



**MP-60
COMPUTER SYSTEM**

**FORTRAN
REFERENCE MANUAL**

LIST OF EFFECTIVE PAGES

New features, as well as changes, deletions, and additions to information in this manual, are indicated by bars in the margins or by a dot near the page number if the entire page is affected. A bar by the page number indicates pagination rather than content has changed.

Page	Rev.	Page	Rev.	Page	Rev.	Page	Rev.
Title Page	E	5-7/5-8	A	8-21	A	C-4	A
ii	E	6-1	A	8-22	A	C-5	A
iii	E	6-2	A	8-23	A	C-6	A
iv	E	6-3	A	8-24	A	C-7	A
v/vi	E	6-4	D	9-1	A	C-8	A
vii	E	6-5	A	9-2	A	C-9	A
viii	E	6-6	A	9-3	A	C-10	A
ix	E	6-7	B	9-4	A	C-11	A
x	E	6-8	D	9-5	A	C-12	A
xi	E	7-1	A	9-6	A	C-13	A
xii	E	7-2	D	9-7	A	C-14	A
1-1	A	7-3	A	9-8	A	C-15/C-16	A
1-2	A	7-4	A	9-9	A	D-1	A
2-1	A	7-5	A	9-10	A	D-2	A
2-2	A	7-6	A	9-11	A	D-3/D-4	A
2-3	A	7-7	A	9-12	A	E-1	A
2-4	C	7-8	A	9-13	A	E-2	A
2-5	C	7-9/7-10	D	9-14	A	F-1	C
2-6	A	8-1	A	9-15	A	F-2	C
3-1	A	8-2	A	9-16	A	F-3	C
3-2	A	8-3	A	9-17/9-18	A	F-4	C
3-3	A	8-4	A	10-1	A	F-5	C
3-4	A	8-5	A	10-2	A	F-6	C
3-5	A	8-6	A	10-3	A	F-7	C
3-6	A	8-7	A	10-4	A	F-8	C
3-7	A	8-8	A	10-5	A	F-9	C
3-8	A	8-9	A	10-6	B	F-10	C
4-1	A	8-10	A	10-7/10-8	A	F-11	C
4-2	A	8-11	A	A-1	B	F-12	C
4-3	A	8-12	A	A-2	B	F-13	C
4-4	A	8-13	A	A-3	B	F-14	C
5-1	A	8-14	A	A-4	B	F-15	C
5-2	A	8-15	A	B-1	A	F-16	C
5-3	A	8-16	A	B-2	A	F-17	C
5-4	A	8-17	A	C-1	A	F-18	C
5-5	D	8-18	A	C-2	A	G-1	D
5-6	A	8-19	A	C-3	A	G-2	D
		8-20	A			G-3	D

LIST OF EFFECTIVE PAGES

New features, as well as changes, deletions, and additions to information in this manual, are indicated by bars in the margins or by a dot near the page number if the entire page is affected. A bar by the page number indicates pagination rather than content has changed.

Page	Rev.	Page	Rev.	Page	Rev.	Page	Rev.
G-4	E						
I-1	A						
I-2	B						
I-3	A						
I-4	A						
I-5	A						
I-6	C						
I-7	A						
I-8	A						
I-9	D						
I-10	A						

PREFACE

This publication is a reference manual for the programmer using the MP-60 FORTRAN compiler. The manual defines the external characteristics of MP-60 FORTRAN, that is, those features which are observable or under the active control of the user.

Knowledge of the MP-60 computer operation is assumed for this manual. The operation of the MP-60 machine instructions is described in the MP-60 hardware reference manual (publication No. 14306500).



CONTENTS

Section	Page
1 INTRODUCTION	1-1
Optional Outputs	1-1
Coding Format	1-1
FORTRAN Character Set	1-2
2 BASIC ELEMENTS	2-1
Constants	2-1
Integer	2-1
Hexadecimal	2-1
Real	2-2
Double Precision Real	2-2
ASCII	2-2
Logical	2-3
Variables	2-3
Simple Variables	2-3
Subscripted Variables	2-3
3 EXPRESSIONS	3-1
Arithmetic Expressions	3-1
Relational Expressions	3-2
Logical Expressions	3-4
Masking Expressions	3-5
Evaluation of Expressions	3-7
Mixed Mode Arithmetic	3-8
4 REPLACEMENT STATEMENTS	4-1
Replacement Statement	4-1
Multiple Replacement Statement	4-3
Mixed-Mode Replacement Statement	4-4

CONTENTS (CONT.)

Section	Page
5 DECLARATIVE STATEMENTS	5-1
Type Statements	5-1
Dimension Statement	5-2
Variable Dimensions	5-3
COMMON Statements	5-4
EQUIVALENCE Statement	5-5
DATA Statement	5-7
6 CONTROL STATEMENTS	6-1
GO TO Statements	6-1
Unconditional GO TO	6-1
Computed GO TO	6-1
Assigned GO TO	6-2
Assign	6-2
IF Statements	6-3
Arithmetic IF	6-3
Logical IF	6-3
DO Statement	6-4
DO Loop Execution	6-4
DO Nests	6-5
DO Loop Transfer	6-6
CONTINUE Statement	6-7
PAUSE Statement	6-7
STOP Statement	6-8
7 SUBPROGRAMS	7-1
Main Program and Subprograms	7-1
Subroutine Subprograms	7-1
SUBROUTINE Statement	7-2
CALL Statement	7-2

CONTENTS (CONT.)

Section	Page
Function Subprograms	7-4
Defining Function Subprograms	7-4
Referencing Function Subprograms	7-5
EXTERNAL Statement	7-7
ENTRY Statement	7-8
ENTRY Call or Reference	7-8
RETURN Statement	7-9/7-10
END Statement	7-9/7-10
Program Arrangement	7-9/7-10
 8 FORMAT SPECIFICATIONS	 8-1
FORMAT Statement	8-1
Field Descriptors	8-1
Field Separators	8-2
Conversion Specifications	8-3
Ew.d Output	8-3
Ew.d Input	8-4
Fw.d Output	8-7
Fw.d Input	8-9
Dw.d Output	8-10
Dw.d Input	8-11
Iw Output	8-11
Iw Input	8-12
\$w Output	8-12
\$w Input	8-13
Lw Output	8-13
Lw Input	8-13
Aw Output	8-13
Aw Input	8-14
Rw Output	8-14
Rw Input	8-14

CONTENTS (CONT.)

Section	Page
nP Scale Factor	8-15
Fw.d Scaling	8-15
Ew.d Scaling	8-16
Scaling Restrictions	8-16
Editing Specifications	8-16
Space (wX)	8-17
wH Output	8-18
wH Input	8-18
New Record (/)	8-19
Repeated Specifications	8-21
Variable Format	8-22
Format Control	8-23
Carriage Control	8-24
9 INPUT/OUTPUT STATEMENTS	9-1
I/O Lists	9-1
DO-Implying Segments	9-1
Transmission of Arrays	9-5
I/O Units	9-5
Partial Records	9-5
Output Statements	9-6
Print Record	9-6
Write Binary Record	9-6
Write ASCII Record	9-7
Input Statements	9-8
Read Card Record	9-8
Read Binary Record	9-8
Read ASCII Record	9-9

CONTENTS (CONT.)

Section	Page
Buffer Statements	9-10
BUFFER IN	9-10
BUFFER OUT	9-11
File Control Statements	9-12
REWIND	9-12
BACKSPACE	9-12
ENDFILE	9-13
Internal Transmission Statements	9-13
ENCODE	9-13
DECODE	9-15
Status Checking Routine	9-17/9-18
I/O Complete Check.....	9-17/9-18
10 PROGRAM OPERATION	10-1
FORTRAN Control Card	10-1
Control Card Notes	10-3
Calling Sequences	10-3
Sample Deck Structures	10-5

APPENDIXES

A	Library Routines
B	FORTRAN Standard Output
C	FORTRAN Diagnostics
D	FORTRAN Statements
E	Character Codes
F	FORTRAN Interface Routines
G	MPX/OS Special Features
INDEX	

TABLES

Table	Parameter/LU	Page
10-1	Parameter/LU	10-2

INTRODUCTION

1

MP-60 FORTRAN (FTN-60) is a FORTRAN compiler that is hosted on the CONTROL DATA® MP-60 Computer and generates object code for execution on the MP-60. FTN-60 runs in conjunction with the MPX-RT, MPX-MP, and MPX-MC operating systems.

OPTIONAL OUTPUTS

Outputs that may be selected include:

- Relocatable binary cards or card images
- Source program listing
- Assembly language listing of machine instructions
- Load-and-go object program for immediate execution
- Symbolic cross reference list

Diagnostic messages are printed when the compiler detects coding errors.

CODING FORMAT

Statements are coded in one of the following formats.

<u>Columns</u>	<u>Content</u>
1 through 5	Statement label
6	Continuation designator (nonzero character)
7 through 72	Statement
or	
1	Comment designator (C)
2 through 72	Comments
73 through 80	Identification and sequencing

A line contains a string of up to 72 FORTRAN characters. The character positions in a line (referred to as columns) are numbered consecutively 1 through 72.

The character C in column 1 identifies the line as comment. Comments are for the convenience of the programmer and permit him to describe the program steps; they do not affect program execution. A comment may be inserted at any point in the program. Comment cards are listed along with the source statements when the source list option is selected.

Columns 1 through 5 may be blank or may contain a label that identifies the line for reference elsewhere in the program. A statement number (label) must be unique within a subprogram and in the range 1 through 99999.

A statement that is labeled and never referenced (a null) causes an informative diagnostic during compilation but does not inhibit execution of the compiled program.

The compiler ignores blanks and leading zeros in statement numbers.

Statements in the formats outlined in this manual appear in columns 7 through 72. A statement that exceeds the 66 characters allowed on a single card may be continued on successive (continuation) cards. A maximum of 19 continuation cards is allowed per statement. Continuation cards are not labeled; columns 1 through 5 must be blank. A character other than a zero or a blank in column 6 designates continuation.

FORTTRAN statements within a program, subroutine, or function must appear in the order of their classes (e.g., every class 1 statement in a program must come before a class 2 or higher statement, etc.). For a list of the FORTRAN statement classes, refer to Appendix D.

The compiler does not interpret columns 73 through 80. These columns appear on the source listing and are for sequencing or program identification.

FORTTRAN CHARACTER SET

The character set has two subsets, alphanumeric characters and special characters.

Alphanumeric characters are as follows:

A through Z
0 through 9

Special characters are as follows:

blank	(left parenthesis
+ plus)	right parenthesis
- minus	,	comma
= equals	.	period
* asterisk	\$	currency symbol
/ slash		

An FTN-60 program or subprogram consists of source language statements necessary to define a problem and the steps in its solution. The source language consists of basic language elements that make up expressions and statements.

CONSTANTS

FTN-60 accepts six basic types of constants: integer, hexadecimal, real, double precision real, ASCII, and logical. The type of a constant is determined by its form. Double precision real constants occupy two consecutive computer words (64 bits); integer, hexadecimal, real, and ASCII constants occupy one computer word (32 bits); and logical constants occupy one bit.

If a constant exceeds the allowed range, the statement that contains it is rejected during a compilation and a diagnostic is provided.

INTEGER

An integer constant consists of 1 through 10 decimal digits in the range $-2^{31} < M < 2^{31}$ (-2,147,483,648 to 2,147,483,647). The constant is translated as a 31-bit value and a sign bit. A negative number is stored in two's complement format.

HEXADECIMAL

A hexadecimal constant is up to eight hexadecimal digits preceded by a currency symbol in the form: $\$n_1 \dots n_i$.

Each digit corresponds to four bits of the translated constant. If fewer than eight digits are written when expressing a hexadecimal constant, the constant is right justified and zero filled. When a minus sign precedes the constant, the constant is stored in two's complement format.

REAL

A real constant is represented by a string of decimal digits. It is expressed as real either with a decimal point or with an exponent representing a power of ten, or with both, in the forms:

$\pm nE\pm s$	$\pm n.$	$\pm n.E\pm s$	
$\pm n.n$	$\pm n.nE\pm s$	$\pm .n$	$\pm .nE\pm s$

s is the exponent to the base ten. The constant is translated into the MP-60 single precision floating point format.

DOUBLE PRECISION REAL

A double precision real constant is expressed in the same manner as a real constant but expressed with a D in the forms:

$\pm nD\pm s$	$\pm n.D$	$\pm n.D\pm s$	
$\pm n.nD\pm s$	$\pm .nD$	$\pm .nD\pm s$	

The constant is translated into the MP-60 double precision floating point format.

ASCII

An ASCII constant is a string of characters of the form nHw, nAw, or nRw.

nHw - left justified blank filled

nAw - left justified zero filled

nRw - right justified zero filled

The constant is translated into four 8-bit ASCII codes within a single computer word. If n is greater than four, a diagnostic is issued.

LOGICAL

A logical constant is a truth value:

.TRUE. or .FALSE.

A logical constant occupies one bit of storage: 1 for true and 0 for false.

For example:

LOGICAL X1, X2:

X1 = .TRUE.

X2 = .FALSE.

VARIABLES

A variable name consists of 1 to 8 alphanumeric characters; the first character must be alphabetic. It represents a specific storage location.

The FTN-60 compiler recognizes simple and subscripted variable names. A simple variable name represents a single quantity; a subscripted variable name represents a single quantity within an array of quantities. The type of a variable is designated either explicitly in a type declaration or implicitly by the first letter of the variable name. A first letter of I, J, K, L, M, or N indicates an integer (fixed point) variable; any other first letter indicates a single precision real (floating point) variable.

SIMPLE VARIABLES

A simple variable name identifies the location where a variable value can be stored. A variable which has been defined as double precision real occupies two consecutive memory locations. Integer and single precision variable names refer to single memory locations. Variable names which have been declared as character or logical types correspond to character addresses and bit addresses, respectively.

SUBSCRIPTED VARIABLES

A subscripted variable name identifies the location in an array where a variable value can be stored.

An array is a block of successive memory locations comprising the elements of the array. Each element of an array is referenced by the array name plus a set of subscripts. The type of an array is determined by the array name or a type declaration.

Arrays may have one, two, or three dimensions; the maximum number of array elements is the product of the dimensions. The maximum number of words used in an array cannot exceed 65,535. The array name and its dimensions must be declared at the beginning of the program in a DIMENSION, COMMON, or SCRATCH COMMON statement.

Subscript Forms

A subscript has one of the following forms; c and d are integer constants and I is a simple integer variable.

(c*I±d) (I±d) (c*I)
 (I) (c)

More than three subscripts cause a compiler diagnostic. Program errors may result if subscripts are larger than the dimensions initially declared for the array. A single subscript notation may also be used for a two- or three-dimensional array if it is the structural location of the variable. However, diagnostics will occur if the elements of a one-dimensional array, $A(d_1)$, are referred to as A(I, J, K) or A(I, J).

For example:

A(I,J) B(I+2, J+3, 2*K+1) Q(14)
 P(KLIM, J, LIM+5) SAM(J-6) A(133)
 B(1, 2, 3) A(233)

At no time during program execution can a simple integer variable used as an index variable take on a value greater than 65,535.

Array Structure

Elements of an array are stored by column in ascending storage locations. The location of an array element with respect to the first element is determined by the maximum array dimensions and the type of the array.

The first element of array A(I, J, K) is (1,1,1). The location of element A(i, j,k) with respect to A(1,1,1) is:

$$\text{Loc } A(i, j, k) = \text{loc } A(1, 1, 1) + ((i-1)+(j-1)*I + (k-1)*I*J) * E$$

The quantity in brackets is the subscript expression. E is the element length (the number of storage locations required for each element of the array). For integer and real arrays, E = 1; for double precision real arrays, E = 2; for character arrays, E = 1; and for logical arrays, E = 1. Subscripts i, j, and k may be any of the allowed subscript forms.

Factoring the expression produces:

Base address	S, the first location of A(1,1,1)
Constant addend	$-(1-I+I*J)*E$
Index function	$(i+I*j+I*J*k)*E$

When i, j, k are other than simple variables (for example, $C*I+d$), constants such as d appear in the constant addend. For example:

In the array declared as A(3,3,3); where A is type double precision real:

I=3; i=1,2,3

J=3; j=1,2,3

K=3; k=1,2,3

The elements of this double precision real array are stored two words per element starting with A(1,1,1) in S, the lowest location reserved for the array.

<u>Locations</u>	<u>Array Element A_{i,j,k}</u>
S, S+1	A ₁₁₁
S+2, S+3	A ₂₁₁
S+4, S+5	A ₃₁₁
S+6, S+7	A ₁₂₁
S+8, S+9	A ₂₂₁
S+10, S+11	A ₃₂₁
S+12, S+13	A ₁₃₁
.	.
.	.
.	.

<u>Locations</u>	<u>Array Element A_{i,j,k}</u>
S+48, S+49	A ₁₃₃
S+50, S+51	A ₂₃₃
S+52, S+53	A ₃₃₃

Referring to the example, if loc A(1,1,1)=S, the locations of A(2,2,3) with respect to A(1,1,1) are:

$$\text{Loc } A(2,2,3) = \text{loc } A(1,1,1) + [(2-1)+(2-1)*3+(3-1)*3*3] *2 = S+44, S+45$$

The following relaxation on the representation of subscripted variables is permissible (d_i represents an integer constant):

$A(d_1, d_2, d_3)$

$A(I, J, K)$ implies $A(I, J, K)$

$A(IMJ)$ implies $A(IMJ, 1, 1)$

A implies $A(1, 1, 1)$

$A(d_1, d_2)$

$A(I, J)$ implies $A(I, J)$

$A(I)$ implies $A(I, 1)$

A implies $A(1, 1)$

An expression is formed from elements and operators. The four kinds of expressions are arithmetic, relational, logical, and masking. Arithmetic expressions have numeric values. Relational and logical expressions have truth values. Masking expressions have 32-bit logical arithmetic values. Each type of expression is associated with a specific group of operators and operands.

ARITHMETIC EXPRESSIONS

The arithmetic operators are:

- + addition
- subtraction
- * multiplication
- / division
- ** exponentiation

The arithmetic elements are:

- Constants
- Simple or subscripted variables
- Function references

Any constant, variable, or function reference by itself can be an arithmetic expression. If X is an expression, then (X) is an expression. If X and Y are expressions, then the following are expressions.

$X + Y$	$X - Y$	$-X$
$X * Y$	X / Y	$X ** Y$

If op is an arithmetic operator and X and Y are arithmetic expressions, then X op op Y is not a valid expression.

Examples of expressions:

A

3.14159

B + 16.427

XBAR + (B(I,J+1,K)/3.)

-(C + DELTA*AERO)

(B-SQRT(X/Y))/(2.0*A)

GROSS - (TAX*0.04)

(TEMPT+V(M,MAX)*Y**C)/(H-FACT(K+3))

RELATIONAL EXPRESSIONS

A relational expression is:

$e_1 \text{ op } e_2$

e_1 and e_2 are arithmetic expressions and op is an operator that belongs to the set. The relational expressions are:

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

A relation is true if e_1 and e_2 satisfy the relation specified by op; otherwise, it is false. A false relational expression is assigned the value of the logical constant .FALSE.; a true relational expression is assigned the value of the logical constant .TRUE..

Rules for relational expressions are as follows:

- 1) Evaluation is from left to right in a relation of the form:

$$e_1 \text{ op } e_2$$

The following relations are equivalent:

$$e_1 \text{ op } e_2$$

$$e_1 \text{ op } (e_2)$$

$$(e_1) \text{ op } e_2$$

$$(e_1) \text{ op } (e_2)$$

- 2) Use a relational operator between two arithmetic expressions.
- 3) In a relational expression, do not use more than two arithmetic expressions connected by a relational operator. Not allowed:

$$e_1 \text{ op } e_2 \text{ op } e_3$$

- 4) Separate two relational expressions with a logical connector, .AND. or .OR., in the forms:

$$e_1 \text{ op } e_2 \left\{ \begin{array}{l} \text{.AND.} \\ \text{.OR.} \end{array} \right\} e_2 \text{ op } e_3$$

$$e_1 \text{ op } e_2 \left\{ \begin{array}{l} \text{.AND.} \\ \text{.OR.} \end{array} \right\} e_3 \text{ op } e_4$$

Examples of relational expressions:

A .GT. 16.

R(I) .GE. R(I-1)

R-Q(I)*Z .LE. 3.141592

K .LT. 16

B-C .NE. D+E

I .EQ. JJ(K)

LOGICAL EXPRESSIONS

A logical expression is formed with logical operators and logical elements and has the value true or false (the values have the same internal representation as relational expressions).

The logical operators are:

- .OR. logical disjunction
- .AND. logical conjunction
- .NOT. logical negation

A logical expression has the general form:

$L_1 \text{ op } L_2 \text{ op } L_3 \dots$

L_i are logical variables, logical constants, logical functions, logical expressions enclosed in parentheses, or relational expressions; and op is the logical operator .AND. (indicating conjunction) or .OR. (indicating disjunction).

The logical operator that indicates negation appears in the form:

.NOT. L_1

Each expression is evaluated by scanning from left to right, with logical operations being performed according to the following hierarchy of precedence.

- 1) .NOT.
- 2) .AND.
- 3) .OR.

A logical variable, logical constant, or a relational expression is, in itself, a logical expression. If L_1 and L_2 are logical expressions, then the following are logical expressions:

.NOT. L_1

L_1 .AND. L_2

L_1 .OR. L_2

If L is a logical expression, then (L) is a logical expression. If L₁, L₂ are logical expressions and op is .AND. or .OR., then L op L₂ is never legitimate. However, .NOT. may appear in combination with .AND. or .OR. only as follows:

L₁ .AND. .NOT. L₂

L₁ .OR. .NOT. L₂

L₁ .AND. (.NOT....)

L₁ .OR. (.NOT....)

.NOT. may appear with itself only in the form .NOT. (.NOT. (.NOT. L))
Other combinations cause compilation diagnostics.

If L₁, L₂ are logical expressions, the logical operators are defined as follows:

.NOT. L₁ is false only if L₁ is true

L₁ .AND. L₂ is true only if L₁, L₂ are both true

L₁ .OR. L₂ is false only if L₁, L₂ are both false

Examples:

1) B - C ≤ A ≤ B + C

is written B - C .LE. A .AND. A .LE. B+C

2) FICA greater than 737.10 and PAYNMB equal to 12600.00

is written FICA .GT. 737.10 .AND. PAYNMB .EQ. 12600.00

MASKING EXPRESSIONS

Masking expressions consist of masking operators and elements; they resemble logical operations in appearance only.

In a masking expression, 32-bit logical arithmetic is performed bit-by-bit on the operands within the expression. The operands may be any type variables, constants, or expressions other than logical. No mode conversion is performed during evaluation. If the operand is double precision, operations are performed on the higher order word. The result of a masking operation is integer. Although the masking operators are identical in appearance

to the logical operators, their meanings are different. They are listed according to hierarchy. The following definitions apply:

- .NOT. bit-by-bit logical negation
- .AND. bit-by-bit logical multiplication
- .OR. bit-by-bit logical addition
- .XOR. bit-by-bit exclusive OR

The operations are as follows:

<u>p</u>	<u>v</u>	<u>p.AND.v</u>	<u>p.OR.v</u>	<u>p.XOR.v</u>	<u>.NOT.p</u>
1	1	1	1	0	0
1	0	0	1	1	0
0	1	0	1	1	1
0	0	0	0	0	1

If B_1 are masking expressions, variables, or constants of any type other than logical, the following are masking expressions:

- .NOT. B_1
- B_1 .AND. B_2
- B_1 .OR. B_2

If B is a masking expression, then (B) is a masking expression. .NOT. may appear with .AND. or .OR. only as follows:

- .AND..NOT.
- .OR..NOT
- .AND. (.NOT. ...)
- .OR. (.NOT. ...)

Masking expressions of the following forms are evaluated from left to right.

- A .AND. B .AND. C...
- A .OR. B .OR. C...

Masking expressions must not contain logical operands.

EVALUATION OF EXPRESSIONS

Evaluation of expressions is generally from left to right with the precedence of the operators and parentheses (the deepest nested parenthetical subexpression is evaluated first) controlling the sequence of operation. The precedence of operators for arithmetic evaluation is as follows:

** (exponentiation)	Class 1
/ (division)	Class 2
* (multiplication)	Class 2
+ (addition)	Class 3
- (subtraction)	Class 3
Relationals	Class 4
.NOT.	Class 5
.AND.	Class 6
.OR.	Class 7
.XOR.	Class 7

In an expression with no parentheses or within a pair of parentheses in which unlike classes of operators appear, evaluation proceeds in the order indicated above (lowest class operators first). In expressions containing like classes of operators, evaluation proceeds from left to right ($A**B**C$) is evaluated as $(A**B)**C$.

When writing an integer expression, it is important to remember the left-to-right scanning process; in addition, if dividing an integer quantity by an integer quantity yields a remainder, the result will be truncated (thus $11/3 = 3$).

An array element name (a subscripted variable) used in an expression requires the evaluation of its subscript. The type of expression in which a function reference or subscript appears does not affect, nor is it affected by, the evaluation of the actual arguments or subscripts.

MIXED-MODE ARITHMETIC

Arithmetic expressions can contain mixed types of constants and variables. FTN-60 performs the arithmetic operations by converting to one type according to established rules.

