up (if the field width permits).

Examples:

Internal No. (Real or DP)	F Conversion Specification	Resulting Output
+123.456	F8.4	123.4560
	F8.3	b123.456
	F7.4	$123.45
	F7.3	123.456
	F7.2	b123.46
	F6.3	$123.4
	F6.2	123.45
-123.456	F9.4	-123.4560
	F9.3	b-123.456
	F8.4	=123.456
	F8.3	-123.456
	F8.2	b-123.46
	F7.3	=123.45
	F7.2	-123.46
+.123456	F8.6	0.123456
	F7.6	$0.1234
	F7.3	bb0.123
0	F6.2	bb0.00
	F4.2	0.00

F4.Ø bbØ.

Example:

Using the statement READ(1,1)A,B,C,D,E, construct a suitable FORMAT to specify the location of five real numbers, 14 digits per number with 3 decimal places each.

Solution:

```
1  FORMAT(5F14.3)
```

Scaled Number-E Output

A type E field descriptor causes a real or double precision number to be output as a scaled number, i.e., as a decimal fraction beginning with Ø, a decimal point, and up to seven significant digits, followed by the letter E, a space, and a decimal exponent (power of 1Ø) of one or more digits. Examples:

ØO.1234567EbØ2

Ø.234Eb99

Ø.ØØØ1Eb12

Field width and the number of decimal places are specified in the same way as type F descriptors. For type E, however, it is also essential to allow room for the leading Ø, the decimal point, and the four characters of the exponent. Truncated numbers are preceded by $ or =, as in type F. Examples:

Internal No. (Real or DP)	E Conversion Specification	Resulting Output
+123.456	E13.6	bØ.123456EbØ3
	E12.6	Ø.123456EbØ3
	E11.4	bØ.1235EbØ3
	E1Ø.4	Ø.1235EbØ3
	E9.4	$Ø.1235Eb
	E8.4	$Ø.1235E
	E7.4	$Ø.1235
	E2.4	$Ø
	E1.4	$
-123.456	E14.6	b-Ø.123456EbØ3
	E13.6	-Ø.123456EbØ3
	E12.4	b-Ø.1235EbØ3
	E11.4	-Ø.1235EbØ3
	E1Ø.4	=Ø.1235EbØ
	E9.4	=Ø.1235Eb

	E8.4	=0.1235E
	E7.4	=0.1235
	E3.0	=0.
0	E9.4	$0.0000Eb
	E6.0	0.Eb00
	E5.0	$0.Eb

NOTE: For illustrative program example, see X12 in Appendix D.

Mixed Number or Scaled Number-G Output

A type G field descriptor causes a real or double precision number to be output in the same way as either an F or an E field descriptor, depending on the magnitude of the number and the ′d′ part of the Gw.d field descriptor.

Table 6-4 shows how the descriptor is interpreted. If the magnitude of the internal number lies between 0.1 and 1, the G descriptor acts as an effective F conversion, in which w is reduced by d and 4 spaces are added at the right. Values between 1 and 10 are converted in the same manner but d is reduced by 1. For other ratios of magnitude and field d specification, see Table 6-4. Note the values outside a particular range are output by an equivalent E conversion. Examples:

Internal Data Magnitude (Real or DP)	Actual Conversion Specification	Equivalent Conversion	Examples of Output
0.123456x10(0) (1=<S=<10)	G12.6	F8.6,4X	0.123456bbbb
	G11.6	F7.6,4X	$0.123456bb
0.123456x10(1) (.1=<S=<1)	G11.6	F7.5,4X	1.23456bbbb
	G10.6	F6.5,4X	$1.23456bb
0.123456x10(5) (10**(S-2)= <S=<10**(S-1))	G11.6	F7.1,4X	12345.6bbbb
	G10.6	F6.1,4X	$1.23456bb
0.123456x10(6) (10**(S-1)= <S=<10**(S))	G11.6	F7.0,4X	123456.bbbb
	G10.6	F6.0,4X	$12345.6bb
0.123456c10(7)	G12.6	E12.6	0.123456Eb07

G11.6	E11.6	$0.123456Eb

Decimal Exponent-D Output

A type D field descriptor converts an internal double precision or complex number to a scaled number, as in E conversions. However, the letter D identifies the decimal exponent to show that the internal value is double precision. Type D values may contain up to 14.5 significant figures. Type D is identical to type E with regard to truncation. Examples:

Internal No. (Double Precision)	D Conversion Specification	Resulting Output
+123.456789	D15.9	Ø.123456789DbØ3
	D12.6	Ø.123457DbØ3
	D8.4	$Ø.1235D
-123.456789	D16.9	-Ø.123456789DbØ3
	D13.6	-Ø.123457DbØ3
	D8.4	=Ø.1235D

D,E,F, and G Input

External numbers to be input under control of these field descriptors can be expressed in integer, mixed, or scaled notation. The format is quite flexible; blanks are ignored; blanks contained within number strings, or trailing, are ignored; a decimal point and decimal exponent are optional. (See Figure 6-1). If a decimal point is present, it overrides the positional decimal point set by the format specification. The implied decimal point is assumed to be placed to the left of the first D places from the right (i.e., count from right to left.

All external numbers are converted to the internal double-precision floating point format, but if the descriptor is type E, F, or G, the result is truncated to a single-precision real value.

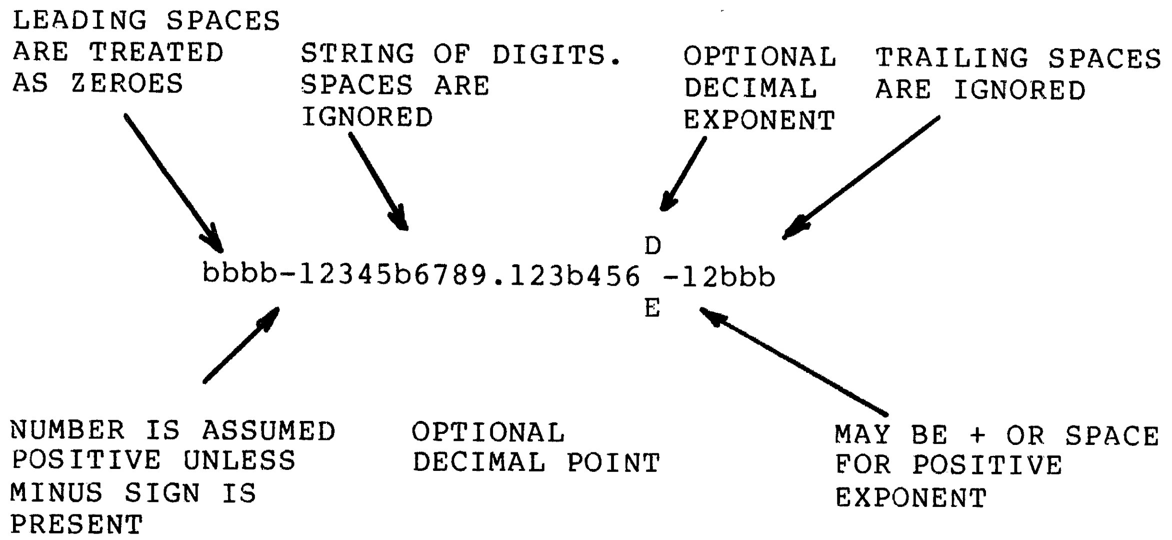
If the input string contains any format errors, range errors, or illegal characters, an error flag is set and a message is printed. (See Appendix A.) The flag can be checked and reset by the OVERFL function (See Section 7). The result of such an input is undefined.

Examples:

Input Characters	Input Format Specification	Resulting Internal Number
bbbbbbb	D,E,F or G7.3	+0.0
bbbbbb1		+0.001
bbbbb1.		+1.0
1.23bbb		+1.23
bb1bbbb		+0.001
bb1b23b		+0.123
bb1023b		+10.23
1234567		+1234.567
123456		+1234.567
b123456		+123.456
12345678		+1234.567
1.234E3		+1234.000
1.234D3		+1234.000
1.23D3b		+1.23X10(3)
1.23E-3		+0.123X10(-2)
1234E-3		+01234X10(-2)
12345-3		+0.12345X10(-1)
b123456	E7.3	+123.456
	F7.2	+1234.56
	G7.1	+12345.6

Table 6-4. Interpretation of Gw.d Descriptors

Internal Data Magnitude	Effective Conversion
1 → 10	F(w-4).(d-1),4X
.1 → 1	F(w-4).d,4X
10(d-2) → 10(d-1)	F(w-4).1,4X
10(d-1) → 10(d)	F(w-4).0,4X
Other	Ew.d



NOTE:

D or I is not optional if the exponent is specified.

Figure 6-1. Format of External Input to Type D,E,F, or G Field Descriptors

F,E,G, and D Scale Factor Designator

A scale factor designator for use with the F,E,G, and D descriptors causes a multiplication by a power of 10. The form is:

nP

where n, the scale factor, is an integer constant with an optional minus sign.

Once a scale factor has been specified, it applies to all subsequent F,E,G, and D field descriptors, until another scale factor is encountered. If n=0, an existing scale factor is removed. The scale factor has no effect on type I,A,H,X, or L descriptors.

E and D Output Scale Factor

Before output conversion, the fractional part of the internal number is multiplied by 10**n and the exponent is decreased by n. Examples:

Internal No.	Format Descriptor	Resulting Output
+123.456	E12.6	0.123456Eb03
	2PE12.6	12.34560Eb01
	-2PE12.6	0.001235Eb05
	-2PE14.6	bb0.001235Eb05
	7PE12.6	1234560.E-04
+123.456789	D15.9	0.123456789Db03
	7PD15.9	1234567.890D-04

F Output Scale Factor

The internal number is multiplied by 10**n, as in:

Internal No.	Format Descriptor	Resulting Output
+123.456	F7.3	123.456
	3PF7.0	123456.
	-2PF7.3	1.235

G Output Scale Factor

The scale factor has an effect only if the internal number is in a range that uses effective E conversion for output. In this case, the effect of the scale factor is the same as in the corresponding E conversion:

Internal No.	Format Descriptor	Resulting Internal No.
Ø.123456X1Ø(7)	G12.6	Ø.123456EbØ7
	3PG12.6	123.456ØEbØ4

D,E,F,G Input Scale Factor

The internal value is formed by dividing the external number by 1Ø**n. However, if the external number contains a D or E exponent, the scale factor has no effect.

Examples:

Input Characters	Input Format Descriptor	Resulting Internal Number
b123456	F7.3	+123.456
	3PF7.3	+.123456
	-3PF7.3	+123456.

H (Hollerith) Field Descriptor

The type H descriptor is used primarily to output headings, messages, and other literal text strings. This descriptor also provides vertical spacing control for the Teletype and line printer. The standard form is:

nHstring

where ′string′ is a series of ASCII characters and n is a non-zero integer equal to the number of characters following the H. If n is omitted it is assumed to be 1. This compiler also accepts Hollerith strings enclosed by single quotes, as in:

′string′

The H symbol and character count are not required.

H Output

The character string following H is written in the output record:

H Descriptor	Characters Output
11HTEXTbSTRING ´TEXTbSTRING´	TEXTbSTRING
15HbbTEXTbSTRINGbb ´bbTEXTbSTRINGbb´	bbTEXTbSTRINGbb

The H descriptor does not require a corresponding item in the output list of a WRITE statement:

```
      WRITE (1,10) A, AARGH, MAT(1)

10    FORMAT (5HLABEL,I13,I10,F15.5)
```

Other types of descriptor may follow a Hollerith string without an intervening comma or slash. The statement:

```
50    FORMAT (´INCOME=b$b´F8.2)
```

could be used to print a line:

```
      INCOME= $ 12345.67
```

A (Alphanumeric) Field Descriptor

Type A conversion transfers ASCII character codes between integer, real, or double-precision variables or arrays and external devices.

The form is:

nAw

where n is the repeat count and w is the number of characters to be transferred per variable or array element. ASCII characters are stored two per integer variable, four per real variable, or eight per double precision variable.

ASCII-A Output

Each type A descriptor provides for output of the ASCII character content of one variable in the output list of the accompanying WRITE statement. If n is greater than the number of characters in the variable, spaces precede the content of the variable.

Examples:

Internal Data	Descriptor	Resulting Output

Xb	A1	X
	A2	Xb
XY	A1	X
	A2	XY
	A3	bXY

ASCII-A Input

Each Type A descriptor provides for input of one or two ASCII characters to one variable in the input list of the accompanying READ statement.

If more characters are input than the variable can hold, only the last n characters are accepted.

Examples:

Input Data	Descriptor	Resulting Internal Data
X	A1	Xb
	A2	Xb
	A3	bb
XY	A1	Xb
	A2	XY
	A3	Yb
bXY	A3	XY
bXYb	A3	Yb
bXYb	A4	Yb
XYbb	A4	bb

Character Array String Transfer

If a repeat count is specified, a type A descriptor can transfer strings of ASCII characters to or from an array. For example, the following statements could be used to enter a line of up to 72 characters from the Teletype keyboard:

```
      INTEGER LINE (36)
      .
      .
      READ (1,1Ø) LINE
      .
      .
1Ø    FORMAT (36A2)
```

If fewer than 72 characters are entered, the rest of the array is filled with blanks.

Logical-L Field Descriptor

Logical variables are represented internally by integer variables and externally by the letters T and F:

Internal Value	External Representation
.FALSE.	F
.TRUE.	T

Truth values formed by the logical and relational operators are +1 for .TRUE. and Ø for .FALSE.

Logical-L Output

The internal value is converted to the letter F (if zero) or the letter T (if non-zero) and output right-justified in a field w characters wide. Examples:

Internal Value	Format Descriptor	Resulting Output
.FALSE.	L1	F
.TRUE.	L1	T
	L5	bbbbT
	LØ	(None)

Logical-L Input

A field w characters from the external record is examined. Leading spaces are ignored. The internal variable is set according to the first non-space character:

Character	Effect on Variable
T	Set to +1
F	Set to Ø
Other	Set to Ø and error flag is set

Any other characters in the field are ignored. The flag can be sensed by the OVERFL function described in Section 7.

Repetition of Descriptors

All descriptors except H and X can be assigned a repeat count, n, that causes the descriptor to be used n times in succession:

```
FORMAT (3D1Ø.5)
```

is equivalent to:

```
FORMAT (D1Ø.5,D1Ø.5,D1Ø.5)
```

Using Parentheses

Groups of descriptors including H and X descriptors may also be enclosed in parentheses and assigned in repeat count:

```
FORMAT (2(3D1Ø.5,X3))
```

is equivalent to:

```
FORMAT (3D1Ø.5,X3,3D1Ø.5,X3)
```

Nesting Repeat Groups

Repeat groups may be nested up to two levels deep:

```
FORMAT (3(2(1ØF.7,3X),I2,5X))
```

Example of Formatted Output

The following example illustrates the repetition of format descriptors and the resulting typewriter or line printer output:

```
      I = 5
      J = 6
      K = 7
      L = 8
1     WRITE(1,1Ø6)I,J,K,L,I,J,K,L,I,J,K
2     WRITE(1,1Ø6)I,J,K,L,I,J
1Ø6   FORMAT(/4H ABC/2(3H XY,I4,2(I2),3(I3)/))
3     WRITE(1,1Ø6)I,J,K,L,I,J,K,L,I,J,K,L
4     WRITE(1,1Ø6)I,J,K,L,I,J,K,L,L,I,J,L,I
5     WRITE(1,106)I,J,K,L,I,J,K,L,
```

The following output on a typewriter or line printer would result:

```
      ABC
      XY   5 6 7  8  5  6    Result of statement 1
      XY   7 8 5  6  7

      ABC
      XY   5 6 7  8  5  6    Result of statement 2
      XY

      ABC
      XY   5 6 7  8  5  6    Result of statement 3
      XY   7 8 5  6  7  8

      ABC
      XY   5 6 7  8  5  6    Result of statement 4
      XY   7 8 5  6  7  8

      XY   5

      ABC
      XY   5 6 7  8  5  6    Result of statement 5
      XY   7 8 5  6  7  8

      XY   5 6 7  8  5  6
```

Rescanning Format Lists

If a format list is exhausted before all items on an input/output list are processed, the format list is repeated, starting at the opening parenthesis that matches the last closing parenthesis in the list. (The parentheses around the format list itself are used only if there are *no* other parentheses.) Any repeat count preceding the selected opening parenthesis is effective as usual.

During output, when a rescan of a format list is required, the current record is padded with blanks and a new record is started. During input, if a rescan is required, the rest of the current record is skipped and

the device is advanced to the beginning of the next record.

Entering Format Statements at Run Time

It is possible to enter format statements at run time by using a READ statement to load the format statement into an array. The array can later be referenced in lieu of a FORMAT statement, by the READ or WRITE statement that handles the data. Arrays to be used for this purpose must be assigned as integer type and must be dimensioned to accomodate the format description, at two characters per word. The format description is loaded into the array by a READ statement that references a type A format statement:

```
          DIMENSION FORM (6 ),TEXT(80)
          INTEGER FORM
          READ (1,20)FORM
      20  FORMAT (6A2)
          .
          .
          .
          WRITE(1,FORM)(ARG(I),K(I),I=1,3)
```

These statements provide for an output format specification such as (3(F7.3,I7)) to be entered at run time. Note that the specification must include opening and closing parentheses but not the word FORMAT.

PRINT & PRINTER CONTROL

PRINT Statement

PRINT is an alternate method of specifying information be printed at the user console.

The compiler supplies the logical unit number of 1 (user console). The format is:

```
PRINT f list
```

where 'f' is the number of a format statement included in the program being compiled, and 'list' is the list of variables to be printed at the user console.

Example:

```
PRINT 5,I,J,K
```

is equivalent to:

```
WRITE(1,5) I,J,K
```

Vertical Spacing Control Symbols (Line Printer)

The first character of each ASCII output record controls the number of vertical spaces to be inserted before printing begins on a line printer. The codes are:

First Character	Vertical Spacing
Space	One line
Ø	Two lines
1	Form feed (advance to first line of next page)*

*Note: Effective only on devices with mechanical form feed.

First Character	Vertical Spacing
+	No advance - print over previous line (line printer only)
Other	One line (character is

printed also)

A Ø, 1, or + character is used for vertical spacing only and is not printed.

A straight forward way to control spacing is to start a FORMAT statement for an ASCII record with 1HØ, where c is the desired spacing control character, as in:

```
      WRITE (4,2Ø) TEXT

   2Ø FORMAT (1HØ, 36A2)
```

The 1HØ entry inserts two line feeds before the output line is printed.

END AND ERROR RETURNS

End and Error Returns in READ/WRITE Statements

READ and WRITE statement syntax has been extended to allow the following forms:

```
READ(d,END=a)             READ(d,f,END=a)
READ(d,ERR=b)             READ(d,f,ERR=b)
READ(d,ERR=b,END=a)       READ(d,f,ERR=b,END=a)
READ(d,END=a,ERR=b)       READ(d,f,END=a,ERR=b)
WRITE(d,ERR=b)            WRITE(d,f,ERR=b)
```

where:

d - device specifier

f - format specifier

a - statement number that control is to be transferred to if an end of file condition is detected in the READ

b - statement number that control is to be transferred to if a device error occurs in the READ/WRITE operation

B FORMAT STATEMENT

The B format has the form:

B´<character string>´

No repeat count is allowed associated with format specifier itself, but a B format may be included in a parenthetical repeat group. The length of the character string defines the length of the field in the output record. The character string is a template for the output field and may consist of the following characters:

+ - $, * Z # . CR

The characters are interpreted as follows:

Plus sign (+):

1) A single leading plus sign (fixed sign) will be replaced by a plus sign if the output number is positive; a minus sign if the output number is negative.

2) Multiple leading plus signs indicate a floating sign. As many of the plus signs as are required by the magnitude of the output number will be used for digits of the number. The

one preceding the M.S.D. of the number will contain a sign character as above, the remainder will be replaced with spaces.

3) A trailing plus sign will be replaced by a sign character as described above.

Minus sign (-):

The minus sign behaves the same as a plus sign except that for positive numbers a space is inserted instead of a plus sign.

Dollar sign ($):

1) A leading dollar sign (preceded by at most a single fixed sign) will cause a dollar sign to be placed in the corresponding position in the output field.

2) Multiple leading dollar signs (preceded by at most a single fixed sign) indicate a floating dollar sign. As many of the dollar signs as required by the magnitude of the output number will be used for digits of the number. The dollar sign will be placed to the left of the M.S.D. and the rest will be replaced with spaces.

Asterisk (*):

Multiple asterisks will be used for digits of the output number as required and the remainder will be included in the output field. Asterisks may be preceded by at most a fixed sign and/or a fixed dollar.

Z:

Z is used to indicate a zero suppress digit position. If the corresponding digit in the output number is a leading zero, a space will be placed in the output field. Otherwise, the digit in the number will be used.

#:

#'s are used to indicate non-zero suppress digit positions. The corresponding digit in the output number will be placed in the output field.

Decimal Point (.):

A decimal point indicates the placement of the decimal point in the output number. The decimal point may be followed only by # characters and/or trailing sign.

Comma (,):

Commas may be placed in the field after any leading characters and prior to the decimal point. If a significant digit precedes

a comma, a comma will be placed in the output field; if not, a space will be output unless the comma is contained in an asterisk field in which case an asterisk will be output.

Credit (CR):

The characters CR may be used as the final two characters in the string. If the output number is positive, they will be replaced with spaces; if negative, they will be printed.

Examples:

Number	Format	Output Field
123	B'####'	0123
12345	B'####'	****
0	B'####'	0000
123	B'ZZZZ'	123
1234	B'ZZZZ'	1234
0	B'ZZZZ'	
0	B'ZZZ#'	0
1.035	B'#.##'	1.04
0	B'#.##'	0.00
1234.56	B'ZZZ,ZZZ,ZZ#.##'	1,234.56
123456.78	B'ZZZ,ZZZ,ZZ#.##'	123,456.78
0	B'ZZZ,ZZZ,ZZ#.##'	0.00
2	B'+###'	+002
-2	B'+###'	-002
2	B'-ZZ#'	2
-2	B'-ZZ#'	- 2
234	B'ZZZZZ+'	234+
-234	B'ZZZZZ+'	234-
234	B'ZZZZZ-'	234
-234	B'ZZZZZ-'	234-
12345	B'ZZZ,ZZ#CR'	12,345
-12345	B'ZZZ,ZZ#CR'	12,345CR
123	B'+++,++#.##'	+123.00
-123	B'+++,++#.##'	-123.00
98	B'$ZZZZZZ#'	$ 98
98	B'$$$$$$$#'	$98
156789	B'$***,***,**#.##'	$****156,789.00

UNFORMATTED (BINARY) RECORDS

Memory-image data consisting of 16-bit binary words can be processed

READ and WRITE statements without reference to a FORMAT statement. The READ and WRITE statements are in the form:

```
READ (u) List

WRITE (u) List
```

where 'u' is a logical device number and 'List' is a list of variables or array names containing the data to be transferred.

An unformatted WRITE operation writes all words specified by the list in binary format. If the list elements do not fill a record, the record is padded with zero bits. If the list elements require more than one record, multiple records are written automatically. The last record is padded with zeroes, if necessary.

An unformatted READ operation reads records from the specified device and enters the binary information into the items in the list. Enough records are read to satisfy all the list items. If a record contains more items than are required by the items in the list, the surplus items are ignored. If no list is present, one record is read but ignored, for an effective one-record forward spacing operation.

DEVICE CONTROL STATEMENTS

The REWIND, BACKSPACE, and ENDFILE statements are used for physical positioning of sequential access devices such as magnetic tape transports. DOS disk files are also treated as sequential records.

REWIND Statement

A REWIND statement of the form:

```
REWIND u
```

causes unit u to be positioned at its initial point.

BACKSPACE Statement

A BACKSPACE statement for a magnetic tape unit is of the form:

```
BACKSPACE u
```

If the unit is positioned at its initial point, this statement has no effect. Otherwise, the statement positions unit u so that the preceding record becomes the next record.

ENDFILE Statement

An ENDFILE statement for a magnetic tape unit is of the form:

```
ENDFILE u
```

causes the recording of an endfile record on unit u. The endfile record is a unique record signifying a demarcation of a sequential file. Action is undefined when an endfile record is encountered during execution of a READ statement.

ENCODE/DECODE STATEMENTS

The ENCODE statement converts the elements of the I/O list into ASCII data according to the specified format and store the first c characters of the resultant line buffer into the specified array. The DECODE statement has the opposite effect, converting the c character record in the specified array into the I/O list elements according to the specified format.

Their syntax is:

```
ENCODE (c,f,a)  list

DECODE (c,f,a)  list
```

where:

c - number of ASCII characters to be transferred

f - format specifier

a - array name

list - I/O list (as in READ/WRITE statements)

DECODE Statement ERR=Option

The DECODE statement accepts an ´ERR=sn´ parameter as in the READ/WRITE statements. At run-time, the ERR= branch will be taken if a FORMAT/ DATA mismatch is detected in processing the DECODE operation.

SECTION 7

FUNCTIONS AND SUBPROGRAMS

GENERAL OVERVIEW

Programming efficiency is usually increased if often-used calculations or data processing operations can be coded once, and then referenced at several points in a main program by a brief calling statement. New arguments, which are the only elements that are different each time the operation is performed, are supplied by each calling statement. Programs organized in this way do not need to repeat identical sections of code, and the same building blocks may be used in other programs, or in alternate versions of the same program.

This effect can be obtained within a single program by using GO TO statements which refer to the same statement number. However, code used in this way is not accessible to other programs. An alternate way is to define the operation as a function or subprogram. Either of these can be invoked by a simple calling statement. In addition, subprograms can be compiled separately and placed on a library tape for use with other programs.

Functions are called by specifying a symbolic name followed by a list of the arguments, in parentheses. For example, in the statement Y=SIN (A), Y is set equal to the SIN (trigonometric sine) function of the argument A. The function name may refer to one of the Prime FORTRAN library functions, a user-defined statement function, or a subprogram defined by a FUNCTION statement. As an extension to FORTRAN IV, Prime also provides several intrinsic functions (XOR, AND, LOC, etc.) to aid system programming.

Functions and subprograms discussed so far have in common the fact that they produce a single output, or result. Calculations that produce multiple results must be defined by SUBROUTINE statements and be compiled separately from the main program. A subroutine subprogram is referenced within a main program by a CALL statement which identifies the subroutine by name and provides a list of arguments. For example:

```
CALL GRAPH(X,Y)
```

might involve a subprogram that calculates point plotting data for arrays X and Y.

This section provides functions, subprograms and subroutines that are available for use during compilation time. These include:

PRIMOS SUBROUTINES,

LIBRARY FUNCTIONS,

INTRINSIC FUNCTIONS,

STATEMENT FUNCTIONS,

FUNCTIONS SUBPROGRAMS,

PROTECTED FUNCTIONS AND SUBROUTINES,

BLOCK DATA SUBPROGRAM,

LIBRARY SUBROUTINES,

SENSE LIGHT/SWITCH SUBROUTINES,

LINKING FORTRAN AND ASSEMBLY LANGUAGE PROGRAMS.

PRIMOS SYSTEM SUBROUTINES

PRIMOS subroutines are those which invoke PRIMOS II or PRIMOS III to do the actual work. Each of these subroutines are identified in Appendix E. Each subroutine does a specific job from attaching to a file directory to transferring data from one device to another. These system routines greatly enhance the capability of a user's FORTRAN program.

LIBRARY FUNCTIONS

Supplied with the Prime FORTRAN IV compiler is a collection of library subroutines. These are identified in Appendix G of this manual. Some of these subroutines are required by the compiler itself; they are called during compilation and appended to the main program during loading. The library also includes a collection of mathematical subroutines that can be called as functions by statements within a main program. Subroutines that execute library functions referenced in a main program are also appended during loading.

Using the Library Subroutines

Library function references are of the form:

```
NAME (ARG1, ARG2, . . . .ARGn)
```

where NAME is one of the library function names listed in Prime's Software Library User Guide, and each ARG is one of the arguments to be processed by the function. Functions require at least one argument.

Arguments may be constants, variables, or expressions. Constants and

variables must be of the modes specified in Prime's Software Library User's Guide. Expressions within the argument parentheses are evaluated and the function is performed on the result. The mode of expression must also agree with expected mode. Following are some examples of the use of functions:

```
M=SIN ( + B)

VORT=AMOD(A,B*72.931+V)

PRIPE=AMAX1(A,B,C,D 5.,ARGH/37.31E3,PI**3)
```

Argument lists may contain references to other library functions (or statement functions or FUNCTION subprograms, defined later). Examples:

```
R=SIN(B+AMOD(D,D))

P=ABS(R*FUNC(1.2E3+S))
```

INTRINSIC FUNCTIONS

This group of functions is an extension feature to increase the efficiency of Prime FORTRAN IV in logical processing and system-level programming on 16-bit integers. Also in this group is the LOC function which returns absolute memory addresses.

Many of these functions are executed by two or three instructions of in-line assembly language code; others result in calls to library subroutines.

All of these functions except LOC are intended to process integer arguments and form integer results. (If any argument is non-integer, an error message will result.)

XOR (Logical Exclusive OR)

This function performs a logical exclusive OR of any number of arguments. The result is integer mode. Examples:

```
XOR(ARG, I)

XOR(J, K, L, M)

XOR(I)
```

AND (Logical AND)

This function performs a logical AND of any number of arguments. The result is integer mode. Examples:

```
AND(ARG, I)

AND(J, K, L, M)

AND(I)
```

OR (Logical OR)

This function performs a logical OR of two arguments, forming an integer-mode result. Examples:

```
OR(ARG, I)

OR(J, 16)
```

NOT (Logical Negation)

The NOT function generates an integer mode value consisting of the 1's complement of its single argument. Examples:

```
NOT(I)

NOT(ARG)
```

SHFT (Logical Shift)

The logical shift function is capable of fetching an integer variable and performing one or two independent logical shifting operations in either direction. The form is:

```
SHFT (IVAR,I,J)
```

where VAR is the name of an integer variable and I and J are integer constants or variables that represent the number and direction of shifts to be performed.

The function may have one, two, or three arguments. If only the first variable is identified, the result of the function is the variable itself (no function is performed). If I is specified, one shift operation is performed. The sign of I determines the direction (+ is right, - is left) and the absolute value of I determines the number of places. Shifting is logical - i.e., vacated bit positions are filled with zeroes. If J is also specified, a second independent shift operation is performed, according to the sign and magnitude of J.

When three arguments are present, the first argument is fetched, shifted I places, then shifted again by J places. This double shift feature is useful for masking or masking-and-positioning as shown in the following examples:

SHFT(IVAR,11,-11) .... Set the right 11 bits to zero.

SHFT(IVAR,-11,11) .... Set the left 11 bits to zero.

SHFT(IVAR,-4,8) ...... Set the left 4 bits to zero and position to the right byte.

SHFT(IVAR,4,-8) ...... Set right 4 bits to zero and position to the left byte.

If I and/or J is not a constant, a call to the SHFT library subroutine is generated. Otherwise the function is implemented by in-line code.

Examples:

```
SHFT(IVAR, -6)

SHFT(A,3)

SHFT(INT, +3, -10)

SHFT(K(5,27),J)

SHFT(M,4,K)

SHFT(1HX, 8)

SHFT(15, -7)
```

LOC (Location)

The LOC function generates an integer value that represents the computer memory address where the function's argument is located. Only one argument (a constant, a variable r array name, a subscripted array element) is allowed. Examples:

NOTE:

> Allowed only in arguments to functions or subroutines of P400 64V mode. In 64V mode, result of LOC is two word quantity representing segment number, and word number of argument.

```
I = LOC(A)

J = LOC(3)

K = LOC(DUM) + 1

L = LOC (AR(3,115))
```

RS, LS, RT, LT

These functions are special versions of the SHFT function and are provided for compatibility with other FORTRAN compilers. These functions result in an integer mode value and require two arguments, as in:

```
I = RS(IVAR,I)
```

where VAR is the value to be shifted, and I specifies the amount and direction of the shift. The equivalent SHFT for each of the functions is shown in the following table:

Operation	Function	Equivalent SHFT Function
Right Shift	RS(IVAR,I)	SHFT(I,S)
Left Shift	LS(IVAR,I)	SHFT(I, -S)
Right Truncate	RT(IVAR,I)	SHFT(I, S-16, 16-S)
Left Truncate	LT(IVAR,I)	SHFT(K, 16-S, S-16)

Intrinsic Function Library Subroutines

Several of the functions previously described will generate a call to a library subroutine if any arguments are non-constant. In all cases, the compiler assumes the result is an integer value in the A register on exit from the subroutine. The subroutines are:

Library Function	Procedure
OR(A1,A2)	OR 16-bit values A1 and A2
SHFT(A1,A2)	Shift A1 by A2 bits
SHFT(A1,A2,A3)	Double shift A1 and A2 then by A3 (bits)
LT(A1,A2)	Save left A2 bits of A1
RT(A1,A2)	Save right A2 bits of A1
LS(A1,A2)	Shift A1 left by A2 bits
RS(A1,A2)	Shift A1 right by A2 bits

STATEMENT FUNCTIONS

Any calculation that can be expressed in a single statement, and produces a single result, may be assigned a function name and referenced in the same way as a library function. A statement function is defined in the form:

```
NAME (ARG1, ARG2, . . . . ARGn) = Expression
```

where NAME is the symbolic name assigned to the function, and each ARG is a dummy variable that represents one of the arguments. The following rules apply to statement functions:

1. The name may consist of one to six alphanumeric characters, the first of which is alphabetic. It must differ from all other function names and variable names used in the main program.

2. The argument list follows the name and is enclosed in parentheses. There must be at least one argument. Multiple arguments are separated by commas. Each argument must be a single nonsubscripted variable. These arguments are only dummy variables, so their names may be the same as names appearing elsewhere in the program. The dummy variable names do indicate argument mode, however, by implicit or explicit mode typing.

3. During each call of a function, the values supplied as the argument variables must be in the same mode as the arguments were when the function was defined.

4. Implicit mode typing of the result of a function is determined by the first letter of the function name. Functions that begin with I,J,K,L,M, or N product INTEGER results; others produce real results. Regardless of the first letter, the result mode can be set to REAL, INTEGER, DOUBLE PRECISION, COMPLEX or LOGICAL by an appropriate mode specification preceding the statement. (See Section 5.)

5. The expression that defines the function may use library functions, previously defined function statements, or FUNCTION subprograms, but not the function itself. Dummy variables cannot be subscripted.

6. Variables in the expression that are not stated as arguments are treated as parameters - i.e., are assumed to be variables appearing elsewhere in the main program.

7. Statement functions must be defined following specification and DATA statements but before the first executable statements of a program.

The following example shows how a statement function HYP might be defined and used:

```
      HYP(A,B)=SQRT(A**2+B**2)
      .
      .
      A=PI*1.23
      .
      .
50    YDS=HYP(HGT,3.724)
      .
      .
60    MAT=1+ABS(HYP(A,3.724))
```

In statement 50 the HYP function is defined using A and B as dummy variables. In statement 50, the actual variable HGT is substituted for the dummy variable A, and the dummy variable B is equated to 3.724. Statement 60 shows the HYP function nested within a standard library ABS function. Note that an actual variable A, defined elsewhere in the main program, has no relationship to the dummy variable A (except the same mode).

FUNCTION SUBPROGRAMS

Statement functions are limited to a single statement and must be coded within and compiled with a main program. FUNCTION subprograms, on the other hand, can consist of many statements and be coded and compiled separately. This permits them to be used in the same way as library functions.

FUNCTION subprograms must be prepared as separately compiled subprograms that produce a single result, in the following format:

```
Mode FUNCTION 'Name' (Arg1, Arg2, . . .Argn)
.
.
(Any number of FORTRAN statements which
perform the required calculations, using
the supplied arguments as values.)
.
.
'Name' = Final Calculation
.
.
RETURN
```

FUNCTION Statement

The FUNCTION statement, which must be the first statement of a FUNCTION subprogram, assigns the name of the function and identifies the dummy arguments. In the preceding example, 'Name' is a symbolic name assigned to identify the function, and each 'Arg' is a dummy argument.

The function name must conform to the normal rules for all symbolic

names (Section 2) with regard to number of characters, etc. Implicit result mode typing occurs according to the first letter of the name. Implicit mode typing can be overridden by preceding the word FUNCTION with one of the mode specifications, INTEGER, REAL, DOUBLE PRECISION, COMPLEX or LOGICAL. The function name must differ from any variables used in the function subprogram or in any main program which references the function.

There must be at least one dummy argument, in the form of a non-subscripted variable or array name. Array names must be cited in a DIMENSION statement within the subprogram. The arguments may be any of the variable names that appear in executable statements of the function subprogram.

Body of Subprogram

The body of the function subprogram can consist of any legal FORTRAN statements except SUBROUTINE, BLOCK DATA, or other FUNCTION statements. The statements that evaluate the function use constants, variables, and expression in the normal way. The program must produce a single result for a given set of argument values. The subprogram must equate the assigned symbolic function name ´name´ to the result, by using ´name´ on the left side of an assignment statement. It is the function name itself, used as a variable, that returns the result to the main program.

RETURN Statement

The RETURN statement consists of a single word RETURN. It terminates the subprogram and returns control to the main program. The RETURN statement must be the last statement in the subprogram (logically, not physically; that is it must be the last statement to which control passes).

Calling Function Subprograms

FUNCTION subprograms are referenced within main program expressions in the following form:

```
Name (V1, V2, . . . Vn)
```

where ´Name´ is the function name assigned by the FUNCTION statement that begins the subprogram, and each V is a value expression to be substituted for the corresponding dummy argument in the argument list of the FUNCTION statement.

The list of values may contain any legitimate constant, variable (subscripted or not), expression, subprogram name, or name of any array provided the corresponding dummy variable in the subprogram has the same mode. The argument list following the function name in the main

program and the list of dummy variables in the FUNCTION statement must agree in number, order and mode. The subprogram must contain the same DIMENSION statements as the main program. Function names included in argument lists must also appear in an EXTERNAL statement in the main program.

For example, a function subprogram that determines the Ith root of a real number R might start with the following statement:

```
FUNCTION ROOT (I,R)
```

A main program would call this function with a statment such as:

```
ANS=ROOT(9,1.22793E-11)
```

In this example, 9 is substituted for the dummy variable I and the other value is substituted for dummy variable R. The function subprogram calculates the root using these for arguments I and R, and returns control to the main program with the answer in variable ANS.

Values contained in arrays are passed to subprogram functions in the same way. For example, a subprogram that determines the median value of data in a 100-item array might begin with the statement:

```
FUNCTION MED (DAT)
```

where DAT refers to a 100-item array dimensioned within the subprogram. The calling main program might call this function with a statement like:

```
I=J+MED(STD)
```

where STD is a 100-item array dimensioned and assigned values by the main program. STD (in the main program) and DAT (in the subprogram) must be of the same mode.

Examples:

Function subprogram for function AVRG:

```
1    FUNCTION AVRG(ALIST,N)

     DIMENSION ALINT (N)

     SUM = ALIST(1)

     DO 10 I=2,N


10   SUM = SUM + ALIST(I)

     AVRG = SUM/FLOAT(N)
```

```
      RETURN
      END
```

Main program call to AVRG function:

```
      DIMENSION SET(500)
      READ(2,5) (SET(I), I=1,200)
    5 FORMAT(6F12.8)
      TEXT = AVRG(SET,200)
      WRITE(2,10)TEXT
   10 FORMAT(20H1 AVERAGE OF SET IS E14.5)
      STOP
      END
```

SUBROUTINE SUBPROGRAMS

Subroutine subprograms are very similar to FUNCTION subprograms. They are prepared in the form:

```
SUBROUTINE NAME (ARG1, ARG2, . . . ARGn)
.
.
(any number of FORTRAN statements which perform
the required calculations, using the supplied
arguments (if any) as values).
.
.
RETURN
END
```

SUBROUTINE Statement

The SUBROUTINE statement, which must be the first statement of a SUBROUTINE subprogram, assigns the name of the subprogram and identifies the dummy arguments, if any.

The subprogram name must conform to the normal rules for symbolic names with regard to number of characters, but the first letter does not set the data mode of the results. The name must be unique to both the supbrogram and a main program which calls it.

The argument list usually consists of a series of dummy variables which are processed by the subroutine and return arguments to the main program. Each argument may be a variable, array, or function name. If an argument is the name of an array, it must be mentioned in a DIMENSION statement following the SUBROUTINE statement. Arguments that return values to the main program must not be constant or expressive in . call.

A subroutine with no arguments is allowable. Such a subroutine might obtain arguments from, and return results to, common. Or it might be used to output a message or control function to a peripheral device.

Body of Subroutine

The body of the subroutine can consist of any legal FORTRAN statements except SUBROUTINE, BLOCK DATA, or FUNCTION statements. The results of calculations may be stored in variables used by both the subprogram and main program, or they may be placed in common. Variables may be used freely on either the right or left side of the equals sign in assignment statements. Each variable that represents a result must appear on the left side of at least one assignment statement, in order to present the result to the main program.