In arithmetic expressions, the operand types are double precision real, real, integer, and character. The following rules establish the relationship between the type of an evaluated expression and the types of the operands it contains:

- 1) The order of dominance of the operand types within an expression from highest to lowest is:
 - a) Double
 - b) Real
 - c) Integer
 - d) Character
- 2) The dominant operand type determines the type of an evaluated arithmetic expression. The mode/type relationships are:

Type A \ Type B	Character	Integer	Real	Double
Character	Integer	Integer	Real	Double
Integer	Integer	Integer	Real	Double
Real	Real	Real	Real	Double
Double	Double	Double	Double	Double

For example, when A is real and B is integer, the mode of A op B is real. In mixed-mode arithmetic, the mode used to evaluate any portion of an expression is determined by the dominant type thus far encountered within the expression and the normal hierarchy of arithmetic operations; integer mode will be used when an integer type is first encountered and will be converted to the appropriate real mode when a real type is encountered.

REPLACEMENT STATEMENTS

4

Expressions, operators, and operands may be combined to form two types of statements, executable and nonexecutable. An executable statement performs a calculation or directs control of the program; a nonexecutable statement provides the compiler with information regarding variable structure, array allocation, and storage-sharing requirements. A source program is a group of FTN-60 statements.

Statements can be divided into four classes:

- Replacement
- Declarative
- Control
- Input/output and encode/decode

Arithmetic replacement statements incorporate expressions for addition, subtraction, multiplication, division, and exponentiation. Logical replacement statements may include relational and logical operators.

Declarative statements permit a programmer to define the mode of a variable as logical, character, real, integer, or double precision; enter data; reserve common storage and overlay the same storage locations with variables and arrays during program execution.

Control statements conditionally or unconditionally alter the sequence of program execution.

Input/output and encode/decode statements permit transfer of data from one storage location to another or between computer storage and external equipment. Conversion and editing specifications provide diversity in input/output formats.

REPLACEMENT STATEMENT

The general form of the arithmetic replacement statement is:

$$v = e$$

e is an arithmetic, logical, or relational expression, and v is any simple or subscripted variable name. The operator $=$ means that v is replaced by the value of expression e , with conversion for mode if necessary.

For example:

1	5	7
		RESLT=X+Y-2. *R
		SUMX=X+Y+Z
		ARG(LAB)=2 *X+COMP**2
		SCEL=BIG .LE. SMALL
		TAB= .NOT. X .OR. Y .OR. Z .AND. A
		EE=AA.GE.BB.OR.CC.GE.DD
		BOOL= .TRUE.

MULTIPLE REPLACEMENT STATEMENT

The multiple replacement statement is an extension of the arithmetic replacement statement. It is evaluated right to left.

$$v_n = v_{n-1} = \dots = v_2 = v_1 = e$$

e must be an arithmetic expression; v_i represents simple or subscripted variables and may be any of the standard types.

The multiple replacement statement indicates that each of the variables $v_1 \dots v_n$ is replaced by the value of the succeeding variable or by the value of e , in a manner analogous to that employed in mixed-mode arithmetic statements.

For example:

Problem: Convert radians to degrees and minutes.

Solution:

1	5	7
		DEGI=IDEG=DEG=BETA*57.296
		IMIN=(DEG-DEGI)*60

Where:

BETA = 2.6 radians (real)

DEG = 148.9696 degrees (real)

IDEG = 148 degrees (integer)

DEGI = 148.000 degrees (real)

IMIN = 58 minutes (integer)

Result: 2.6 radians = 148 degrees 58 minutes

MIXED-MODE REPLACEMENT STATEMENT

Although the type of an evaluated expression is determined by the type of the dominant operand, this does not restrict the types that an identifier may assume. The following chart shows the v-to-e relationship for all of the standard modes.

Arithmetic replacement statement: $v = e$

v Identifier

e Evaluated arithmetic expression

Type of e Type of v	Double	Real	Integer	Character
Double	$e \rightarrow v$	Convert e to double precision $real \rightarrow v$	Convert e to double precision $real \rightarrow v$	Convert e to double precision $real \rightarrow v$
Real	Convert e to $real \rightarrow v$	$e \rightarrow v$	Convert e to $real \rightarrow v$	Convert e to $real \rightarrow v$
Integer	Convert e to $integer \rightarrow v$	Convert e to $integer \rightarrow v$	$e \rightarrow v$	Right justify and zero fill; $e \text{ then } \rightarrow v$
Character	Convert to integer; lower 8 bits $\rightarrow v$	Convert e to integer; lower 8 bits $\rightarrow v$	Lower 8 bits of e $\rightarrow v$	$e \rightarrow v$

DECLARATIVE STATEMENTS

5

Declarative statements are nonexecutable statements that:

- Assign word structure to variables (type)
- Reserve storage for arrays and single variables (DIMENSION, COMMON, and SCRATCH COMMON)
- Assign initial values to variables (data)

The declarative COMMON, SCRATCH COMMON, EXTERNAL, TYPE, and DIMENSION statements may be placed in any order prior to the first executable statement of a program or subprogram. If the statements EQUIVALENCE and DATA are used, they must follow other declaratives in the indicated order and must precede the first executable statement.

TYPE STATEMENTS

A type statement designates the word structure of variable and function identities. FTN-60 recognizes five standard types of fixed length as follows:

<u>Type Declaration</u>	<u>Word Structure</u>
DOUBLE PRECISION	Two words per element
REAL	One word per element
INTEGER	One word per element
CHARACTER	Eight bits per element
LOGICAL	One bit per element

Rules:

- 1) Place the type declaration with other declaration statements prior to the first executable statement in a program or subprogram.
- 2) Unless declared, a variable is integer if the first character of its identifier is I, J, K, L, M, or N and real if the first character is any other letter.

- 3) In type statements, an identifier declared more than once causes a compiler diagnostic.
- 4) An array identifier in the list designates the entire array.

For example:

1	5	7	
			DOUBLE PRECISION EL, CAMINO
			REAL IDE63, JEWEL
			INTEGER QUID, PRO, QUO
			CHARACTER ALPHA, BETA, GAMMA
			LOGICAL A1, B2 (4), A147

DIMENSION STATEMENT

The nonexecutable statements DIMENSION, COMMON, and SCRATCH COMMON reserve storage for arrays. A subscripted variable in an expression represents an element of an array of variables.

DIMENSION $v_1 (s_1, s_2, s_3), v_2 (s_4, s_5, s_6), \dots$

An array name, v_1 , has up to three unsigned integer subscripts, s_1 , separated by commas.

The number of storage locations reserved for a given array is determined by the product of the subscripts in the subscript string and the number of words per type. An array may occupy a maximum of 65,535 words.

For example:

1	5	7	
			REAL HERCULES
			CHARACTER BEAT
			DIMENSION HERCULES (10,20), BEAT (5,3)

VARIABLE DIMENSIONS

If an entry in a declarator subscript is an integer variable name, the array is variable and the variable names are called variable dimensions. Such an array may appear only in a procedure subprogram. The dummy argument list of the subprograms must contain the array name and the integer names that represent the variable dimensions. The values of the actual parameter list of the reference must be defined prior to calling the subprogram and may not be redefined or undefined during execution of the subprogram. The maximum size of the actual array may not be exceeded. Every array in an executable program requires at least one associated constant array declaration through subprogram references.

For example:

```
SUBROUTINE XMAX (DATA, K, J)
```

```
  DIMENSION DATA (K, 6, J)
```

In a subprogram, a symbolic name that appears in a COMMON statement may not identify a variable array.

COMMON STATEMENTS

A program may be divided into independently compiled subprograms that use the same data. The COMMON statements reserve storage areas, scratch or data, that can be referenced by more than one subprogram.

COMMON/name/list assigns data common storage locations to variables and arrays designated in the list. Values in data common may be preset with a DATA statement.

COMMON, COMMON// or SCRATCH COMMON/name/list assigns scratch common locations to variables and arrays designated in the list. These may not be preset with data.

A maximum of 30 common (scratch and data) storage blocks may be allocated per subprogram. The number of common blocks in a program is limited only by available memory.

A subprogram may declare and use some of the same common blocks defined by another subprogram and at the same time define additional common storage.

Associated with each reserved storage block is an alphanumeric name that identifies the storage area. The name may be up to eight characters in length, and the first character must be alphabetic. Common with no name is treated as scratch common and is given a unique storage block.

Rules:

- 1) Place COMMON and SCRATCH COMMON statements with other declarative statements prior to the first executable statement in the program.
- 2) The list is composed of subscripted or nonsubscripted variable identifiers. If a nonsubscripted array name appears on the list, the dimensions must be defined by a DIMENSION statement in the subprogram.
- 3) Attempting to list an identifier in both scratch and data common doubly defines the variable and causes a diagnostic.
- 4) The order of identifiers in the COMMON and SCRATCH COMMON statements determines their order in the common storage block.
- 5) At the beginning of program execution, the contents of common (if not preset with a DATA statement) are undefined.
- 6) The type and quantity of identifiers determine the length of the common block.
- 7) A subprogram may rearrange the allocation of storage locations in common.

- 8) A subprogram may not increase the length of a common block assigned by the first subprogram loaded. However, it may use less common than the first program. The first program defining a common block must declare the maximum size of the block.
- 9) When a subprogram does not need all of the locations reserved in common, dummy variables in the statement achieve correspondence or reserved areas.

For example:

SCRATCH COMMON list

COMMON list

SCRATCH COMMON/BLK/list

COMMON/list

COMMON/CAT/list

COMMON/CAT/list/BLK/list

EQUIVALENCE STATEMENT

An EQUIVALENCE statement permits storage to be shared by two or more entities; it does not imply equality of entities. Each element in a given list is assigned the same storage (or part of the same storage) by the processor.

EQUIVALENCE (k_1), (k_2), ..., (k_n)

Each k is a list of the form:

a_1, a_2, \dots, a_m

Each a is either a variable name or an array element name (but not a formal parameter); the subscripts may contain only constants. m is greater than or equal to 2.

EQUIVALENCE may not be used to reorder COMMON or reposition the base. The base of an equivalence group is the element with the smallest address, or, if in common, it is the beginning element of the common block. The effect of an EQUIVALENCE statement upon a common assignment may be the lengthening of a common block beyond the last assignment for that block made by a COMMON statement. An element or array is brought into COMMON if it is equivalenced to an element in COMMON. Two elements in COMMON must not be equivalenced to one another.

The following example illustrates changes in block lengths as the result of an EQUIVALENCE declaration.

Given: Arrays A and B

Sa subscript of A

Sb subscript of B

For example:

A and C in common; B not in common

$S_b \leq S_a$ is a permissible subscript arrangement

$S_b > S_a$ is not a permissible subscript arrangement

Block 1

origin	A(1)		COMMON/BLOCK1/A(4), C
	A(2)	B(1)	DIMENSION B(5)
	A(3)	B(2)	EQUIVALENCE (A(3), B(2))
	A(4)	B(3)	
	C	B(4)	
		B(5)	

EQUIVALENCE statements must appear before DATA or executable statements in the program unit.

DATA STATEMENT

The DATA statement enables the programmer to assign values to program variables or variables in data common at compile time. The DATA statements appear after the specification statement group and prior to the first executable statement of the program unit. The general form of the DATA statement is:

$$\text{DATA } k_1/d_1/, k_2/d_2/,\dots,k_n/d_n/$$

Each k_i is a list containing names of variables, array elements, or array names implying the entire array. Each d_i is a list of constants, any one of which may be preceded by a constant repeat factor, j^* , where j is an integer constant. If a list contains more than one entry, the entries are separated by commas.

Rules:

- 1) The form of the constants in the list d_i , rather than the type of the identifier in the list k_i , determines the data type of the stored constants.
- 2) If the specified constant has a different storage requirement than the corresponding variable, a usage diagnostic is given and the constant is ignored.
- 3) Any subscript expression must be an unsigned integer constant.
- 4) Formal parameters may not appear in a list k_i .
- 5) When the form j^* appears before a constant, it indicates that the constant is to be repeated j times.
- 6) An ASCII or hexadecimal constant may appear in the list d_i .
- 7) A variable or array defined by a DATA statement may not be in scratch common.

For example:

1	5	7
		DIMENSION ARRAY(10)
		DATA X, Y, Z/1., 1.5, 2.0/, FACT(1)/1./
		DATA ARRAY/10*0./, BLANK/\$20202020/, LIST/4HDATA/



Program execution normally proceeds from one statement to the statement immediately following it in a FORTRAN program. Control statements are used to alter this sequence or to repeat program segments.

GO TO STATEMENTS

The statement labels used in the GO TO statements must be associated with executable statements in the same program unit as the GO TO statement.

UNCONDITIONAL GO TO

The form of the unconditional GO TO statement is:

GO TO k

k is a statement label.

Execution of this statement discontinues the current sequence of execution and resumes execution at the statement labeled k.

COMPUTED GO TO

The form of the computed GO TO statement is:

GO TO (k_1, k_2, \dots, k_n), i

k_i represents statement labels, and i is a simple integer variable. Variable i may only assume values $1 \leq i \leq n$. The transfer of control is to statement k_i .

This statement acts as a many-branch GO TO; i is preset or computed prior to its use in the GO TO statement. The variable i must not be specified by an ASSIGN statement. No compilation diagnostic is provided for this error, but the branching is undefined.

For example:

```
K = 4
.
.
.
GO TO (100, 200, 300, 400, 500), K
```

Statement number 400 will be executed next.

ASSIGNED GO TO

The form of the assigned GO TO statement is:

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

k_1 represents statement labels, and m is a simple integer variable.

This statement acts as a many-branch GO TO. The variable m is assigned an integer value k in a preceding ASSIGN statement; m cannot be defined as the result of a computation. No diagnostic is given if m is computed, but the branching is undefined. A compile-time diagnostic is given if the list is missing.

ASSIGN

The form of the GO TO assignment statement is:

```
ASSIGN k TO m
```

k is one of the statement labels appearing in the GO TO list; m is the simple integer variable in the assigned GO TO statement. At the time of execution of an assigned GO TO statement, the current value of m must have been assigned by an ASSIGN statement.

For example:

```
ASSIGN 10 TO NN
.
.
.
GO TO NN, (5, 10, 15, 20)
```

Statement number 10 will be executed next.

IF STATEMENTS

Conditional transfer of control is provided with the IF statements.

ARITHMETIC IF

The form of the arithmetic IF statement is:

IF (e) k_1, k_2, k_3

e is an arithmetic expression of type integer, real or double precision, and k_1 represents statement labels.

This statement causes expression e to be evaluated and control transferred according to that value.

e < 0 jump to k_1
e = 0 jump to k_2
e > 0 jump to k_3

For example:

IF (A*B-CSIN(X)) 10, 10, 20

IF (N) 5, 6, 7

IF (A/B*C) 10, 11, 12

LOGICAL IF

The form of the logical IF statement is:

IF (l) S

l is a logical expression and S is an executable statement (not a statement label). S must not be a DO or another IF. If e is true, S is executed. If e is false, S is treated as a CONTINUE statement.

For example:

IF (L) GO TO 3

IF (A.GT.16.0.OR.A.EQ.0.) A=B

DO STATEMENT

The DO statement makes it possible to repeat a group of statements immediately following the DO statement a number of times, changing the value of a simple integer variable for each repetition. The form of the DO statement is:

$$\text{DO } n \text{ } i = m_1, m_2, m_3$$

n is the label (number) of the statement ending the DO loop; i is a simple integer index variable. The m_i are indexing parameters; they must be unsigned nonzero integer constants or simple integer variables. i is initially set equal to m_1 , and after each execution of the DO loop, m_3 is added to i . (When omitted, m_3 assumes a value of 1.) When i becomes greater than m_2 , the DO loop is satisfied.

The DO statement, the statement labeled n , and any intermediate statements constitute a DO loop. Statement n may not be a GO TO, FORMAT, another DO statement, an arithmetic IF statement, RETURN, STOP, PAUSE, or a logical IF containing any one of these statements.

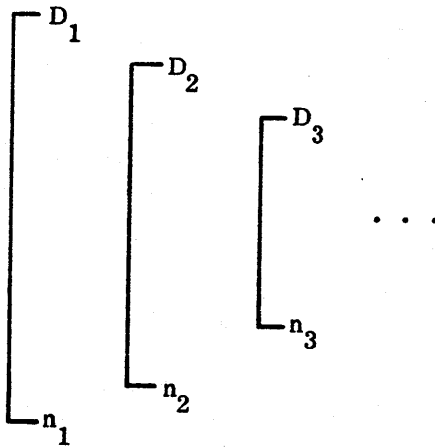
DO LOOP EXECUTION

Should m_1 exceed m_2 on the initial entry to the loop, the loop is executed once and control passes immediately to the statement following statement n . If it does not exceed m_2 , the loop is executed. The value of i is increased by m_3 and again compared with m_2 . The process continues until i exceeds m_2 . The DO loop is then satisfied, and control passes to the statement immediately following statement n .

If a transfer out of the DO loop occurs before the DO is satisfied, the value of i is preserved and may be used in subsequent statements.

DO NESTS

When a DO loop contains another DO loop, the grouping is called a DO nest. The last statement of a nested DO loop must either be the same as the last statement of the outer DO loop or occur before it. If D_1, D_2, \dots, D_m represent DO statements where the subscripts indicate that D_1 appears before D_2 (which appears before D_3 , etc.) and n_1, n_2, \dots, n_m represent the corresponding limits of the D_i , then n_m must appear before (or coincide with) n_{m-1}, \dots, n_{m-m} and n_2 must appear before (or coincide with) n_1 . Nesting may be to 50 levels.

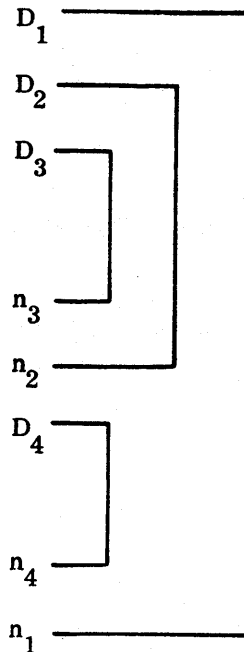


When two or more DO loops end with the same statement, the innermost DO loop is satisfied first.

Examples:

```

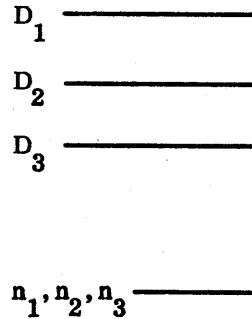
1) DO 1 I=1,10,2
    DO 2 J=1,5
      DO 3 K=2,8
        .
        .
        .
      3 CONTINUE
    2 CONTINUE
    DO 4 L=1,3
      .
      .
      .
    4 CONTINUE
  1 CONTINUE
  
```



```

2) DO 5 I=1,5
   DO 5 J=1,10
   DO 5 K=J,15
   .
   .
   .
5 CONTINUE

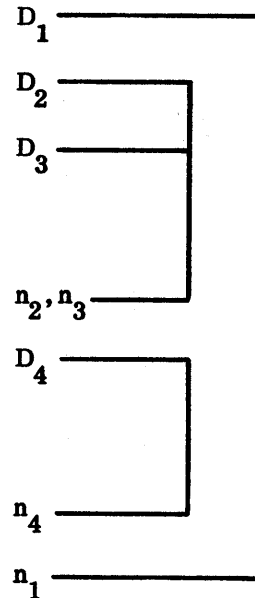
```



```

3) DO 100 L=2, LIMIT
   DO 10 I=1,10
   DO 10 J=1,10
   .
   .
   .
10 CONTINUE
   DO 20 K=K1, K2
   .
   .
   .
20 CONTINUE
100 CONTINUE

```

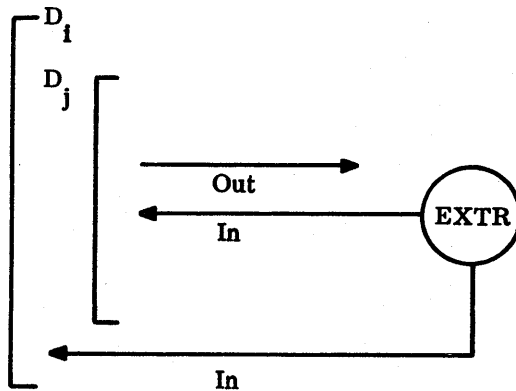


DO LOOP TRANSFER

In a DO nest, a transfer may be made from one DO loop into a DO loop that contains it and a transfer out of a DO nest is permissible.

The special case is transferring out of a nested DO loop and then transferring back to the nest. In a DO nest, if the range of D_i includes the range of D_j and a transfer out of the range of D_j occurs, then a transfer into the range of D_i or D_j is permissible.

In the following diagram, EXTR represents a portion of the program outside of the DO nest.



If two or more DO loops terminate at the same logical point and a transfer is made to the terminal statement for the outer DO loop, then the inner DO should have its own terminal statement.

CONTINUE STATEMENT

The form of the CONTINUE statement is:

CONTINUE

The CONTINUE statement acts as a do-nothing instruction; control passes to the next sequential program statement. The CONTINUE statement is frequently used as the last statement of a DO loop to provide a loop termination when a GO TO or IF would normally be the last statement of the loop.

PAUSE STATEMENT

The forms of the PAUSE statement are:

PAUSE

PAUSE Message

The PAUSE statement transfers control to a system subroutine. The following message is outputted on the console CRT.

PAUSE Message

The operator response is to accept the line.

14061100 B

STOP STATEMENT

The forms of the STOP statement are:

STOP

STOP Message

A STOP statement terminates object program execution, prints STOP Message on the standard output unit, and exits to the MPX.

PROGRAMMING CONSIDERATIONS

The MP-60/20 and MP-60/30 CPUs include lookahead stack (LAS) hardware and supporting firmware and software. The LAS significantly increases the instruction execution rate of the CPU by the incorporation of integrated circuit memory in the CPU for the retention of program loops. Each MP-60 program state is allocated a stack of 128 instructions.

The MP-60 FORTRAN compiler has been designed to generate object code that uses the capabilities of the LAS. The FORTRAN programmer does not generate any special statements, but should be aware of internal compiler operation.

DO LOOP statements are analyzed by the compiler to determine if the loop can execute completely in the stack. A DO LOOP qualifies if it contains no external references (e.g., CALL statements or functions) and consists of 125 or less machine instructions (one instruction is needed for loop control and two instructions following are automatically loaded into the stack). The loop length can be determined by obtaining an assembly listing (A-option on FTN control statement) or a cross-reference listing (R-option) and computing the length from relative program locations. A program branch instruction (GO TO, IF, etc.) potentially causes the LAS pointers to be reset. The LAS pointers are reset (and the stack effectively reloaded) when a branch is attempted beyond stack limits. The compiler, therefore, substitutes stack fill branch instructions for normal branch instructions in DO LOOPS that qualify for stack execution. Thus, the first pass through the loop causes it to be loaded entirely into the stack. Normal sequential program execution can also take advantage of the LAS. This is due to the automatic read of the next two sequential instructions. However, each branch causes a reload of stack pointers and excessive branching can degrade performance.

SUBPROGRAMS

7

MAIN PROGRAM AND SUBPROGRAMS

A main program is a set of FORTRAN statements bounded by a PROGRAM statement and an END statement. Execution starts at the beginning of the main program.

Subprograms (subroutines and functions) are sets of instructions that may be written and compiled separately from the main program and may be referred to by the main program.

A calling program is a main program or subprogram that refers to subroutines and functions.

A subprogram name must be unique within the subprogram. The name of a function determines the type of a subprogram in the same manner that variable names determine types of variables. Names of subroutine subprograms are not classified by type.

The PROGRAM statement is the first statement of a main program; it can be used only once within a main program. The form of the PROGRAM statement is:

PROGRAM name

name is a one-to-eight alphanumeric identifier beginning with an alphabetic character.

SUBROUTINE SUBPROGRAMS

A subroutine subprogram is composed of a set of FORTRAN statements bounded by a SUBROUTINE statement and an END statement. A subroutine subprogram performs operations or calculations that may or may not return values to the calling program.

Subroutine subprograms are compiled independently of the main program and may be compiled in a separate run.

SUBROUTINE STATEMENT

A subroutine begins with one of the following statements:

SUBROUTINE name

SUBROUTINE name (P_1, P_2, \dots, P_n)

A subroutine name contains up to eight characters; the first character is alphabetic. The name must not appear in a declarative statement or within the subroutine subprogram.

A SUBROUTINE statement can contain one to 63 formal parameters, p_i ; they may be array names, unsubscripted variables, or names of other functions or subroutine subprograms. Formal parameters must not appear in COMMON, DATA, or EQUIVALENCE declarative statements within the subroutine subprogram.

CALL STATEMENT

A reference to a subroutine is a call upon a computational or operational procedure. No resultant value is identified or associated with the name of the subroutine. The subroutine subprogram returns values, if any, to the main program through formal parameters or common. The executable statement in the calling program for referring to a subroutine is:

CALL name

CALL name (p_1, p_2, \dots, p_n)

The CALL statement transfers control to the subroutine named. A RETURN or END statement in the subroutine subprogram returns control to the calling program. A called subroutine may not call the calling program or itself.

The actual parameters, p_i , of a subroutine call must agree in order, number (1 to 63), and type with the formal parameters of the subroutine subprogram. The following forms are acceptable for actual parameters:

- Arithmetic expression
- Array name
- Constant
- Function reference
- Variable, simple or subscripted
- Subroutine name
- Relational expression
- Logical expression

A function reference, used as an actual parameter, must also be used in an EXTERNAL statement in the calling program.