The subroutine is terminated by a RETURN statement (described previously). The last physical record in a subroutine must be an END statement.

Calling Subroutines (CALL Statement)

SUBROUTINE functions are referenced within main programs by CALL statements, of the form:

```
CALL Name (V1, V2. . . . . Vn)
```

where 'Name' is the symbolic name assigned by the SUBROUTINE statement that begins the subroutine, and each V is a value expression to be substituted for the corresponding dummy argument in the argument list of the SUBROUTINE statement. Each value may be a constant, variable (including array name), subscripted variable, array, expression, or function name. Arguments used by the subroutine and the main program must agree in number, order, and mode; and the main program must contain the same DIMENSION statements as the subroutine. Address constants ($n) can be used to specify statement numbers of alternate returns Examples:

Main Program:

```
 1   DIMENSION X(10,15), Y(15,12), Z(10,12), JOB(3)
     DATA JOB/'MATMPY'/
     READ(2,4) ((X(I,J), J=1,15), I=1,10),
    X          ((X(I,J), J=1,12), I=1,15)
 4   FORMAT(6E12.6)
 5   CALL MATMPY(X,10,15,12,Z)
     DO 13 J=1,12
13   WRITE(4,15)  (Z(I,J), I=1,10
15   FORMAT(2H0 6E17.6)
     CALL EXIT (JOB)
     END
```

Subroutine MATMPY:

```
 1   SUBROUTINE MATMPY(A,N,M,B,L,C)
     DIMENSION A(N,M), B(M,L), C(N,L)
     DO 5 I=1,N
     DO 5 J=1,L
     C(I,J) = 0.0
     DO 5 K=1,M
 5   C(I,J) = C(I,J) + A(I,K)*B(K,J)
     RETURN
     END
```

Subroutine EXIT:

```
     SUBROUTINE EXIT(JOBA)
```

```
      DIMENSION JOBA(3)
      WRITE(1,5) LIST
    5 FORMAT(12H END OF JOB , 3A2, / )
      RETURN
      END
```

PROTECTED FUNCTIONS AND SUBROUTINES

This feature prevents FUNCTION and SUBROUTINE subprograms used in a real time environment from being interrupted before they have completed their calculations. It is only necessary to place the word PROTECTED before the statement that introduces the SUBROUTINE or FUNCTION. Interrupts are disabled when a protected subroutine is entered and enabled when control returns to the main program. Examples:

PROTECT/PROTECTED SUBROUTINE ALPHA (A1, A2)

```
PROTECTED SUBROUTINE BELL
PROTECTED FUNCTION BETA (X,Y,Z)
PROTECTED DOUBLE PRECISION FUNCTION JAM (M)
```

BLOCK DATA SUBPROGRAMS

This type of subprogram labels common areas and then initializes data values within the area by means of DATA statements. Any COMMON block area that overlaps memory used by loader cannot be initialized.

The first statement of such a program must be a BLOCK DATA statement of the form:

```
BLOCK DATA
```

BLOCK DATA subprograms are processed per ANSI standard.

The body of the block data subprogram may contain only type statements plus EQUIVALENCE, DATA, DIMENSION, and COMMON statements.

If any element of a given common block is initialized, the subroutine must include a complete set of specification statements for the entire block, even though some of the elements do not appear in DATA statements. More than one block may be initialized by a single subprogram. Examples:

```
BLOCK DATA
COMMON /COM1/C2,C3,ARR/COM2/X,Z,C
```

```
      DIMENSION ARR (40)
      EQUIVALENCE(C1,ARR(1),(C4,ARR(2))
      INTEGER X
      COMPLEX C
      DATA C1, C2, C3, C4 /4*0.0/,
     X 1,C/45,(1.3,3.14)/
      END
```

LIBRARY SUBROUTINES

Library subroutines are referenced using the CALL statement described previously. Standard Prime library subroutines are summarized in Appendix B.

SENSE LIGHT/SWITCH SUBROUTINES

Subroutines identified in Appendix J permit the program to communicate with the control panel sense switches, lights, and error flag.

These routines allow the program to test for error conditions and report any errors to the front panel lights.

LINKING FORTRAN AND ASSEMBLY LANGUAGE PROGRAMS

FORTRAN and assembly language programs may be intermixed freely in a memory load, provided the proper calling conventions (Reference MAN 1880) are observed and communication links are set up to pass the arguments back and forth.

In the object code of a compiled FORTRAN program, every subroutine call (CALL statement) is converted into an assembly-language CALL pseudo-operation. (The actual object coding is equivalent to a JST instruction followed by an EXT pseudo-op, both of which specify the subroutine name.) If any arguments are specified, the compiler enters a series of DACS containing pointers to the argument variables. This can be seen in Figure 7-1, which illustrates a FORTRAN bench- mark program timed by two assembly language routines that turn the real time clock on and off.

SENSE LIGHT/SWITCH SUBROUTINES

Subroutines identified in Appendix J permit the program to communicate with the control panel sense switches, lights, and error flag.

These routines allow the program to test for error conditions and report any errors to the front panel lights.

LINKING FORTRAN AND ASSEMBLY LANGUAGE PROGRAMS

FORTRAN and assembly language programs may be intermixed freely in a memory load, provided the proper calling conventions (Reference MAN 188Ø) are observed and communication links are set up to pass the arguments back and forth.

In the object code of a compiled FORTRAN program, every subroutine call (CALL statement) is converted into an assembly-language CALL pseudo-operation. (The actual object coding is equivalent to a JST instruction followed by an EXT pseudo-op, both of which specify the subroutine name.) If any arguments are specified, the compiler enters a series of DACS containing pointers to the argument variables. This can be seen in Figure 7-1, which illustrates a FORTRAN bench- mark program timed by two assembly language routines that turn the real time clock on and off.

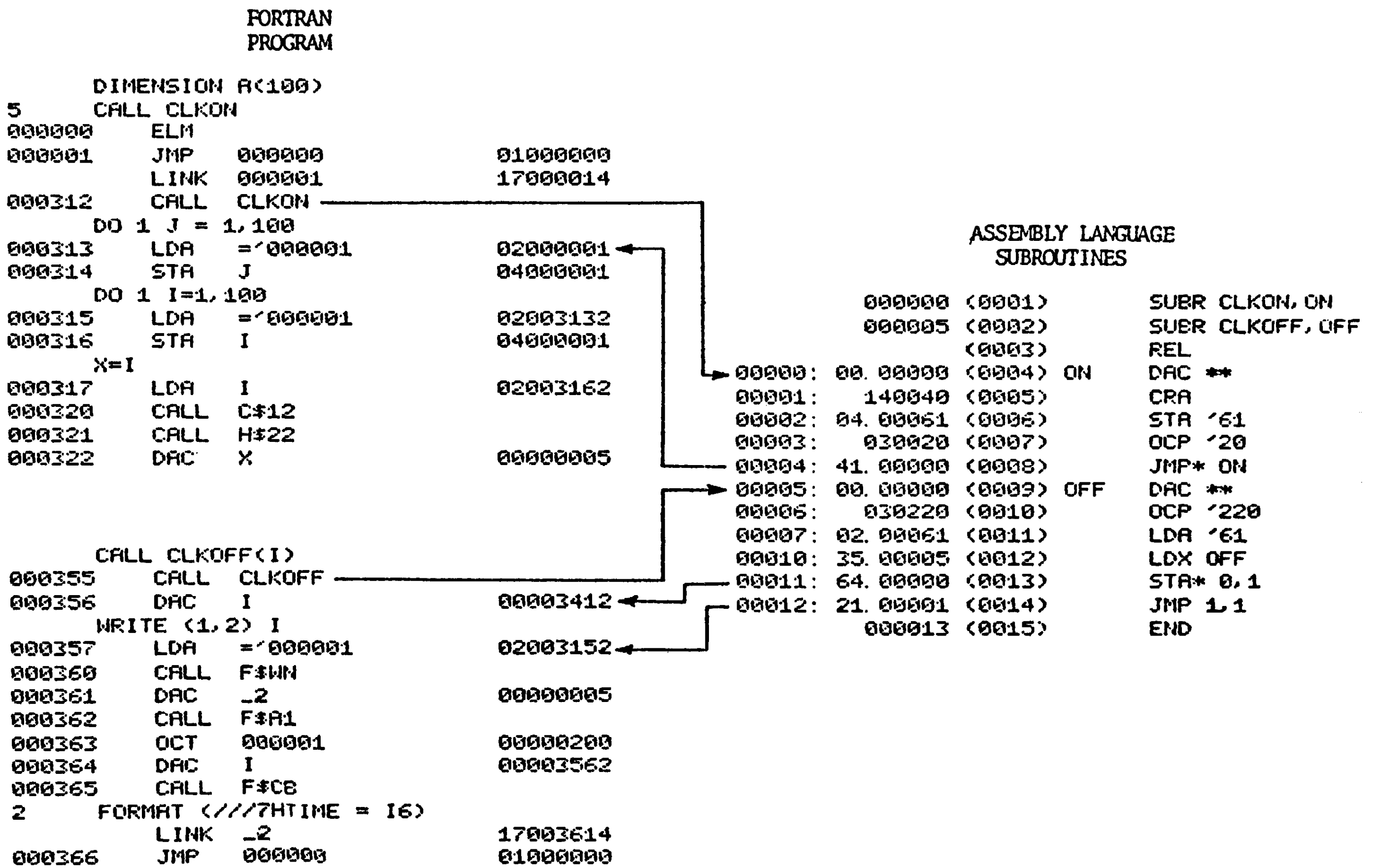


Figure 7-1 FORTRAN/Assembly Language Argument Transfer (Without F$AT)

In the assembly language subroutine, entry points are defined by SUBR pseudo-ops. A DAC is provided at each entry point to hold an address value, to be deposited by a JST from the calling program. If no arguments are to be passed, the value is a pointer to the next executable instruction of the calling program. Customarily, the subroutine returns to the calling program by an indirect jump through this address vector.

However, if arguments are to be passed, the value deposited by the JST is a pointer to the first of the argument variables. That is the case for the CALL CLKOFF statement; the CLKOFF subroutine returns one argument (the elapsed time count) to the FORTRAN program. This is done by storing the A register indirectly through the pointer placed in the OFF entry point by the FORTRAN program. The subroutine then returns to that location plus one. Note the use of indexing to obtain the offset.

Assembly Language Interface

To call a FORTRAN subprogram from an assembly language program, use the PCL (procedure call) instruction followed by APs (argument pointers) for each argument. The last bit must be set in the AP of the last argument.

Example:

```
        SEG
        EXT   FTNSUB
              .
              .
              .
        PCL   FTNSUB
        AP    ARG1,S
        AP    ARG2,S
        AP    ARG3,SL
```

An assembly language subprogram callable from a FORTRAN program must contain an ECB pseudo-op. If the subprogram has arguments, ARGT (argument transfer) must be the first instruction in the procedure frame.

Example:

```
             SEG
             SUBR     PMASUB
  ENTRY      ARGT
             .
             .
             .
             PRTN
             DYMN     ARG1(3), ARG2(3), ARG3(3)
             .
```

```
                .
                .
                LINK
  PMASUB        ECB         ENTRY,, ARG1,3
                .
                .
                .
                END
```

Using F$AT*

The library subroutine F$AT may be used in assembly language subroutines to simplify the tracking down of argument addresses. The assembly language calling sequence is:

```
        CALL F$AT
        OCT  n
ARG1    DAC **
ARG2    DAC **
 .       .
 .       .
 .       .
ARGn    DAC **
```

* Does not apply to P400 64V mode program.

where 'n' is the number of arguments to be transferred, and each ARG is an argument name. The CALL F$AT statement must be the first statement in the subroutine following the DAC ** entry point. F$AT could have been used in the CLKOFF subroutine of Figure 7-1 as follows:

```
OFF    DAC **
       CALL F$AT
       OCT 1
TIME   OCT 0      F$AT WILL PUT ADDRESS LINK HERE
       OCP '220
       LDA '61
       STA* TIME
       JMP* OFF   NORMAL RETURN
```

,his coding has exactly the same effect as the example in Figure 7-1 but execution time is slower. However, it is more convenient to use F$AT when several arguments are involved.

NOTE:

If the subroutine has more than one argument, DAC in call is followed by 0 word.

SECTION 8

PROGRAMMING TECHNIQUES

MAXIMUMS

Accuracy of Numbers:

Single Precision: Any number of digits may be used and the seven most significant digits are stored internally.

Double Precision: Any number of digits may be used and fourteen most significant digits are stored internally.

ATTACHING TO ANOTHER USER FILE DIRECTORY (UFD)

A user program can operate in more than one UFD by calling the ATTACH subroutine. However, care should be taken that the rules for ATTACH (see Section 3, PRIMOS II and III File System User Guide (MAN 2604).

The ATTACH subroutine has the same effect as the ATTACH internal command. The calling sequence is:

```
CALL ATTACH (Ufd, Ldisk, Password, Key, Altrtn)
```

NOTE:

The reference subkeys are shown in Appendix E.

In attaching to a directory, the subroutine ATTACH specifies where to look for the directory. ATTACH either specifies a user file directory in the master file directory (MFD) on a particular logical disk or a file directory in the current UFD, or the home UFD as the directory to be attached. ATTACH may specify a file unit number on which a segment directory is open. In the segment directory reference, the file directory to be attached is the one whose beginning disk address is given by the word at the file pointer of the file unit.

CLOSING AND OPENING FILES

CONTROL Subroutine

A CONTROL subroutine call (i.e., CALL CONTRL (key, name, logical-device, altrtn) is a method of operning and closing files. Functions not applicable to a certain device are ignored. This means that with CONTRL subroutine calls, functions can be requested in a device independent way.

SEARCH Subroutine

A SEARCH subroutine call (i.e., CALL SEARCH (KEY, NAME, FUNIT, ALTRTN) can be written within a FORTRAN program to either open or close a PRIMOS File.

The following information describes the search operation. However, the rules for using the SEARCH subroutine are describd in Section 3 of the PRIMOS II and PRIMOS III File System Users Guide (MAN 2604).

SEARCH is used to connect a file to a file unit (open a file) or disconnect a file from a file unit (close a file). After a file is connected to a unit; PRWFIL and other routines may be called, either to position the current- position pointer of a file unit (file pointer) or to transfer information to or from the file (using the file unit to reference the file).

On opening a file, SEARCH specifies allowable operations that may be performed by PRWFIL, and other routines. These operations are read only, write only or both read and write.

On opening a file, SEARCH also specifies where to look for the file or where to add the file, if the file does not already exist, and also SEARCH specifies the file is to be opened for writing or both reading and writing. SEARCH either specifies a filename in the currently attached user file directory or a file unit number on which a segment directory is open. In the segment directory reference, the file to be opened or closed is the one whose beginning disk address is given by the word at the current position pointer of the file unit.

On creating a newfile, the user specifies to SEARCH the file type of the new file.

The subroutine SEARCH may be used to perform actions other than opening and closing a file. SEARCH may delete a file, rewind a file unit, or truncate a file.

On a call to close a file, SEARCH attempts to close file NAME and generates an error message or goes to the alternate return if NAME is not found. FUNIT is ignored unless NAME is 0. If NAME is 0,

SEARCH ensures that FUNIT is closed. That is, it closes FUNIT if FUNIT is open but does not generate an error message if the file unit is closed. Example:

```
    CALL SEARCH (1, 'OBJECT'm 1, ERR)
```

Searches for a file, OBJECT, in the current UFD and opens it for reading.

The user is allowed to open the current UFD for reading via a call to SEARCH. The calling sequence for this feature is:

```
    CALL SEARCH (1, -1, Funit, Altrtn)
```

This call opens the current UFD for reading on Funit. The user must have owner access rights to the UFD; i.e., the owner password must have been given in the most recent call to ATTACH (or ATTACH command). Control goes to Altrtn if there is no UFD attached, if Funit is already in use, or the user does not have owner rights to the UFD.

Direct Positioning Subroutine - 'POSFIL'

A standard FORTRAN IV (or assembly language) library subroutine exists that allows direct positioning to any record in a disk file. This subroutine 'POSFIL', functions with both sequential access (SAM) and direct access (DAM) files, although it is most often used with the latter. The only requirement in file organization is that all records in a file must be of equal length. This presents no problem with unformatted (binary) files, as the binary disk output routine (Ø$BDØ7) automatically generates fixed length records. Formatted (ASCII) files must be specified by the programmer as fixed length records when defining the record size. See example below.

'POSFIL' operates under control of any of Prime's operating systems (single user DOS, multi-user DOS/VM, Real Time Operating System and, virtual DOS running is a background task under RTOS).

Calling Sequence:

```
       CALL POSFIL  (I,J,K)
```

where:

I = FORTRAN logical unit number. This must be a disk file open for reading only or, open for reading and writing.

J = Integer expression representing desired record. (Record numbering starts a 1)

K = Optional alternate return. If present, this represents the statement to which control is passed if error or end of file conditions are detected. If this value is Ø or omitted, errors will cause the program to abort to operating system level and print an error message.

End of file positioning may be done with 'POSFIL' by specifying a record number of 32767 and an alternate return value corresponding to the normal return statement. An appropriate error statement should immediately follow the 'CALL POSFIL'.

'POSFIL' postions the file pointer at the start of the desired record by locating the record size in IOCS, adding 1 to compensate for the word added by the FORTRAN disk output drivers, multiplying by the record number minus 1, dividing by the number of words per physical record and making a call to 'PRWFIL' to position the file at an absolute record and word.

Examples: The following program illustrates how to create an ASCII file suitable for direct access and reading, which is performed in the second part of the program. The exact same coding could be used with unformatted reads and writes in the case of a binary file.

Example:

```
C       PROGRAM TO WRITE AN ASCII FILE SUITABLE FOR POSFIL
C       AND TO READ BACK RANDOM RECORDS
C
        INTEGER TEXT(6)
C
C       DEFINE UNIT #5 AS DISK FILE OF FIXED LENGTH ASCII RECORDS, DOS
        FILE UNIT #1, SIX WORDS PER LOGICAL RECORD
C
        CALL ATTDEV (5,8,1,6)     /* 6 IS RECORD LENGTH
C
C       OPEN NEW DAM FILE ON DOS FILE UNIT #1 FOR READING AND WRITING
C
        CALL SEARCH (:2003,'SAMPLE',1)
C
C       WRITE 9999 FIXED LENGTH ASCII RECORDS
C
        DO 10 I=1,9999
10      WRITE (5,1000) I
1000    FORMAT ('RECORD #',I4)
C
L       ASK FOR RECORD NUMBER, LOCATE, READ, AND WRITE TO USER TERMINA
C
20      CALL TNOU ('ENTER RECORDS $',I4)
        CALL TIDEC (I)
        IF (I) 20,30,25           /* 0 CAUSES PROGRAM TO TERMINATE
25      CALL POSFIL (5,I,$35)
        READ (5,1001) TEXT
        CALL TNOU (TEXT,12)
        GO TO 20
C
C       CLOSE FILE AND EXIT
C
30      CALL SEARCH (4,0,1)
        CALL EXIT
        GO TO 20
C
C       CONTROL PASSES TO HERE IF POSFIL ENCOUNTERS AN ERROR OR EOF
C
35      CALL PRERR                /* PRINT ERROR MESSAGE
        GO TO 20                  /* TRY AGAIN
C
1001    FORMAT (6A2)
C
        END
```

RECORD LENGTH OPTION

ATTDEV Subroutine

While the formatted record length is 120 characters (maximum), a user can define a larger size using ATTDEV subroutine.

```
CALL ATTDEV (logical unit, device, unit, buffer size)
```

Subroutine ATTDEV can be used:

1) To change the record size associated with a unit number.

2) To change the unit number to physical device mapping.

ATTDEV performs these functions by manipulating entries in tables in library module CONIOC.

Argument Explanation

logical unit - FORTRAN unit number (used in READ and WRITE statements) (See Table 6-1 in Section 6)

device - Position of physical device in CONIOC device-type tables. The default configuration:

1 -> user terminal

7 -> file system (disk)

unit - For multi-unit devices (i.e., mag tape). If device is the file system, the unit is FUNIT (see Table 6-1)

buffer size - The maximum record size in words (number of characters plus 1)/divided by two) for logical unit.

F$IO Subroutine

F$IO provides a buffer equal to the maximum record length to be used for FORTRAN transfers. The default buffer size is 132 characters. When a larger buffer is required, it is defined by the following statement:

```
COMMON/F$IOBF/IBUF (size)
```

where: SIZE is number of characters divided by 2.

NOTE: Only the common block name, F$IOBF, and the size of the array specified in the common block are significant; the array name itself is arbitrary.

APPENDIX A

COMPILER ERROR MESSAGES

COMPILER ERROR MESSAGES FOR LARGE SYSTEMS (LFTN)

Table A-1 lists each compiler error and the corresponding definition of the error.

The general format of the error messages is:

    ****LINE nnnn [context] name - message

where:

LINE nnnn - nnn is the source line number that the statement in error started on. All lines read from an insert file have the same source line number. In the case of an undefined statement number error, nnnn is the line number of the last reference to the undefined statement number, not the line number of the END statement where it was detected. If an error detected in an EQUIVALENCE statement, the word ´EQUIVALENCE´ is substituted for ´LINE nnnn´.

context - context consists of the last 1-10 nonblank characters processed by the compiler before detecting the error. This field can be used to isolate the position in the statement that error occurs.

name - If the error is directly related to the misuse of a specific name, that name will be included in the error message. Otherwise the field will be omitted.

message - A message of up to 20 characters in length describing the error. A list of all messages is included below.

COMPILER ERROR MESSAGES FOR SMALL SYSTEMS (SFTN)

Table A-2 lists each compiler error message code and the corresponding error definition.

Table A-1. Compiler Error Messages (LFTN)

Error Messages	Definition
ARG LIST REQUIRED	Argument list not specified in FUNCTION statement.
ARRAY NAME REQUIRED	Something other than an array name appeared in a position where only an array name is allowed.
CHAR STRING SIZE	A character string was not terminated, or a string in a DATA statement was longer than the associated name list.
COMMON NAME ILL.	Illegal use of a name already declared in common.
COMPILER OVERFLOW	Insufficient memory to compile program.
CONSTANT REQUIRED	A name appeared where only a constant is allowed.
CONSTANT TOO LARGE	Constant exponent excessive for data type.
DATA MODE ERROR	Illegal mode mixing in an expression, expression mode not of required type, or constant in DATA statement is of different mode than associated name in variable list.
EXCESS SUBSCRIPTS	Too many subscripts in EQUIVALENCE or DATA list item.
FUNCT VAL UNDEFINED	The function name was not assigned a value in a FUNCTION subprogram.
ILL.DO TERMINATION	Improper DO loop nesting, or an illegal statement terminating a DO loop.
ILL. EQUIVALENCE	EQUIVALENCE group violates EQUIVALENCE rules or specifies an impossible equivalencing.
ILL. LOGICAL IF	A logical IF contained in a logical IF, or a DO statement contained in a logical IF.
ILL. STMT NO. REF	Reference to a specification statement number.
ILL. UNARY OP USAGE	Improper use of an operator in an expression.
ILL. USE OF ARG	SUBROUTINE or FUNCTION argument used in COMMON, EQUIVALENCE, or DATA statement.
ILL. USE OF STMT	Statement illegal in context of the program. For example, RETURN in a main program, SUBROUTINE not the first statement, or specification statements out of order. If an undeclared array name is used on the left in an assignment statement, the compiler will assume it is a statement function definition and therefore generate this error.
INCONSISTENT USAGE	The use of the name listed in the error message conflicts with earlier usage. This message also will be generated at the END

	statement in a SUBROUTINE subprogram if the subroutine name is used within the subprogram.
INTEGER REQUIRED	A non-integer name or constant appeared where only an integer name or constant is allowed.
MULT DEF STMT NO.	The statement number of the current line has already been defined.
NAME REQUIRED	A constant appeared where only a name is allowed.
NO END STMT	The last statement in a subprogram was not an END statement.
NO PATH TO STMT	The current statement does not have a statement number d the previous statement was an unconditional transfer of control.
NONCOMMON DATA	A BLOCK DATA subprogram initialized data not defined in common or contained executable statements.
PARENTHESIS MISSING	Incorrect parenthesis used in an implied DO loop in an I/O statement.
STMT NAME SPELLING	A statement name was recognized by its first four characters, but the remaining spelling was incorrect.
STMT NO. MISSING	A FORMAT statement appeared without a statement number.
SUBPGM/ARR NAME ILL	Illegal usage of subprogram or array name.
SUPBROGRAM NAME ILL	Illegal usage of subprogram name.
SYMBOLIC SUBSCR ILL	Illegal usage of a symbolic subscript in a specification statement.
SYNTAX ERROR	General syntax error, context usually shows offending character(s).
UNDECLARED VARIABLE	The listed variable did not appear in a specification statement (generated when the undeclared variable check option is enabled).
UNDEFINED STMT NO.	The listed statement number was not defined in the subprogram. The listed line number is the line number of the last reference to the statement number.
UNRECOGNIZED STMT	The compiler could not identify the statment.

Table A-2. Compiler Error Messages (SFTN)

CODE	DEFINITION
AR	Item not an array name.
BD	Code generated within a block data subprogram.
BL	Block data not first statement.
CE	Constant´s exponent exceeds 8 bits (Over 255).
CH	Improper terminating character (punctation).
CM	Comma outside parenthesis, not in a DO statement.
CN	Improper constant (data initialization).
CR	Illegal common reference.
DA	Illegal use of dummy argument.
DD	Dummy item appears in an equivalence or data list.
DM	Data and Data Name mode do not agree.
DT	Improper DO termination.
EC	Equivalence group not followed by comma or CR.
EQ	Expression to left of equals, or multiple equals.
EX	Specification statement appears after cleanup.
FA	Function has no arguments.
FD	Function name not defined by an arithmetic statment.
FS	Function/Subroutine not the first statement.
HD	Hollerith string too long in DATA statement.
HS	Hollerith data string extends past end of statement.
IC	Impossible common equivalencing.
ID	Unrecognizable statement.
IE	Impossible Equivalence grouping.
IF	Illegal IF statement type.
IN	Integer required at this position.
IO	Error in Read/Write statement syntax.
IT	Item not an integer.
MM	Mode mixing error.
MO	Data pool overflow.
MS	Multiply defined statement number.
NA	Name required.
NC	Constant must be present.
ND	Wrong number of dimensions.
NE	No END statement prior to Control statement.
NS	Subroutine name not allowed
NT	Logical NOT, not an unary operator.
NU	Name already being used.
OP	More than one operator in a row.
PA	Operation must be within parenthesis.
PH	No path leading to this statement.
PR	Parenthesis missing in a DO statement.
PW	* preceded by an operator other than a *.
RL	More than 1 relational operator in a relational example.
RN	Reference to a specification statement´s number.
RT	Return not allowed in main program.
SC	Statement number on a continuation card.

SP	Statement name misspelled.
ST	Illegal statement number format.
SU	Subscript incrementer not a constant.
TF	"TYPE" not followed by "FUNCTION" or List.
TO	Assign statement has word TO missing.
UO	Multiple + or - signs, not as unary operators.
US	Undefined statement number.
VD	Symbolic subscript not dummy in dummy array, or symbolic subscript appears on a non-dummy array.

APPENDIX B

RUN-TIME ERROR MESSAGE

When a library subroutine detects an error condition, it types a two character error message on the ASR, then continues, usually with unpredictable results. Some subroutines do not check for errors but call other subroutines which do. For example, DLOG10 does not check for arguments less than or equal to 0, but it calls DLOG2 which does. Error codes are preceded by "****".

Code	Definition
AD	Overflow/underflow occurred (A$66) (S$66).
AT	ARG = ARG2 = Ø for ATAN2
BN	Device error in REWIND command (Note 1).
DE	Double precision exponent overflow decode FORMAT/DATA MISMATCH (literal)
DL	Argument is not greater than zero (DLOG, DLOG2).
DN	Device error (end file). (Note 1).
DT	Second argument is zero (DATAN2).
DZ	Division by zero (D$22).
EQ	Exponent overflow occurred (A$81).
EX	Exponent overflow (EXP).
FE	Format error (F$IO).
FN	Device error in BACKSPACE command, (Note 1).
II	Improper power value (E$11).
IM	Overflow or underflow occurred (M$11, E$11).
LG	Argument is not greater than Ø (ALOG, ALOG1Ø).
RI	Number too large for integer conversion (C$12).
RN	Device error or end of file in READ statement. (Note 1.)
SE	Single precision exponent overflow.
SQ	Argument is negative (SQRT).
SZ	Single precision divide by zero.
WN	Device eror or end of file in WRITE statement. (Note 1.)
XX	ARG > 32767

Notes:

1. Device error codes are printed in the form: ****cc n
where n is the FORTRAN logical unit number of the device.

P-300	P-400	Definition	Explanation
DT	DATAN	Bad Argument	A2 = 0
SQ	DSQRT	Argument < 0	A < 0
RI	DSIN/DCOS	Argument range error	A too hi/too low
EX	DEXP	Overflow/under-flow	Result too lg/sm
RI	DEXP	Argument too large	A too lg
DL	DLOG/DLOG2	Argument < = 0	A < or = 0
SQ	SQRT	Argument < 0	A < 0
RI	SIN/COS	Argument too large	A too lg
AT	ATAN2	Both arguments = 0	A1 & A2 = 0
LG	ALOG1/ALOG10	Argument < = 0	A < or = 0
RI	EXP	Argument too large	A >> 0
EX	EXP	Overflow	Result >> 0
II	I**I	Argument error	A1**A2 > 32767
BN	F$BN	Bad logical unit	LU out of range
FE	F$IO	Format error	Bad FMT stmt
--	F$IO	Null Read Unit	Read Lunit not configured
--	F$IO	Format/Data Mismatch	Input data doesn't correspond with FMT statement
ST	STOP (n)		STOP Start Encountered

PA	PAUSE (n)		PAUSE Start Encountered
--	ATTDEV	Bad Unit	Bad Lunit to ATTDEV

APPENDIX C

LIST OF STATEMENTS

ENCODE (c,f,a) list

The ENCODE statement converts the elements of the I/O list into ASCII data according to the specified format and stores the first c characters of the resultant line buffer into the specified array where c is the number of ASCII characters to be transferred, f is the format specifier and a is the array name.

DECODE (c,f,a) list

The DECODE statement converts the c character in the specified array into the I/O list elements according to the specified format, where c is the number of ASCII characters to be transferred, f the format specifier, and a the array name.

PRINT f list

The PRINT statement performs the same function as the WRITE statement.

FORMAT SN FORMAT (dF1 dF2 dF3....... Fn)

The FORMAT STATEMENTS DO THE TRANSLATION BETWEEN THE EXTERNAL FORM OF DATA AND THE WAY IT IS STORED INTERNALLY WITHIN THE PROCESSOR, where SN is a mandatory statement number, each F is a format field description and each d is a delimiter (slash or comma).

REWIND u

A REWIND statement causes unit u to be positioned at its initial point.

BACKSPACE u

A BACKSPACE statement positions unit u so that the preceding record becomes the next record.

ENDFILE u

The ENDFILE statement causes the recording of an ENDFILE record on unit u.

Mode v1,v2,v3...,vn

where mode is INTEGER, REAL, DOUBLE PRECISION, COMPLEX, or LOGICAL and each v is a variable name, an array name, a function name, or an array declarator.

DIMENSION v1(i1), v2(i2),...,vn(in)

Establishes the name and maximum storage requirement of an array. The variable ´v´ is an assigned name of an array. Each ´i´ is a series of one, two or three subscripts that define the dimensions and size of the array.

EQUIVALENCE (k1), (k2),...,(kn)

Equates single variables to each other. Each k is a list of 2 or more variables, subscripted variables or array names separated by commas.

COMMON/x/a1/.../xn/an/

Data items are assigned sequentially within a COMMON block in the order of appearance. Each a is a nonempty list of variable names, array names, or array declarators and each x is a COMMON block name or is empty.

EXTERNAL v1,v2,...,vn

Permits the name of an external function subprogram (library or user defined) to be passed as an argument in a subroutine call or function reference, where each v is declared to be an external procedure name.

DATA k1/d1/,k2/d2/,....,kn/dn/

Sets variables or array elements k to initial values during loading of the object program along with corresponding constants d.

TRACE v1,v2,...vn

Causes diagnostic printouts of the results of computation to facilitate debugging, where each v is a variable name or array.

GO TO k

Causes the statement identified by the statement label k to be executed next.

GO TO (k1,k2, ... ,kn), i

A computed GO TO statement causes the selection of the 1st, 2nd,... or nth label according to the computed value of i (1,2,....,n).

GO TO i, (k1,k2,...,kn)

Assigned GO TO statement which assigns a current value of i to a statement label. The statement identified by that label is executed next.

ASSIGN k TO i

A GO TO assignment statement causes the statement identified by the assigned statement label to be executed next.

IF (e) k1,k2,k3

An arithmetic IF statement where e is any arithmetic expression of integer real, or double precision type, and the k's are statement labels. The value of e determines one of 3 possible branches as follows:

value of e	Statement Executed Next
< Ø (negative)	k1
= Ø	k2
> Ø (positive)	k3

IF (e) S

A logical IF statement where e is a logical expression and S is any executable statement except a DO statement or another logical If statement. The logical expression e is evaluated. If e has the value .TRUE., statement S is executed. Otherwise, control passes to the next statement.

DO n i = m1,m1,m3 or DO n i = m1,m2

A DO statement where n is the label of an executable statement following the DO statement in the same program unit; i is an integer variable called the index and m1,m2,and m3 are the initial, limit, and increment values of the index respectively. Default value of m3 is one.

CONTINUE

The CONTINUE Statement terminates the current execution of a DO loop. If no DO loop is in effect, control transfers to the next executable statement.

STOP (n)

where n is an optional string of one to five decimal digits. A STOP statement transfers control to the subroutine F$HT.

PAUSE (n)

where n is an optional decimal constant. A PAUSE statement transfers control to F$HT. A subsequent START command continues the operation.

END

The END statement is placed at the end of every program.

FULL LIST

The FULL LIST statement specifies a source listing, and a symbolic listing.

$INSERT Filename

The INSERT statement causes the text in file to be compiled in place of the $INSERT statement.

WRITE (u,f)list

The WRITE statement is used for data output where u is the unit number of the I/O device and f is the format statement number.

NOLIST

The NOLIST statement specifies no source listing, and no symbolic listing.

LIST

The LIST statement specifies a source listing and no symbolic listing.

END AND ERROR RETURNS IN READ/WRITE STATEMENTS

```
READ(d,END=a)              READ(d,f,END=a)
READ(d,ERR=b)              READ(d,f,ERR=b)
READ(d,ERR=b,END=a)        READ(d,f,ERR=b,END=a)
READ(d,END=a,ERR=b)        READ(d,f,END=a,ERR=b)
WRITE(d,ERR=b)             WRITE(d,f,ERR=b)
```

where:

d - device specifier

f - format specifier

a - statement number that control is to be transferred to if an end of file condition is detected in the READ

b - statement number that control is to be transferred to if a device error occurs in the READ/WRITE operation

APPENDIX D

PROGRAM EXAMPLES

Appendix D contains numerous FORTRAN IV illustrated examples that serve as an aid in understanding the FORTRAN IV specification.

X1 ARRAY EXAMPLE

A. PROGRAM

```
(0001)        INTEGER T(100),L(100),A,B
(0002)        DO 31 I=1,100
(0003) 31     T(I)=0
(0004)        DO 20 I=1,100
(0005)        L(I)=I**2
(0006)        T(I)=T(I)+L(I)
(0007) 20     CONTINUE
(0008)        A=1
(0009)        B=5
(0010) 21     WRITE(1,1)(L(K),K=A,B),(L(KK),KK=A,B)
(0011) 1      FORMAT(2(5(I5)))
(0012)        A=A+5
(0013)        B=B+5
(0014)        IF(A. GE. 100)CALL EXIT
(0015)        GO TO 21
(0016)        END
```

B. COMPILE & RUN

```
OK, FTN X1
GO
0000 ERRORS (FTN-1082.L11)

OK, LOAD
GO
$ LO B_X1
$LI
LC
$ SA *X1
```

```
$QU

OK, R *X1
GO
```

C. OUTPUT

```
    1    4    9   16   25    1    4    9   16   25
   36   49   64   81  100   36   49   64   81  100
  121  144  169  196  225  121  144  169  196  225
  256  289  324  361  400  256  289  324  361  400
  441  484  529  576  625  441  484  529  576  625
  676  729  784  841  900  676  729  784  841  900
  961 1024 1089 1156 1225  961 1024 1089 1156 1225
 1296 1369 1444 1521 1600 1296 1369 1444 1521 1600
 1681 1764 1849 1936 2025 1681 1764 1849 1936 2025
 2116 2209 2304 2401 2500 2116 2209 2304 2401 2500
 2601 2704 2809 2916 3025 2601 2704 2809 2916 3025
 3136 3249 3364 3481 3600 3136 3249 3364 3481 3600
 3721 3844 3969 4096 4225 3721 3844 3969 4096 4225
 4356 4489 4624 4761 4900 4356 4489 4624 4761 4900
 5041 5184 5329 5476 5625 5041 5184 5329 5476 5625
 5776 5929 6084 6241 6400 5776 5929 6084 6241 6400
 6561 6724 6889 7056 7225 6561 6724 6889 7056 7225
 7396 7569 7744 7921 8100 7396 7569 7744 7921 8100
 8281 8464 8649 8836 9025 8281 8464 8649 8836 9025
 9216 9409 9604 980110000 9216 9409 9604 980110000
```

X2 FORMAT STATEMENTS

A. PROGRAM

```
(0001) C     THIS PROGRAM ILLUSTRATES THE USE OF IF STATEMENTS
(0002) C     AND FORMAT STATEMENTS.
(0003)       I=100
(0004) 5     WRITE(1,10)
(0005)       READ(1,20) X
(0006)       IF(X-I)30,40,50
(0007) 30    WRITE(1,60)I
(0008)       GOTO5
(0009) 40    WRITE(1,70)I
(0010)       GOTO5
(0011) 50    WRITE(1,80)I
(0012)       GOTO5
(0013) 10    FORMAT('INPUTANYNUMBER')
(0014) 20    FORMAT(I6)
(0015) 60    FORMAT('YOURNUMBERWASLESSTHAN',I6)
(0016) 70    FORMAT('YOURNUMBERWAS',I6)
(0017) 80    FORMAT('YOUR NUMBER WAS GREATER THAN   ',I6)
(0018)       CALL EXIT
(0019)       END
```

B. COMPILE & RUN

```
OK,

FTN X2

GO
0000 ERRORS (FTN-1082.L11)

OK,LOAD
GO
$ LO B_X2
$ LI
LC
$ SA *X2
$ QU

OK, R *X2
GO
```

C. OUTPUT

```
INPUT ANY NUMBER
```

```
77
YOUR NUMBER WAS LESS THAN 1ØØ
INPUT ANY NUMBER
-77
YOUR NUMBER WAS LESS THAN 1ØØ
INPUT ANY NUMBER
Ø
YOUR NUMBER WAS LESS THAN 1ØØ
INPUT ANY NUMBER

QUIT,
```

X3 ASSIGN GO TO

A. PROGRAM

```
(0001) C     *COMPUTED GO TO EXAMPLE*
(0002) 205   WRITE (1,50)
(0003) 50    FORMAT('TYPE I HERE')
(0004)       READ(1,55) I
(0005) 55    FORMAT(I5)
(0006)       X =48
(0007)       B = I*5
(0008)       C = I+5
(0009)       A = B + C
(0010)       Y = X + A
(0011) 20    GO TO(100,310,320),I
(0012)       WRITE(1,105)
(0013) 105   FORMAT('ERROR - I OUT OF RANGE')
(0014)       GO TO 200
(0015) 100   WRITE(1,106) Y
(0016) 106   FORMAT('Y = 'I3,' WHEN I = 1')
(0017)       GO TO 205
(0018) 320   PRINT 110,Y
(0019) 110   FORMAT('Y = 'I3,' WHEN I = 3')
(0020)       GO TO 205
(0021) 310   PRINT 115,Y
(0022) 115   FORMAT('Y = 'I3,' WHEN I = 2')
(0023)       GO TO 205
(0024) 200   CALL EXIT
(0025)       END
0000 ERRORS (FTN-1082.L11)
```

B. COMPILE & RUN

```
OK, FTN X3
GO
0000 ERRORS (FTN-1082.L11)

OK, LOAD
GO
$ LO B_X3
$LI
LC
$ SA *X3
$ QU

OK, R *X3
GO
```

C. OUTPUT

```
TYPE I HERE
1
Y = 59 WHEN I = 1
TYPE I HERE
2
Y = 65 WHEN I = 2
TYPE I HERE
3
Y = 71 WHEN I = 3
TYPE I HERE
4
ERROR - I OUT OF RANGE
OK,
```

X4 SIMPLE CALCULATION

This example illustrates format statements within a simple program to compute and print the square of a number.