When a subroutine is used with a parameter list, the subroutine name and its parameters must appear as separate actual parameters.

Examples:

1) Subroutine Subprogram

1	5	7
		SUBROUTINE ISHTAR (Y, Z)
		COMMON//X(100)
		Z=0
		DO 5 I=1,100
	5	Z=Z+X(I)
		CALL Y
		RETURN
		END

Calling Program Reference

1	5	7
		COMMON//A(100)
		EXTERNAL PRNTIT
		.
		.
		.
		CALL ISHTAR (PRNTIT, SUM)

The formal parameters, Y and Z, in the subroutine subprogram are replaced by PRNTIT and SUM. CALL Y is a call to subroutine PRNTIT; PRNTIT must appear in an EXTERNAL statement for the compiler to recognize it as a subroutine name.

2) Subroutine Subprogram

1	5	7
		SUBROUTINE BLVDLR (A,B,W)
		W=2. *B/A
		END

Calling Program Reference

1	5	7
		CALL BLVDLR (X(I), Y(I), W)
		.
		.
		CALL BLVDLR (X(I)+H/2., Y(I)+C(1)/2., W)
		.
		.
		CALL BLVDLR (X(I)+H, Y(I)+C(3), Z)

FUNCTION SUBPROGRAMS

A function subprogram is defined externally to the program unit that references it. It is headed by a FUNCTION statement and terminated with an END statement. At least one RETURN statement is required to return control to the referencing program unit.

DEFINING FUNCTION SUBPROGRAMS

The form of the FUNCTION statement is:

t FUNCTION name (a_1, a_2, \dots, a_n)

t is either INTEGER, REAL, DOUBLE PRECISION, LOGICAL, or empty; name is the symbolic name of the function to be defined. a_i , called a dummy argument, represents a variable name, an array name, or an external procedure name.

The following restrictions apply to the construction of a function subprogram.

- 1) A function must be typed in the calling program unit and in the function subprogram. Typing may be implicit or explicit (refer to Section 5). In the function subprogram, explicit typing may be done with the FUNCTION statement or with a type statement. Character functions can only be explicitly defined with a type statement.
- 2) The symbolic name of the function must also appear as a simple variable name in the defining subprogram. During every execution of the subprogram, this variable must be assigned a value at least once, either by appearing on the left-hand side of an arithmetic assignment statement or through its inclusion in an input list. The value of the variable at the time of execution of any RETURN statement in this subprogram is called the value of the function.
- 3) The symbolic names of the dummy arguments may not appear in an EQUIVALENCE, COMMON, or DATA statement in the function subprogram.
- 4) The function subprogram may assign values to one or more of its arguments in order to return results in addition to the value of the function.
- 5) The function subprogram may contain any statement except the following: PROGRAM, SUBROUTINE, another FUNCTION statement, or any statement that directly or indirectly references the function being defined.
- 6) The argument list must not be empty.

REFERENCING FUNCTION SUBPROGRAMS

A function reference consists of the function name followed by an actual argument list enclosed in parentheses. This reference may be used as a variable in an arithmetic or logical expression. The actual arguments, which constitute the argument list, must agree in order, number, and type with the corresponding dummy arguments in the defining program unit. The function name must assume type implicitly or appear in a type statement.

An actual argument in an external function reference may be one of the following:

- Variable name
- Array element name
- Array name
- Expression
- Name of an external procedure

If an actual argument is an external function name or a subroutine name, then the corresponding dummy argument must be used within the called function as an external function name or a subroutine name, respectively. If the actual argument is an expression, then this association is by value rather than by name. If an actual argument corresponds to a dummy argument that is assigned a value in the referenced function subprogram, the actual argument must be a variable name, an array element name, or an array name. Unless it is also a dummy argument, an external function name that is used as an actual argument must be specified in an EXTERNAL statement.

Examples:

1) Function Subprogram

```
FUNCTION GREATER (A, B,)  
  
IF (A .GT. B) GO TO 2  
  
1 GREATER = A+B  
  
RETURN  
  
2 GREATER = A-B  
  
RETURN  
  
END
```

Function Reference

```
W(I, J) = FA + FB - GREATER(C-D, 3. *AX/BX)
```

2) Function Subprogram

```
FUNCTION PSYCHE (A, B, X)  
  
CALL X(A)  
  
PSYCHE = A/B*2.0+(A-B)  
  
RETURN  
  
END
```

Function Reference

EXTERNAL EROS

.
. .
. . .

R = S-PSYCHE (TLIM, ULIM, EROS)

NOTE:

X is not the name of an external subroutine, but is a parameter that represents the name of a subroutine. The subroutine that is eventually called is EROS.

EXTERNAL STATEMENT

When a CALL statement or function reference contains the name of a subroutine or function in its list of actual parameters, the name must be declared in an EXTERNAL statement as follows:

EXTERNAL name₁, name₂, , name_n

name₁ is a function or subroutine name used as a parameter.

The EXTERNAL statement must precede the first executable statement of any program in which it appears.

For example:

To make function reference $\text{PHI}(p_1, p_2)$ in the statement $C=D-\text{PHI}(Q(K), \text{SINF})$, function SINF is an actual parameter of the function PHI and must be declared in the EXTERNAL statement.

1	5	7
		EXTERNAL SINF

PHI, the function originally referenced, begins with the following statements.

1	5	7
		FUNCTION PHI(ALFA, PHI2)
		PHI=PHI2 (ALFA)

Formal parameter ALFA takes the value Q(K); formal parameter PHI2 calls for SINF. Thus, the function subprogram PHI calculates the sine of Q(K).

ENTRY STATEMENT

The form of the ENTRY statement is:

ENTRY name

The ENTRY statement identifies an alternate entry point in a subprogram to be entered if the name of the entry point rather than the normal function name is referenced in a statement. ENTRY may not be labeled or be within a DO loop.

ENTRY CALL OR REFERENCE

To enter a subprogram at the ENTRY statement, the name of the entry point is called (subroutine) or referenced (function) in the same way as a subroutine or function.

ENTRY names must agree with the type of the function name when used in a function subprogram.

The actual parameters with the ENTRY statement must agree in type and mode with the formal parameters in the FUNCTION or SUBROUTINE statement for the subprogram.

For example:

1	5	7
	45	R=S+JAM (Q,2.*P)

Subprogram execution would begin at ENTRY JAM in the subprogram.

1	5	7
	10	FUNCTION JOE(X, Y) JOE=X+Y RETURN ENTRY JAM IF END

RETURN STATEMENT

The form of the RETURN statement is:

RETURN

A subprogram normally contains one or more RETURN statements to indicate the end of logic flow within the subprogram and return control to the calling program.

In function references, control returns to the statement containing the function. In most subprograms, control returns to the calling program. A RETURN statement in the main program causes an exit to the monitor.

END STATEMENT

The form of the END statement is:

END

The END statement marks the physical end of a program, subroutine subprogram, or function subprogram. The END statement acts as a return to the calling program or function reference.

PROGRAM ARRANGEMENT

FORTRAN compilation assumes that all statements and comments inserted between a PROGRAM, SUBROUTINE, or FUNCTION statement and an END statement belong to one program. Comments inserted between END and a SUBROUTINE or FUNCTION statement are associated with the preceding program. A FINIS card follows the last program.



FORMAT SPECIFICATIONS

8

Data transmission between internal and external storage requires an I/O control statement and, in some cases, a FORMAT statement. The I/O statement specifies the I/O device and the process and a list of data to be moved. The FORMAT statement specifies the manner in which the data is to be converted, edited, and moved. In binary and buffered I/O statements, no FORMAT statement is used.

FORMAT STATEMENT

The ASCII I/O control statements (except BUFFER IN and BUFFER OUT) require a format list for internal and external conversion of the I/O list elements. A FORMAT statement is:

FORMAT ($q_1 t_1 z_1 t_2 z_2 \dots t_n z_n q_2$)

($q_1 t_1 z_1 t_2 z_2 \dots t_n z_n q_2$) is the format specification. Each q is a series of slashes or is empty. Each t is a field descriptor or group of field descriptors. Each z is a field separator. n may be zero.

A FORMAT statement must be labeled. It is nonexecutable and can appear anywhere in the program.

FIELD DESCRIPTORS

The format field descriptors are:

nPrFw.d	Real conversion without exponent
nPrEw.d	Real conversion with exponent
nPrDw.d	Double precision conversion with exponent
rIw	Integer conversion
r\$w	Hexadecimal integer
rLw	Logical conversion

rAw	ASCII conversion, left justified
rRw	ASCII conversion, right justified
wHh ₁ h ₂ ...h _w	Heading and labeling
wX	Intra-line spacing

Where:

- 1) The letters F, E, D, I, \$, L, A, R, H, and X indicate the manner of conversion and editing between the internal and external representations and are called the conversion codes.
- 2) w is a nonzero integer constant 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.
- 4) r, the repeat count, is an optional nonzero integer constant indicating the number of times to repeat the succeeding basic field descriptor.
- 5) nP 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.

FIELD SEPARATORS

The format field separators (also called delimiters) are the slash and the comma. A series of slashes is also a field separator. The field descriptors or groups of field descriptors are separated by a field separator.

The slash is used not only to separate field descriptors, but to specify demarcation of formatted records.

CONVERSION SPECIFICATIONS

The data elements in I/O lists are converted from external to internal or from internal to external representation according to FORMAT conversion specifications.

Ew.d OUTPUT

The Ew.d specification converts floating-point numbers in storage to the ASCII character form for output. The field occupies w positions in the output record; the corresponding rounded floating-point number appears right justified in the field as:

$$\wedge . \alpha \dots \alpha E \pm ee \quad \text{when } -38 \leq ee \leq 38$$

$\alpha \dots \alpha$ are the most significant digits of the integer and fractional part; ee are the digits in the exponent. If d is zero, the digits to the right of the decimal do not appear, but the exponent does. The fractional part contains a maximum of 11 digits. Field w must be wide enough to contain the integer portion, sign, decimal point, E, and the exponent; that is, $w \geq 6 + d$ is required.

When the field is not wide enough to contain the output value, asterisks are inserted in the entire field. A field width, w, of less than five causes a format error. If the field is longer than the output value, the quantity is right justified with blanks in the excess positions to the left.

Examples:

- 1) Proper use of Ew.d specification:

A contains +67.32 or -67.32

Right

```
WRITE (10,10) A
```

```
10 FORMAT (E10.3)
```

Result: $\wedge \wedge .673E+02$ or $\wedge -.673E+02$

- 2) Minus sign not provided for:

A contains +67.32 or -67.32

Wrong

```
WRITE (10,10) A
```

```
10 FORMAT (E8.3)
```

Result: .673E+02 or *****

- 3) w is larger than required:

A contains 412.679

```
WRITE (10,25) A
```

```
25 FORMAT (E14.4)
```

Result: ^^^^^ .4127E+03

Ew.d INPUT

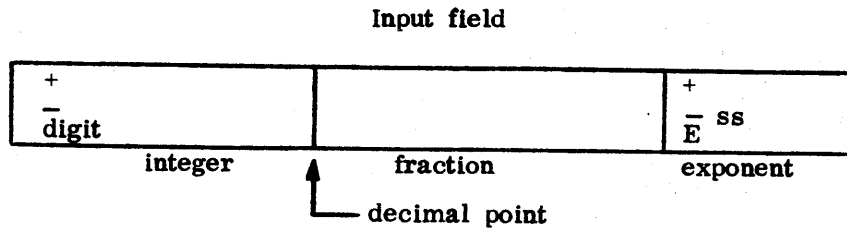
The Ew.d input specification converts the number in the input field to real and stores it in the appropriate location in memory.

w specifies the total number of characters in the input field. In the left-to-right scanning process, blanks in the field are interpreted as zeros.

The subfields for an input value may include integer, fraction, and exponent.

$\pm n. m$	$\pm n. m \pm s$	$\pm n. m E s$	$\pm n. m E \pm s$
$\pm n.$	$\pm n. \pm s$	$\pm n. E s$	$\pm n. E \pm s$
$\pm. m$	$\pm. m \pm s$	$\pm. m E s$	$\pm. m E \pm s$
$\pm n$	$\pm n \pm s$	$\pm n E s$	$\pm n E \pm s$

Subfield structure of the input field:



An integer subfield begins with a sign (+ or -) or a digit followed by a string of digits and ends with a decimal point, E, sign, or the end of w.

A fraction subfield begins with the decimal point, includes a string of digits, and ends with a sign, E, or the end of w.

An exponent subfield may begin with E or a sign. When it begins with E, the sign may appear between E and the digits in the exponent. The digits in the exponent must be less than or equal to 38; the entire input quantity must be in the range of 10^{-38} to 10^{38} .

When no decimal point is present in the input field, d in the Ew.d specification is a negative power factor of ten. The internal representation of the input quantity becomes:

$$(\text{integer subfield}) \times 10^{-d} \times 10^{(\text{exponent subfield})}$$

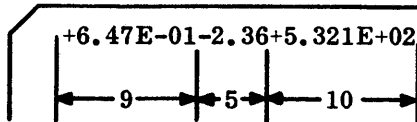
For example, if the specification is E7.7, the input quantity 3267+05 is converted and stored as $3267 \times 10^{-7} \times 10^5 = 32.67$.

When E conversion is specified and a decimal point occurs in the input field, the decimal point overrides d. The input quantity 3.67294+5 may be read by any specification allotting necessary field length but will always be stored as 3.67294×10^5 .

The field length w must be the same as the length of the input field. When w is too long, incorrect numbers may be read, converted, and stored as shown in the following example. When w is too short, a portion of the input field may be left unread. The field w includes significant digits (maximum of 17), sign, decimal point, E, and exponent.

For example:

Input Card

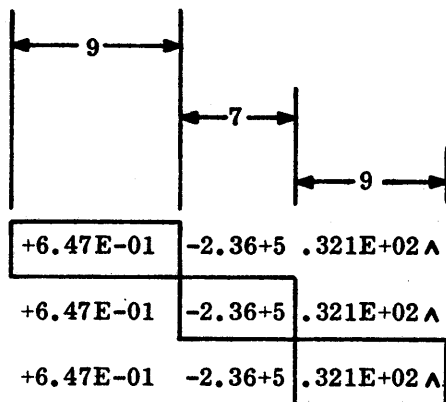


Incorrect Specification

READ 20, A, B, C

20 FORMAT (E9.3, E7.2, E9.3)

Reading proceeds as follows:



First, `+6.47E-01` is read, converted, and placed in location A.

Next, `-2.36+5` is read, converted, and placed in location B. The number desired was `-2.36` but the specification error (E7.2 instead of E5.2) caused the two extra characters to be read. The number read (`-2.36+5`) is legitimate under the definitions and restrictions.

Finally, `.321E+020` is read, converted, and placed in location C. Here again, the input number is legitimate although it is not the number desired.

In this example, numbers are incorrectly read, converted, and stored, yet there is no immediate indication that an error has occurred.

Examples:

<u>Input Field</u>	<u>Ew.d Input Specification</u>	<u>Converted Value</u>	<u>Remarks</u>
+143.26E-03	E11.2	.14326	Subfields all present
-12.437629E+1	E13.6	-124.37629	Subfields all present
8936E+004	E9.9	.08936	Input number converted as $8936 \times 10^{-9} \times 10^4$
327.625	E7.3	327.625	No exponent subfield
4.376	E5.0	4.376	Decimal point overrides specification
-.0003627+5	E11.7	-36.27	Integer subfield contains only minus
-.0003627E5	E11.7	-36.27	Integer subfield contains only minus
1E1	E3.0	10.	Input number converted as $1. \times 10^1$
E+06	E10.6	0.	No integer or fraction subfield; zero stored regardless of exponent
1.E ^ 1	E6.3	10.	Blanks interpreted as zeros

Fw.d OUTPUT

The Fw.d specification converts floating-point numbers in storage to ASCII form for output. The field occupies w positions in the output record; the corresponding list element must be a floating-point quantity that is converted and rounded to a decimal number, right justified in the field, as:

$\Delta \alpha \dots \alpha . \alpha \dots \alpha$

α represents the most significant digits of the number (maximum of 17 significant digits). The number of decimal places to the right of the decimal point is specified by d. If d is zero, the digits to the right do not appear. If the number being converted is positive, the + sign is suppressed. w must be large enough to include d, the decimal point, and the sign, in addition to the digits to the left of the decimal.

If the field w is too short to accommodate the number, asterisks appear in the field.

If the field w is longer than required to accommodate the number, it is right justified with blanks occupying the excess field positions to the left.

Examples:

- 1) Proper specification:

A contains +123.45678 or -123.45678

Right

```
WRITE (10,10) A
```

```
10 FORMAT (F10.5)
```

Result: ^ 123.45678 or -123.45678

- 2) w too small to accommodate integer portion:

A contains +123.45678

Wrong

```
WRITE (20,10) A
```

```
10 FORMAT (F8.5)
```

Result: *****

- 3) w too small to accommodate sign:

A contains -67.460

Wrong

```
WRITE (42,12) A
```

```
12 FORMAT (F6.3)
```

Result: *****

4) w larger than required

A contains 412.6727

Right

WRITE (10,25) A

25 FORMAT (F10.3)

Result: AAA 412.673

Fw.d INPUT

The Fw.d specification converts a number in an input field (w columns wide) to real and stores it in memory. The input field consists of an integer and a fraction subfield. An omitted subfield is assumed to be zero.

Permissible subfield combinations are:

- Integer and fraction
- Integer by itself
- Fraction by itself
- Any of the above followed by an exponent subfield

An integer subfield begins with a digit, + or -; blanks in the field are interpreted as zeros. The integer field is terminated by a decimal point or by the end of the input field.

A fraction subfield begins with a decimal point; it is terminated by the end of the input field.

In the Fw.d specification, d acts as a negative power factor of ten when the fraction subfield is not present. The internal representation is (integer subfield) $\times 10^{-d}$. For example, the specification F4.3 causes the input quantity 3267 to be converted and stored as $3267 \times 10^{-3} = 3.267$.

A decimal point in the input quantity causes d to be ignored. For example, 3.6789 may be read under any F6.d specification but will always be stored as 3.6789.

The maximum number of significant digits that may appear in the combined interger-fraction field is 19. Excess digits are discarded from the right during the conversion process.

The field length specified by w in Fw.d should always be the same as the actual length of the input field containing the input number. When it is too long, incorrect numbers may be read, converted, and stored. When it is too short, significant digits may be lost.

Examples:

<u>Input Field</u>	<u>Fw.d Input Specification</u>	<u>Converted Value</u>	<u>Remarks</u>
367.2593	F8.4	367.2593	Integer and fraction field
37925	F5.5	.37925	No fraction subfield; input number converted as 37925 x 10 ⁻⁵
-4.7366	F7.0	-4.7366	Decimal point overrides specification
.62543	F6.5	.62543	No integer subfield
.62543	F6.2	.62543	Decimal point overrides d of specification
+144.15E-03	F11.2	.14415	Exponents are allowed in F input

Dw.d OUTPUT

D conversion corresponds to Ew.d output. The field occupies w positions of the output record; the list item is a double precision quantity which appears as a decimal number, right justified.

$\Delta . \alpha \dots \alpha E \pm ee$ when $-38 \leq ee \leq 38$

Dw.d INPUT

D conversion corresponds to E conversion except that the list variables should be double precision names. D is acceptable in place of E as the beginning of an exponent subfield.

For example:

Input Card

```
-6.31675298443E-03+2.7189264531476293477528869D-09
```

18 15 17

DOUBLE Z, Y, X

READ 1, Z, Y, X

1 FORMAT (D18.11, D15.0, D17.4)

Iw OUTPUT

The Iw specification converts decimal integer values in the output list to ASCII character form for output. The output quantity occupies w output record positions; it will appear right justified in the field w, as:

$\alpha_1 \alpha_2 \dots \alpha_n$

α_i represents decimal digits (maximum 19) of the integer. When the integer is positive, the + sign is suppressed.

When the field w is larger than required, the output quantity is right justified with blanks occupying excess positions to the left. When the field is too short, asterisks are inserted in the field.

For example:

J contains -3762

K contains +4762937

L contains +13

WRITE (11, 10) J, K, L

10 FORMAT (I8, I10, I5)

Result: -3762 4762937 13
 8 10 5

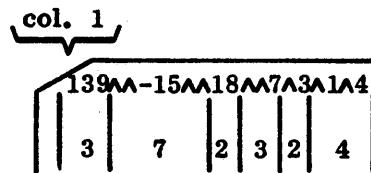
Iw INPUT

The Iw input specification converts the input field to a decimal integer. The field is w characters in length, and the corresponding list element must be a decimal integer quantity.

Input field w consists of an integer subfield that contains only a plus or minus, 0 through 9, or blank which is interpreted as zero. When a sign appears, it must precede the first digit in the field. The value is stored right justified in the specified variable.

For example:

Input Card



READ (13,10) I,J,K,L,M,N

10 FORMAT (I3,I7,I2,I3,I2,I4)

Result: I contains 139 L contains 7
 J contains -1500 M contains 3
 K contains 18 N contains 104

\$w OUTPUT

The \$w output specification converts internal binary to ASCII hexadecimal integers. The output quantity occupies w output record positions and appears as:

$\alpha_1 \dots \alpha_w$

α_i represents hexadecimal digits (maximum sixteen). No sign appears; a negative hexadecimal number is represented as it appears in storage in two's complement form. If w is greater than required, the number is right justified. If w is too small, the rightmost hexadecimal digits in storage occupy the output field; the left portion of the word is lost.

\$w INPUT

The \$w input specification provides a method of entering hexadecimal quantities into storage. The input field w has a maximum of 16 hexadecimal digits. The string of hexadecimal digits may be preceded by a sign; a negative sign causes the two's complement of the quantity to be stored. Blanks in the field are interpreted as zeros. Only hexadecimal digits (0-9, A-F) may appear.

Lw OUTPUT

The Lw specification is used to output logical values. The output field is w characters long, and the list item must be a logical element. A value of TRUE or FALSE in storage causes w-1 blanks followed by a T or F to be outputted.

For example:

I, K, L are TRUE; J is FALSE

LOGICAL I, J, K, L

PRINT 5, I, J, K, L

5 FORMAT (4L3)

Result: ^^T ^^F ^^T ^^T

Lw INPUT

This specification accepts logical quantities as list items. The field is considered true if the first nonblank character in the field is T; otherwise, it is false.

Aw OUTPUT

With the Aw output specification, internal code is converted to external alphanumeric characters.

$\alpha_i \dots \alpha_w$

α_i represents alphanumeric characters (maximum is eight characters). When w is larger than required, the character string is right justified with blank fill to the left. When the field is too small, the leftmost characters appear in the output field; any other characters are lost.

Aw INPUT

The Aw input specification accepts up to eight 8-bit characters. A blank in the input field is converted to the 8-bit equivalent ASCII code for blanks (20). If w exceeds the number of characters for the storage word, only the rightmost characters are stored in the variable defined in the I/O list. If w is less than the allowed number, the characters in the input field are stored left justified in the variable with blank fill to the right.

Rw OUTPUT

The Rw output specification is similar to the Aw specification for converting internal codes to external alphanumeric characters. The w output field is:

$$\alpha_1 \dots \alpha_w$$

α_i represents alphanumeric characters (maximum is eight characters). When w is larger than required to represent the characters, the character string is right justified with zero fill to the left. When the field is too small, the rightmost characters appear in the output field; any other characters are lost.

Rw INPUT

The Rw input specification, like the Aw specification, accepts up to eight 8-bit characters. A blank in the input field is converted to the 8-bit equivalent ASCII code for blanks (20). If w exceeds the allowed number of characters for the storage word, only the rightmost characters are stored in the variable defined in the I/O list. If w is less than the allowed number, the characters in the input field are stored right justified with binary zero fill to the left.

nP SCALE FACTOR

The D, E, and F conversions may be preceded by a scale factor:

$$\text{external number} = \text{internal number} \times 10^{\text{scale factor}}$$

The scale factor applies to Fw.d on both input and output and to Ew.d and Dw.d on output only. A scaled specification is written as:

nPDw.d

nPEw.d

nPFw.d

n is a signed integer constant.

Fw.d SCALING

For scaled input, the number in the input field is multiplied by 10^{-n} and stored. For example, if the input quantity 314.1592 is read under the specification 2PF8.4, the internal number is $314.1592 \times 10^{-2} = 3.141592$.

For scaled output, the number in the output field is the internal number multiplied by 10^n . In the output representation, the decimal point is fixed; the number moves to the left or right depending on whether the scale factor is plus or minus. For example, the internal number 3.1415926536 may be represented on output under scaled F specifications as follows:

<u>Specification</u>	<u>Output Representation</u>
F13.6	3.141593
1PF13.6	31.415927
3PF13.6	3141.592654
-1PF13.6	.314159

Ew.d SCALING

The scale factor has the effect of shifting the output number left n places while reducing the exponent by n. Using 3.1415926536, some output representations corresponding to scaled E specifications are:

<u>Specification</u>	<u>Output Representation</u>
E20.2	.31E+01
1PE20.2	3.14E+00
2PE20.2	31.42E-01
3PE20.2	314.16E-02
4PE20.2	3141.59E-03
5PE20.2	31415.93E-04
-1PE20.2	.03E+02

SCALING RESTRICTIONS

The scale factor is assumed to be zero if no other value has been given; however, once a value has been given, it will hold for all D, E, and F specifications following the scale factor within the same FORMAT statement. To nullify this effect in subsequent D, E, and F specifications, a zero scale factor, 0P, must precede a D, E, or F specification. Scale factors which result in exponents that exceed ± 38 may cause overflow conditions.