A. PROGRAM

```
ON001) C     THIS IS A SIMPLE PROGRAM WHICH ILLUSTRATES THE INTERACTI
(0002) C     BETWEEN THE READ, THE WRITE AND THE FORMAT STATEMENTS.
(0003) 5     WRITE(1,10)
(0004)       READ(1,20)R
(0005)       R=R**2
(0006)       WRITE(1,30)R
(0007)       GO TO 5
(0008) 10    FORMAT('GIMME A NUMBER!!')
(0009) 20    FORMAT(F20.6)
(0010) 30    FORMAT('THE SQUARED NUMBER IS:'F20.6/)
(0011)       END
```

B. COMPARE & RUN

```
OK, FTN X4
GO
0000 ERRORS (FTN-1082.L11)
OK, LOAD
GO
$ LO B_X4
$ LI
LC
$ SA *X4
$ QU

OK, R *X4
GO
```

C. OUTPUT

```
GIMME A NUMBER!!
37.999
THE SQUARED NUMBER IS:  1443.923340
GIMME A NUMBER!!
45.987
THE SQUARED NUMBER IS:  2114.803711
```

X5 FORMULA CALCULATION

Given:

The equation for determining the current flowing through an alternate current circuit is:

$$I = \frac{E}{\sqrt{R^2 + (2\pi fL - \frac{1}{2\pi fc})^2}}$$

A. PROGRAM

```
(0001) C     THIS PROGRAM NAME IS X5
(0002) C
(0003) C
(0004)       WRITE(1,10)
(0005) 10    FORMAT(/'GIVE ME THE FOLLOWING VALUES')
(0006)       WRITE(1,15)
(0007) 15    FORMAT(/'OHMS:')
(0008)       READ(1,20) OHMS
(0009) 20    FORMAT(E20.6)
(0010)       WRITE(1,25)
(0011) 25    FORMAT('FREQ:')
(0012)       READ(1,30) FREQ
(0013) 30    FORMAT(E20.6)
(0014)       WRITE(1,35)
(0015) 35    FORMAT(//'HENRY:')
(0016)       READ(1,30) HENRY
(0017)       WRITE(1,36)
(0018) 36    FORMAT('VOLTS: ')
(0019)       READ(1,40) VOLT
(0020) 40    FORMAT(E5.2)
(0021)       WRITE(1,45)
(0022) 45    FORMAT('FARADS: ')
(0023)       READ(1,50) FARAD
(0024) 50    FORMAT(E20.12)
(0025)       AMPS = VOLT/SQRT(OHM**2 + ((6.2832*FREQ*HENRY)-1/
(0026)       X(6.2832*FREQ*FARAD))**2)
(0027)       WRITE(1,55) AMPS
(0028) 55    FORMAT('AMPS =',E20.12)
(0029)       CALL EXIT
(0030)       END
```

B. COMPILE & RUN

```
OK, FTN X5
GO
ØØØØ ERRORS 8FTN-1Ø82.L11)

OK, LOAD
GO
$ LO
$ LI
LC
$ SA *X5
$QU

OK, R *X5
GO
```

C. OUTPUT

```
GIVE ME THE FOLLOWING VALUES

OHMS:  75 FREQ:  6Ø.

HENRY:   Ø.ØØ75 VOLTS:  68.  FARADS:  Ø.ØØØ57 AMPS = Ø.372357Ø25146E
Ø2

OK,
```

X6 FLOATING POINT FORMAT

This example illustrates input and output formats.

A. PROGRAM

```
(0001) C     *CALCULATES VOLUME OF A BØX.*
(0002)       REAL LENGTH
(0003)       WRITE(1,5Ø)
(0004) 5Ø    FORMAT('GIVE LENGTH, WIDTH, AND HEIGHT')
(0005)       READ(1,1ØØ) LENGTH, WIDTH, HEIGHT
(0006) 1ØØ   FORMAT(3F1Ø.3)
(0007)       VOLUME = LENGTH*WIDTH*HEIGHT
(0008)       WRITE(1,1Ø5) LENGTH, WIDTH,HEIGHT,VOLUME
(0009) 1Ø5   FORMAT('LGTH:',F6.2,' X W:',F6.2,' X HT:',F6.2,'=VOL:',
C      F1Ø.3)
(001Ø)       CALL EXIT
(0011)       END
```

B. COMPILE & RUN

```
OK, FTN X6 ØØØØ ERRORS (FTN-1Ø82.L11)  OK, LOAD
GO
$ LO B_X6
$ LI
LC
$ SA *X6
$ QU

OK, R *X6
GO
```

C. OUTPUT

```
GIVE LENGTH, WIDTH, AND HEIGHT
12.,56.,87.
LGTH:  12.ØØ X W:  56.ØØ X HT:  87.ØØ=VOL:  58464.ØØØ
```

X7 DO LOOP

This example illustrates output formats.

A. PROGRAM

```
(ØØØ1) C     *THIS PROGRAM EXAMPLE DEMONSTRATES A DO LOOP*
(ØØØ2) C     *CALCULATES VOLUME OF A BØX.*
(ØØØ3)       REALLENGTH
(ØØØ4)       WRITE(1,18)
(ØØØ5) 18    FORMAT('HOW MANY BOXES?')
(ØØØ6)       READ(1,19) N
(ØØØ7) 19    FORMAT(I5)
(ØØØ8)       DO 1ØØ I = 1,N
(ØØØ9) 5     FORMAT('ENTER LENGTH: ')
(ØØ1Ø)       WRITE(1,5)
(ØØ11)       READ(1,6)LENGTH
(ØØ12) 6     FORMAT(F6.3)
(ØØ13)       WRITE(1,1Ø)
(ØØ14) 1Ø    FORMAT('ENTER WIDTH: ')
(ØØ15)       READ(1,7) WIDTH
(ØØ16) 7     FORMAT(F6.3)
(ØØ17)       WRITE(1,15)
(ØØ18) 15    FORMAT('ENTER HEIGHT: ')
(ØØ19)       READ (1,8)HEIGHT
(ØØ2Ø) 8     FORMAT(F6.3)
(ØØ21) 9     VOLUME = LENGTH*WIDTH*HEIGHT
(ØØ22)       WRITE(1,2Ø) LENGTH, WIDTH,HEIGHT,  VOLUME
(ØØ23) 2Ø    FORMAT('LGTH:',F1Ø.3,3X,'WI:',F1Ø.3,3X,'HT:'F1Ø.3,3X,
(ØØ24)       *'VOL:',F1Ø.3,5X,///,5X)
(ØØ25) 1ØØ   CONTINUE
(ØØ26)       WRITE(1,1Ø5) N
(ØØ27) 1Ø5   FORMAT('I HAVE CALCULATED' ,I2,3X 'BOXES')
(ØØ28) 21    CALL EXIT
(ØØ29)       END
```

B. RUN AND COMPILE

```
OK, FTN X7 ØØØØ ERRORS (FTN-1Ø82.L11)  OK, LOAD
GO
$ LØ B_X7
$ LI
LC
$ SA *X7
$ QU

OK, R*X7
GO
```

C. PROGRAM OUTPUT

```
HOW MANY BOXES?
3
ENTER LENGTH:
3.
ENTER WIDTH:
5.  ENTER HEIGHT:
6.
LGTH:  3.000 WI:  5.000 HT:  6.000 VOL:  90.000


ENTER LENGTH:
5.
ENTER WIDTH:
7.
ENTER HEIGHT:
9.
LGTH:  5.000 WI:  7.000 HT:  9.000 VOL:  315.000


ENTER LENGTH:
5.
ENTER WIDTH:
2.
ENTER HEIGHT:
9.
LGTH:  5.000 WI:  2.000 HT:  9.000 VOL:  90.000


I HAVE CALCULATED 3 BOXES

OK,
```

X8 FINDING THE SQUARE ROOT

A. PROGRAM

```
MAT01) C     THIS PROGRAM ILLUSTRATES A VARIETY OF READ/WRITE AND FOR
C       STATEMENTS
(0002) 5     WRITE(1,10)
(0003)       WRITE(1,20)
(0004)       READ(1,30)A
(0005)       WRITE(1,30)
(0006)       WRITE(1,40)
(0007)       READ(1,50)B
(0008)       WRITE(1,60)
(0009)       READ(1,70)C
(0010)       R=SQRT(A**(B-C+2))
(0011)       WRITE(1,90)A
(0012)       WRITE(1,100)B
(0013)       WRITE(1,110)C
(0014)       WRITE(1,80)R
(0015)       GO TO 5
(0016) 10    FORMAT(/'FIND YOUR SQRT OF FUNCTION A')
(0017) 20    FORMAT('INPUT A     ' )
(0018) 30    FORMAT(F5.3)
(0019) 40    FORMAT(/'INPUT B')
(0020) 50    FORMAT(F5.3)
(0021) 60    FORMAT(/'INPUT C')
(0022) 70    FORMAT(F5.3)
(0023) 90    FORMAT(//'A='F5.3)
(0024) 100   FORMAT('B='F5.3)
(0025) 110   FORMAT('C='F5.3)
(0026) 80    FORMAT(//'ANSWERIS:'F7.3)
(0027)       CALL EXIT
(0028)       END
```

B. COMPILE & RUN

```
OK, FTN X8
0000 ERRORS (FTN-1082,L11)

OK, LOAD
GO
$ LO B_X8
$ LI
LC
$ SA *X8
$ QU

OK, R *X8
GO
```

C. OUTPUT

```
FIND YOUR SQRT OF FUNCTION A
INPUT A
9.5

INPUT B
8.7

INPUT C
7.7


A= 9.500
B= 8.700
C= 7.700

ANSWER IS:  29.281
```

X9 IF EXAMPLE

This example illustrates the use of IF statements and the corresponding output. The following program examines each given number to determine if the number is negative, zero or positive. With the appropriate GO TO statement, it directs the sequence to the appropriate output.

A. PROGRAM

```
(0001) C     IF STATEMENT EXAMPLE.
(0002) C     *THIS IS AN EXAMPLE DEMONSTRATING THE ARITHMETIC IF
       C     STATEMENT*
(0003)       DO 400 I=1,5
(0004)       WRITE(1,5)
(0005) 5     FORMAT('GIVE ME A NUMBER'//)
(0006)       READ(1,10) Y
(0007) 10    FORMAT(I5)
(0008)       IF (Y) 100,310,320
(0009) 100   WRITE(1,35)
(0010) 35    FORMAT('YOU GAVE A NEGATIVE NUMBER!!'//)
(0011)       GO TO 400
(0012) 310   WRITE(1,40)
(0013) 40    FORMAT('YOU GAVE A ZERO!!'//)
(0014)       GO TO 400
(0015) 320   WRITE(1,25)
(0016) 25    FORMAT('YOU GAVE A POSITIVE NUMBER!!'//)
(0017) 400   CONTINUE
(0018) 900   CALL EXIT
(0019)       END
```

B. COMPILE & RUN

```
OK, FTN X9
0000 ERRORS (FTN-1082.L11)
OK, LOAD
GO
$ L0 B_X9
$ LI
LC
$ SA *X9
$ QU

OK, R*X9
GO
```

C. OUTPUT

```
GIVE ME A NUMBER

45
YOU GAVE A POSITIVE NUMBER!!

GIVE ME A NUMBER

-99
YOU GAVE A NEGATIVE NUMBER!!

GIVE ME A NUMBER

Ø
YOU GAVE A ZERO!!

GIVE ME A NUMBER
YOU GAVE A POSITIVE NUMBER!!

GIVE ME A NUMBER

19
YOU GAVE A POSITIVE NUMBER!!
```

X1Ø COMPUTED GO TO EXAMPLE

A. PROGRAM

```
(ØØØ1) 31Ø   X =48
(ØØØ2)       B = I*5
(ØØØ3)       C = 5-I
(ØØØ4)       I=3
(ØØØ5) 2Ø    GO TO(1ØØ,31Ø,32Ø),I
(ØØØ6) 32Ø   A = B + C
(ØØØ7)       I=1
(ØØØ8)       GO TO 2Ø
(ØØØ9) 1ØØ   Y = A*X
(ØØ1Ø)       WRITE (1,11Ø)Y
(ØØ11) 11Ø   FORMAT(I5)
(ØØ12)       CALL EXIT
(ØØ13)       END
```

B. COMPILE & RUN

```
OK, FTN X1Ø
ØØØØ ERRORS (FTN-1Ø82-L11)
OK, LOAD
GO
$ LO B_X1Ø
$ LI
LC
$ SA *X1Ø
$ QU

OK, R *X1Ø

GO
```

C. OUTPUT

```
24Ø

OK,
```

X11 SIMPLE CALCULATIONS

This example illustrates the relationships of simple input and output statements.

A. PROGRAM

```
A0001) C      *THIS PROGRAM WILL CALCULATE THE AREA OF A RECTANGLE OR
       C      TRIANGLE*
(0002) 5      FORMAT('ENTER BASE:')
(0003)        WRITE(1,5)
(0004)        READ(1,6)BASE
(0005) 6      FORMAT(F6.3)
(0006) 10     FORMAT('ENTER HEIGHT:')
(0007)        WRITE(1,10)
(0008)        READ(1,12)HEIGHT
(0009) 12     FORMAT(F6.3)
(0010) 15     AREA=BASE*HEIGHT
(0011)        WRITE(1,20) BASE,HEIGHT,AREA
(0012) 20     FORMAT('BA:',3X,F6.3,5X, 'TIMESHT:'3X,F6.3,5X,'= AREA:
       C      F10.3)
(0013)        CALL EXIT
(0014)        END
```

B. COMPILE & RUN

```
OK, FTN X11
0000 ERRORS (FTN-1082.L11)
OK, LOAD
GO
$ LO B_X11
$ LI
LC
$ SA *X11
$ QU

OK, R *X11
GO
```

C. OUTPUT

```
ENTER BASE:
3.0098
ENTER HEIGHT:
9.435
BA:  3.010 TIMES HT:  9.435 = AREA:  28.397
```

X12 OPENING & CLOSING FILES

The following example illustrates how files are opened and closed in a FORTRAN program.

A. PROGRAM

```
(0001)        DOUBLE PRECISION ANSWER(10)
(0002)        CALL CONTRL(1,'DATA   ',6,$20)
(0003)        READ(6,10,END=100)(ANSWER(I),I=1,7)
(0004)        GO TO 102
(0005) 10     FORMAT(E20.7/E20.7,6X,E20.7,6X,E20.7/E20.7,6X,E20.7/E20.7)
(0006) 102    WRITE(1,5) (ANSWER(I),I=1,7)
(0007) 5      FORMAT( 'HERE IS YOUR DATA   ' //(E20.7,5X,E20.7,5X,E20.7))
(0008)        GO TO 200
(0009) 20     CALL TNOU('ERROR',5)
(0010)        GO TO 200
(0011) 100    WRITE(1,101)
(0012) 101    FORMAT('END OF FILE READ BEFORE DATA TERMINATED')
(0013) 200    CALL CONTRL(4,'DATA   ',6)
(0014)        CALL EXIT
(0015)        END
```

B. DATA

```
OK, SLIST DATA GO
1234567890987.7654321
2976546789075.7654321    -987654321234.9876543    -0.0005678
1234567890123.1234567    1234567890123.1234567
1234567890123.1234567
```

C. COMPILE & RUN

```
OK, FTN X12
0000 ERRORS (FTN-1082.L11)
OK, LOAD
GO
$ LO B_X12
$ LI
LC
$ SA *X12
```

```
$ QUIT

OK, R *X12
GO
```

D. OUTPUT

```
HERE IS YOUR DATA

 Ø.1234568E 13                 Ø.2976547E 13-Ø.9876543E 12
-Ø.5678ØØØE-Ø3                 Ø.1234568E 13 Ø.1234568E 13

OK,
```

X13 EXAMPLE OF SUBROUTINE CALLS

The following example illustrataes how subroutines are called. The purpose of this program is to print out the time when the program was started, the time the program ended and the time used. The actual time to run this program was 68 ticks. Two subroutines were called: SEARCH and TIMDAT. SEARCH (line 0007) opens the file called 'OUTPUT' and the TIMDAT routine calculates the actual time.

A. PROGRAM

```
(0001)         INTEGER  OUTPUT
(0002)         INTEGER UTM,UTS,UTT,UCS,UCT,UPS,UPT
(0003)         INTEGER STM,STS,STT,SCS,SCT,SPS,SPT
(0004)         INTEGER TTM,TTS,TTT,TCS,TCT,TPS,TPT
(0005)         DIMENSION IAREA(9),ITIME(15)
(0006)         DATA IAREA/1,2,3,4,5,6,7,8,9/
(0007)         CALL SEARCH(2,'OUTPUT',2,$10)
(0008)         CALL TIMDAT(ITIME,15)
(0009)         UTM=ITIME(4)
(0010)         UTS=ITIME(5)
(0011)         UTT=ITIME(6)
(0012)         UCS=ITIME(7)
(0013)         UCT=ITIME(8)
(0014)         UPS=ITIME(9)
(0015)         UPT=ITIME(10)
(0016)         WRITE(6,1)(ITIME(I),I=1,15)
(0017) 1       FORMAT('DATE:',10X,2A2,A1,/
(0018)         *'TIME MINUTES:',2X,I5,/
(0019)         *'TIME SECONDS:',2X,I5,/
(0020)         *'TIME TICKS:',4X,I5,/
(0021)         *'CPU SECS USED:',1X,I5,/
(0022)         *'CPU TICKS USED:',I5,/
(0023)         *'PAGING SECONDS:',I5,/
(0024)         *'PAGING TICKS:',2X,I5,/
(0025)         *'TICKS PER SEC:',1X,I5,/
(0026)         *'USER NUMBER:',3X,I5,/
(0027)         *'USER NAME:',5X,3A2)
(0028)         DO 90 K=1,10
(0029) 90      WRITE(6,2)IAREA
(0030) 2       FORMAT(9I2)
(0031)         CALL TIMDAT(ITIME,15)
(0032)         STM=ITIME(4)
(0033)         STS=ITIME(5)
(0034)         STT=ITIME(6)
(0035)         SCS=ITIME(7)
(0036)         SCT=ITIME(8)
(0037)         SPS=ITIME(9)
(0038)         SPT=ITIME(10)
(0039)         WRITE(6,1)(ITIME(I),I=1,15)
```

```
(0040)      TTM=STM-UTM
(0041)      TTS=STS-UTS
(0042)      TTT=STT-UTT
(0043)      TCS=SCS-UCS
(0044)      TCT=SCT-UCT
(0045)      TPS=SPS-UPS
(0046)      TPT=SPT-UPT
(0047)      WRITE(6,3)TTM,TTS,TTT,TCS,TCT,TPS,TPT
(0048)      CALL SEARCH(4,'OUTPUT',2)
(0049) 3    FORMAT('TOTAL TIME MINUTES:',I5/
(0050)      *       'TOTAL TIME SECONDS:',I5/
(0051)      *       'TOTAL TIME TICKS:   ',I5/
(0052)      *       'TOTAL CPU SECONDS:  ',I5/
(0053)      *       'TOTAL CPU TICKS:    ',I5/
(0054)      *       'TOTAL PAGING SECS:  ',I5/
(0055)      *       'TOTAL PAGING TICKS:',I5)
(0056)      CALL EXIT
(0057) 10   CALL TNOU('PROBLEMS WITH OPENING FILE',25)
(0058)      END
0000 ERRORS (FTN-1082.L11)
```

B. OUTPUT

```
DATE:             05046
TIME MINUTES:     808
TIME SECONDS:     33
TIME TICKS:       169
CPU SECS USED:     19
CPU TICKS USED:   82
PAGING SECONDS:   28
PAGING TICKS:     135
TICKS PER SEC:    330
USER NUMBER:        9
USER NAME:        DAWES
 1 2 3 4 5 6 7 8 9
 1 2 3 4 5 6 7 8 9
 1 2 3 4 5 6 7 8 9
 1 2 3 4 5 6 7 8 9
 1 2 3 4 5 6 7 8 9
 1 2 3 4 5 6 7 8 9
 1 2 3 4 5 6 7 8 9
 1 2 3 4 5 6 7 8 9
 1 2 3 4 5 6 7 8 9
 1 2 3 4 5 6 7 8 9
DATE:             05046
TIME MINUTES:       808
TIME SECONDS:      33
TIME TICKS:       237
CPU SECS USED:     19
CPU TICKS USED:150
```

```
PAGING SECONDS: 28
PAGING TICKS:  135
TICKS PER SEC: 330
USER NUMBER:     9
USER NAME:      DAWES
TOTAL TIME MINUTES:  0
TOTAL TIME SECONDS:  0
TOTAL TIME TICKS: 68
TOTAL CPU SECONDS:  0
TOTAL CPU TICKS: 68
TOTAL PAGING SECS:  0
TOTAL PAGING TICKS:  0
```

X14 INTEGER/DO LOOP EXAMPLE

The object of this example is to illustrate a simple DO loop and to show the relationship of the format statement and actual output.