EDITING SPECIFICATIONS

Editing specifications define spacing between characters and lines, skip records, begin new records, and provide a method of adding headings and comments.

SPACE (wX)

The wX specification produces blanks in an output record or skips w characters of input quantities.

Examples:

1) Output spacing:

A contains 7
B contains 13.6
C contains 1462.37

INTEGER A

WRITE (11,10) A,B,C

10 FORMAT (I2,6X,F6.2,6X,E12.5)

Result: ^7^^^^^^13.60^^^^^^1.46237E+03
 | 2 | 6 | 6 | 6 | 12 |

2) Skipping on input:

Input Card

15.62	\$13.78	COST	15.97	
5	3	5	6	5

READ (K3,11) R,S,T

11 FORMAT (F5.2,3X,F5.2,6X,F5.2)

Result: R contains 15.62
 S contains 13.78 (^\$ spaced over)
 T contains 15.97 (ACOST^ spaced over)

wH OUTPUT

The wH output specification provides a method of including a set of w 8-bit ASCII characters in the output record in the form of comments, titles, heading, and carriage control characters.

An unsigned integer w specifies the number of characters to the right of the H to be included in the output record.

Examples:

- 1) No I/O list:

```
WRITE (17,20)
```

```
20 FORMAT (28H BLANKS COUNT IN AN H FIELD.)
```

```
Output record: ^BLANKS ^COUNT ^IN ^AN ^H ^FIELD.
```

- 2) Mixed specifications:

```
A contains 1.5
```

```
WRITE (14,30) A
```

```
30 FORMAT (6H LMAX=, F5.2)
```

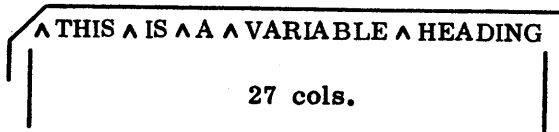
```
Output record: ^LMAX = ^1.50
```

wH INPUT

With the H specification, a new heading is read into an existing H field. When the new characters on an input record are read, the corresponding characters are placed into the format list designated in the I/O statement. A subsequent output statement puts the new characters in the output record. The field width, w, specifies the number of characters in the input field.

For example:

Input Card



READ (K4,10)

10 FORMAT (27H^^^^^^ ...^^^^^^)
 . 27 spaces
 .
 .

WRITE (K5,10)

Result: ^ THIS ^ IS ^ A ^ VARIABLE ^ HEADING

NEW RECORD (/)

A slash, signaling the end of an ASCII record, may appear anywhere in the specifications list. It may be separated from other list elements by commas; consecutive slashes may appear. During output, the slash is used to start new records, cards, or lines. During input, a slash specifies the beginning of the next record.

Examples:

1) WRITE (12,10)

10 FORMAT (20X,7HHEADING///6X,5HINPUT,19X,6HOUTPUT)

Output record:

^^ HEADING	line 1
	line 2
	line 3
^^^^^^ INPUT^^ OUTPUT	line 4

Each line corresponds to an ASCII record. The second and third records are null and produce the line spacing illustrated.

- 2) A contains -11.6
B contains .325
C contains 46.327
D contains -14.261

WRITE (11,11) A,B,C,D

11 FORMAT (2E10.2/2F7.3)

Result: ^-.116E+02^^.325E+00
^46.327-14.261

line 1
line 2

WRITE (12,11) A,B,C,D

11 FORMAT (2E10.2//2F7.3)

Result: ^-1.16E001^^3.25E-01
^46.327-14.261

line 1
line 2

- 3) AMAX(1) contains 3.62
AMAX(2) contains -4.03
AMAX(3) contains -9.78

WRITE (11,15) (AMAX(I),I=1,3)

15 FORMAT (8H ANSWERS,2(/),3F8.2)

Result: ^ ANSWERS

^^^^ 3.62^^ -4.03^^ -9.78

line 1
line 2
line 3

4) Read a pair of cards:

DIMENSION ARRAY(16)

READ 10, ARRAY

10 FORMAT (8F10.4)

Result: Two cards are read, both in the same format, to fill the 16 elements of ARRAY.

DIMENSION A(10), NT(20)

READ 11, A, NT

11 FORMAT (10F8.2/20I4)

Result: Two cards are read; the first fills the 10 elements of A; the second fills the 20 elements of NT.

REPEATED SPECIFICATIONS

Repetition of the field descriptors (except nH and nX) is accomplished by using the repeat count. If the I/O list warrants, the specified conversion will be interpreted repetitively up to the specified number of times.

Repetition of a group of field descriptors or field separators is accomplished by enclosing them within parentheses and optionally preceding the left parenthesis with an integer constant, called the group repeat count, indicating the number of times to interpret the enclosed grouping. If no group repeat count is specified, a group repeat count of one is assumed. This form of grouping is called a basic group.

A further grouping may be formed by enclosing field descriptors, field separators, or basic groups within parentheses. Again, a group repeat count may be specified. The parentheses enclosing the format specification are not considered as group delineating parentheses. MP-60 FORTRAN allows three inner levels of nesting.

For example, if two quantities, K and L, are to be printed, the program could be written:

WRITE (11,10) K, L

10 FORMAT (I2, I2)

Since the specifications for K and L are identical, the FORMAT statement may be written:

```
10 FORMAT (2I2)
```

When a group of format specifications repeats itself, a group repeat count may be used to simplify the statement.

```
FORMAT (E15.3, F6.1, I4, I4, E15.3, F6.1, I4, I4)
```

```
FORMAT (2(E15.3, F6.1, 2I4))
```

VARIABLE FORMAT

Format lists need not be supplied by FORMAT statements; instead, they can be placed in integer arrays. Placing format lists in arrays and referencing the arrays instead of the FORMAT statement permit the programmer to change, index, and specify formats at the time of execution.

Format arrays are prepared by storing a format list, including left and right parentheses, as it would otherwise appear with a FORMAT statement. Variable specifications can be read in from cards, changed with assignment statements, or preset in labeled common with a DATA statement.

For example:

Prepare an array for format list:

```
(E12.2, F8.2, I7, 2E20.3, F9.3, I4)
```

```
DIMENSION IVAR(8)
```

```
READ (K1,1) (IVAR(I), I=1,8)
```

```
1 FORMAT (8A4)
```

Result: IVAR(1) contains E12	IVAR(5) contains E20.
IVAR(2) contains .2, F	IVAR(6) contains 3, F9
IVAR(3) contains 8.2,	IVAR(7) contains .3, I
IVAR(4) contains I7, 2	IVAR(8) contains 4) ^^

When using the specifications, reference the array:

```
WRITE (12,IVAR) A,B,I,C,D,E,J
```

Specifications can be changed with assignment statements:

```
IVAR(4) = 4H^^^2
```

This removes I7 from the format list, permitting:

```
WRITE (12,IVAR) A,B,C,D,E,J
```

FORMAT CONTROL

Execution of a formatted READ or formatted WRITE statement initiates format control. Each action of format control depends on information jointly provided by the next element of the I/O list, if one exists, and the next field descriptor obtained from the format specification. If there is an I/O list, at least one field descriptor other than nH or nX must exist.

When a READ statement is executed under format control, one record is read when the format control is initiated, and thereafter additional records are read only as the format specification demands.

When a WRITE statement is executed under format control, a record is written each time the format specification demands that a new record be started. Termination of format control causes writing of the current record.

Except for the effects of repeat counts, the format specification is interpreted from left to right.

To each I, F, E, D, \$, A, R, or L basic descriptor interpreted in a format specification, there is one corresponding element specified by the I/O list. To each H or X basic descriptor there is no corresponding element specified by the I/O list, and the format control communicates information directly with the record. Whenever a slash is encountered, the format specification demands that a new record start or the preceding record terminate. During a READ operation, any unprocessed characters of the current record will be skipped at the time of termination of format control or when a slash is encountered.

Whenever the format control encounters an I, F, E, D, \$, A, R, or L basic descriptor in a format specification, it determines if there is a corresponding element specified by the I/O list. If there is such an element, it transmits appropriately converted information between the element and the record and then proceeds. If there is no corresponding element, the format control terminates.

If the format control proceeds to the last outer right parenthesis of the format specification, a test is made to determine if another list element is specified. If not, control terminates. However, if another list element is specified, the format control demands a new record start and control reverts to the group repeat specification terminated by the last preceding right parenthesis, or if none exists, then to the first left parenthesis of the format specification. This action of itself has no effect on the scale factor.

Examples of format control interaction with an I/O list are as follows:

- 1) List longer than format specifications:

DIMENSION NP(4)

WRITE (6,10) NP, TMAX, PMAX, DRPD, G, C, P1, T1, H1, S1

10 FORMAT (4I10/(4F10.2))

Result:	101	98	121	97
	1337.28	540.68	-.47	53.70
	71.20	123.00	-823.23	.00
	-.25			

- 2) List shorter than format specifications:

WRITE (6,10) N1, N2, I, J, A, B

10 FORMAT (4I10/(4F10.2))

Result:	100	200	10	20
	.23	100.00		

CARRIAGE CONTROL

The first character of a listable output record is used for printer carriage control and is not printed. Usually, this character is in H format in a FORMAT specification used to print or write on the standard output file.

<u>Control Character</u>	<u>Action Before Print</u>	<u>Action After Print</u>
1	Eject page	No space
0 (zero)	Space two lines	No space
+	No space	No space
-	Space three lines	No space
(blank)	Space one line	No space
other	Space one line	No space

INPUT/OUTPUT STATEMENTS

9

I/O statements control the transfer of information between the storage unit and an external device.

In the I/O control statements:

- i Indicates the unit number and must be a simple integer variable or an integer constant.
- n Identifies the format list and is either a FORMAT statement number or the name of the array containing the format list. Binary and buffered data transmission do not require n.
- list Indicates the variables for input or output.

I/O LISTS

The list portion of an I/O control statement indicates the data elements and the order of transmission from left to right. Elements may be variable names, array names, array elements, or DO-implying segments. If ASCII transmission is indicated, the type of each element must correspond to an appropriate conversion specification in the FORMAT statement.

DO-IMPLYING SEGMENTS

A DO-implying segment consists of one or more list element and indexing values. Dimensioned arrays may appear in the list with values specified for the range of the subscripts in an implied DO loop.

The general form for a DO-implying segment is:

$(\dots ((\text{list}, \gamma_1 = m_1, m_2, m_3), \gamma_2 = n_1, n_2, n_3), \dots, \gamma_i = z_1, z_2, z_3)$

m_k, n_k, \dots, z_k Unsigned integer constants or predefined positive integer variables. When the third indexing parameter (m_3, n_3, \dots, z_3) is omitted, a value of one is used for incrementing.

γ_i Index variables which must be simple integer variables.

A list element may be a simple variable, a dimensioned variable, or an array name.

The first index variable (γ_1) defined in the list is incremented first. Data named in the implied DO loops is transmitted by increments of m_3 until m_2 is exceeded. (When m_3 is omitted, the increment value is 1.) When the first index variable reaches m_2 , it is reset; the next index variable to the right (γ_2) is then incremented, and the process is repeated until the last index variable (γ_i) has been incremented.

The general form for arrays is:

$((A(I, J, K), \gamma_1=m_1, m_2, m_3), \gamma_2=n_1, n_2, n_3), \gamma_3=p_1, p_2, p_3)$

I, J, K Subscripts of A; must be in standard form

$\gamma_1, \gamma_2, \gamma_3$ May represent I, J, K; $\gamma_1 \neq \gamma_2 \neq \gamma_3$

A DO-implying segment for an array may replace a nest of DO loops.

DO 10 $\gamma_3 = p_1, p_2, p_3$

DO 10 $\gamma_2 = n_1, n_2, n_3$

DO 10 $\gamma_1 = m_1, m_2, m_3$

·
·
·

Transmit list elements by an input or output statement

·
·
·

10 CONTINUE

An implied DO loop may also be used to transmit a simple variable, a sequence of variables, or an array a number of times. In the segment $((A), K=1, 10)$, A will be transmitted 10 times.

The limit to which implied DO loops may be nested is determined by the length of the statement.

Examples:

- 1) Example of a DO-implying segment nested five deep:

```
((((A(I, J, K), B(M), C(N), N=1, 10, 1), M=1, 5), K=K1, K2, K3), J=1, 60, 15), I=1, 10, 1)
```

During execution, each subscript (index variable) is set to the initial index value: I=1, J=1, K=K1, M=1, N=1. The segment replaces a DO loop nest:

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

```
DO 15 J=1, 60, 15
```

```
DO 15 K=K1, K2, K3
```

```
DO 15 M=1, 5
```

```
DO 15 N=1, 10, 1
```

```
.
```

```
.
```

```
.
```

```
READ (61, 1) A(I, J, K), B(M), C(N)
```

```
1 FORMAT (...)
```

```
.
```

```
.
```

```
.
```

```
15 CONTINUE
```

- 2) Elements of A, a three-by-three matrix, will be transmitted by columns using:

```
((A(I, J), I=1, 3), J=1, 3)
```

- 3) Elements of A will be transmitted by rows using:

```
((A(I, J), J=1, 3), I=1, 3)
```

- 4) In the list segment (B(J), L, (A(I, L), I=1, L), J=3, 9, 3), L must have a value before it can be used as an index variable. The segment replaces a DO loop nest:

```
DO 11 J=3, 9, 3
.
.
.
READ (61, 1) B(J), L
1 FORMAT (...)
.
.
.
DO 11 I=1, L
.
.
.
READ (61, 2) A(I, L)
2 FORMAT (...)
.
.
.
11 CONTINUE
```

- 5) CAT, DOG, and RAT will each be transmitted 10 times with the segment:

```
(CAT, DOG, RAT, I=1, 10)
```

TRANSMISSION OF ARRAYS

In an I/O list, an array name without subscripts causes the entire array to be transmitted.

For example:

```
DIMENSION SPECS (7, 5, 3)
```

```
.  
. .  
. .
```

```
WRITE (20) SPECS
```

This example transmits the array SPECS as if under control of the following implied DO loop or nested DO loops.

```
..., ((SPECS(I, J, K), I=1, 7), J=1, 5), K=1, 3), ...
```

```
DO 10 K=1, 3
```

```
DO 10 J=1, 5
```

```
DO 10 I=1, 7
```

```
WRITE (20) SPECS (I, J, K)
```

```
10 CONTINUE
```

I/O UNITS

In FORTRAN programming, the user performs his I/O in terms of integer unit numbers. This unit number may be either an integer constant or a simple integer variable.

PARTIAL RECORDS

Information processed by the I/O statements is divided into records; each time an I/O statement is executed, a new record is processed. Thus, it is not possible to read or write several parts of a single record with more than one statement.

OUTPUT STATEMENTS

PRINT RECORD

The form of the print record statement is:

```
PRINT n, list
```

Transfers information from the storage locations in the list to the standard output unit. This information is transferred as line printer images, 136 characters or less per line, in accordance with format list n. The maximum record length is 136 characters; the first character of every record is used for carriage control on the printer and is not printed.

For example:

```
PRINT 16, A
```

```
16 FORMAT (10H ^ RESULT ^ = ^ ,F7.3)
```

WRITE BINARY RECORD

The form of the write binary record statement is:

```
WRITE (i) list
```

Transfers binary information from the storage locations given by the list identifiers to the specified unit i. If the list is omitted, the statement acts as a no-operation.

The number of elements in the list determines the number of physical records to be written on the unit. A physical record contains 118 words; the first word is a count word, and the remaining 117 words contain the transmitted data. The physical records written by one write binary record statement constitute one logical record. Physical records are blocked in 480-word blocks (to mass storage devices only).

For physical records in the logical record, the first word of all records except the k^{th} contains zero; the first word of the k^{th} record contains the integer k. If there is only one physical record, the first word contains the integer 1.

For example:

```
DIMENSION A(260), B(4)
```

```
WRITE (10) A, B
```

```
DO 5 I=1,10
```

```
5 WRITE (6) AMAX(I), (M(I,J), J=1,5)
```


WRITE ASCII RECORD

The form of the write ASCII record statement is:

WRITE (i,n) list

Transfers information from storage locations given by identifiers in the list to unit i, according to the format list n.

A logical record containing up to 136 characters is written on unit i as ASCII characters. Each logical record is one physical record. The number of elements in the I/O list and the format list n determines the number of records to be written on a unit. If the logical record is less than 136 characters, the remainder of the record is filled with blanks. Records are blocked in 480-word blocks (to mass storage devices only).

When the contents of the output file is to be printed, the first character of a record is a printer control character that will not be printed. If the programmer fails to allow for a printer control character, the first character of each output record is lost on the printed listing.

INPUT STATEMENTS

READ CARD RECORD

The form of the read card record statement is:

```
READ n, list
```

Reads one or more card images from the standard input unit, converts the information from left to right in accordance with format list n, and stores the converted data in the locations named in the I/O list. The images are in the form of 80-column Hollerith cards read from the standard input unit.

For example:

```
READ 10, A,B,C  
10 FORMAT (3F10.4)
```

READ BINARY RECORD

The form of the read binary record statement is:

```
READ (i) list
```

Transfers one logical record of information from unit i to storage locations named by the list identifiers.

The record being read must have been written in binary mode by a WRITE (i) list statement. The word count generated by the write statement is not transmitted to the input area. The number of words in the list must be equal to or less than the number of words transmitted in the corresponding write statement.

When the list is omitted, the binary read statement spaces over one logical record.

For example:

```
DIMENSION C(264), BMAX(10), M2(10, 5), A(100, 50)
.
.
.
READ (10) C
.
.
.
DO 7 I=1, 10

7 READ (6) BMAX(I), (M2(I, J), J=1, 5)

READ (5)      (skip one logical record on unit 5)

READ (6) ((A(I, J), I=1, 100), J=1, 50)
```

READ ASCII RECORD

The form of the read ASCII record statement is:

```
READ (i, n) list
```

Transfers one logical record of information from unit i to storage locations specified in the list, according to format list n.

The number of words in the list and the format specifications must conform to the record structure on the input unit (up to 136 characters). The record being read must have been written in ASCII mode.

Examples:

```
DIMENSION MB(1)
.
.
.
READ (10, 11) X, Y, Z
11 FORMAT (3F10.6)
.
.
.
READ (2, MB) (Z(K), K=1, 8)
```

MB(1) contains (F7.2) ^^

14061100 A

BUFFER STATEMENTS

The differences between the buffer statements and the read/write statements are:

- 1) In a buffer statement, mode must be specified (ASCII or binary).
- 2) The buffer statements are not associated with I/O lists or formatted lists. Data transmission occurs to or from an area of storage.
- 3) Only one physical record is transferred for each buffer request (for mass storage, a block is considered a record).
- 4) A buffer statement initiates data transmission and then returns control to the program, permitting it to perform other tasks while data transmission is in progress. Before the buffered data is used, the status of the buffer operation must be checked (by calling the IFUNIT function).
- 5) Buffer I/O is not allowed on unit numbers 61, 62, and 63.
- 6) Logical arrays cannot be transmitted with a buffer statement.

BUFFER IN

The form of the BUFFER IN statement is:

`BUFFER IN (i,p) (a,b)`

Where:

- i Unit number; an integer constant or variable $1 \leq i \leq 60$
- p Mode; an integer constant or variable
 - 0 Mode is ASCII
 - 1 Mode is binary
- a First variable in the block to be transmitted; a variable, array name, or array element reference (must not be type logical)
- b Last variable in the block to be transmitted; a variable, array name, or array element reference (must not be type logical)

This statement transmits one physical record of information in mode p from unit i to storage locations a through b. If the file being read was written by an ASCII write statement, only one physical record of 136 characters is read. Provision must be made for the count word buffered in with the data that was written with binary write statements.

When BUFFER IN requests transmission of more words than are on the record, the words are stored from a to k, where k is less than b and contains the last word on the record. If the record is larger than the buffer size, the remainder of the record is ignored.

The first word address must be less than or equal to the last word address or the job is terminated.

For example:

```
DIMENSION TEMP(50)
```

```
·  
·  
·
```

```
BUFFER IN (NT,0)(TEMP,TEMP(25))
```

```
BUFFER IN (10,0)(TEMP(26), TEMP(50))
```

BUFFER OUT

The form of the BUFFER OUT statement is:

```
BUFFER OUT (i,p) (a,b)
```

Where:

- i Unit number; an integer constant or variable $1 \leq i \leq 60$
- p Mode; an integer constant or variable
 - 0 Mode is ASCII
 - 1 Mode is binary
- a First variable in the block to be transmitted; a variable, array name, or array element reference (must not be type logical)
- b Last variable in the block to be transmitted; a variable, array name, or array element reference (must not be type logical)

This statement transmits information from storage locations a through b and writes one physical record on unit i in mode p. The physical record contains all the words from a to b.

The first word address must be less than or equal to the last word address or the job is terminated.

Because only one physical record is transmitted, the block size for a mass storage file must be large enough to contain the largest data block buffered out.

For example:

```
DIMENSION A(100)
.
.
.
BUFFER OUT (10,1)(A(1),A(100))
```

FILE CONTROL STATEMENTS

The auxiliary I/O statements REWIND, BACKSPACE, and ENDFILE may not reference unit numbers 61, 62, and 63.

REWIND

The form of the REWIND statement is:

```
REWIND i
```

Rewinds magnetic tape file *i* to the load point. When the tape is already rewound, the statement acts as a do-nothing statement.

A REWIND statement on a mass storage file is interpreted as a locate to block one.

BACKSPACE

The form of the BACKSPACE statement is:

```
BACKSPACE i
```

Backspaces magnetic tape file *i* one physical block. When the tape is already at the load point (rewound), BACKSPACE acts as a do-nothing statement.

A BACKSPACE statement on a mass storage file is interpreted as a locate to the beginning of the previous block.

ENDFILE

The form of the ENDFILE statement is:

```
ENDFILE i
```

Writes an end-of-file on magnetic tape or mass storage file *i*. The file is positioned at the end-of-file upon completion of the request.

INTERNAL TRANSMISSION STATEMENTS

The ENCODE and DECODE statements are comparable to ASCII read/write statements except that no peripheral equipment is used in the data transfer. Information is transferred under format specifications from one area of storage to another.

ENCODE

The form of the ENCODE statement is:

```
ENCODE (c,n,v) list
```

c is an integer constant or variable describing the number of characters per record in storage; *n* is the statement label of a FORMAT statement or the name of an integer array containing the format specifications; *v* is a variable, array element, or array name at which the first record is to start; and *list* is the list containing the data to be converted. The list has the same format as an I/O list.

The execution of this statement converts the information in the list according to the FORMAT statement and stores it in records starting at v, with c ASCII characters per record. If the format list attempts to convert more than c characters per record, a diagnostic is given. If the number of characters converted by the format list is less than c, the remainder of the record is filled with blanks.

Examples:

- 1) ENCODE may be used to calculate a field definition in a format specification at object time. Assume that in the statement FORMAT (2A8, 1m) the programmer wishes to specify m at some point in the program, subject to the restriction $2 \leq m \leq 9$. The following program permits m to vary.

```
PROGRAM VARY
INTEGER FMT(1)
.
.
.
IF (M .LT. 10 .AND. M .GT. 1) GO TO 1
STOP
1 ENCODE (8,100,FMT) M
100 FORMAT (6H(2A8,I, 11, 1H))
.
.
.
PRINT FMT, A, B, J
.
.
.
```


- 2) ENCODE may be used to convert internal integer data to ASCII codes and suppress printing of zero data.

```
PROGRAM CONVERT

DIMENSION IA(100)
.
.
.
DO 20 I=1,100

IF (IA(I) .EQ. 0) GO TO 10

ENCODE (4,100,IA(I)) IA(I)

100 FORMAT (A4)

GO TO 20

10 IA(I) = 4H

20 CONTINUE

PRINT 101, IA

101 FORMAT (10A12)

END
```

DECODE

The form of the DECODE statement is:

```
DECODE (c,n,v) list
```

c is an integer constant or variable indicating the number of characters per record in storage; n is the statement label of a FORMAT statement or the name of an integer array containing format specification; v is a variable, array element, or array name at which the first record starts; and list is the list in which converted information is stored. The list has the same format as an I/O list.

The execution of this statement converts and edits the information from records, starting at v and consisting of c ASCII characters each, and stores it in the variables specified by the list. When the FORMAT statement specifies more than c characters per record, a diagnostic is given. When fewer than c characters per record are specified, the remainder of the record is ignored.

For example:

DECODE may be used to read an input card under one of three different formats depending on the code punched in column one. Cards are read until -1 appears in column one.

```
PROGRAM READIN

DIMENSION BUFFER(10), DATA(20), IDATA(20), LDATA(20)

EQUIVALENCE (DATA(1), IDATA(1), LDATA(1))

5 READ 100, KEY, BUFFER

100 FORMAT (I2, 9A8, A6)

    IF (KEY .EQ. -1) GO TO 40

    GO TO (10, 20, 30), KEY

10 DECODE (78, 101, BUFFER) IDATA

101 FORMAT (I2, 19I4)

    GO TO 5

20 DECODE (78, 102, BUFFER) DATA

102 FORMAT (F2.0, 19F4.0)

    GO TO 5

30 DECODE (78, 103, BUFFER) LDATA

103 FORMAT (L2, 19L4)

    GO TO 5

40 STOP

END
```

STATUS CHECKING ROUTINE

The status checking routine provides extra capabilities in the I/O phase of FORTRAN programming.