A. PROGRAM

```
(0001)        INTEGER BUFF(18,3)
(0002)        NUM=1
(0003)        L=1
(0004)        M=2
(0005)        N=3
(0006)        DO 10 I=1,18
(0007)        DO 20 II=1,3
(0008)        BUFF(I,II)=NUM
(0009)        NUM=NUM+1
(0010) 20     CONTINUE
(0011) 10     CONTINUE
(0012)        DO 30 I=1,18
(0013)        WRITE(1,15)BUFF(I,L),BUFF(I,M),BUFF(I,N)
(0014) 30     CONTINUE
(0015) 15     FORMAT(3I3)
(0016)        CALL EXIT
(0017)        END
```

B. RUN & COMPILE

```
OK, FTN X14
0000 ERRORS (FTN-1082.L11)
OK, LOAD
GO
$ LO B_X14
$ LI
LC
$ SA *X14
$ QU

OK, R *X14
```

C. OUTPUT

```
 1              2              3
 4              5              6
 7              8              9
10             11             12
13             14             15
```

```
16            17            18
18            20            21
22            23            24
25            26            27
28            29            30
31            32            33
34            35            36
37            38            39
40            41            42
43            44            45
46            47            48
49            50            51
52            53            54

OK,
```

X15 OVERLAY EXAMPLE

The following example illustrates a simple program overlay. Program B in this example overlays program A. B is called using a CALL RESUME routine after program A has been run. Each program is compiled separately.

A. PROGRAM

```
(0001) C     PROGRAM A
(0002)       COMMON /A/P1D,IC1,IC2,IC3,IC4,IC5
(0003)       INTEGER P1D(10),P2D(10)
(0004)       DATA P2D/1HA,1HB,1HC,1HD,1HE,1HF,1HG,1HH,1HI,1HJ/
(0005)       WRITE(1,100)
(0006) 100   FORMAT('START PROGRAM A')
(0007)       DO 5 I=1,10
(0008) 5     P1D(I)=P2D(I)
(0009)       WRITE(1,1)
(0010) 1     FORMAT('INPUT ANY FIVE CHARACTERS')
(0011)       READ(1,2)IC1,IC2,IC3,IC4,IC5
(0012) 2     FORMAT(5A1)
(0013)       WRITE(1,3)P1D
(0014) 3     FORMAT(10(1X,A1))
(0015)       WRITE(1,4)IC1,IC2,IC3,IC4,IC5
(0016) 4     FORMAT(5(1X,A1))
(0017)       WRITE(1,101)
(0018) 101   FORMAT('END PROGRAM A')
(0019)       CALL RESUME('PROGB ')
(0020)       CALL EXIT
(0021)       END
0000 ERRORS (FTN-1082.L11)
```

```
(0001) C     PROGRAM B
(0002)       COMMON /A/ P1D,IC1,IC2,IC3,IC4,IC5
(0003)       INTEGER P1D(10)
(0004)       WRITE(1,100)
(0005) 100   FORMAT('START PROGRAM B')
(0006)       WRITE(1,1)
(0007) 1     FORMAT('OUTPUT FROM PROGRAM B')
(0008)       WRITE(1,2)IC1,IC2,IC3,IC4,IC5
(0009) 2     FORMAT(5(1X,A1))
(0010)       WRITE(1,3)P1D
(0011) 3     FORMAT(10(1X,A1))
(0012)       WRITE(1,4)
(0013) 4     FORMAT('END PROGRAM B')
```

```
(0014)       CALL EXIT
(0015)       END
0000 ERRORS (FTN-1082.L11)
```

C. OUTPUT

```
OK, R PROGA
GO
START PROGRAM A
INPUT ANY FIVE CHARACTERS
DUMMY
 A B C D E F G H I J
 D U M M Y
END PROGRAM A
START PROGRAM B
OUTPUT FROM PROGRAM B
 D U M M Y
 ABCDEFGHIJ
END PROGRAM B

OK,
```

X16 ASSIGN STATEMENT

A. PROGRAM

```
GO     EXAMPLE 16
C      *THIS IS AN EXAMPLE DEMONSTRATING THE ASSIGN STATEMENT*
       ASSIGN 320 TO I
20     GO TO I,(100,310,320)
       ASSIGN 310 TO I
       GO TO 20
100    WRITE (1,35)
35     FORMAT('I = 100')
310    WRITE(1,40)
40     FORMAT('I = 310')
320    WRITE(1,25)
25     FORMAT('STATEMENT 320 RSULTED IN THIS ACTION')
       CALL EXIT
       END
```

B. COMPILE & RUN

```
OK,
     FTN17
GO
0000 ERRORS (FTN-1082.L11)

OK,

     LOAD
GO
$ LO B X18
$ LI
LC
$ SA *X17
$ QU

OK, R *X17
GO
```

C OUTPUT

```
STATEMENT 320 RESULTED IN THIS ACTION

OK,
```

APPENDIX E

PRIMOS SUBROUTINES SUMMARY

The following table summarizes most of the PRIMOS subroutines. For a detailed description of each subroutine, refer to the PRIMOS II and III interactive users guide (MAN 2602).

FUNCTION	SUBROUTINE	Calling Sequence
ATTACH to a UFD	ATTACH	CALL ATTACH (Ufd,Ldisk, Passwd,Key,Altrtn)
To interrupt a running program	BREAK$	CALL BREAK$(.TRUE.) or CALL BREAK$(.FALSE.)
Command line READ	CMREAD	CALL CMREAD(Array)
Rename File	CNAME$	CALL CNAME$(Oldnam, Newnam,Altrtn)
Reads a Command Line	COMANL	CALL COMANL
Allows PRIMOS to read commands from a file	COMINP	CALL COMINP(Name,Funit, Altrtn)
Compares left-most characters of 2 strings	COMEQV	Value-COMEQV (String1,String2)
Gets next character from terminal	C1IN	CALL C1IN(Char)
Initialize disk devices	D$INIT	CALL D$INIT(Pdisk)
Sets system vector then takes an alternative return	ERRSET	CALL ERRSET(Altval, Altrtn,Messag,Num) CALL ERRSET(Altval, Altrtn,Name,Messag,Num)
	ERRSET	CALL ERRSET(Altval, Altrtn)
Return to PRIMOS	EXIT	CALL EXIT
Updates disk	FORCEW	CALL FORCEW(0,Funit)

Moves words from ERRVEC into Xervec	GETERR	CALL GETERR(Xervec,n)
Moves characters from buffer	GETWRD	CALL GETWRD(Buff,Array)
Moves n words in Xervec	GINFO	CALL GINFO(Xervec,n)
Compares two 6-character names	NAMEQV	Value=NAMEQV(name1, name2)
Prints error message on user terminal	PRERR	CALL PRERR
Copies a file	PRWFIL	CALL PRWFIL(Key, Funit, Pbuff,Nwords,Position, Altrtn)
Inserts characters into a buffer to form a six-character name	PUTC	CALL PUTC(Buf,Char)
Tells system to cycle next user	RECYCL	CALL RECYCL
Reads up to 80 characters	RDCOM	CALL RDCOM(Buf)
Inverse of the SAVE	RESTOR	CALL RESTORE(Rvec,Name, Altrtn)
Same as RESUME command	RESUME	CALL RESUME(Filname)
READS one disk record	RREC	CALL RREC(Bptrs,Blen, N,Ra,Pdisk,Altrtn)
SAVEs disk file	SAVE	CALL SAVE(Vect, Filename)
Connects a file to a file unit	SEARCH	CALL SEARCH(Key,Name, Funit,Altrtn)
Reads a char and echoes	T1IN	CALL T1IN(Char)
Supplies time information	TIMDAT	CALL TIMDAT(Array,Num)
Reads a number from a terminal	TIOCT TIDEC TIMEX	CALL TIOCT, TIDEC, TIMEX(Num)
Moves a card of data	T$CMPC	CALL T$CMPC(Unit,Buffer-

from Card Reader		Address,Word-Count, Instruction,Status Vector)
One line of data from terminal to line printer	T$LMPC	CALL T$LMPC(Unit,Buffer-Address,Word-Count, Instruction,Status-Vector)
Magnetic tape input or output, read a 'Raw' tape	T$MT	CALL T$MT(Unit,Buffer-Address,Word-Count, Instruction,Status-Vector)
Provides control over a SYN multi-line communications device	T$SLC	CALL T$SLC(Key,Line, Loc,(block),Mwds)
Gould Printer/Plotter control	T$VG	CALL T$VG(Unit,Loc (buffer),words,inst,statv)
Types out character	T1OU	CALL T1OU(Char)
Prints characters from array with carriage return	TNOU	CALL TNOU(Array,Nchars)
Types octal converted ASCII number	TOOCT	CALL TOOCT(Number)
Prints characters from array-no carriage return	TNOUA	CALL TNOUA(Array,Nchar)
Updates current UFD	UPDATE	CALL UPDATE(Key,1,Ø)
Writes record onto disk	WREC	CALL WREC(Bptrs,Blen, N,Ra,Altrtn)

APPENDIX F

SUMMARY OF IOCS SUBROUTINES

The following tables summarize the input and output device handling subroutines. Refer to Section 6 of the Software Library User Guide (MAN 1880) for detailed information. Because this summary does not include the respective rules for each element in subroutine calls, it is necessary that the Software Library User Guide be consulted before using these subroutines.

DEVICE OPERATION	FUNCTION	SUBROUTINE CALL
Disk files	Open named file for reading	CALL SEARCH(1,'Name',Funit, Altrtn)
Disk files	Open named file for writing	CALL SEARCH(2,'Name',Funit, Altrtn)
Disk files	Open named file for both reading and writing	CALL SEARCH(3,'Name',Funit,
Disk files	Close named file	CALL SEARCH(4,'Name',Funit, Altrtn)
Disk files	Delete named file	CALL SEARCH(5,'Name',Funit, Altrtn)
Disk files	Rewind file (repositions to first record of FUNIT)	CALL SEARCH(7,'Name',Funit, Altrtn)
Disk files	Truncate named file	CALL SEARCH(8,'Name',Funit, Altrtn)
Disk files	Performs a tree search(performs ATTACH and SEARCH calls) needed to open the specified file. Use SEARCH function numbers.	CALL TRSRCH(Func,'Name', funit,Altrtn)

Disk files	Read or write disk file	CALL PRWFIL (Key, Funit, Pbuffer,Nwords,Position,Altrtn)
Write Disk files	Write ASCII data from Buff onto a disk file(in compressed ASCII format)opened on unit.	CALL O$AD07(Unit,buff, Count,Altrtn)
Write Disk files	Write ASCII data from Buff onto a disk file (in fixed length records).	CALL O$AD08(Unit,Buff, Count,Altrtn)
Read Disk files	Read ASCII data from file open on unit	CALL I$AD07)Unit,Buff, Count,Altrtn)
Write Disk files	Write Binary data to a file opened on unit	CALL O$BD07(Unit,Buff, Count,Altrtn)
Moves Raw Data	Move raw data from the terminal, or command file n to user program's address space	CALL CNIN$(Buffer,Char-Count,Actual Count)
GETA Terminal	Gets next character from the terminal data	CALL C1IN(char)
ATTACH DEVICE	Attaches specified device by initializing both LUTEL, associating logical device to physical device.	CALL ATTDEV(logical-device, physical-device,unit, FTN-Buff-size) where FTN-Buf-Size is the maximum size of the I/O Buffer. Default is 120 bytes (240 bytes with VFTN)
ATTACH DEVICE	Attaches logical units 1-5 under control of the rightmost five octal digits of the argument to SETIOS, FLAG	CALL SETIOS(FLAG)
WRITE DEVICE	Contents of Buff (ASCII data) moved from memory to named output device.	CALL WRASC(logical-device, Buff,Count,Altrtn)

READ DEVICE	Fetches one (ASCII) record into memory	CALL RDASC(logical-device, Buff,Count,Altrtn)
WRITE DEVICE	Count of words of Buff are written to the specified named output device	CALL WREIN(logical-device Buff,,Count,Altrtn)
READ DEVICE	Count of words of Buff are initialized to zero.	CALL RDBIN(logical-device, Buff,Count,Altrtn)
SEARCH DISK	Calls SEARCH with same arguments	CALL CONTRL(Key,Name, Logical-device,Altrtn)
POSITION DISK	Allows direct positioning to any record in a disk.	CALL POSFIL(Logical-device, Rec,Altrtn)
USER TERMINAL	Outputs ASCII data to user terminal or ASR Punch.	CALL O$AA01(sub-unit, Buff,Count,Altrtn)
CONTROL USER TERMINAL	Close Device	CALL C$A01(1,Name,Unit,Altrtn)
CONTROL USER TERMINAL	Turn on Punch and Punch Leader	CALL C$A01(2,Name,Unit,Altrtn)
CONTROL USER TERMINAL	Punch Trailer	CALL C$A01(4,Name,Unit,Altrtn)
CONTROL H.S. Paper Tape Reader or Punch	Close Device	CALL C$P02(1,Name,Unit,Altrtn)
CONTROL H.S. Paper Tape Reader or Punch	Turn on Punch 1& Punch Leader	CALL C$P02(12,Name,Unit,Altrtn)
CONTROL H.S. Paper Tape Reader or Punch	Punch trailer	CALL C$P02(4,Name,Unit,Altrtn)

CONTROL 9-track mag tape	Open for Read	CALL C$MØ5(1,Name,Unit,Altrtn)
READ USER TERMINAL	Input ASCII data from user terminal or ASR reader	CALL I$AAØ1(Sub unit,Buff, Count,Altrtn)
READ USER TERMINAL OR STORAGE DEVICE	Input ASCII data from the command stream(either from a user terminal or from a command file)	CALL I$AA1(Sub unit,Buff, Count,Altrtn)
WRITE TERMINAL	Outputs binary data to an ASCR Punch	CALL O$BAØ1(Sub unit,Buff, Count,Altrtn)
WRITE H.S.Paper Tape Punch	Outputs ASCII data to the H.S.Punch	CALL O$APØ2(Sub unit,Buff, Count,Altrtn)
READ H.S.Paper Reader	Inputs ASCII data from the H.S.	CALL I$APØ2(Sub unit,Buff, Count,Altrtn)
WRITE H.S.Paper Tape Punch	Outputs binary data to the H.S.Punch	CALL O$BPØ2(Sub unit,Buff Count,Altrtn)
CONTROL LINE PRINTER	Centronics line Printer	CALL Ø$ALØ4(Unit,Buff, Count,Altrtn)
CONTROL Line Printer	Parallel Interface line printer	CALL O$ALØ6(Unit,Buff, Count,Altrtn)

DEVICE OPERATION	FUNCTION	CALLING SEQUENCE
READ H.S. Paper Tape reader	Input 1 character to the A register	CALL P1IB(Char)
WRITE H.S. Paper Tape Punch	Output 1 character from A register to punch	CALL P1ØB(Char)
READ H.S. Paper Tape Reader	Input 1 character from paper tape	CALL P1IN(Char)

WRITE H.S. Punch	Output 1 character to H.S.Punch	CALL P1OU(Char)
Write User Terminal	Outputs Count characters to the user terminal	CALL TNOU(Buff,Count) CALL TNOUA(Buff,Count)
Read user terminal	Reads one character from the user terminal to the A register	CALL T1IB(Char)
WRITE user terminal	Writes one character from A register to user terminal	CALL T1OB(Char)
Read user terminal	Reads one character from user terminal (New line for CR)	CALL T1IN(Char)
Control Line Printer	Versatec Printer/ Plotter	CALL O$AL14(Unit,Buff, Count,Altrtn)
Read Card Reader	Reads input from parallel card	CALL I$AC03(Unit,Buff-Address, Name,Word-Count,Altrtn)
Read Card reader	Reads input from serial card Reader	CALL I$AC09(Unit,Buffer-Name, Word-Count,Altrtn)
Writes User terminal	Writes one character to the users terminal	CALL T1OU(Char)
Read Card Reader	Raw data mover from MPC card reader to user's spare	CALL T$CMPC
Read or Write a Mag tape	Moves Raw Data between Mag tape and user address space	CALL T$MT(Unit,Pba,Word-Count, instruction, statv)
User terminal to line printer	Moves information from the user to the line printer	CALL T$LMPC(Unit,Buffer-Address,word-count,instruction statv)

Raw Data to Versatec Printer/ Plotter	Moves data from Buffer to Versatec Printer	CALL T$VG(unit,LOC(Buff), Nwds,instruction,statv)

MAGNETIC TAPE OPERATIONS

DEVICE	FUNCTION	CALLING SEQUENCE
CONTROL 9-track mag tape	Open for Write	CALL C$MØ5(2,Name,Unit, Altrtn)
CONTROL 9-track mag tape	Open for Read & Write	CALL C$MØ5(3,Name,Unit,Altrtn)
CONTROL 9-track mage tape	Rewind and Close file	CALL C$MØ5(4,Name,Unit,Altrtn)
9-track Tape (ASCII & BINARY)	Write ASCII	CALL Ø$AMØ5
	Read ASCII	CALL I$AMØ5
	Write Binary	CALL O$BMØ5
	Read Binary	CALL I$BMØ5
	Control see Key Functions)	CALL C$MØ5(Key,Name,Altrtn)
9-Track Tape EBCDIC & BINARY	Write EBCDIC	CALL O$AM13
	Read EBCDIC	CALL I$AM13
	Write Binary	CALL O$BM13
	Read Binary	CALL I$BM13

	Control (See Key functions)	CALL C$M13(Key,Name,Unit, Altrtn)
7-Track Tape ASCII & BINARY	Write ASCII	CALL O$AM1Ø
	Read ASCII	CALL I$AM1Ø
	Write Binary	CALL O$BM1Ø
	Read Binary	CALL I$BM1Ø
	Control (see Key functions)	CALL C$M1Ø(Key,Name,Unit, Altrtn)
Control 7-track mag tape. ASCII Drivers Read BCD data format	Open for Read	CALL C$M1Ø(1,Name,Unit,Altrtn)
Control 7-track mag tape. ASCII drivers read BCD data format	Open for Write	CALL C$M1Ø(2,Name,Unit,Altrtn)
Control 7-track mag tape. ASCII drivers read BCD data format	Open for Read & Write	CALL C$M1Ø(3,Name,Unit,Altrtn)
Control 7-track mag tape. ASCII drivers Read BCD data format	Rewind & Close file	CALL C$M1Ø(4,Name,Unit,Altrtn)
SEARCH Mag tape	Open for Read	CALL SEARCH(1,Name,Unit,Altrtn)
SEARCH Mag tape	Open for Write	CALL SEARCH(2,Name,Unit,Altrtn)
SEARCH	Open for Read	CALL SEARCH(3,Name,Unit,Altrtn)

Mag tape	& Write	
SEARCH Mag Tape	Rewind & Close file	CALL SEARCH(4,Name,Unit,Altrtn)
7-track tape BCD & Binary	Write BCD	CALL O$AM11
	Read BCD	CALL I$AM11
	Write Binary	CALL O$BM11
	Read Binary	CALL I$BM11
	Control (see Key functions)	CALL C$M11(Key,Name,Unit Altrtn)

Key Functions:

Since the subroutines are similar, they will be described in groups.

C$M05 (Key, Name, Unit, Altrtn)
C$M10
C$M11

Select Key for appropriate operation:

Key = -4 for Rewind to BOT (Beginning of Tape)
-3 for Backspace one file mark
-2 for Backspace one record
-1 for Write file mark
1 for Open to read
2 for Open to write
3 for Open to read/write
4 for Close (Write file mark and rewind)
5 for Move forward one record
6 for Move forward one file mark
7 for Rewind to BOF (Beginning of File)
8 for Select device and read status

Name = Not Applicable (may be anything)

Unit = 0, 1, 2, or 3 (Depending on which device is ASSIGNed)

Altrtn = Is the alternate return. If Altrtn=0, it means that alternate return is not desired.

APPENDIX G

FORTRAN LIBRARY FUNCTIONS

The following list of functions are available to perform a variety of mathematical operations. Refer to Section 2, Prime Software Library Users Guide (MAN188Ø) for more details on the FORTRAN library functions (e.g., arguments).