I/O COMPLETE CHECK

Function: IFUNIT(I)

The I/O complete function is referenced with a unit number as its only parameter. It returns an integer result reflecting the status of the previous buffer operation on that unit. The result is not returned until the buffer operation is complete. The possible results are:

- 1 I/O operation is complete with no errors.
- 2 End-of-file mark was encountered
- 3 End of allocated area was detected during last operation
- 4 End of device was detected on the last operation
- 5 Irrecoverable I/O error was detected on the last operation

An example of a reference to the function is:

```
I=IFUNIT(3)
```

```
GO TO (10,20,30,40,50),I
```



The MP-60 FORTRAN compiler (FTN-60) is called into execution via an MPX library task control card. Parameters on this control card define compile options and are passed to the compiler through the PARM area assigned to the job.

A description of the FORTRAN control card and other information needed to compile and execute MP-60 FORTRAN programs under MPX are contained in this section. For detailed descriptions of control cards, both necessary and optional, refer to the MP-60 Reference Manual, Control Data publication No. 14306500.

FORTRAN CONTROL CARD

The MPX task name control card that causes MP-60 FORTRAN to be called, loaded, and executed appears as follows:

```
*FTN (field1, field2, ..., field6)
```

All fields are optional and may appear in any order on the card. Blanks are ignored, and illegal characters are assumed to be commas. Fields are of the following formats:

Parameter = logical unit,

Parameter,

Parameter can be one of the following letters: I, L, R, A, X, or P (refer to Table 10-1). Refer to the MPX/RT Reference Manual, Control Data publication No. 14062300, for legal logical unit (LU) numbers.

Sample *FTN card:

```
*FTN (L, A, X, P)
```

TABLE 10-1. PARAMETER/LU

Parameter	No LU	LU	No Field Present
I	Source input from LU 63, standard input unit.	Source input from named LU. If LU is other than 63, user must ensure that file is OPEN to job.	Source input from LU 63.
L	Source language listing and diagnostics appear on LU 62, standard output unit.	Source language listing written on specified LU.	No source listing provided.
R	Cross-reference list written on LU 62.	Cross-reference list written on specified LU. Note: it is recommended that cross reference unit be same as source listing unit.	No cross-reference listing provided.
A	Assembly language listing written on LU 62.	Assembly language listing written on specified LU.	No listing of assembly language produced.
X	Relocatable binary output written on LU 57, standard load and go file.	Relocatable binary card images of compiled programs written on named LU.	No relocatable binary file written.
P	Relocatable binary output written on LU 61, standard punch unit.	Relocatable binary card images of compiled programs written on named LU.	No binary deck provided.

Source language and assembly listings are written on the standard output unit, LU 62. The punchable output is written on the standard punch unit, LU 61, and automatically punched. Executable output is written on the standard load and go file, LU 57. Input source was read from the standard input unit, LU 63.

*FTN (I = 10, L, R, X = 11

Source input is from LU 10. Source output listing with cross-reference listing is on LU 62. Executable binary is written on LU 11.

CONTROL CARD NOTES

For a detailed description of all MPX control cards, refer to the MPX/RT Reference Manual (Section 2). Information relevant to FORTRAN is, however, contained in this section.

Core memory assigned to a job should be requested by the user via the MPX schedule statement (*SCHED). A minimum of 9 pages is needed to compile small FORTRAN programs. Larger programs require more memory, up to a maximum of 16 pages. When executing FORTRAN-compiled programs containing ASCII input or output statements, memory must be scheduled for I/O buffers (a 480-word buffer for each unique logical unit).

The FORTRAN compiler uses system scratch files 1 and 2 (LUs 59 and 60) as intermediate files. The user should reposition these files if they are to be used later in the job.

A FINIS card is used to notify the compiler that there are no more programs to be compiled. The word FINIS must begin in column 10.

1 10
FINIS

CALLING SEQUENCES

Programs written in MP-60 assembly language (COMPASS) may call or be called by programs compiled by MP-60 FORTRAN. Calling sequence conventions have been established for this purpose.

The calling sequence compiled for an external reference of the form
 CALL NAME (p_1, p_2, \dots, p_n) is:

RTJ	NAME
UJP	*(n+1)
VFD	16/0, 16/ p_1
VFD	16/0, 16/ p_2
.	
.	
VFD	16/0, 16/ p_n

Where:

NAME	Entry point of the program unit being referenced
p_i^*	Address of the i^{th} parameter

The main program, the set of FORTRAN statements bounded by a PROGRAM statement and an END statement, is entered initially by MPX. If the main program contains either ASCII I/O statements or STOP statements, a FORTRAN library routine with entry points Q8QENTRY and Q8QEXITS is provided to interface with MPX. The initialization performed by the main program includes clearing register X1, which is needed for double precision arithmetic operations and tests.

Subroutines, entered by the CALL statement or from COMPASS programs, use all registers. Functions save and restore all registers except RA-RF (registers 26 through 31) and return the function value in register RE (or RE-RF if the function is a type double precision). Both subroutines and function require register X to be zero upon entrance. Refer to Section 7 for further explanation of subprogram relationships.

*Note that if p_i is type character, address field would be 14/0, 18/ p_i .

SAMPLE DECK STRUCTURES

Compile only from standard input:

*JOB (ID=RLD, AC=645)

*SCHED(TL=30, CM=10)

*FTN(L, R)

[FORTRAN source deck]

END

FINIS

*EOJ

Compile and execute:

*JOB (ID=JYG, AC=645)

*SCHED(TL=200, CM=12)

*FTN(L, R, X, A)

[FORTRAN source deck]

END

FINIS

*LOAD

*RUN

[DATA]

*EOJ

Object time execution with PCC:

*JOB(LD=RLD,AC=645)

*SCHED(TL=200,CM=14)

*EQUIP(01=DP)

*TASK(ID=JYG, PCC=01)

[FORTRAN binary deck]

*RUN

[DATA]

*EOJ

FORTRAN program with COMPASS subprogram:

*JOB(ID=RLD,AC=645)

*SCHED(TL=200,CM=12,MT=1)

*EQUIP(01=MT)

*FTN(X=01) (Note: load and go to magnetic tape 01)

[FORTRAN source deck]

END

FINIS (Note: FINIS starts in column 10)

*REWIND(01)

*CMP(L,X)

[COMPASS source deck]

END

FINIS (Note: FINIS starts in column 10)

*REWIND(57)

*LOAD(01, 57)

*RUN

[DATA]

*EOJ



LIBRARY ROUTINES

A

<u>Library Function</u>	<u>Definition</u>	<u>Type of Argument</u>	<u>Type of Result</u>
ABS(a)	$ a $ (absolute value)	Real	Real
AINT(a)	Truncation	Real	Real
ALOG(a)	$\log_e(a)$	Real	Real
ALOG10(a)	$\log_{10}(a)$	Real	Real
AMAX0(a ₁ , a ₂ , ...)	$\max(a_1, a_2, \dots)$	Integer	Real
AMAX1(a ₁ , a ₂ , ...)	$\max(a_1, a_2, \dots)$	Real	Real
AMIN0(a ₁ , a ₂ , ...)	$\min(a_1, a_2, \dots)$	Integer	Real
AMIN1(a ₁ , a ₂ , ...)	$\min(a_1, a_2, \dots)$	Real	Real
AMOD(a ₁ , a ₂)	$a_1 \pmod{a_2}^*$	Real	Real
AND(a ₁ , a ₂)	$a_1 \wedge a_2$	Integer	Integer
ATAN(a)	arctan(a)	Real	Real
ATAN2(a ₁ , a ₂)	arctan (a ₁ /a ₂)	Real	Real
COS(a)	cos(a)	Real	Real
DABS(a)	$ a $	Double	Double
DATAN(a)	arctan(a)	Double	Double
DATAN2(a ₁ , a ₂)	arctan(a ₁ /a ₂)	Double	Double

<u>Library Function</u>	<u>Definition</u>	<u>Type of Argument</u>	<u>Type of Result</u>
DBLE(a)	Express single precision argument in double precision form	Real	Double
DCOS(a)	cos(a)	Double	Double
DDIM(a ₁ , a ₂)	a ₁ - min(a ₁ , a ₂)	Double	Double
DEXP(a)	e ^a	Double	Double
DIM(a ₁ , a ₂)	a ₁ - min(a ₁ , a ₂)	Real	Real
DINT(a)	Sign of a times largest integer ≤ a	Real	Double
DFLOAT(a)	Convert from integer to double	Integer	Double
DLOG(a)	log _e (a)	Double	Double
DLOG10(a)	log ₁₀ (a)	Double	Double
DMAX1(a ₁ , a ₂ , ...)	max(a ₁ , a ₂ , ...)	Double	Double
DMIN1(a ₁ , a ₂ , ...)	min(a ₁ , a ₂ , ...)	Double	Double
DMOD(a ₁ , a ₂)	a ₁ (mod a ₂)*	Double	Double
DSIGN(a ₁ , a ₂)	Sign of a ₂ times a ₁	Double	Double
DSIN(a)	sin(a)	Double	Double
DSQRT(a)	√a	Double	Double
ENABLE%	Initialize fault indicators and enable arithmetic class interrupts	Not Applicable	Integer
EXP(a)	e ^a	Real	Real
FDATE(a)	Subroutine to obtain system data	Any Type	ASCII (2 words)
FLOAT(a)	Conversion from integer to real	Integer	Real

<u>Library Function</u>	<u>Definition</u>	<u>Type of Argument</u>	<u>Type of Result</u>
IABS(a)	a	Integer	Integer
IDIM(a ₁ , a ₂)	a ₁ - min(a ₁ , a ₂)	Integer	Integer
FTIME(a)	Subroutine to obtain system time	Any Type	ASCII (2 words)
IARCHK(I)	Determine if arithmetic overflow has occurred. Returns 1 if there is a fault, 2 if there is no fault	Integer variable	Integer
IDINT(a)	Sign of a times largest integer ≤ a	Double	Integer
IDVCHK(I)	Determine if divide fault has occurred. Returns 1 if there is a fault, 2 if there is no fault	Integer variable	Integer
IFIX(a)	Conversion from real to integer	Real	Integer
IFNCHK(I)	Determine if function fault has occurred. Returns 1 if there is a fault, 2 if there is no fault	Integer variable	Integer
INT(a)	Sign of a times largest integer ≤ a	Real	Integer
IOVERFL(I)	Determine if exponent overflow has occurred. Returns 1 if there is a fault, 2 if there is no fault	Integer variable	Integer
ISHFT(a ₁ , a ₂)	Value is first argument shifted by second. If second argument is negative, shift is right; if positive, shift is left circular	Any Type	Integer
ISIGN(a ₁ , a ₂)	Sign of a ₂ times a ₁	Integer	Integer
MAX0(a ₁ , a ₂ , ...)	max(a ₁ , a ₂ , ...)	Integer	Integer
MAX1(a ₁ , a ₂ , ...)	max(a ₁ , a ₂ , ...)	Real	Integer

<u>Library Function</u>	<u>Definition</u>	<u>Type of Argument</u>	<u>Type of Result</u>
MIN0(a ₁ , a ₂ , ...)	min (a ₁ , a ₂ , ...)	Integer	Integer
MIN1(a ₁ , a ₂ , ...)	min(a ₁ , a ₂ , ...)	Real	Integer
MOD(a ₁ , a ₂)	a ₁ (mod a ₂)*	Integer	Integer
NOT(a)	\bar{a}	Integer	Integer
OR(a ₁ , a ₂)	a ₁ ∨ a ₂	Integer	Integer
SECOND(a)	System time in seconds	Any Type	Real
SIGN(a ₁ , a ₂)	Sign of a ₂ times a ₁	Real	Real
SIN(a)	sin(a)	Real	Real
SNGL(a)	Obtain most significant part of double precision argument	Double	Real
SQRT(a)/SQRTF(a)	\sqrt{a}	Real	Real
TAN(a)	tan(a)	Real	Real
TANH(a)	tanh(a)	Real	Real
XOR(a ₁ , a ₂)	a ₁ ⊕ a ₂	Integer	Integer

*a₁ (mod a₂) is defined as a₁ - $\left[\frac{a_1}{a_2} \right] a_2$, where [X] is the integer whose magnitude does not exceed the magnitude of X and whose sign is the same as X.

FORTRAN STANDARD OUTPUT

B

MP-60 FORTRAN DIAGNOSTIC RESULTS

The diagnostic results consist of null statement numbers, error messages, and program and common lengths. Null statements are statement labels that are not referenced in either assigned GO TO, format, or DO statements. Error messages are one of the following types: I - informative, E - fatal to execution, or F - fatal to compilation. The program length is given in decimal and does not include any associated object routines. All data and scratch common blocks are listed with their decimal lengths.

MP-60 FORTRAN CROSS-REFERENCE TABLE (SYMBOLS)

The cross-reference table contains a sorted listing of all symbols and their references. The relative program address, type, and line number of each appearance are printed for each symbol. The relative address is prefaced by a character representing the relocation type: C - common, P - program, X - external. In addition, for character or logical symbol types, the character or bit position is included. The type may be integer, real, double precision, character, or logical. All references refer to the line number on which the symbol appears. Multiple references per line are also listed. The character in parentheses indicates the type of reference:

- C Symbol appears in a common statement
- D Symbol appears in a dimension statement
- E Symbol appears in an equivalence statement
- O Symbol is used as operand
- S Symbol appears on lefthand side of replacement statement
- P Symbol appears in parameter list
- U Symbol represents LU number
- F Symbol represents format array

MP-60 FORTRAN CROSS-REFERENCE TABLE (LABELS)

The cross reference table also contains a sorted listing of all statement labels. The relative program address and all references are given for each label. In addition, labels representing formats are flagged as such. All references refer to the line number where the label appears. The character in parentheses refers to the type of reference:

- L Label is defined at this line
- A Label appears in ASSIGN statement
- J Label appears in GO TO or IF statement
- F Label is referenced as format
- D Label represents end of DO loop

FORTRAN DIAGNOSTICS

C

DIAGNOSTIC CLASSIFICATION AND ERROR MESSAGE CONVENTION

FORTRAN diagnostics are listed at the end of the source listing. Each diagnostic is classified by the code letter in the first column of the diagnostic. The code letters are as follows:

- I Informative diagnostic
- E Fatal to execution
- F Fatal to compilation

The error messages use the following conventions:

- \$0 Line number
- \$1 Integer number
- \$2 Integer number
- \$3 Single character
- \$4 Single character
- \$5 Symbol
- \$6 Symbol
- \$7 Label
- \$8 Start next line

MISCELLANEOUS ERROR MESSAGES

- 1) \$0 TABLE OVERFLOW, CROSS REFERENCE TABLE DISCARDED
- 2) \$0 MANAGED MEMORY TABLES OVERFLOWED (\$1)

Remarks: The integer number (\$1) is the absolute address of the compiler subroutine that increased the table size. This number is intended mainly for compiler checkout.

- 3) \$0 NO HEADER CARD

Remarks: Every FORTRAN compilation must begin with a PROGRAM, SUBROUTINE, or FUNCTION header card. The statement PROGRAM % JOB% is provided if the header card is missing.

- 4) \$0 UNRECOGNIZABLE STATEMENT
- 5) \$0 CARD OUT OF ORDER

Remarks: The required order of statements is:

Class 1: HEADER STATEMENT

PROGRAM

SUBROUTINE

FUNCTION

Class 2: DECLARATIVE STATEMENTS

DIMENSION

COMMON

TYPE

EXTERNAL

Class 3: EQUIVALENCE STATEMENTS

Class 4: DATA STATEMENTS

Class 5: EXECUTABLE STATEMENTS

Class 6: END

- 6) \$0 R-LIST TABLE OVERFLOWED
Remarks: The R-list table is a fixed length table; statement could be shortened.
- 7) \$0 THIS STATEMENT CANNOT HAVE A LABEL
- 8) LABEL \$7 IS UNDEFINED
- 9) VARIABLE \$5 DID NOT START ON A CHARACTER BOUNDARY
Remarks: The variable \$5 must be equivalenced to a logical variable that forced it off a character boundary.
- 10) VARIABLE \$5 DID NOT START ON A WORD BOUNDARY
Remarks: The variable \$5 must be equivalenced to a character or logical variable that forced it off a word boundary.
- 11) VARIABLE \$5 ATTEMPTS EXTEND ORIGIN OF COMMON
- 12) ARRAY \$6 HAS TOO MANY ELEMENTS
Remarks: An array may have a maximum of 65K elements. A double precision array may have a maximum of 32K elements.
- 13) \$0 ILLEGAL CHARACTER
- 14) \$0 ILLEGAL SYMBOL NAME
- 15) \$0 SYMBOL NAME EXCEEDS 8 CHARACTERS
- 16) \$0 UNBALANCED PARENTHESIS
- 17) \$0 ASCII CONSTANT CONTAINS MORE THAN 4 CHARACTERS
- 18) \$0 END OF STATEMENT REACHED BEFORE END OF ASCII CONSTANT
- 19) \$0 E-LIST OVERFLOW -- SIMPLFY STATEMENT
- 20) \$0 TWO . IN NUMERIC CONSTANTS
- 21) \$0 SYNTAX ERROR
- 22) \$0 CONSTANT TABLE OVERFLOW, SIMPLFY STATMENT
- 23) \$0 MORE THAN 255 CHARACTERS IN A CONSTANT

24) \$0 COMPILER ERROR -- CONVERT, SCANNER

Remarks: This error should never occur.

25) \$0 CONSTANT TOO LARGE

26) \$0 MORE THAN 4 CHARACTERS IN AN ASCII CONSTANT

27) \$0 SYNTAX ERROR

HEADER STATEMENT ERROR MESSAGES

1) \$0 SYNTAX ERROR

2) \$0 SYNTAX ERROR IN PARAMETER LIST

3) \$0 PARAMETER LIST ILLEGAL

4) \$0 PARAMETER LIST MISSING

DECLARATIVE STATEMENT ERROR MESSAGES

1) \$0 CANNOT DIMENSION AN EXTERNAL

2) \$0 ARRAY PREVIOUSLY DIMENSIONED

3) \$0 CANNOT DIMENSION A PROGRAM NAME

4) \$0 ILLEGAL CONSTANT IN DIMENSION

5) \$0 SYNTAX ERROR IN DIMENSIONS

6) \$0 ARRAY NAME MUST BE A FORMAL PARAMETER

Remarks: Refer to variable dimensions

7) \$0 VARIABLE DIMENSION MUST BE A FORMAL PARAMETER

8) \$0 DIMENSION MUST BE TYPE INTEGER

9) \$0 MORE THAN 3 DIMENSIONS

10) \$0 SYMBOL PREVIOUSLY TYPED

- 11) \$0 ILLEGAL EXTERNAL
- 12) \$0 NUMBER OF COMMON BLOCK NAMES IS LIMITED TO 30
- 13) \$0 BLOCK NAME IN BOTH DATA AND SCRATCH COMMON
- 14) \$0 PROGRAM NAME CANNOT BE IN COMMON
- 15) \$0 ILLEGAL SYMBOL IN VARIABLE LIST

EQUIVALENCE STATEMENT ERROR MESSAGES

- 1) \$0 EXTERNAL OR FORMAL PARAMETER IN EQUIVALENCES
- 2) \$0 EQUIVALENCE RELATION ERROR

Remarks: The relation error is caused by inconsistent equivalencing, such as:

EQUIVALENCE (A(1), B(1)), (A(2), B(3))

- 3) \$0 PROGRAM OR FUNCTION NAME IN EQUIVALENCE
- 4) \$0 TWO ELEMENTS OF SET IN COMMON
- 5) \$0 CANNOT SUBSCRIPT A SIMPLE VARIABLE
- 6) \$0 SYNTAX ERROR IN SUBSCRIPTS
- 7) \$0 TOO MANY SUBSCRIPTS IN ARRAY

Remarks: The number of subscripts must not exceed the number of dimensions.

DATA STATEMENT ERROR MESSAGES

- 1) \$0 SYMBOL \$5 IS A FORMAL PARAMETER
- 2) \$0 VARIABLE \$5 IS IN SCRATCH COMMON
- 3) \$0 VARIABLE \$5 HAS A CONSTANT OF DIFFERENT WORD SIZE
- 4) \$0 LIST CONTAINING \$5 HAS TOO MANY CONSTANTS
- 5) \$0 LIST CONTAINING \$5 HAS TOO MANY VARIABLES
- 6) \$0 REPEAT FACTOR MUST BE TYPE INTEGER

EXECUTABLE STATEMENT ERROR MESSAGES

- 1) \$0 LEFT SIDE OF REPLACEMENT MUST BE A VARIABLE
- 2) \$0 SYNTAX ERROR IN LEFT SIDE OF REPLACEMENT
- 3) \$0 SYNTAX ERROR IN LIST
- 4) \$0 VARIABLE MUST BE TYPE INTEGER
- 5) \$0 ILLEGAL USE OF SYMBOL
- 6) \$0 ILLEGAL STATEMENT AFTER THE LOGICAL IF
- 7) \$0 ENTRY NAME PREVIOUSLY USED
- 8) \$0 ILLEGAL SUBROUTINE NAME
- 9) \$0 SYNTAX ERROR IN EXPRESSION
- 10) \$0 ILLEGAL USE OF PROGRAM OR ENTRY POINT NAME
- 11) \$0 FUNCTION MUST HAVE ARGUMENTS
- 12) \$0 FUNCTION NAME MUST BE AN EXTERNAL
- 13) \$0 ILLEGAL USE OF FUNCTION NAME
- 14) \$0 ILLEGAL USE OF EXTERNAL PROCEDURE NAME
- 15) \$0 SUBSCRIPTED VARIABLE NOT DIMENSIONED
- 16) \$0 TOO MANY SUBSCRIPTS
- 17) \$0 SYNTAX ERROR IN SUBSCRIPTS
- 18) \$0 CONSTANT IN SUBSCRIPT OUT OF RANGE
- 19) \$0 R-LIST TABLE OVERFLOWED
- 20) \$0 OPERATOR TABLE OVERFLOWED
- 21) \$0 OPERAND TABLE OVERFLOWED
- 22) \$0 FUNCTION TABLE OVERFLOWED
- 23) \$0 ILLEGAL SYMBOL USED AS AN INDEX

- 24) \$0 NO PATH TO THIS STATEMENT
- 25) \$0 DOUBLY DEFINED STMT LABEL
- 26) \$0 STMT LABEL MUST BE BETWEEN 1 AND 99999
- 27) \$0 ILLEGAL STATEMENT LABEL
- 28) \$0 STMT LABEL USED AS A FORMAT NUMBER
- 29) \$0 ILLEGAL MODE CONVERSION
- 30) \$0 ILLEGAL OPERATION LOGICAL .OP. ARITHMETIC
- 31) \$0 LOGICAL OPERAND IN AN ARITHMETIC EXPRESSION
- 32) \$0 .XOR. IS MASKING ONLY
- 33) \$0 RELATIONAL OPERATOR CANNOT BE USED WITH LOGICAL OPERANDS
- 34) \$0 ILLEGAL EXPRESSION LOGICAL .OP. LOGICAL .OP. = + -*/
- 35) \$0 WRONG ARGUMENT COUNT FOR INTRINSIC FUNCTION
- 36) \$0 ILLEGAL EXPONENTIATION
- 37) \$0 RIGHT-HAND SIDE OF LOGICAL REPLACEMENT STMT . OR LOGICAL IF EXPRESSION IS NOT TYPE LOGICAL

DO LOOP STATEMENT ERROR MESSAGES

- 1) \$0 SYNTAX ERROR IN DO STMT
- 2) \$0 ILLEGAL DO-LOOP CONTROL VARIABLE
- 3) \$0 DO-LOOP CONTROL VARIABLE USED IN A PREVIOUS LOOP
- 4) \$0 ILLEGAL DO-LOOP PARAMETER
- 5) \$0 DO-LOOP DEPTH EXCEEDED
- 6) \$0 THE TERMINAL LABEL FOR THIS DO IS PREVIOUSLY DEFINED OR IS A FORMAT NUMBER
- 7) THE DO-LOOP DEFINED AT LINE \$1 TERMINATES AT THE END STMT.