DESCRIPTION	FUNCTION
Compute absolute value of SP number giving SP result	ABS
Convert the imaginary part of a CP number to an SP number	AIMAG
Truncate fractional bits of an SP number	AINT
Compute logarithm (base e) of an SP number	ALOG
Compute logarithm (base 1Ø) of an SP number giving an SP number	ALOG1Ø
Find maximum of a list of integers	AMAXØ
Find maximum of a list of SP numbers	AMAX1
Find a minimum of a list of integers	AMINØ
Find minimum of a list of SP numbers	AMIN1
Compute remainder of	AMOD

quotient of two SP numbers	
Compute the principal value of the arctangent of a SP number	ATAN
Compute the principal value of the arctangent of an SP number divident byu an SP number	ATAN2
Compute the absolute	CABS
Compute cosine of a CP number	CCOS
Compute the exponential of a CP number	CEXP
Convert two SP numbers to a CP number	CMPLX
Compute the conjugate of a CP number	CONJ
Compute the cosine of an SP number	COS
Compute sine of a CP number	CSIN
Compute the square root of a CP number	CSQRT
Compute the absolute value of a DP number	DABS
Compute the principal value of the arctangent of a DP number	DATAN
Compute the principal value of the arctangent of a DP number divided by another	DATAN2
Convert SP number to DP number	DBLE
Compute the cosine of a DP number	DCOS

Compute exponential of a DP number	DEXP
Compute positive difference of two SP numbers	DIM
Truncate fractional	DINT
Compute logarithm (base e) of a DP number giving a DP number	DLOG
Compute logarithm (based 2) of a DP number	DLOG2
Compute logarithm (base 1Ø) of a DP number	DLOG1Ø
Find MAX of a variable list of DP numbers	DMAX1
Find MIN of a variable list of DP numbers	DMIN1
Compute the remainder of a DP number divided by another DP number	DMOD
Combine magnitude of a DP number and the sign of another DP number	DSIGN
Compute the sine of a DP number	DSIN
Compute the square root of a DP number	DSQRT
Compute exponential of SP number giving SP result	EXP
Convert integer to SP number	FLOAT
Compute positive difference of two	IDIM

integers	
Convert DP number to an integer	IDINT
Convert user specified SP number to an integer	IFIX
Convert an SP number number to an integer	INT
Invoke REAL random number generator giving integer result	IRND
Combine magnitude of an integer with sign of another integer	ISIGN
Shift an integer left by a specified number of bits (i.e., left shift	LS
Save the specified number of left most bits of an integer (i.e., left-truncated	LT
Find maximum of a variable list of integers	MAXØ
Finde maximum of a variable list of SP numbers	MAX1
Find minimum of a variable list of integers	MINØ
Find minimum of a variable list of SP numbers	MIN1
Compute the remainder of one integer divided by another	MOD
Perform a logical OR of 2 16-bit integers giving an integer result	OR

Convert real part of a CP number to an SP number	REAL
Invoke REAL random number generator giving SP result	RND
Shift an integer right by a specified number of bits (i.e., right shift	RS
Save a specified number of right-most bits of an integer (i.e., right truncate)	RT
Shift an integer by a specified number of bits	SHFT
Combine magnitude of an SP number and the sign of another SP number giving an SP result	SIGN
Compute the sine of a SP number	SIN
Convert a DP number to a SP fixed point number	SNGL
Compute square root of an SP number giving SP result	SQRT
Compute hyperbolic tangent of an SP number	TANH

APPENDIX H

FORTRAN MATH LIB SUMMARY

The Math Library contains subroutines to solve math problems such as determinants, permutations, and combinations. For more detailed information about each subroutine, refer to Section 9 of the Software Library User Guide (MAN 188Ø).

FUNCTION	CALL SEQUENCE
Sets the sqare matrix MAT equal to the N by N identity matrix.	CALL MIDN [DMIDN,IMIDN,CMIDN] (MAT,N)
Sets the N by M matrix MAT equal to the constant value CON.	CALL MCON [DMCON,IMCON,CMCON] (MAT,N,M,CON)
Sets the N by M matrix MATO equal to the scalar product of the N by M matrix MATI and the scalar constant SCON.	CALL MSCL [DMSCL,IMSCL,CMSCL] (MATO,N,M,SCON)
Sets the N by N square matrix MATD equal to the transpose of the N by N matrix MATI.	CALL MTRN [DMTRN,IMTRN,CMTRN] (MATO,MATI,N)
Sets the N by M matrix MATS equal to the matrix sum of the N by M matrices MAT1 and MAT2.	CALL MADD [DMADD,IMADD,CMADD] (MATS,MAT1,MAT2,N,M)
Sets the N by M matrix MATD equal to the matrix difference of the N by M matrices MAT1 and MAT2.	CALL MSUB [DMMSUB,IMSUB,CMSUB] (MATD,MAT1,MAT2,N,M)
Sets the N1 by N3 matrix	CALL MMLT [DMMLT,IMMLT,CMMLT]

MATP equal to the matrix product of the N1 by N2 matrix MATL (left) and the N2 by N3 matrix MATR (right).

(MATP,MATL,MATR,N1,N2,N3)

Sets N by N square matrix MATO equal to the inverse of the N by N matrix MATRI.

CALL MINV [DMINV,CMINV] (MATO,MATI,N,WORK,NP1, NPPN,IERR)

Sets N by N square matrix MATO equal to the adjoint of the N by n matrix MATI.

CALL MADJ [DMADJ,IMADJ,CMADJ[(MATO,MATI,N,IW1,IW2,IW3,IW4,IERR)

Sets DET equal to the determinant of the N by N square size N used as work arrays.

CALL MDET [DMDET,IMDET,CMDET] (DET,MAT,N,IW1,IW2,IW3,IW4,IERR)

Sets COF equal to the (I,J) signed cofactor of the square N by N matrix MAT.

CALL MCOF [DMCOF,IMCOF,CMCOF] (COF,MAT,N,IW1,IW2,IW3,IW4,I,J,IERR)

Sets the N by L column vector XVECT equal to the solutions (X1,X2,X3,..., Xn) of the system linear equations.

CALL LINEQ [DLINEQ,CLINEQ] (XVECT,YVECT,CMAT,,WORK<N<NP1,IERR)

PERM is a loopless algorithm for computing the next permutation of N elements (N>2) with a single interchange of adjacent elements.

CALL PERM (IPERM,N,IW1,IW2,IW3, LAST,RESTRT)

COMB is a algorithm (not loopless) for computing the next combination of NR out of N elements with single interchange of elements.

CALL COMB (ICOMB,N,NR,IW1,IW2, IW3,LAST,RESTRT)

APPENDIX I

FORTRAN COMPILER SUBROUTINES

SUBROUTINES INTERNAL TO FORTRAN The following programs are called by the compiler:

Subroutine	Function
F$TR	Performs the function of the FORTRAN TRACE routine.
F$RN	Read with no alternate returns.
F$RNX	Read with ERR= and END= alternate returns.
F$WN`	Write with no alternate returns.
F$WNX	Write with ERR= alternate return.
F$DN	Close (END-FILE) logical device specified.
F$FN	Provides backspace function to FORTRAN run-time programs.
F$BN	Rewinds logical device specified.
F$IO F$CB F$A1 F$A2 F$A3 F$A5 F$A6	Interprets the format last character by character.
F$IOBX	Checks record size.
F$CG	FORTRAN computed GOTO processor.
F$RA	Read ASCII, no alternate returns
F$RB	Read BINARY, no alternate returns
F$RAX	Read ASCII, with ERR= and END= alternate returns
F$RBX	Read BINARY, with ERR= and END= alternate returns
F$RX	Common rad handler.
F$WA	Write ASCII, no alternate returns
F$WB	Write BINARY, no alternate returns

F$WAX Write ASCII with ERR= and END= alternate returns
F$WBX Write BINARY, with ERR= and END= alternate returns
F$WX Common write handler
F$EN Encode statement processor
F$DE Decode statement processor
F$DEX Decode statement processor with ERR=
F$IOBF F$IO buffer definition
AC1
AC2 Storage locations to hold complex accumulator
AC3
AC4
AC5 Storage locations to hold error code
F$RTE FORTRAN RETURN statement processor
F$AT FORTRAN argument transfer subroutine
F$ATI FORTRAN argument transfer subroutine for
PROTECTED subroutine

INTRINSIC FUNCTIONS

The following subroutines are the FORTRAN library intrinsic function handlers:

F$LT Left truncate

F$RT Right truncate

F$LS Left shift

F$RS Right shift

F$SH General shift

F$OR Inclusive OR

FLOATING POINT EXCEPTIONS
The FLEX (and F$FLEX) subroutines are invoked by the compiler or system. This subroutine is the floating point exception interrupt processor. It determines the exception type, which may be:

Exponent overflow/underflow

Divide by zero

Store exception

Real-integer exception

APPENDIX J

INDICATOR/CONTROLS

SUMMARY

The Indicator and Control subroutines allow a program to test for error conditions and report errors to the front panel lights. For more details, see the Software Library User Guide (MAN 1880).

FUNCTION	CALLING SEQUENCE
Updates the sense light settings according to an argument A1: 1=ON; 0=OFF	CALL DISPLY (A1) CALL DISPLY (0) CALL DISPLY (1)
Check the overflow condition. If an error has occurred, A1 is set to 1. Otherwise it is set to 2	CALL OVERFL (A1)
Sets specified sense light ON or sets all sense lights OFF. Iff A1=0, all sense lights are reset to off.	CALL SLITE (A1) CALL SLITE (0)
Tests the setting of a sense light specified by the argument A1. The result of this test (1 for ON, 2 for OFF) is stored in the location specified by the argument R.	R=SLITET (A1)
Tests the setting of a sense switch specified by the argument A1. The result of this test (1 for ON, 2 for OFF) is stored in the location specified by the argument R.	R=SSWTCH (A1)

APPENDIX K

SUMMARY OF SORT ROUTINES

This appendix summarizes the SORT and SEARCH LIBRARY routines contained in Section 8 of the Software Library User Guide (MAN 1880).

TYPE OF SORT	CALLING SEQUENCE
Based on a non-threaded binary tree structure	CALL HEAP (PTABLE,NENTRY,NWRDS,FWORDS, NKWORDS,TARRAY,NPASS,ALTBP)
Partition exchange sort	QUICK (PTABLE,NENTRY,NWRDS,FWORD, NKWORDS,TARRAY,NPASS,ALTBP)
Diminishing increment sort. SHELL utilizes the straight insertion sort (INSERT) on each of its passes.	SHELL (PTABLE,NENTRY,NWORDS,FWORD,NKWORDS, NPASS,ALTBP)
Straight insertion sorting is based upon ´percolating´ each element into its final position.	INSERT (PTABLE,NENTRY,NWORDS,FWORD,NKWRDS, NPASS,ALTBP,INCR)
A simple interchange SORT.	BUBBLE (PTABLE,NENTRY,NWORDS,FWORD, NKWORDS,ARRAY,NPASS,ALTBP,INCR)
Binary Sort	BNSRCH (PTABLE,NENTRY,NWORDS,FWORD, NKWORDS,SKEY,FENTRY,INDEX,OPFLAG,ALTNF, ALTBP)

PARAMETERS	DESCRIPTION
PTABLE	Integer pointer to thefirst word of the table.
NENTRY	Numbers of table entries (not words).
NWORDS	Number of words per entry.
NKWORDS	Number of words in key field.