- 8) THE CONTROL VARIABLE FOR THE LOOP DEFINED AT LINE \$1 IS MODIFIED IN THE LOOP
- 9) DO PAR M2 FOR THE LOOP DEFINED AT LINE \$1 IS MODIFIED IN THE LOOP
- 10) DO PAR M3 FOR THE LOOP DEFINED AT LINE \$1 IS MODIFIED IN THE LOOP
- 11) THE LOOP DEFINED AT LINE \$1 IS ENTERED FROM OUTSIDE OF ITS RANGE
- 12) THE TERMINAL LABEL FOR THE LOOP DEFINED AT LINE \$1 IF REFERENCED FROM OUTSIDE OF THE LOOP

I/O STATEMENT ERROR MESSAGES

- 1) \$0 ILLEGAL FORMAT NUMBER
 - 2) \$0 FORMAT NO. MUST BE BETWEEN 1 AND 99999
 - 3) \$0 FORMAT NO. NOT ASSIGNED TO A FORMAT
 - 4) \$0 SYNTAX ERROR, FORMAT
 - 5) \$0 VARIABLE FORMAT NOT A VARIABLE
 - 6) \$0 I/O SYNTAX ERROR, UPC NOT TYPE INTEGER
 - 7) \$0 I/O SYNTAX ERROR, UPC GREATER THAN 65535
 - 8) \$0 I/O SYNTAX ERROR, UPC MUST BE INTEGER CONSTANT OR INTEGER VARIABLE
- Remarks: UPC = unit mode (ASCII or binary) or characters per record.
- 9) \$0 ENCODE/DECODE/BUFFER STMT. SYNTAX ERROR, \$8 RECORD OR BUFFER ADDRESS MUST BE A VARIABLE
 - 10) \$0 ENCODE/DECODE/BUFFER STMT., \$8 RECORD OR BUFFER ADDRESS IS A FUNCTION OR ROUTINE NAME
 - 11) \$0 ENCODE/DECODE/BUFFER STMT., \$8 RECORD OR BUFFER ADDRESS IS NOT DIMENSIONED
 - 12) \$0 R-LIST OVERFLOW, SIMPLFY STATEMENT
 - 13) \$0 SYNTAX ERROR, I/O STMT (UNIT, FORMAT) LIST OR (UNIT) LIST

14) \$0 SYNTAX ERROR, I/O STMT (UNIT, PARITY) (A,B)

Remarks: A = First variable of the block

B = Last variable of the block

15) \$0 SYNTAX ERROR IN I/O LIST

16) \$0 SYMBOL IN I/O LIST IS NOT A VARIABLE

17) \$0 SYNTAX ERROR, I/O STMT.

18) \$0 SYNTAX ERROR, I/O STMT. (C, F, W)L

Remarks: Reference to encode/decode statements:

(C, F, W)L = (characters per record, format statement number, start of record) list.

19) \$0 ILLEGAL ARRAY REFERENCE, VARIABLE IS NOT DIMEN.

20) \$0 UNBALANCED PARENTHESIS IN I/O LIST

21) \$0 FORMAT DOES NOT HAVE A LABEL

22) \$0 ILLEGAL FORMAT LABEL

23) \$0 MULTIPLY DEFINED FORMAT LABEL

MAIN CONTROL TASK ERRORS

1) FTN ABORT - INSUFFICIENT CORE

Remarks: There is not enough core scheduled to open all the managed internal compiler tables.

2) FTN ABORT - BLK/DEBLK ERROR XX, ON LU YY

Remarks: An irrecoverable blocker or deblocker error has occurred on LU YY. Refer to MPX/RT Reference Manual, Section 6, for an explanation of the error XX.

3) FTN ABORT - INSUFFICIENT SCRATCH

Remarks: An end-of-file status was encountered on system scratch 1 (LU 60). The size of this file is a system parameter. To compile under the current system, the program must be segmented.

4) MORE THAN 19 CONTINUATION CARDS IGNORED

Remarks: The previous statement contained more than 19 continuation cards. Subsequent cards are ignored.

5) END OF FILE REACHED ON LU XX

Remarks: An end-of-file status was received on LU XX when one was not expected.

6) ERROR TABLE OVERFLOW - SUBSEQUENT ERRORS NOT LISTED

Remarks: The error table may be expanded to allow more errors by reassembling FTN.

OBJECT TIME INFORMATIVE ERROR MESSAGES

Format: ERROR IN (Name) CALLED FROM XXXX (Message)

Where: (Name) = The name of the object routine called by the user

XXXX = The address of the user call

(Message) = The unique error message

Messages:

1) RECORD OVERFLOW

Remarks: The format specification associated with the last I/O call causes a record overflow. For ASCII read or write, the record limit is 136 characters; for ENCODE/DECODE, the record limit is C characters.

2) ILLEGAL CHARACTER

Remarks: An illegal character appears in an input field (i.e., character not 0 through 9 for format type I). The input record is displayed.

3) NUMBER OUT OF RANGE

Remarks: The input value cannot be represented without loss of significant digits. Zero is returned to the user. The input record is displayed.

OBJECT TIME FATAL ERROR MESSAGES

Format: ERROR IN (Name) CALLED FROM XXXX (Message)

Where: (Name) = The name of the object routine called by the user

XXXX = The address of the user call

(Message) = The unique error message

Messages:

1) ILLEGAL LOGICAL UNIT

Remarks: The LU number is not in the range 1 through 63.

2) REFERENCE CONFLICT ON LU, XX

Remarks: LU XX cannot be referenced by both buffered and nonbuffered statements.

3) UNCHECKED END OF FILE ON LU, XX

Remarks: An end of file was encountered on LU XX on the previous read, and function IFUNIT was not called to clear the status.

4) UNCHECKED END OF ALLOCATED AREA ON LU, XX

Remarks: An end of allocated area status was received on LU XX on the previous read/write, and function IFUNIT was not called to clear the status.

5) UNCHECKED END OF DEVICE ON LU, XX

Remarks: An end of device status was received on LU XX on the previous read/write, and function IFUNIT was not called to clear the status.

6) BLOCKER/DEBLOCKER ERROR XX, ON LU YY

Remarks: Blocker/deblocker returned the fatal error status XX on LU YY. Refer to MPX/RT Reference Manual for explanation of blocker/deblocker error codes.

7) EXECUTION DELETED - NO TRANSFER ADDRESS

Remarks: A FORTRAN PROGRAM statement or COMPASS END card with a name in the address field generates the transfer address.

8) EXECUTION TERMINATED - INSUFFICIENT MEMORY

Remarks: More memory is needed for I/O buffers.

9) ILLEGAL SPECIFICATION

Remarks: The output field width for the associated format statement is not large enough to accommodate the value (i. e., $w \geq 6 + d$ is required).

10) BUFFER LENGTH ERROR

Remarks: The first word address specified in a BUFFER IN/OUT statement is greater than the last word address.

OBJECT TIME FORMAT ERRORS

Format: FORMAT ERROR N - XXXX

Where: N = The unique format error number

XXXX = The address of the format statement

All format errors are fatal.

<u>Error Number</u>	<u>Significance</u>
1	Format list does not begin with a left parenthesis.
2	Illegal repeat factor was encountered: a) Repeat factor = 0 b) Repeat factor is not an integer
3	Unrecognizable format conversion; the format conversion is designated by a symbol other than E, F, D, I, A, R, \$, L, H, X, or P.
4	Illegal field width or missing field width: a) Field width = 0 b) No field width present c) Illegal character in field width specification
5	A number precedes a slash, comma, or right parenthesis.
6	Parenthesis error: a) Repeat groups may not be nested b) A parenthetical grouping may not appear within a repeat group
7	More than one decimal point appears in the numeric field.
8	Numeric value exceeds 32767.
9	No non-H, non-X specification present in format when needed.

MATHEMATICAL LIBRARY INFORMATIVE ERROR MESSAGES

- 1) ERROR IN DSQRT CALLED FROM hhhh NEG ARGUMENT
- 2) ERROR IN SQRT CALLED FROM hhhh NEG ARGUMENT
- 3) ERROR IN ATAN2 CALLED FROM hhhh BOTH ARGUMENTS ARE ZERO
- 4) ERROR IN DATAN2 CALLED FROM hhhh BOTH ARGUMENTS ARE ZERO
- 5) ERROR IN MOD CALLED FROM hhhh SECOND ARG IS ZERO
- 6) ERROR IN AMOD CALLED FROM hhhh SECOND ARG IS ZERO
- 7) ERROR IN DMOD CALLED FROM hhhh SECOND ARG IS ZERO
- 8) ERROR IN ALOG CALLED FROM hhhh ARGUMENT TOO SMALL
- 9) ERROR IN ALOG10 CALLED FROM hhhh ARGUMENT TOO SMALL
- 10) ERROR IN DLOG10 CALLED FROM hhhh ARGUMENT TOO SMALL
- 11) ERROR IN DLOG CALLED FROM hhhh ARGUMENT TOO SMALL

Remarks: For arguments less than or equal to zero, messages 8 through 11 are reported for logarithms.

- 12) ERROR IN ISIGN CALLED FROM hhhh FIXED POINT OVERFLOW

Remarks: For the argument \$80000000, the above message is reported.

- 13) ERROR IN SIN CALLED FROM hhhh ARGUMENT TOO BIG
- 14) ERROR IN COS CALLED FROM hhhh ARGUMENT TOO BIG

Remarks: For arguments whose absolute value is greater than 2^{16} , messages 13 and 14 are reported.

- 15) ERROR IN TAN CALLED FROM hhhh ARGUMENT IS MULTIPLE OF $\pi/2$ -- TAN IS UNDEFINED
- 16) ERROR IN TAN CALLED FROM hhhh ARGUMENT TOO BIG

Remarks: For arguments whose absolute value is greater than 2^{19} , the above message is reported for tangent.

- 17) ERROR IN DSIN CALLED FROM hhhh ARGUMENT MAGNITUDE TOO BIG

- 18) ERROR IN DCOS CALLED FROM hhhh ARGUMENT MAGNITUDE TOO BIG
Remarks: For arguments whose absolute value is greater than 2^{50} , messages 17 and 18 are reported.
- 19) ERROR IN EXP CALLED FROM hhhh ARGUMENT TOO BIG
- 20) ERROR IN DEXP CALLED FROM hhhh ARGUMENT TOO BIG
Remarks: For arguments greater than $127 \ln 2$, messages 19 and 20 are reported.
- 21) ERROR IN IEI CALLED FROM hhhh NEG ARGUMENT
- 22) ERROR IN IEI CALLED FROM hhhh BASE AND POWER ARE ZERO
- 23) ERROR IN REI CALLED FROM hhhh BASE AND POWER ARE ZERO
- 24) ERROR IN DEI CALLED FROM hhhh BASE AND POWER ARE ZERO
- 25) ERROR IN RED CALLED FROM hhhh BASE IS ZERO AND POWER IS NEGATIVE
- 26) ERROR IN RED CALLED FROM hhhh BASE IS NEGATIVE AND POWER IS NONZERO
- 27) ERROR IN RED CALLED FROM hhhh BASE AND POWER ARE ZERO
- 28) ERROR IN RED CALLED FROM hhhh ARG MAG TOO BIG
- 29) ERROR IN RER CALLED FROM hhhh BASE IS ZERO AND POWER IS NEGATIVE
- 30) ERROR IN RER CALLED FROM hhhh BASE IN NEGATIVE AND POWER IS NONZERO
- 31) ERROR IN RER CALLED FROM hhhh BASE AND POWER ARE ZERO
- 32) ERROR IN RER CALLED FROM hhhh ARG MAG TOO BIG
- 33) ERROR IN DER CALLED FROM hhhh BASE IS ZERO AND POWER IS NEGATIVE
- 34) ERROR IN DER CALLED FROM hhhh BASE IS NEGATIVE AND POWER IS NONZERO
- 35) ERROR IN DER CALLED FROM hhhh BASE AND POWER ARE ZERO
- 36) ERROR IN DER CALLED FROM hhhh ARG MAG TOO BIG
- 37) ERROR IN DED CALLED FROM hhhh BASE IS ZERO AND POWER IS NEGATIVE

- 38) ERROR IN DED CALLED FROM hhhh BASE IS NEGATIVE AND POWER IS NONZERO
- 39) ERROR IN DED CALLED FROM hhhh BASE AND POWER ARE ZERO
- 40) ERROR IN DED CALLED FROM hhhh ARG MAG TOO BIG

Remarks: Messages 21 through 40 are error messages for exponentiation with I representing integer arguments, R representing real arguments, and D representing double precision arguments.



FORTRAN STATEMENTS

D

CLASS 1 STATEMENTS

- 1) PROGRAM s
- 2) SUBROUTINE s
- 3) SUBROUTINE $s(a_1, a_2, \dots, a_n)$
- 4) FUNCTION $f(a_1, a_2, \dots, a_n)$
- 5) REAL FUNCTION $f(a_1, a_2, \dots, a_n)$
- 6) DOUBLE PRECISION FUNCTION $f(a_1, a_2, \dots, a_n)$
- 7) INTEGER FUNCTION $f(a_1, a_2, \dots, a_n)$
- 8) LOGICAL FUNCTION $f(a_1, a_2, \dots, a_n)$

CLASS 2 STATEMENTS

- 1) EXTERNAL v_1, v_2, \dots, v_n
- 2) CHARACTER v_1, v_2, \dots, v_n
- 3) INTEGER v_1, v_2, \dots, v_n
- 4) REAL v_1, v_2, \dots, v_n
- 5) DOUBLE PRECISION v_1, v_2, \dots, v_n
- 6) LOGICAL v_1, v_2, \dots, v_n
- 7) DIMENSION $v_1, (l_1), v_2, (l_2), \dots, v_n, (l_n)$
- 8) COMMON //a/
- 9) COMMON /x₁/a₁/.../x_n/a_n

10) SCRATCH COMMON $/x_1/a_1/\dots/x_n/a_n$

11) COMMON a_1, a_2, \dots, a_n

CLASS 3 STATEMENTS

1) EQUIVALENCE $(k_1), (k_2), \dots, (k_n)$

CLASS 4 STATEMENTS

1) DATA $k_1/d_1/, k_2/d_2/, \dots, k_n/d_n/$

CLASS 5 STATEMENTS

1) $v_1 = v_2 = \dots = v_n = e$

2) GO to k

3) GO TO m, (k_1, k_2, \dots, k_n)

4) GO TO $(k_1, k_2, \dots, k_n), i$

5) IF (e) k_1, k_2, k_3

6) IF (e) s

7) DO n i = m_1, m_2, m_3

8) ASSIGN k TO i

9) CONTINUE

10) PAUSE

11) PAUSE n

12) STOP

13) STOP n

14) ENTRY s

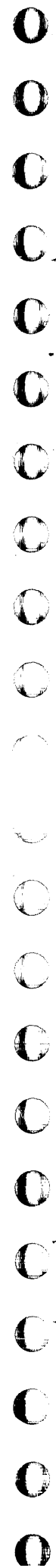
- 15) RETURN
- 16) CALL s
- 17) CALL s (e_1, e_2, \dots, e_n)
- 18) FORMAT ()
- 19) READ (u, f) 1
- 20) WRITE (u, f) 1
- 21) READ (u)1
- 22) WRITE (u) 1
- 23) REWIND u
- 24) BACKSPACE u
- 25) ENDFILE u

Class 5 statements that are not USA standard FORTRAN include the following:

- 1) READ f 1
- 2) PRINT f, 1
- 3) BUFFER IN (u, p) (a, b)
- 4) BUFFER OUT (u, p)(a, b)
- 5) ENCODE (c, n, w)1
- 6) DECODE (c, n, w)1

CLASS 6 STATEMENTS

- 1) END
- 2) END s



CHARACTER CODES

E

<u>Character</u>	<u>Keypunch</u>	<u>ASCII</u>
A	12-1	41
B	12-2	42
C	12-3	43
D	12-4	44
E	12-5	45
F	12-6	46
G	12-7	47
H	12-8	48
I	12-9	49
J	11-1	4A
K	11-2	4B
L	11-3	4C
M	11-4	4D
N	11-5	4E
O	11-6	4F
P	11-7	50
Q	11-8	51
R	11-9	52
S	0-2	53
T	0-3	54
U	0-4	55
V	0-5	56
W	0-6	57

<u>Character</u>	<u>Keypunch</u>	<u>ASCII</u>
X	0-7	58
Y	0-8	59
Z	0-9	5A
0	0	30
1	1	31
2	2	32
3	3	33
4	4	34
5	5	35
6	6	36
7	7	37
8	8	38
9	9	39
Blank	None	20
=	3-8	3D
+	12	2B
-	11	2D
*	11-4-8	2A
/	0-1	2F
(0-4-8	28
)	12-4-8	29
.	12-3-8	2E
,	0-3-8	2C

FORTRAN INTERFACE ROUTINES

F

The FORTRAN interface library subroutines are designed to allow the FORTRAN user to directly call routines within MPX from FORTRAN. Thus, the need for assembly level code to perform functions previously not available under MP-60 FORTRAN has been eliminated. These subroutines will pick up the parameters passed by the FORTRAN call, set up the necessary registers, and perform either a monitor call or jump and reset index to the appropriate MPX routine. Upon completion of the MPX routine, control will return to the interface subroutine which will return control to the calling FORTRAN user.

The FORTRAN interface library subroutines have been divided into four groups. The groupings are as follows:

- Group 1 - Blocker/deblocker interface subroutines
- Group 2 - Console display, status, logical unit to hardware type correlation interface subroutines
- Group 3 - OCARM (FILE MANAGER) and miscellaneous I/O function interface subroutines
- Group 4 - Miscellaneous subroutines.

Table F-1 contains the FORTRAN interface library subroutine names, the MPX Operating System routines associated with each, and a brief description of the function(s) performed by the MPX routine. For a detailed description of the MPX Operating System routines referenced in Table F-1, refer to the MP-60 Computer System MPX/RT Reference Manual, or the MP-60 Computer System MPX/OS Reference Manual.

GROUP 1 FTNINT1

FTNPACK

This subroutine is called as follows:

```
CALL FTNPACK (P1, P2, P3, P4, P5)
```

TABLE F-1. FORTRAN INTERFACE LIBRARY SUBROUTINES

Grp	Subroutine	MPX Routine	Function Performed Within MPX Routine
1	FTNPACK	PACK	Transfer record to PACKD buffer area
1	FTNPACKC	PACKC	Remove LU from blocker/deblocker tables
1	FTNPACKD	PACKD	Establishes blocking area (buffer)
1	FTNPACKO	PACKO	Output partially filled buffer
1	FTNPICK	PICK	Transfer record to user's record area
1	FTNPICKC	PICKC	Remove LU from blocker/deblocker tables
1	FTNPICKD	PICKD	Establish the deblocking area (buffer)
1	FTNPICKI	PICKI	Skip record(s)
2	FTNCTOC	CTOC	Send command message to operator via CRT
2	FTNCTOI	CTOI	Send information message to operator via CRT
2	FTNULOC	ULOC	Locate to specified block
2	FTNUST	UST	Status logical unit
2	FTNUTYP	UTYP	Determine hardware type assigned to LU
*2	FTNXSTAT	XSTAT	Status logical unit (expanded)
*2	FTNCLEAR	CLRIO	Cancel the last command to specific device
*2	FTNRDSTA	RDSTAT	Obtain RTC status for a particular device
*2	FTNFVFC	FVFC	Vertical format control for printer
*2	FTNFMODE	FMODE	Select or suppress echo mode to plasma
*2	FTNFHS	FSS	Select high speed mode on printer/plotter
3	FTNALLOC	ALLOCATE	Create file label in system label directory
3	FTNCLOSE	CLOSE	Remove reference to a file from the system tables

*This routine not available to MPX/OS users.

TABLE F-1. FORTRAN INTERFACE LIBRARY SUBROUTINES (Cont.)

Grp	Subroutine	MPX Routine	Function Performed Within MPX Routine
3	FTNERASE	ERASE	Erase specific area on logical unit
3	FTNMODFY	MODIFY	Change the label of an existing closed file
3	FTNOPEN	OPEN	Prepare existing file for data transmission
3	FTNREAD	READLU	Data transfer from logical unit to user designated buffer area
3	FTNRELES	RELEASE	Release some or all of the space allocated to a file
*3	FTNSELDN	SELDEN	Select recording density of LU
*3	FTNSELTR	SELTRK	Select specified track of logical unit
3	FTNSEOF	SEOF	Search for end-of-file (forward, backward)
*3	FTNSFNCT	SFNCT	Issue special function command to LU
3	FTNUNLD	UNLD	Unload logical unit
3	FTNWRITE	WRITLU	Data transfer to logical unit from user designated buffer area
4	FTNPARM		Transfer specified area of PARM to user designated area.

*This routine not available to MPX/OS users.

where P1 = Mode of record (0=ASCII, 1 = binary)

P2 = Logical unit number

P3 = First byte of record to be transferred

P4 = Number of bytes to be transferred

P5 = Location (2 words) where status will be returned.

This subroutine will load the appropriate parameters, P1 through P4, into registers RB, RC and RD and jump to MPX routine pack. Upon return of control, this subroutine will retrieve status from PARM and store it in the area specified by parameter P5 before returning controller to the calling program.

FTNPACKC

This subroutine is called as follows:

CALL FTNPACKC (P1,P2)

where P1 = Logical unit number

P2 = Location (2words) where status will be returned.

This subroutine will load the parameter P1 into register RB and jump to MPX routine PACKC. Upon return, this subroutine will retrieve status from PARM and store it in the area specified by parameter P2 before returning to the caller.

FTNPACKD

This subroutine is called as follows:

CALL FTNPACKD (P1,P2,P3,P4,P5,P6)

where P1 = Type of buffering (0=double, 1=single)

P2 = Logical unit number

P3 = First word of record to be transferred

P4 = Number of words to be transferred

P5 = Block number of first write (mass storage files only):

Less than 0 = file is positioned to highest block + 1

Equal to 0 = file is not positioned

Greater than 0 = file is positioned to specified block

P6 = Location (2 words) where status will be returned.

This subroutine will load the appropriate parameters, P1 through P5, into registers RB, RC, RD and RE and jump to MPX routine PACKD. Upon return of control, this subroutine will retrieve status from PARM and store it in the area specified by P6 before returning to caller.

FTNPACKO

This subroutine is called as follows:

CALL FTNPACKO (P1, P2, P3)

where P1 = Logical unit number

P2 = Block number of first write (mass storage files only):

Less than 0 = output to highest block written +1

Equal to 0 = output to next sequential block

Greater than 0 = output to specified block

P3 = Location (2 words) where status will be returned.

This subroutine will load parameters P1 and P2 into registers RB and RE, respectively, and jump to MPX routine PACKO. Upon return of control, this subroutine will retrieve status from PARM and store it in the area specified by P3 before returning to caller.

FTNPICK

This subroutine is called as follows:

CALL FTNPICK (P1, P2, P3, P4)

where P1 = Logical unit number

P2 = First byte address of the record to be transferred

P3 = Length of record in bytes

P4 = Location (2 words) where status will be returned.

This subroutine will load the appropriate parameters, P1 through P3, into registers RB, RC and RD and jump to MPX routine PICK. Upon return of control, this subroutine will retrieve status from PARM and store it in the area specified by P4 before returning to the caller.

FTNPICKC

This subroutine is called as follows:

CALL FTNPICKC (P1,P2)

where P1 = Logical unit number

P2 = Location (2 words) where status will be returned.

This subroutine will load parameter P1 into register RB and jump to MPX routine PICKC. Upon return of control, this subroutine will retrieve status from PARM and store it in area specified by P2 before returning control to caller.

FTNPICKD

This subroutine is called as follows:

CALL FTNPICKD (P1,P2,P3,P4,P5,P6)

where P1 = Type of buffering (0=double, 1=single)

P2 = Logical unit number

P3 = First word of record to be transferred

P4 = Number of words to be transferred

P5 = Block number of first read (mass storage files only):

Less than or equal to 0 = file is not positioned

Greater than 0 = file is positioned to specific block

P6 = Location (2 words) where status will be returned.

This subroutine will load the appropriate parameters, P1 through P5, into registers RB, RC, RD and RE and jump to MPX routine PICKD. Upon return of control, this subroutine will retrieve status from PARM and store it in the area specified by P6 before returning to caller.

FTNPICKI

This subroutine is called as follows:

CALL FTNPICKI (P1,P2,P3)

where P1 = Logical unit number

P2 = Block number of block to be input (mass storage only):

Less than or equal to 0 = input next sequential block

Greater than 0 = input specified block

P3 = Location (2 words) where status will be returned.

This subroutine will load parameters P1 and P2 into registers RB and RE, respectively, and jump to MPX routine PICKI. Upon return of control, this subroutine will retrieve status from PARM and store it in area specified by P3 before returning to caller.

GROUP 2 FTNINT2

FTNCTOC

This subroutine is called as follows:

CALL FTNCTOC (P1,P2)

where P1 = The first byte of data to be output to CRT

P2 = The location where accept or reject status is to be returned.

NOTE

Data pointed to by P1 must be characters ending with hexadecimal 03. On accept location, P2 will contain hexadecimal 00000041. On reject location, P2 will contain hexadecimal 00000052.

This subroutine will set value of P1 into register, perform monitor call to MPX routine CTOC; upon return, this subroutine will load appropriate accept or reject code into P2 and return control to the caller.

FTNCTOI

This subroutine is called as follows:

CALL FTNCTOI (P1)

where P1 = The first byte of data to be output to CRT.

NOTE

Data pointed to by P1 must be characters ending with hexadecimal 03.

This subroutine will set value of P1 into register, perform monitor call to MPX routine CTOI; upon return, this subroutine will return control to the caller.

FTNULOC

This subroutine is called as follows:

CALL FTNULOC (P1, P2)

where P1 = Logical unit number

P2 = Block number to locate to.

NOTE

If P2 equals value of -1, the file is positioned to the last block written + 1.

This subroutine will set the values of P1 and P2 into registers and perform a monitor call to MPX routine ULOC. Upon return of control, this subroutine will return to caller.

FTNUST

This subroutine is called as follows:

CALL FTNUST (P1, P2)

where P1 = Logical unit number to status

P2 = Location where status word is to be stored.

This subroutine will load the addresses of P1 and P2 into registers and perform a monitor call to MPX routine UST. Upon return of control, this subroutine will return to caller.

FTNUTYP

This subroutine is called as follows:

CALL FTNUTYP (P1, P2)

where P1 = Logical unit number to be tested for

P2 = Location where hardware type code is to be returned.

NOTE

Refer to MPX/RT Reference Manual Section 3 for hardware type codes or MPX/OS Reference Manual Section 4.

This subroutine will set the address of P1 into a register and perform a monitor call to MPX routine UTYP. Upon return this subroutine will retrieve the hardware type code from PARM and store into P2 before returning to caller.

FTNXSTAT

This subroutine is called as follows:

CALL FTNXSTAT (P1, P2)

where P1 = Logical unit number of device to be tested

P2 = Location where 2 words of returned status is to be written.

This subroutine will set the values of P1 and P2 into registers and perform a monitor call to MPX routine XSTAT. Upon return of control, this subroutine will return to caller.

FTNCLEAR

This subroutine is called as follows:

CALL FTNCLEAR (P1)

where P1 = Logical unit number of device to be cleared.

This subroutine will set value of P1 into a register and perform a monitor call to MPX routine CLRIO. Upon return of control this subroutine will return to caller.

FTNRDSTA

This subroutine is called as follows:

CALL FTNRDSTA (P1, P2)

where P1 = Logical unit number of device for which status is desired

P2 = Location where 2 words of returned status is to be written.

This subroutine will set the value of P1 into a register and perform a monitor call to MPX routine RDSTAT. Upon return this subroutine will retrieve status from PARM + 1 and PARM + 2 and store this status in address specified by P2 before returning control to caller.

FTNFVFC

This subroutine is called as follows:

CALL FTNFVFC (P1, P2)

where P1 = Logical unit number of device

P2 = Format control command, defined as follows:

Format Command	Description
0-7	Space to channel 0-7 of tape, respectively
16-31	Space 0-15 lines, respectively

This subroutine will set the values of P1 and P2 into registers and perform a monitor call to MPX routine FVFC. Upon return of control this subroutine will return to caller.

FTNFMODE

This subroutine is called as follows:

CALL FTNFMODE (P1, P2)

where P1 = Logical unit number of device

P2 = Enable/suppress echo mode designator (0 = suppress, 1 = enable).

This subroutine will set the values of P1 and P2 into registers and perform a monitor call to MPX routine FMODE. Upon return of control this subroutine will return to caller.

FTNFHS

This subroutine is called as follows:

CALL FTNFHS (P1, P2)

where P1 = Logical unit number of device

P2 = Turn ON/OFF high speed mode (0 = OFF, 1 = ON).

This subroutine will set values of P1 and P2 into registers and perform a monitor call to MPX routine FSS.

GROUP 3 FTNINT3

FTNALLOC

This subroutine is called as follows:

CALL FTNALLOC (P1, P2)

where P1 = The first word of the parameter list

P2 = The location where OCARM/File Manager status is to be returned.

This subroutine will set value of P1 into register, perform monitor call to MPX routine ALLOCATE; upon return, this subroutine will retrieve OCARM/File Manager status from PARM region, store OCARM/File Manager status in address specified by P2, and return control to the caller.

FTNCLOSE

This subroutine is called as follows:

CALL FTNCLOSE (P1, P2)

where P1 = The logical unit number of the file to be closed

P2 = The location where OCARM/File Manager status is to be returned.

This subroutine will set value of P1 into register, perform monitor call to MPX routine CLOSE; upon return, this subroutine will retrieve OCARM/File Manager status from PARM region, store OCARM/File Manager status in address specified by P2, and return control to the caller.

FTNERASE

This subroutine is called as follows:

```
CALL FTNERASE (P1)
```

where P1 = Logical unit number of device to erase on.

This subroutine will set value of P1 into register, perform monitor call to MPX routine ERASE and upon regaining control will return to caller.

FTNMODFY

This subroutine is called as follows:

```
CALL FTNMODFY (P1, P2)
```

where P1 = First word of parameter list

P2 = Location where OCARM/File Manager status will be returned.

This subroutine will set the value of P1 into register, perform monitor call to MPX routine MODIFY and upon return will load the OCARM/File Manager status from PARM and store into P2 before returning control to caller.

FTNOPEN

This subroutine is called as follows:

```
CALL FTNOPEN (P1, P2, P3)
```

where P1 = First word of parameter list

P2 = Logical unit number to be assigned to file

P3 = Location where OCARM/File Manager error status will be returned.

This subroutine will load P1 and P2 parameters into registers and perform a monitor call to MPX routine OPEN. Upon return of control, this subroutine will retrieve OCARM

status from PARM and save this status in the area specified by P3. Control will then be returned to the caller.

FTNREAD

This subroutine is called as follows:

```
CALL FTNREAD (P1, P2, P3, P4)
```

where P1 = First word/byte of buffer area (see P3 for word/byte explanation)

P2 = Number of words/bytes in buffer (see P3 for word/byte explanation)

P3 = Mode control (2 digits in hexadecimal format):

00 = ASCII record, word format

10 = ASCII record, byte format

20 = BINARY record, word format

P4 = Logical unit number.

NOTE

Caution must be exercised when calling this subroutine that the FORTRAN user does not intermix with FORTRAN I/O statements without checking for completion status.

This subroutine will load the address of P1 and the values of P2 through P4 into registers and perform a monitor call to MPX routine READLU. Upon return of control, this subroutine will return to caller.

FTNRELES

This subroutine is called as follows:

```
CALL FTNRELES (P1, P2)
```

where P1 = First word of parameter list

P2 = Location where OCARM/File Manager error status will be returned.

This subroutine will load P1 parameter into a register and perform a monitor call to MPX routine RELEASE. Upon return of control, this subroutine will retrieve OCARM/ File Manager status from PARM and save this status in the area specified by P3. Control will then be returned to the caller.

FTNSELDN

This subroutine is called as follows:

CALL FTNSELDN (P1, P2)

where P1 = Logical unit number of device density is to be selected on

P2 = Density control code: 0 = low density (556 BPI NRZI - 667)

1 = high density (800 BPI NRZI - 667/669)

2 = hyperdensity (1600 BPI PE -669).

This subroutine will set values of P1 and P2 into registers and perform a monitor call to MPX routine SELDEN. Upon return, this subroutine will return control to the caller.

FTNSELTR

This subroutine is called as follows:

CALL FTNSELTR (P1, P2)

where P1 = Logical unit number of device on which track select is to be done

P2 = Track control code: 0 = SELECT TRACK 0

1 = SELECT TRACK 1

2 = SELECT TRACK 2

3 = SELECT TRACK 3.

This subroutine will load P1 and P2 values into registers and perform monitor call to MPX routine SELTRK. Upon return of control, this subroutine will return to the caller.

FTNSEOF

This subroutine is called as follows:

CALL FTNSEOF (P1, P2)

where P1 = Logical unit number of device search is to be performed on

P2 = Mode control code: 0 = SEARCH FORWARD

 1 = SEARCH BACKWARD.

This subroutine will load P1 and P2 parameters into registers and perform a monitor call to MPX routine SEOF. Upon return of control, this subroutine will return to caller.

FTNSFNCT

This subroutine is called as follows:

CALL FTNSFNCT (P1, P2, P3, P4)

where parameters P1 through P4 will be specified by the system into which the MPX routine SFNCT is used.

This subroutine will set the values of P1 through P4 into registers RB through RE, respectively, and perform a monitor call to MPX routine SFNCT. Upon return of control, this subroutine will return to caller.

FTNUNLD

This subroutine is called as follows:

CALL FTNUNLD (P1)

where P1 = Logical unit number to be unloaded.

This subroutine will load the value of P1 into a register and perform a monitor call to MPX routine UNLD. Upon return of control, this subroutine will return to caller.

FTNWRITE

This subroutine is called as follows:

```
CALL FTNWRITE (P1, P2, P3, P4)
```

where P1 = First word/byte of data to be written (see P3 for word/byte)

P2 = Number of words/bytes to be written (see P3 for words/bytes)

P3 = Mode control (2 digits in hexadecimal format):

00 = ASCII record, word format

10 = ASCII record, byte format

20 = BINARY record, word format

P4 = Logical unit number.

NOTE

Caution must be exercised when calling this subroutine that the FORTRAN user does not intermix with FORTRAN I/O statements without checking for completion status.

This subroutine will load the address of P1 and the values of P2 through P4 into registers and perform a monitor call to MPX routine WRITLU. Upon return of control, this subroutine will return to caller.

GROUP 4 FTNINT4

FTNPARM

This subroutine is called as follows:

```
CALL FTNPARM (P1, P2, P3)
```

where P1 = Address of area where data from PARM is to be saved

P2 = This value plus PARM will be first word retrieved from PARM

P3 = Number of words to be retrieved from PARM starting at word specified by P2.

This subroutine will load the appropriate parameters, P1 through P3 into registers. The subroutine will retrieve the locations within PARM specified by P2 and P3 and store the contents of these locations in the area specified by P1. Upon completion of this function, control will be returned to the calling program.

MPX/OS SPECIAL FEATURES

G

MPX/OS INTERFACE ROUTINES

The interface routines pertaining to FORTRAN are called the FORTRAN Interface Utility (ESRUTIL). This module consists of five individual routines which allow the caller (usually FORTRAN) to perform standard system executive service request (ESR) functions.

A brief description of each routine follows.

Routine: IESR

Description: Generate monitor call to specified ESR.

Calling Sequence:

CALL IESR (TYPE, P1, P2, P3, P4) or
K = IESR (TYPE, P1, P2, P3, P4)

Input Parameter:

TYPE = ESR to perform (must be declared in an
EXTERNAL statement)

P1-P4 = Parameters one through four, dependent on ESR.

Output Parameter:

K = Contents of PARM+0, dependent on ESR.

Routine: IAPAW

Description: Specifies relative location in PARM (word access).

Calling Sequence:

K = IAPAW (OFFSET); access in word mode

Input Parameter:

OFFSET = Relative location in PARM (i.e., PARM + OFFSET).
OFFSET starts from 1, as in FORTRAN indexing (i.e., OFFSET = 1
for PARM+0).

Output Parameter:

K = Contents of word from PARM

Routine: IAPAH

Description: Specifies relative location in PARM (half-word access).

Calling Sequence:

K = IAPAH (OFFSET); access in half-word mode

Input Parameter:

OFFSET = Relative location in PARM (i.e., PARM + OFFSET).
OFFSET starts from 1, as in FORTRAN indexing (i.e., OFFSET = 1
for PARM+0).

Output Parameter:

K = Contents of half-word from PARM

Routine: IAPAC

Description: Specifies relative location in PARM (character access)

Calling Sequence:

K = IAPAC (OFFSET); access in character mode

Input Parameter:

OFFSET = Relative location in PARM (i.e., PARM + OFFSET).
OFFSET starts from 1, as in FORTRAN indexing (i.e., OFFSET = 1
for PARM+0).

Output Parameter:

K = Contents of character from PARM

Routine: LOCF

Description: Provides the address of a variable instead of the data itself. LOCF actually modifies the caller's code to load the address, hence LOCF is entered one time per occurrence.

Calling Sequence:

K = LOCF (VAR)

Input Parameter:

VAR = Variable for which address is desired.

Output Parameter:

K = Address of VAR

To illustrate the use of the above routines, an example is provided.

```
PROGRAM EXAMPLE
.
.
.
EXTERNAL STATGC
.
.
.
DIMENSION BLOCK(2)
SCRATCH COMMON/GLOBAL/A(4096)
.
.
.
DATA BLOCK/4HBLOC,4HKONE/
.
.
.
C
C STATUS GLOBAL COMMON BLOCKONE
C
ISTATUS = IESR (STATGC, BLOCK(1), BLOCK(2), LOCF(A))
ISIZE = IAPAW(2)
.
.
```

Routine: IBDB

Description: Generate JSX call to specified blocker/deblocker routine.

Calling Sequence:

CALL IBDB (TYPE, P1, P2, P3, P4)

K = IBDB (TYPE, P1, P2, P3, P4)

Input Parameters:

TYPE = blocker/deblocker routine to execute (must be declared in an EXTERNAL statement)

P1-P4 = Parameters one thru four, dependent on blocker/deblocker routine.

Output Parameter:

K = Contents of Parm+0, from blocker/deblocker call.

GLOBAL COMMON DECLARATIONS

A special reserved scratch common block name (GLOBAL) is to be used by the programmer to signal the beginning of global common block declarations. Scratch common blocks encountered by the loader before the occurrence of the common block name GLOBAL constitute local scratch common. All subsequent scratch common blocks including GLOBAL will start on a page boundary and comprise global common.

An example of FORTRAN coding is shown below.

```
SCRATCH COMMON/A(100), B(100)
SCRATCH COMMON/BLOCK 1/C(4096)
SCRATCH COMMON/GLOBAL/D(2048), E(2048)
SCRATCH COMMON/BLOCK 2/F(8192)
```

The example generates two pages of local scratch common and three pages of global scratch common. Global common would start at logical address 2000. Global common blocks could be mapped into addresses 2000 through 4FFF₁₆ and referenced by the arrays D, E, and F.

The following ESRs support the GLOBAL common feature.

```
GETGC
STATGC
RETGC
```

Refer to the MPX/OS Reference Manual, Section 4.1, Executive Service Requests, for the usage of the above ESRs.

INDEX

Item	Page
Aw input	8-14
Aw output	8-13
Actual arguments	7-2, 7-3
Actual parameters	
See actual arguments	
Alphanumeric identifiers	2-3
.AND.	3-7
Arguments	7-2, 7-3
Arithmetic replacement statement	
Form of	4-2
Mixed mode	4-1 through 4-4
Arithmetic elements	3-1
Arithmetic evaluation	3-3, 3-7
Arithmetic expressions	3-1, 3-2
Arithmetic IF statement	6-3
Arithmetic operators	3-1, 3-7
Array	
Declaration	2-4
Dimensioning	2-4
Element	2-4, 2-5, 2-6
Location of elements	2-4, 2-5
Number of subscripts	2-4
Number of words per element	2-5
Structure	2-4, 2-5
Subscripts	2-4, 2-5, 2-6
Transmission of	9-5
Type	2-5

INDEX (CONT.)

Item	Page
ASCII codes	Appendix E
ASCII constants	2-2
ASCII conversions	8-13, 8-14, 8-18, 8-19
ASSIGN statement	6-2
Assigned GO TO statement	6-2
Assignment statements	
See replacement statements	
BACKSPACE statement	9-12
Blanks	1-2
Buffer statements	9-10, 9-11, 9-12
BUFFER IN	9-10, 9-11
BUFFER OUT	9-11, 9-12
Status checking	9-17/9-18
CALL statement	7-2, 7-3, 7-4
Calling program	7-1
Calling sequence	10-4
Card format	1-1, 1-2
Carriage control	8-24
Character codes	Appendix E
Character set	1-2
Codes	
ASCII	E-1
Character	E-1
Keypunch	E-1
Coding format	1-1, 1-2
Comment card format	1-1
COMMON statement	5-3, 5-4, 5-5
Block identifier	5-4, 5-5
Data common	5-4, 5-5
Form of	5-4, 5-5
Order of appearance	5-5
Scratch common	5-4, 5-5

INDEX (CONT.)

Item	Page
COMPASS calling sequence	10-4
COMPASS subprograms	10-4
Compile time options	1-1
Computed GO TO statement	6-1, 6-2
Constants	2-1, 2-2, 2-3
ASCII	2-2
Double precision	2-2
Hexadecimal	2-1
Integer	2-1
Logical	2-3
Real	2-2
Continuation cards	1-2
CONTINUE statement	6-7
Control cards	10-1, 10-2, 10-3
FIN card	10-1, 10-2, 10-3
FINIS card	10-3
Control statements	Chapter 6
CONTINUE statement	6-7
DO statement	6-4
GO TO statements	6-1, 6-2
IF statements	6-3
PAUSE statement	6-7
STOP statement	6-8
Conversion specifications	
Format	8-3 through 8-20
Dw.d input	8-11
Dw.d output	8-10
Data common	5-4
DATA statement	5-7, 5-8
Data transmission	8-1

INDEX (CONT.)

Item	Page
Data types	2-2
Explicit declaration	2-2, 5-1, 5-2
Implicit declaration	2-2
Declarative statements	Chapter 5
COMMON statement	5-4, 5-5
DATA statement	5-7/5-8
EQUIVALENCE statement	5-5, 5-6
EXTERNAL statement	7-7
Type statements	5-1, 5-2
Order of appearance	5-1
DECODE statement	9-14
Delimiters	1-2
Diagnostics	Appendix C
DIMENSION statement	5-2, 5-3
Dimension of arrays	2-4, 2-5
DO-implying segments	9-1, 9-2, 9-3
DO loop execution	6-4, 6-5
DO loop transfer	6-6, 6-7
DO nests	6-5, 6-6
DO statement	6-4 through 6-7
Double-precision constants	2-2
Dummy arguments	7-4 through 7-6
Ew.d input	8-4, 8-5, 8-6
Ew.d output	8-3, 8-4
Ew.d scaling	8-16
Editing specifications	8-16 through 8-21
ENCODE statement	9-13, 9-14, 9-15
END statement	7-1, 7-9/7-10
ENDFILE statement	9-13
ENTRY statement	7-8

INDEX (CONT.)

Item	Page
.EQ.	3-2
EQUIVALENCE statement	5-5, 5-6
Arrays in	5-5, 5-6
Form of	5-5
Interaction with COMMON	5-5, 5-6
Order of appearance	5-6
Error messages	Appendix C
Evaluation of expressions	
Arithmetic	3-7
Logical	3-4
Masking	3-6
Mixed mode	3-8
Relational	3-3
Executable program	7-1
Explicit data type association	5-1, 5-2
Exponentiation	3-1, 3-7
Expressions	Chapter 3
Arithmetic	3-1, 3-2
Evaluation of	3-7, 3-8
Logical	3-4, 3-5
Masking	3-5, 3-6
Mixed mode	3-8
Relational	3-2 through 3-5
Extensions to ANSI FORTRAN	D-3/D-4
EXTERNAL statement	7-7
Fw.d input	8-9, 8-10
Fw.d output	8-7, 8-8, 8-9
Fw.d scaling	8-15
.FALSE.	2-3
Field descriptors	8-1, 8-2

INDEX (CONT.)

Item	Page
Field separators	8-3
File control	9-12, 9-13
BACKSPACE statement	9-12
ENDFILE statement	9-13
REWIND statement	9-12
FINIS card	10-3
Formal parameters	7-2, 7-3
See also dummy arguments	
Format arrays	8-22, 8-23
Format control	8-23, 8-24
Format specifications	Chapter 8
ASCII conversion	8-13, 8-14, 8-18, 8-19
Blank field descriptor	8-17
Carriage control	8-24
Double-precision conversion	8-10, 8-11
Field descriptors	8-1, 8-2
Hexadecimal conversion	8-12, 8-13
Integer conversion	8-11, 8-12
Logical conversion	8-13
New record specification	8-19, 8-20, 8-21
Real conversion	8-3 through 8-10
Repeated specifications	8-21, 8-22
Scale factor	8-15, 8-16
FORMAT statement	8-1
FORTRAN interface routines	Appendix F
FORTRAN standard output	Appendix B
FORTRAN statements	Appendix D
FTN (MP-60 FORTRAN compiler)	10-1
FTN card	10- through 10-5
FUNCTION statement	7-4

INDEX (CONT.)

Item	Page
Function subprogram	7-4, 7-5, 7-6
Defining	7-4
Referencing	7-5
Restrictions on construction	7-5
.GE.	3-2
GO TO statements	6-1, 6-2
Assigned GO TO statement	6-2
Computed GO TO statement	6-1, 6-2
GO TO assignment statement	6-2
Unconditional GO TO statement	6-1
.GT.	3-2
wH input	8-18, 8-19
wH output	8-18
Hexadecimal constants	2-1
Hexadecimal input (\$w)	8-13
Hexadecimal output (\$w)	8-12
Iw input	8-12
Iw output	8-11
IF statements	6-3
Arithmetic IF statement	6-3
Logical IF statement	6-3
IFUNIT (status checking routine)	9-17/9-18
Implicit data type association	2-3
Input statements	9-8, 9-9
READ (i) list	9-8
READ (i, n) list	9-9
READ n, list	9-8

INDEX (CONT.)

Item	Page
I/O lists	9-1 through 9-4
DO-implying segments	9-1, 9-2, 9-3
Transmission of arrays	9-5
I/O format reference	9-1
I/O statements	Chapter 9
I/O units	9-5
Integer constants	2-1
Integer data	2-1
Integer division	3-7
Internal transmission	9-13 through 9-16
DECODE statement	9-15, 9-16
ENCODE statement	9-13, 9-14, 9-15
Lw input	8-13
Lw output	8-13
.LE.	3-2
Library routines	Appendix A
Logical replacement statement	4-2
Logical constants	2-3
Logical data	2-3
Logical expressions	3-1, 3-4, 3-5
Elements	3-4
Hierarchy of evaluation	3-4
Operators	3-4
Value	3-4
Logical IF statement	6-3
Logical unit	10-1
.LT.	3-2

INDEX (CONT.)

Item	Page
Main program	7-1
MPX operating system	1-1, 10-1, 10-5, 10-6, 10-7/10-8
Control cards	10-1
Deck structure examples	10-5, 10-6, 10-7/10-8
Special features	G-1
Mixed-mode arithmetic expressions	3-8
Mixed-mode replacement statements	4-2, 4-3, 4-4
.NE.	3-2
Nested DO loops	6-5, 6-6
New record specification	8-19, 8-20, 8-21
.NOT.	3-2
Null labels	1-2
Optional outputs	1-1
.OR.	3-2
Output statements	9-6, 9-7
PRINT n, list	9-6
WRITE (i) list	9-7
WRITE (i,n) list	9-7
Parameters	
See arguments	
Partial records	9-5
PAUSE statement	6-7
PRINT n, list	9-6
Printer carriage control	8-24
Program communication	7-1, 7-2
Program operation	Chapter 10
PROGRAM statement	7-1

INDEX (CONT.)

Item	Page
Rw input	8-14
Rw output	8-14
READ (l) list	9-8, 9-9
READ (i, n) list	9-9
READ n, list	9-8
Real constants	2-2
Real data	2-2
Relational expressions	3-2 through 3-5
Elements	3-2
Evaluation	3-4
Operators	3-2
Rules	3-3
Value of	3-3
Repeated specifications	8-21, 8-22
Replacement statements	Chapter 4
Arithmetic replacement	4-2
Logical replacement	4-2
Mixed-mode replacement	4-2, 4-3, 4-4
Multiple replacement	4-3
RETURN statement	7-9/7-10
REWIND statement	9-12
Sample deck structure	10-5, 10-6, 10-7/10-8
Scale factor	8-15
Scaling restrictions	8-16
SCRATCH COMMON	5-4
Simple variables	2-3
Source format	1-1, 1-2
Source language	1-2
Source program	1-1
Space (wX)	8-17

INDEX (CONT.)

Item	Page
Specification statements	
See declarative statements	
Statement card format	1-1, 1-2
Statement label	1-2
Statement numbers	1-2
Status checking routine	9-17/9-18
STOP statement	6-8
Storage, reserved for array	2-5
Storage unit size	2-3
Subprograms	2-1, 7-1
Alternate entry	7-8
Arguments	7-2, 7-3
Calling	7-2
Functions	7-4 through 7-7
Returning	7-9/7-10
Subroutine	7-1 through 7-4
Terminating	7-9/7-10
SUBROUTINE statement	7-2
Subroutine subprograms	7-1 through 7-4
CALL statement	7-2, 7-3, 7-4
Defining	7-1
Referencing	7-2
Restrictions on construction	7-2
SUBROUTINE statement	7-2
Subscript expressions	2-3, 2-4
Subscript forms	2-4
Subscript variables	2-3, 2-4

INDEX (CONT.)

Item	Page
Task name card	10-1
.TRUE.	2-3
Type declaration	2-1, 2-3, 5-1
Explicit declaration	5-1
Implicit declaration	2-1, 2-3
Type of function	7-4, 7-5
Type statements	5-1, 5-2
Unconditional GO TO statement	6-1
Variable format	8-22, 8-23
Variable dimensions	5-3
Variables	2-3
WRITE (i) list	9-7
WRITE (i, n) list	9-7
.XOR.	3-7

COMMENT SHEET

MANUAL TITLE MP-60 Computer System FORTRAN Reference Manual

PUBLICATION NO. 14061100 REVISION E

FROM: NAME: _____
BUSINESS ADDRESS: _____

COMMENTS:

This form is not intended to be used as an order blank. Your evaluation of this manual will be welcomed by Control Data Corporation. Any errors, suggested additions or deletions, or general comments may be made below. Please include page number references and fill in publication revision level as shown by the last entry on the Record of Revision page at the front of the manual. Customer engineers are urged to use the TAR.

CUT ALONG LINE

PRINTED IN U.S.A.

A43419 REV. 11/69

NO POSTAGE STAMP NECESSARY IF MAILED IN U. S. A.