INDEX

$ 2-9

$INSERT 5-12

$INSERT STATEMENT 5-12

$N 7-12

'LIST' 5-7

ØSBDØ7 7-14

16-BIT INTEGERS 7-3

5-COLUMN INTEGER 6-9

A FIELD DESCRIPTOR 6-27

A INPUT 6-28

A OUTPUT 6-28

ABSOLUTE MEMORY ADDRESSES 7-3

ABSOLUTE RECORD 7-15

AC1 I-2

AC2 I-2

AC3 I-2

AC4 I-2

AC5 I-2

ACCESS FILES 8-2

ADDRESS CONSTANTS 3-8, 7-12

ADDRESS VALUE 7-18

ADVANTAGES OF FORTRAN 1-5

ALPHANUMERIC CHARACTERS 2-3

ALPHANUMERIC FIELD DESCRIPTOR 6-27

ALTERNATE METHODS OF DECLARING ARRAYS 5-2

AND 3-16, 7-2, 7-3

ANS 7-1Ø

ARCTANGENT G-2

AREA TRACE 5-11

AREA TRACE STATEMENT 5-12

ARG 7-2, 7-7, 7-19

ARGUMENT LIST 7-7, 7-1Ø

ARGUMENTS 7-1, 7-3, 7-18

ARITHMETIC ASSIGNMENT STATEMENTS 3-1

ARITHMETIC CONSTANTS 3-1

ARITHMETIC EXPRESSION 5-12

ARITHMETIC IF STATEMENT 4-3, 4-7

ARITHMETIC OPERATORS 3-12, 3-18

ARRAY 3-1Ø, 7-12

ARRAY DECLARATOR 2-2

ARRAY ELEMENTS 5-6, 6-28

ARRAY NAME 2-2, 5-5, 5-11, 6-5

ARRAY STORAGE ARRANGEMENT 3-11, 8-1

ARRAY X 7-2

ARRAY Y 7-2

ASCII 3-6, 6-1, 6-8

ASCII CHARACTERS 6-2, 6-28

ASCII CONSTANTS 3-7

ASCII DATA 3-7

ASCII FILES 7-14

ASCII-A INPUT 6-28

ASCII-A OUTPUT 6-28

ASSIGN STATEMENT 4-1, 4-2

ASSIGNED GO TO STATEMENT 4-2

ASSIGNMENT RULES 3-19

ASSIGNMENT STATEMENTS 3-1

ATTACH 8-2

ATTACH DEVICE F-2

ATTACHING TO ANOTHER USER FILE DIRECTORY 8-1

B FORMAT STATEMENT 6-35

BACKSPACE STATEMENT 6-39

BASIC TERMINOLOGY 2-1

BINARY FILES 7-14

BINARY RECORDS 6-38

BLANK CHARACTER 2-4

BLANK COMMON 5-6, 5-7

BLANK RECORDS 6-9

BLOCK DATA STATEMENT

2-1, 7-9, 7-12, 7-17

BLOCK DATA SUBPROGRAM 7-17

BLOCK NAME 5-7

BODY OF SUBPROGRAM 7-9

BODY OF SUBROUTINE 7-12

BRIEF CALLING STATEMENT 7-1

CALL ABS G-1

CALL AIMAG G-1

CALL AINT G-1

CALL ALOG G-1

CALL ALOG1Ø G-1

CALL AMAXØ G-1

CALL AMAX1 G-1

CALL AMINØ G-1

CALL AMIN1 G-1

CALL AMOD G-2

CALL ATAN G-2

CALL ATAN2 G-2

CALL CABS G-2

CALL CCOS G-2

CALL CEXP G-2

CALL CLKOFF STATEMENT 7-18

CALL CMPLX G-2

CALL CONJ G-2

CALL COS G-2

CALL CSIN G-2

CALL CSQRT G-2

CALL DABS G-2

CALL DATAN G-2

CALL DATAN2 G-2

CALL DBLE G-2

CALL DCOS G-2

CALL DEXP G-3

CALL DIM G-3

CALL DINT G-3

CALL DISPLY J-1

CALL DLOG G-3

CALL DLOG1Ø G-3

CALL DLOG2 G-3

CALL DMAX1 G-3

CALL DMIN1 G-3

CALL DMOD G-3

CALL DSIGN G-3

CALL DSIN G-3

CALL DSQRT G-3

CALL EXP G-3

CALL F$AT STATEMENT 7-19

CALL FLOAT G-3

CALL FUNCTION SUBPROGRAMS 7-9

CALL IDIM G-4

CALL IDINT G-4

CALL IFIX G-4

CALL INT G-4

CALL IRND G-4

CALL ISIGN G-4

CALL LS G-4

CALL LT G-4

CALL MAXØ G-4

CALL MAX1 G-4

CALL MINØ G-4

CALL MIN1 G-4

CALL MOD G-4

CALL OR G-4

CALL OVERFL J-1

CALL POSFIL 7-14

CALL REAL G-5

CALL RND G-5

CALL RS G-5

CALL RT G-5

CALL SHFT G-5

CALL SIGN G-5

CALL SIN G-5

CALL SLITE J-1

CALL SNGL G-5

CALL SQRT G-5

CALL STATEMENT 7-2, 7-12, 7-18

CALL TANH G-5

CALLING CONVENTIONS 7-18

CALLING SUBROUTINES 7-12

CARD EQUIPMENT 6-2

CARDS 6-2

CARRIAGE RETURN 6-2

CASE A 8-1

CASE B 8-1

CENTRONICS LINE PRINTER F-4

CHARACTER ARRAY STRING TRANSFER 6-29

CHARACTER ARRAYS 6-29

CHARACTER SET 2-2

CLKOFF SUBROUTINE 7-18, 7-19

CLOSE DEVICE F-3

CLOSE NAMED FILE F-1

CLOSING AND OPENING FILES 8-3

CODING 5-11

CODING FORMS 2-4

COMB H-3

COMBINATIONS H-1

COMMA 6-9

COMMENT LINES 2-9

COMMENTS 2-9

COMMON 5-1, 5-3

COMMON AREAS 7-17

COMMON BLOCKS 5-6, 5-7, 5-17

COMMON I 5-2

COMMON STATEMENT 5-6, 7-17

COMMON STORAGE AREAS 5-3

COMMUNICATION LINKS 7-18

COMPILATION AND RUN TIME CONTROL 5-1

COMPILATION AND RUN TIME CONTROL STATEMENTS 5-11

COMPILING AND RUN TIME FEATURES 1-4

COMPLEX 3-9, 5-1, 5-3, 6-8, 7-7

COMPLEX DATA TYPES 3-8

COMPLEX MODE VARIABLE 5-6

COMPLEX MODES 3-18

COMPLEX NUMBERS 6-18

COMPLEX STATEMENTS 5-2

COMPLEX VALUES 3-8

COMPUTE ABSOLUTE VALUE OF SP NUMBER G-1

COMPUTE COSINE OF A CP NUMBER G-2

COMPUTE LOGARITHM OF AN SP NUMBER G-1

COMPUTE REMAINDER OF QUOTIENT OF TWO SP NUMBERS G-2

COMPUTE THE ABSOLUTE G-2

COMPUTE THE EXPONENTIAL OF A CP NUMBER G-2

COMPUTE THE PRINCIPAL VALUE G-2

COMPUTED GO TO STATEMENT 4-2

CONSECUTIVE ITEMS OF THE ARRAY 6-5

CONSOLE TELETYPE 6-2

CONSTANT 7-12

CONSTANTS 7-3

CONSTANTS IN A FORTRAN STATEMENT 3-2

CONTIGUOUS CHARACTERS 6-2

CONTINUATION LINE 2-9

CONTINUE STATEMENT 4-7

CONTROL F-5

CONTROL LINES 2-9

CONTROL PANEL SENSE SWITCHES 7-18

CONTROL STATEMENTS 4-1

CONTROL VARIABLE 4-7

CONVERT THE IMAGINARY PART OF A CP G-1

CORRECTING TYPING ERRORS 2-10

COS 5-8

CREATING A NEW FILE 8-3

CURRENT UFD 8-2

D FIELD DESCRIPTORS 6-17

D OUTPUT 6-21

D, E, F, G, INPUT 6-26

D, E, F, G, INPUT SCALE FACTOR 6-26

DAC ** 7-19

DACS 7-18

DAM 7-14

DAT 7-10

DATA 5-1

DATA DEFINING 5-1

DATA DEFINING STATEMENTS 5-9

DATA MODE 7-12

DATA MODE SPECIFICATION 5-1

DATA MODE SPECIFICATION STATEMENTS 5-4

DATA NAMES 2-2

DATA STATEMENTS 5-9, 7-7, 7-17

DATA TRANSFERS 6-2

DATA TYPE MODE SPECIFICATION STATEMENTS 5-1

DATA VALUES 3-1, 7-17

DECODE STATEMENT 6-39, 6-40

DELETE A FILE 8-3

DELETE NAMED FILE F-1

DELIMITERS 6-9

DETERMINANTS H-1

DEVICE CONTROL 6-1

DEVICE CONTROL STATEMENTS 6-39

DIAGNOSTIC PRINTOUTS 5-11

DIGITS 2-3

DIMENSION 5-1

DIMENSION STATEMENT 5-3, 5-4, 5-5, 7-12, 7-17

DIRECT POSITIONING SUBROUTINE 7-14

DIRECTORY 8-2

DISK 6-2

DISK OPERATING SYSTEM 1-5

DO LOOP 4-5

DO STATEMENT 4-4

DOLLAR SIGN 2-9

DOS 6-39, 7-14

DOS/VM 7-14

DOUBLE PRECISION 3-9, 3-18, 5-1, 5-2, 5-3, 6-8, 7-7

DOUBLE PRECISION DATA TYPES 3-7

DOUBLE PRECISION MODE VARIABLE 5-6

DOUBLE PRECISION STATEMENTS 5-2

DOUBLE PRECISION VARIABLE 6-28

DOUBLE SHIFT FEATURE 7-5

DUMMY ARGUMENT 5-8, 7-12

DUMMY ARGUMENTS FOR A SUBPROGRAM 5-6

DUMMY NAMES 5-4

DUMMY VARIABLES 5-3, 7-7

E AND D OUTPUT 6-25

E AND D OUTPUT SCALE FACTOR 6-25

E FIELD DESCRIPTORS 6-17

E OUTPUT 6-19

ELAPSED TIME COUNT 7-18

ELEMENT X 5-5

ENCODE STATEMENT 6-39

ENCODE/DECODE 6-1

END AND ERROR RETURNS 6-35

END OF FILE POSITIONING 7-14

END STATEMENT 2-4, 4-8, 7-12

ENDFILE STATEMENT 6-39

ENTERING FORMAT STATEMENTS AT RUN TIME 6-32

INDEX

EQ 3-14

EQUIVALENCE 5-1

EQUIVALENCE STATEMENT 5-4, 5-6, 5-7, 7-17

ERASE CHARACTER " 2-10

ERR=OPTION 6-40

ERROR FLAG 7-18

ERROR FLAG SET 6-22

ERROR MESSAGE 8-1

ERROR STATEMENT 7-14

EVAL 5-8

EVALUATION SEQUENCE 3-17

EXECUTABLE 2-2

EXECUTABLE PROGRAM 2-1

EXIT 7-13

EXPONENTIATION 3-14

EXPRESSIONS 3-1, 3-11, 3-12, 4-3, 7-3, 7-7, 7-12, 7-18

EXT PSEUDO-OP 7-18

EXTENDED RANGE 4-6

EXTERNAL 5-1

EXTERNAL DEVICE 6-2

EXTERNAL FORM 6-8

EXTERNAL FUNCTION 2-1

EXTERNAL PROCEDURE 2-1

EXTERNAL PROCEDURE NAME 5-8

EXTERNAL PROCEDURE SPECIFICATION STATEMENT 5-8

EXTERNAL PROCEDURE SPECIFICATION 5-1

EXTERNAL STATEMENT 5-8, 7-10

EXTERNAL SUBPROGRAM NAMES 5-3

EXTERNAL SUBROUTINE 2-1

F FIELD DESCRIPTORS 6-17

F OUTPUT 6-18, 6-25

F OUTPUT SCALE FACTOR 6-25

F$A1 I-1

F$A2 I-1

F$A3 I-1

F$A5 I-1

F$A6 I-1

F$AT I-2

F$ATI I-2

F$BN I-1

F$CB I-1

F$CG I-1

F$DE I-2

F$DEX I-2

F$DN I-1

F$EN I-2

F$FLEX I-2

F$FN I-1

F$HT SUBROUTINE 4-8

F$IO 6-35

F$IO 6-8

F$IO I-1

F$IOBF 6-8

F$IOBF I-2

F$IOBX I-1

F$RA I-1

F$RAX I-1

F$RB I-1

F$RBX I-1

F$RN I-1

F$RNX I-1

F$RTE I-2

F$RX I-1

F$TR I-1

F$WA I-1

F$WAX I-2

F$WB I-1

F$WBX I-2

F$WN I-1

F$WNX I-1

F$WX I-2

F,E,G, AND D SCALE FACTOR DESIGNATOR 6-25

FALSE 3-2

FIELD 2-7

FIND MAXIMUM OF A LIST OF INTEGERS G-1

FIND MAXIMUM OF A LIST OF SP NUMBERS G-1

FIND MINIMUM OF A LIST OF SP NUMBERS G-1

FLEX I-2

FLOATING POINT EXCEPTIONS I-2

FORMAT 6-1

FORMAT CONTROL 6-1

FORMAT CONVERSION SUBROUTINE 6-8, 6-35

FORMAT DESCRIPTOR 6-6

FORMAT ERRORS 6-22

FORMAT FIELD DESCRIPTOR SUMMARY 6-10

FORMAT FIELD DESCRIPTORS 6-9

FORMAT SPECIFICATION 6-5

FORMAT STATEMENT 6-5, 6-7, 6-8, 6-27, 6-32, 6-38

FORMATTED FILES 7-14

FORMATTED RECORD 6-2

FORMATTED RECORD LENGTH 6-8

FORTRAN COMPILER SUBROUTINES 1-4, I-1

FORTRAN DISK OUTPUT DRIVERS 7-15

FORTRAN IV 6-1, 7-2

FORTRAN IV COMPILER 7-2

FORTRAN LIBRARY FUNCTIONS 7-2

FORTRAN LIBRARY SUBROUTINES & FUNCTIONS 1-4

FORTRAN MATH LIBRARY 1-5

FORTRAN STATEMENT 7-12

FORTRAN SUBPROGRAM 5-4

FORTRAN VERSIONS 1-2

FULL LIST 5-1, 5-11

FUNC 7-3

FUNCTION AVRG 7-10

FUNCTION NAME 7-2, 7-8, 7-10, 7-12

FUNCTION STATEMENT 2-1, 7-2, 7-8, 7-9, 7-10, 7-12

FUNCTION SUBPROGRAM 5-8, 7-3, 7-7, 7-8, 7-17

FUNCTIONS 5-11, 7-2

FUNCTIONS AND SUBPROGRAMS 7-1

FUNIT 8-3

G FIELD DESCRIPTORS 6-17

G OUTPUT 6-20, 6-25

G OUTPUT SCALE FACTOR 6-25

GE 3-14

GETS NEXT CHARACTER F-2

GLOBAL MODE ASSIGNMENT 5-2

GO TO ASSIGNMENT STATEMENT 4-2

GO TO STATEMENT 4-7, 7-2

GT 3-14

H FIELD DESCRIPTOR 6-26

H INPUT 6-27

H OUTPUT 6-26

HOLLERITH DESCRIPTOR 6-27

HOLLERITH FIELD DESCRIPTOR 6-26

HORIZONTAL SPACING CONTROL 6-6, 6-8

I FIELD DESCRIPTOR 6-13

I INPUT 6-15

I OUTPUT 6-14

I/O CHARACTERS 3-7

I/O CONTROL SYSTEM 6-3

I/O LIST 6-2, 6-3

IDENTIFIERS 2-2

IF 3-2

IF STATEMENT 5-12

IMPERATIVE VERBS 2-2

INDEX

IMPLICIT MODE 3-9, 7-7

IMPLICIT MODE ASSIGNMENT 3-9

IMPLIED DO 6-3

IMPLIED DO-LOOPS 6-5

INCREMENTATION PARAMETER 4-7

INDICATION AND CONTROL SUBROUTINES 1-5, J-1

INITIAL PARAMETER 4-7

INLINE COMMENTS 2-9

INPUT ASCII DATA FROM THE COMMAND STREAM F-4

INPUT ASCII DATA FROM USER TERMINAL F-4

INPUT F-1 CHARACTER F-7

INPUT F-1 CHARACTER FROM PAPER TAPE F-7

INPUT FIELD DESCRIPTOR 6-1Ø

INPUT STRING 6-22

INPUT/OUTPUT 6-1

INPUT/OUTPUT CONTROL SYSTEM 1-5

INPUT/OUTPUT STATEMENTS 6-2

INPUTS ASCII DATA FROM THE H.S.READER F-4

INSERT 5-1

INSERT STATEMENT 5-12

INSERTING FILES 5-12

INTEGER 3-9, 3-18, 5-1, 5-3, 6-8, 7-7

INTEGER CONSTANT 5-5, 6-3

INTEGER DATA TYPES 3-6

INTEGER MODE 7-6

INTEGER MODE VARIABLE 5-6

INTEGER STATEMENTS 5-2

INTEGER VARIABLES 5-4, 6-28

INTRINSIC FUNCTION LIBRARY SUBROUTINES 7-6

INTRINSIC FUNCTIONS 7-3, I-2

IOCS 1-5, 6-3, 7-15

ITEM TRACE 5-11

ITEM TRACE STATEMENTS 5-11

JST INSTRUCTION 7-18

KEY 8-3

KILL CHARACTER ? 2-1Ø

L FIELD DESCRIPTOR 6-29

L INPUT 6-3Ø

L OUTPUT 6-29

LABEL 2-9

LE 3-14

LETTERS 2-3

LIBRARY FUNCTIONS 7-2, 7-7

LIBRARY SUBROUTINES 7-2, 7-18

LIGHTS 7-18

LINE PRINTER 6-2, 6-33

LINEQ H-3

LINES 2-7

LINKING FORTRAN AND ASSEMBLY LANGAUGE PROGRAMS 7-18

LIST 5-1, 5-11, 6-3

LIST CONTROL STATEMENT 2-4

LISTING CONTROL STATEMENTS 5-12

LISTING FILE 5-12

LITERAL TEXT STRINGS 6-26

LOC 7-2, 7-5

LOCATION ´ØØØØ1 5-7

LOCATION ´ØØØØ6 5-7

LOCATION 7-5

LOG FUNCTION 7-3

LOGICAL 3-9, 5-1, 5-3, 6-8, 7-7

LOGICAL AND 7-3

LOGICAL ASSIGNMENT STATEMENTS 3-1, 3-15, 3-16

LOGICAL CONSTANTS 3-1

LOGICAL DATA TYPES 3-6

LOGICAL EXCLUSIVE OR 7-3

LOGICAL EXPRESSIONS 3-2, 3-6

LOGICAL FIELD DESCRIPTOR 6-29

LOGICAL IF STATEMENTS 3-15, 4-4

LOGICAL MODE VARIABLE 5-6

LOGICAL NEGATION 7-4

LOGICAL OPERATOR APPLICATIONS 3-16

LOGICAL OPERATORS 3-15

LOGICAL OR 7-4

LOGICAL SHIFT 7-4

LOGICAL STATEMENTS 5-2

LOGICAL TRUE 3-2

LOGICAL TRUTH VALUES 3-6

LOGICAL UNIT TABLE 6-3

LOGICAL-L INPUT 6-30

LOGICAL-L OUTPUT 6-29

LS 7-6

LT 3-14, 7-6

LUTBL 6-3

MADD H-2

MADJ H-3

MAGNETIC TAPE 6-2

MAGNETIC TAPE TRANSPORTS 6-39

MASKING 7-5

MASKING AND POSITIONING 7 5

MASTER FILE DIRECTORY 8-2

MATH LIB H-1

MATHEMATICAL SUBROUTINES 7-2

MATHLB 1-5

MATMPY 7-13

MAXIMUM VALUES 8-1

MAXIMUMS 8-1

MCOF H-3

MCON H-2

MDET H-3

MEMORY LOAD 7-18

MEMORY STORAGE 5-5

MFD 8-2

MIDN H-2

MINV H-3

MIXED MODE EXPRESSIONS 3-18

MIXED NUMBER-F OUTPUT 6-18

MMLT H-2

MODE DECLARATION STATEMENT 5-3

MODE SPECIFICATION STATEMENT 5-2

MODE STATEMENT 5-2

MOVE RAW DATA F-2

MOVES DATA FROM BUFFER F-8

MOVES INFORMATION F-8

MOVES RAW DATA F-8

MSCL H-2

MSUB H-2

MTRN H-2

MULTIPLICATION 3-14

NAME 7-2

NAME 7-7

NE 3-14

NESTED 3-13

NESTED DO LOOPS 4-5

NOLIST 5-1, 5-11

NON-ZERO CONSTANT 5-5

NONEXECUTABLE 2-2

NONOWNER STATUS 8-2

NOT 3-16, 7-4

NOT FUNCTION 7-4

NUMBER OF ELEMENTS 5-7

NUMERICAL CONSTANTS 3-2

OBJECT 8-3

OBJECT CODE 1-5

OCTAL DIGIT 2-3

OCTAL INTEGERS 3-6

OFF ENTRY POINT 7-18

ONE-DIMENSIONAL ARRAY

Y 5-5

ONE-PASS COMPILER 1-5

OPEN FOR READ & WRITE F-4, F-6

OPEN FOR READ F-4, F-6

OPEN FOR WRITE F-4, F-6

OPEN NAMED FILE FOR BOTH READING AND WRITING F-1

OPEN NAMED FILE FOR READING F-1

OPEN NAMED FILE FOR WRITING F-1

OPENING A FILE 8-3

OPERATORS 2-2

OPERATORS IN FORTRAN STATEMENTS 3-12

OR 3-16, 7-4

ORDER OF EVALUATION 3-16

OUTPUT CONVERSION 6-25

OUTPUT F-1 CHARACTER F-7

OUTPUT F-1 CHARACTER TO H.S. PUNCH F-7

OUTPUT FIELD DESCRIPTOR 6-10

OUTPUT FIELD DESCRIPTORS 6-11

OUTPUT OF ARRAY ELEMENTS 6-5

OUTPUTS ASCII DATA TO THE H.S.PUNCH F-4

OUTPUTS BINARY DATA F-4

OUTPUTS COUNT CHARACTERS TO THE USER TERMINAL F-7

OWNER PASSWORD 8-2

OWNER RIGHTS 8-3

OWNER STATUS 8-2

PAPER TAPE 6-2

PAPER TAPE UNIT 6-2

PARALLEL INTERFACE LINE PRINTER F-4

PARENTHESES 3-13

PAUSE STATEMENT 4-8

PERFORMS A TREE SEARCH F-2

PERM H-3

PERMUTATIONS H-1

PHYSICAL RECORD 7-15

POINT PLOTTING DATA 7-2

POSFIL 7-14

POSITIONING DISK F-3

POSITIVE CONSTANT 5-5

POSITIVE NON-ZERO INTEGER CONSTANTS 5-3

PRIME FORTRAN EXTENSIONS 1-6

PRIME FORTRAN IV FEATURES 1-5

PRINT & PRINTER CONTROL 6-33

PRINT STATEMENT 6-33

PROCEDURE SUBPROGRAM 2-1

PROCESSING ARRAYS 6-8

PROCESSING ENTIRE ARRAYS 6-8

PROGRAM FORM 2-4

PROGRAM UNITS 2-1

PROGRAMMER 5-12

PROGRAMMING EFFICIENCY 7-1

PROTECTED 7-17

PROTECTED FUNCTIONS AND SUBROUTINES 7-16

PROTECTED SUBROUTINE 7-17

PUNCH TRAILER F-3, F-4

R=SSWTCH J-1

RANGE ERRORS 6-22

RAW DATA MOVER F-8

READ 6-1

READ ASCII DATA F-2

READ ASCII F-5

READ BCD F-6

READ BINARY F-5, F-6

READ DEVICE F-3

READ OR WRITE DISK FILE F-2

READ STATEMENT 6-2, 6-27, 6-28, 6-32, 6-38, 6-39

READ/WRITE STATEMENTS 6-35

READING DATA INTO ARRAYS 6-3

READS INPUT FROM PARALLEL CARD READER F-8

READS INPUT FROM SERIAL CARD READER F-8

READS ONE CHARACTER F-7

REAL 3-9, 3-18, 5-1, 5-2, 5-3, 6-8, 7-7

REAL DATA TYPES 3-7

REAL I 5-2

REAL MODE VARIABLE 5-6

REAL NUMBERS 3-7, 3-8

REAL STATEMENTS 5-2

REAL TIME OPERATING SYSTEM 1-5, 7-14

REAL VARIABLE 6-28

RECORD CHARACTERISTICS 6-1

RECORD LENGTH OPTION 6-7

RECORD NUMBER 7-15

RECORD SIZE 6-2

REFERENCE DOCUMENTS 1-2

REFERENCE SUBKEYS 8-3

RELATIONAL OPERATORS 3-2, 3-14, 3-18

RESCANNING FORMAT

LISTS 6-32

RETURN STATEMENT 7-9

REWIND & CLOSE FILE F-6

REWIND A FILE UNIT 8-3

REWIND FILE F-1

REWIND STATEMENT 6-39

RIGHTS 8-2

RS 7-6

RT 7-6

RTOS 7-14

SAM 7-14

SCALAR 3-1Ø

SCALE FACTOR 6-25, 6-26

SCOPE OF HANDBOOK 1-3

SEARCH 8-3

SEARCH DISK F-3

SEARCH SUBROUTINE 8-3

SEARCHES FOR A FILE 8-3

SENSE LIGHT/SWITCH SUBROUTINES 7-18

SEQUENCE NUMBER 2-1Ø

SEQUENTIAL ACCESS DEVICES 6-39

SEQUENTIAL INPUT 6-5

SHFT 7-4

SHFT FUNCTION 7-6

SHFT LIBRARY SUBROUTINE 7-4

SIMILAR STATEMENTS 3-2

SIMPLE STATMENTS 3-1

SIN 5-8, 7-2

SINGLE VARIABLE 3-1

SINGLE-BIT TRUTH VALUES 7-3

SIZE OF ARRAYS 5-3

SLASH 6-9

SLASHED CHARACTERS 2-1Ø

SLITET J-1

SOURCE PROGRAMS 3-7

SPACES 2-1Ø, 6-15

SPECIAL CHARACTERS 2-3

SPECIAL COMMON BLOCK 5-7

SPECIFICATION STATEMENTS 5-1, 7-17

SPECIFICATION SUBPROGRAM 2-1

STANDARD PRIME LIBRARY SUBROUTINES 7-18

START SWITCH 4-8

STATEMENT FUNCTIONS 7-6, 7-7, 7-8

STATEMENT LABELS 4-1, 4-7

STATEMENT LINE 2-9

STATEMENT NUMBER 2-9

STD 7-10

STOP COMPILATION 2-9

STOP STATEMENT 4-7

STORAGE 6-3

STORAGE LOCATIONS 6-2

STORAGE SPECIFICATION 5-1

STORAGE SPECIFICATION STATEMENTS 5-3

STRING 2-7

SUBPROGRAM 2-1

SUBR PSEUDO-OPS 7-18

SUBROUTINE CALL 8-2

SUBROUTINE CALLS 3-7

SUBROUTINE INTEGER S 5-2

SUBROUTINE STATEMENT 2-1, 7-2, 7-9, 7-11, 7-12

SUBROUTINE SUBPROGRAM 2-2, 7-11, 7-17

SUBROUTINES INTERNAL TO FORTRAN I-1

SUBSCRIPTED VARIABLES 3-10, 5-5, 6-5,. 7-12

SUBSCRIPTS 3-10

SUBSEQUENT SOURCE STATEMENTS 5-12

SUMMARY OF INPUT FIELD DESCRIPTION 6-13

SYMBOLIC NAME 3-1, 7-2

SYNTACTIC ELEMENTS 2-2

TAB 6-16

TABULATION CONTROL 6-16

TERMINAL PARAMETER 4-7

TEXT EDITOR 2-4

TRACE 5-1

TRACE CONTROL STATEMENT 2-4

TRACE OBJECT CODING 5-12

TRACE STATEMENT 5-11, 5-12

TRANSLATION 6-8

TREE FILE NAME 5-12

TREE FILE NAME SPECIFIER 5-12

TRIGONOMETRIC SINE 7-2

TRUNCATE A FILE 8-3

TRUNCATE FRACTIONAL BITS OF AN SP G-1

TRUNCATE NAMED FILE F-1

TRUNCATION 3-13

TURN ON PUNCH AND PUNCH LEADER F-3

TWO-DIMENSIONAL ARRAY 3-11

TWO-DIMENSIONAL ARRAY X 5-5

TYPE A DESCRIPTOR 6-28

TYPE A FORMAT STATEMENT 6-32

TYPE D CONVERSION 6-18

TYPE D FIELD DESCRIPTOR 6-21

TYPE E CONVERSION 6-18

TYPE E FIELD DESCRIPTOR 6-20

TYPE F CONVERSION 6-17

TYPE G CONVERSION 6-18

TYPE G FIELD DESCRIPTOR 6-21

TYPEWRITER 6-2

UNCONDITIONAL GO TO STATEMENT 4-1

UNFORMATTED FILES 7-14

UNFORMATTED RECORDS 6-38

UNIT NUMBER ´U´ 6-3

UNIT RECORD KEYPUNCHING 2-4

USER PROGRAM 8-2

USER TERMINAL F-3

USER-DEFINED STATEMENT FUNCTION 7-2

USING F$AT 7-18

VALUE EXPRESSION 7-12

VAR 7-4, 7-6

VARIABLE 5-5, 5-12, 6-3, 6-28, 7-12

VARIABLE LIST 6-4

VARIABLE NAME 3-8

VARIABLE SUBSCRIPTS