FOLD ON DOTTED LINES AND STAPLE

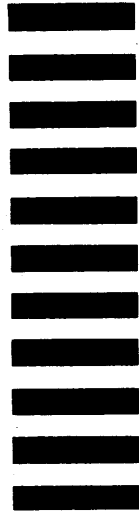
FOLD

FOLD

FIRST CLASS
PERMIT NO. 8241
MINNEAPOLIS, MINN.

BUSINESS REPLY MAIL
NO POSTAGE STAMP NECESSARY IF MAILED IN U.S.A.

POSTAGE WILL BE PAID BY
CONTROL DATA CORPORATION
3101 East 80th Street, HQG 346B
Box 609
Minneapolis, Minnesota 55440



CUT ALONG LINE

FOLD

FOLD



14061100



**MP-60
COMPUTER SYSTEM**

**FORTRAN
REFERENCE MANUAL**

LIST OF EFFECTIVE PAGES

New features, as well as changes, deletions, and additions to information in this manual, are indicated by bars in the margins or by a dot near the page number if the entire page is affected. A bar by the page number indicates pagination rather than content has changed.

Page	Rev.	Page	Rev.	Page	Rev.	Page	Rev.
Title Page	E	5-7/5-8	A	8-21	A	C-4	A
ii	F	6-1	A	8-22	A	C-5	A
iii	F	6-2	A	8-23	A	C-6	A
iv	E	6-3	A	8-24	A	C-7	A
v/vi	E	6-4	F	9-1	A	C-8	A
vii	E	6-5	A	9-2	A	C-9	A
viii	E	6-6	A	9-3	A	C-10	A
ix	E	6-7	B	9-4	A	C-11	A
x	E	6-8	D	9-5	A	C-12	A
xi	E	7-1	A	9-6	A	C-13	A
xii	E	7-2	D	9-7	A	C-14	A
1-1	A	7-3	A	9-8	A	C-15/C-16	A
1-2	A	7-4	A	9-9	A	D-1	A
2-1	A	7-5	A	9-10	A	D-2	A
2-2	A	7-6	A	9-11	A	D-3/D-4	A
2-3	A	7-7	A	9-12	A	E-1	A
2-4	C	7-8	A	9-13	A	E-2	A
2-5	C	7-9/7-10	D	9-14	A	F-1	C
2-6	A	8-1	A	9-15	A	F-2	C
3-1	A	8-2	A	9-16	A	F-3	C
3-2	A	8-3	A	9-17/9-18	A	F-4	C
3-3	A	8-4	A	10-1	A	F-5	C
3-4	A	8-5	A	10-2	A	F-6	C
3-5	A	8-6	A	10-3	F	F-7	C
3-6	A	8-7	A	10-4	A	F-8	C
3-7	A	8-8	A	10-5	A	F-9	C
3-8	A	8-9	A	10-6	B	F-10	C
4-1	A	8-10	A	10-7/10-8	A	F-11	C
4-2	A	8-11	A	A-1	B	F-12	C
4-3	A	8-12	A	A-2	B	F-13	C
4-4	A	8-13	A	A-3	B	F-14	C
5-1	A	8-14	A	A-4	B	F-15	C
5-2	A	8-15	A	B-1	A	F-16	C
5-3	A	8-16	A	B-2	A	F-17	C
5-4	A	8-17	A	C-1	A	F-18	C
5-5	D	8-18	A	C-2	A	G-1	D
5-6	A	8-19	A	C-3	A	G-2	D
		8-20	A			G-3	D

LIST OF EFFECTIVE PAGES

New features, as well as changes, deletions, and additions to information in this manual, are indicated by bars in the margins or by a dot near the page number if the entire page is affected. A bar by the page number indicates pagination rather than content has changed.

Page	Rev.	Page	Rev.	Page	Rev.	Page	Rev.
G-4	E						
I-1	A						
I-2	B						
I-3	A						
I-4	A						
I-5	A						
I-6	C						
I-7	A						
I-8	A						
I-9	D						
I-10	A						

IF STATEMENTS

Conditional transfer of control is provided with the IF statements.

ARITHMETIC IF

The form of the arithmetic IF statement is:

IF (e) k_1, k_2, k_3

e is an arithmetic expression of type integer, real or double precision, and k_i represents statement labels.

This statement causes expression e to be evaluated and control transferred according to that value.

e < 0 jump to k_1
e = 0 jump to k_2
e > 0 jump to k_3

For example:

IF (A*B-CSIN(X)) 10, 10, 20

IF (N) 5, 6, 7

IF (A/B*C) 10, 11, 12

LOGICAL IF

The form of the logical IF statement is:

IF (ℓ) S

ℓ is a logical expression and S is an executable statement (not a statement label). S must not be a DO or another IF. If e is true, S is executed. If e is false, S is treated as a CONTINUE statement.

For example:

IF (L) GO TO 3

IF (A.GT.16.0.OR.A.EQ.0.) A=B

DO STATEMENT

The DO statement makes it possible to repeat a group of statements immediately following the DO statement a number of times, changing the value of a simple integer variable for each repetition. The form of the DO statement is:

$$\text{DO } n \text{ } i = m_1, m_2, m_3$$

n is the label (number) of the statement ending the DO loop; i is a simple integer index variable. The m_i are indexing parameters; they must be unsigned nonzero integer constants or simple integer variables. i is initially set equal to m_1 , and after each execution of the DO loop, m_3 is added to i . (When omitted, m_3 assumes a value of 1.) When i becomes greater than m_2 , the DO loop is satisfied. Maximum value of the index constant is 65,535.

The DO statement, the statement labeled n , and any intermediate statements constitute a DO loop. Statement n may not be a GO TO, FORMAT, another DO statement, an arithmetic IF statement, RETURN, STOP, PAUSE, or a logical IF containing any one of these statements.

DO LOOP EXECUTION

Should m_1 exceed m_2 on the initial entry to the loop, the loop is executed once and control passes immediately to the statement following statement n . If it does not exceed m_2 , the loop is executed. The value of i is increased by m_3 and again compared with m_2 . The process continues until i exceeds m_2 . The DO loop is then satisfied, and control passes to the statement immediately following statement n .

If a transfer out of the DO loop occurs before the DO is satisfied, the value of i is preserved and may be used in subsequent statements.

Source language and assembly listings are written on the standard output unit, LU 62. The punchable output is written on the standard punch unit, LU 61, and automatically punched. Executable output is written on the standard load and go file, LU 57. Input source was read from the standard input unit, LU 63.

*FTN (I = 10, L, R, X = 11

Source input is from LU 10. Source output listing with cross-reference listing is on LU 62. Executable binary is written on LU 11.

CONTROL CARD NOTES

For a detailed description of all MPX control cards, refer to the MPX/RT Reference Manual (Section 2). Information relevant to FORTRAN is, however, contained in this section.

Core memory assigned to a job should be requested by the user via the MPX schedule statement (*SCHED). A minimum of 11 pages is needed to compile small FORTRAN programs. Larger programs require more memory, up to a maximum of 16 pages. When executing FORTRAN-compiled programs containing ASCII input or output statements, memory must be scheduled for I/O buffers (a 480-word buffer for each unique logical unit).

The FORTRAN compiler uses system scratch files 1 and 2 (LUs 59 and 60) as intermediate files. The user should reposition these files if they are to be used later in the job.

A FINIS card is used to notify the compiler that there are no more programs to be compiled. The word FINIS must begin in column 10.

1 10
FINIS

CALLING SEQUENCES

Programs written in MP-60 assembly language (COMPASS) may call or be called by programs compiled by MP-60 FORTRAN. Calling sequence conventions have been established for this purpose.

The calling sequence compiled for an external reference of the form
CALL NAME (p₁, p₂, ..., p_n) is:

RTJ	NAME
UJP	*+(n+1)
VFD	16/0, 16/p ₁
VFD	16/0, 16/p ₂
.	
.	
VFD	16/0, 16/p _n

Where:

NAME	Entry point of the program unit being referenced
p _i *	Address of the i th parameter

The main program, the set of FORTRAN statements bounded by a PROGRAM statement and an END statement, is entered initially by MPX. If the main program contains either ASCII I/O statements or STOP statements, a FORTRAN library routine with entry points Q8QENTRY and Q8QEXITS is provided to interface with MPX. The initialization performed by the main program includes clearing register X1, which is needed for double precision arithmetic operations and tests.

Subroutines, entered by the CALL statement or from COMPASS programs, use all registers. Functions save and restore all registers except RA-RF (registers 26 through 31) and return the function value in register RE (or RE-RF if the function is a type double precision). Both subroutines and function require register X to be zero upon entrance. Refer to Section 7 for further explanation of subprogram relationships.

*Note that if p_i is type character, address field would be 14/0, 18/p_i.

COMMENT SHEET

MANUAL TITLE MP-60 Computer System FORTRAN Reference Manual

PUBLICATION NO. 14061100 REVISION F

FROM: NAME: _____
BUSINESS ADDRESS: _____

COMMENTS:

This form is not intended to be used as an order blank. Your evaluation of this manual will be welcomed by Control Data Corporation. Any errors, suggested additions or deletions, or general comments may be made below. Please include page number references and fill in publication revision level as shown by the last entry on the Record of Revision page at the front of the manual. Customer engineers are urged to use the TAR.

CUT ALONG LINE

PRINTED IN U.S.A.

AA3419 REV. 11/69

NO POSTAGE STAMP NECESSARY IF MAILED IN U. S. A.

FOLD ON DOTTED LINES AND STAPLE

STAPLE

STAPLE

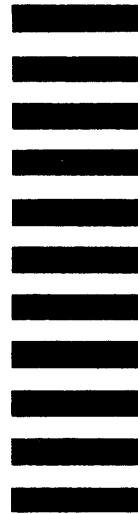
FOLD

FOLD

FIRST CLASS
PERMIT NO. 8241
MINNEAPOLIS, MINN.

BUSINESS REPLY MAIL
NO POSTAGE STAMP NECESSARY IF MAILED IN U.S.A.

POSTAGE WILL BE PAID BY
CONTROL DATA CORPORATION
3101 East 80th Street, HQG 346B
Box 609
Minneapolis, Minnesota 55440



CUT ALONG LINE

FOLD

FOLD



14061100



MP-60
COMPUTER SYSTEM

FORTRAN
REFERENCE MANUAL

LIST OF EFFECTIVE PAGES

New features, as well as changes, deletions, and additions to information in this manual, are indicated by bars in the margins or by a dot near the page number if the entire page is affected. A bar by the page number indicates pagination rather than content has changed.

Page	Rev.	Page	Rev.	Page	Rev.	Page	Rev.
Title Page	E	5-1	A	8-11	A	10-4a	G
ii	G	5-2	A	8-12	A	10-4b	G
iii	G	5-3	A	8-13	A	10-5	A
iv	G	5-4	A	8-14	A	10-6	B
v/vi	E	5-5	D	8-15	A	10-7/10-8	A
vii	E	5-6	A	8-16	A	A-1	G
viii	E	5-7/5-8	A	8-17	A	A-2	G
ix	E	6-1	A	8-18	A	A-3	G
x	E	6-2	A	8-19	A	A-4	G
xi	G	6-3	A	8-20	A	B-1	A
xii	E	6-4	F	8-21	A	B-2	A
1-1	A	6-5	A	8-22	A	C-1	A
1-2	A	6-6	A	8-23	A	C-2	A
2-1	A	6-7	B	8-24	A	C-3	A
2-2	A	6-8	D	9-1	A	C-4	A
2-3	A	7-1	A	9-2	A	C-5	A
2-4	G	7-2	D	9-3	A	C-6	A
2-4a/2-4b	G	7-3	A	9-4	A	C-7	A
2-5	C	7-4	A	9-5	A	C-8	A
2-6	A	7-5	A	9-6	A	C-9	A
3-1	A	7-6	A	9-7	A	C-10	A
3-2	A	7-7	A	9-8	A	C-11	A
3-3	A	7-8	A	9-9	A	C-12	A
3-4	A	7-9/7-10	D	9-10	A	C-13	A
3-5	A	8-1	A	9-11	A	C-14	A
3-6	A	8-2	A	9-12	A	C-15/C-16	A
3-7	A	8-3	A	9-13	A	D-1	A
3-8	A	8-4	A	9-14	A	D-2	A
4-1	A	8-5	A	9-15	A	D-3/D-4	A
4-2	A	8-6	A	9-16	A	E-1	A
4-3	A	8-7	A	9-17/9-18	A	E-2	A
4-4	A	8-8	A	10-1	A		
		8-9	A	10-2	G		
		8-10	A	10-3	G		
				10-4	G		

LIST OF EFFECTIVE PAGES (continued)

Page	Rev.	Page	Rev.	Page	Rev.	Page	Rev.
F-1	C						
F-2	C						
F-3	C						
F-4	C						
F-5	C						
F-6	C						
F-7	C						
F-8	C						
F-9	C						
F-10	C						
F-11	C						
F-12	C						
F-13	C						
F-14	C						
F-15	C						
F-16	C						
F-17	C						
F-18	G						
F-18a/F-18b	G						
G-1	D						
G-2	D						
G-3	G						
G-4	G						
G-4a/G-4b	G						
I-1	A						
I-2	B						
I-3	A						
I-4	A						
I-5	A						
I-6	C						
I-7	A						
I-8	A						
I-9	D						
I-10	A						

CONTENTS (CONT.)

Section	Page
Buffer Statements	9-10
BUFFER IN	9-10
BUFFER OUT	9-11
File Control Statements	9-12
REWIND	9-12
BACKSPACE	9-12
ENDFILE	9-13
Internal Transmission Statements	9-13
ENCODE	9-13
DECODE	9-15
Status Checking Routine	9-17/9-18
I/O Complete Check.....	9-17/9-18
10 PROGRAM OPERATION	10-1
FORTRAN Control Card	10-1
Control Card Notes	10-3
Calling Sequences	10-4a
Sample Deck Structures	10-5

APPENDIXES

A	Library Routines
B	FORTRAN Standard Output
C	FORTRAN Diagnostics
D	FORTRAN Statements
E	Character Codes
F	FORTRAN Interface Routines
G	MPX/OS Special Features
INDEX	

TABLES

Table	Parameter/LU	Page
10-1	Parameter/LU	10-2

LOGICAL

A logical constant is a truth value:

.TRUE. or .FALSE.

A logical constant occupies one bit of storage: 1 for true and 0 for false.

For example:

LOGICAL X1, X2:

X1 = .TRUE.

X2 = .FALSE.

VARIABLES

A variable name consists of 1 to 8 alphanumeric characters; the first character must be alphabetic. It represents a specific storage location.

The FTN-60 compiler recognizes simple and subscripted variable names. A simple variable name represents a single quantity; a subscripted variable name represents a single quantity within an array of quantities. The type of a variable is designated either explicitly in a type declaration or implicitly by the first letter of the variable name. A first letter of I, J, K, L, M, or N indicates an integer (fixed point) variable; any other first letter indicates a single precision real (floating point) variable.

SIMPLE VARIABLES

A simple variable name identifies the location where a variable value can be stored. A variable which has been defined as double precision real occupies two consecutive memory locations. Integer and single precision variable names refer to single memory locations. Variable names which have been declared as character or logical types correspond to character addresses and bit addresses, respectively.

SUBSCRIPTED VARIABLES

A subscripted variable name identifies the location in an array where a variable value can be stored.

An array is a block of successive memory locations comprising the elements of the array. Each element of an array is referenced by the array name plus a set of subscripts. The type of an array is determined by the array name or a type declaration.

Arrays may have one, two, or three dimensions; the maximum number of array elements is the product of the dimensions. The maximum number of words used in an array cannot exceed 65,535. The array name and its dimensions must be declared at the beginning of the program in a DIMENSION, COMMON, or SCRATCH COMMON statement.

Subscript Forms

A subscript indicates the position of a particular element in an array. A subscript consists of a pair of parentheses enclosing from one to three subscript expressions which are separated by commas. The subscript follows the array name. A subscript expression can be any valid arithmetic expression. The value of the expression must be integer.

If the number of subscript expressions is less than the number of declared dimensions, the compiler assumes the omitted subscripts have a value of one. The number of subscript expressions in a reference must not exceed the number of declared dimensions.

The value of a subscript must never be zero or negative. It should be less than or equal to the product of the declared dimensions, or the reference will be outside the array. If the reference is outside the bounds of the array, results are unpredictable.

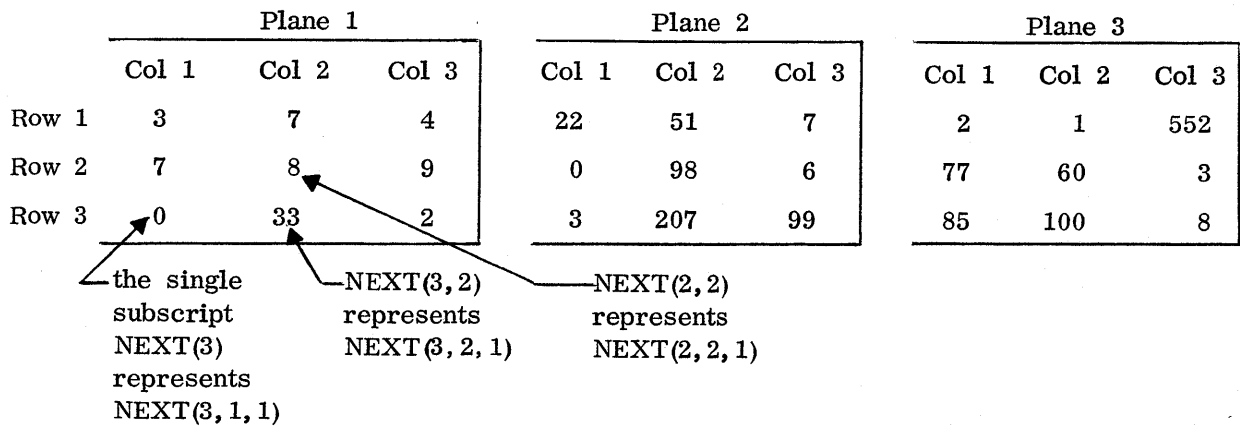
Valid subscript forms:

```
A(1, K)                                ARRY1(ARRY2(I, J*K-M+4), ARRY3(I), 10)
B(1+2, J-3, 6*K+2)
LAST(6)
ARRAYD(1, 3, 2)
STRING(3*K*ITEM+3)
```

Invalid subscript forms:

```
ATLAS(0)      zero subscript causes a reference outside of the array
D(1 .GE. K)   relational or logical expression illegal
A(, 1) or A(1, , K) commas can only be used to separate adjacent subscript
expressions
```

Example:



In the three-dimensional array NEXT when only one or two subscripts are shown, the remaining subscripts are assumed to be one.

At no time during program execution can a simple integer variable used as an index variable take on a value greater than 65,535.

Array Structure

Elements of an array are stored by column in ascending storage locations. The location of an array element with respect to the first element is determined by the maximum array dimensions and the type of the array.

The first element of array A(I, J, K) is (1, 1, 1). The location of element A(i, j, k) with respect to A(1, 1, 1) is:

$$\text{Loc } A(i, j, k) = \text{loc } A(1, 1, 1) + ((i-1) + (j-1)*I + (k-1)*I*J) * E$$



The MP-60 FORTRAN compiler (FTN-60) is called into execution via an MPX library task control card. Parameters on this control card define compile options and are passed to the compiler through the PARM area assigned to the job.

A description of the FORTRAN control card and other information needed to compile and execute MP-60 FORTRAN programs under MPX are contained in this section. For detailed descriptions of control cards, both necessary and optional, refer to the MP-60 Reference Manual, Control Data publication No. 14306500.

FORTNAN CONTROL CARD

The MPX task name control card that causes MP-60 FORTRAN to be called, loaded, and executed appears as follows:

```
*FTN (field1, field2, ..., field6)
```

All fields are optional and may appear in any order on the card. Blanks are ignored, and illegal characters are assumed to be commas. Fields are of the following formats:

Parameter = logical unit,

Parameter,

Parameter can be one of the following letters: I, L, R, A, X, or P (refer to Table 10-1). Refer to the MPX/RT Reference Manual, Control Data publication No. 14062300, for legal logical unit (LU) numbers.

Sample *FTN card:

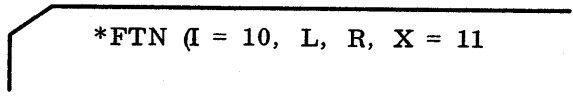
```
*FTN (L, A, X, P)
```

TABLE 10-1. PARAMETER/LU

Parameter	No LU	LU	No Field Present
I	Source input from LU 63, standard input unit.	Source input from named LU. If LU is other than 63, user must ensure that file is OPEN to job.	Source input from LU 63.
L	Source language listing and diagnostics appear on LU 62, standard output unit.	Source language listing written on specified LU.	No source listing provided.
R	Cross-reference list written on LU 62.	Cross-reference list written on specified LU. Note: It is recommended that cross reference unit be same as source listing unit.	No cross-reference listing provided.
A	Assembly language listing written on LU 62.	Assembly language listing written on specified LU.	No listing of assembly language produced.
X	Relocatable binary output written on LU 57, standard load and go file.	Relocatable binary card images of compiled programs written on named LU.	No relocatable binary file written.
P	Relocatable binary output written on LU 61, standard punch unit.	Relocatable binary card images of compiled programs written on named LU.	No binary deck provided.
O	Optimization is performed.	= 2 Associativity will also be performed.	No optimization is performed.

NOTE: The cross reference (R) must be turned on to remove the dead variables when the optimizer is on.

Source language and assembly listings are written on the standard output unit, LU 62. The punchable output is written on the standard punch unit, LU 61, and automatically punched. Executable output is written on the standard load and go file, LU 57. Input source was read from the standard input unit, LU 63.



*FTN (I = 10, L, R, X = 11

Source input is from LU 10. Source output listing with cross-reference listing is on LU 62. Executable binary is written on LU 11.

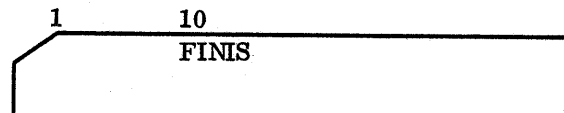
CONTROL CARD NOTES

For a detailed description of all MPX control cards, refer to the MPX/RT Reference Manual (Section 2). Information relevant to FORTRAN is, however, contained in this section.

Core memory assigned to a job should be requested by the user via the MPX schedule statement (*SCHED). A minimum of 11 pages is needed to compile small FORTRAN programs. Larger programs require more memory, up to a maximum of 16 pages. When executing FORTRAN-compiled programs containing ASCII input or output statements, memory must be scheduled for I/O buffers (a 480-word buffer for each unique logical unit).

The FORTRAN compiler uses system scratch files 1 and 2 (LUs 59 and 60) as intermediate files. The user should reposition these files if they are to be used later in the job.

A FINIS card is used to notify the compiler that there are no more programs to be compiled. The word FINIS must begin in column 10.



1 10
 FINIS

OPTIMIZATION

If the O option is on the control card, optimization is performed on the program. The types of optimizations performed are described below.

Whenever the program is changed due to the OPTIMIZATION process, an informative message is output to the list unit, giving the number of the statement that was modified.

1. Redundant code within a sequence of statement (block) is removed. Redundant code is that code that is duplicated in a block of statements that has one entrance.

EXAMPLE

```
DIMENSION A(10,10),B(10,10)
DO 10 I=1,10
    A(I,J)=K
    B(I,J)=O
10 CONTINUE
```

The subscripts for both A and B produce the same code for the index. Therefore the code is redundant, and the calculation of the second index is removed. The index calculation for A is also used for B.

2. Constants within a statement are combined where possible.

EXAMPLE

```
A = 5 + 10 - B + 10
```

This statement would have the 5 + 10 combined and the statement would become:

```
A = 15 - B + 10
```

Note that associativity does not apply

If O=2 is on the control card, associativity will be performed.

3. Code that doesn't change values during execution of a DO loop is moved outside of the loop.

EXAMPLE

```
DO 10 I = 1,10
    J = K
    A ( I ) + J * I
10 CONTINUE
```

In this set of statements, the statement J = K does not change values after the initial evaluation. The evaluation of this statement can be moved outside of the loop to give a statement sequence similar to the sequence below:

```
J = K
DO 10 I = 1,10
    A ( I ) = J * I
10 CONTINUE
```

4. Variables that are not used or referenced are deleted from the generated code.

EXAMPLE

```
PROGRAM EX
INTEGER A (10)
DO 10 I = 1, 10
  A (I) = I
  B (I) = I

10 CONTINUE
   PRINT 100, B

100 FORMAT (10I4)
   STOP
   END
```

The array A is never used in this program. Because it isn't used, it can be removed from the generated code and reduce the amount of memory required for the program.

NOTE: The cross-reference (R) must be turned on to remove dead variables when the optimizer is on.

5. Statements that are not referenced are eliminated. A referenced statement is one that can be executed sometime during a program's execution. A nonreferenced statement usually occurs following a GOTO statement because of a missing label.

EXAMPLE

```
I = J * 10
GOTO 30
A = B + J
...
30 CONTINUE
```

The statement A = B + J (and any other statement following this and prior to the 30 continue) are nonreferenced statements. Because it is never executed, the statement(s) can be deleted and have no effect on the program execution and reduce the amount of memory needed for the program.

CALLING SEQUENCES

Programs written in MP-60 assembly language (COMPASS) may call or be called by programs compiled by MP-60 FORTRAN. Calling sequence conventions have been established for this purpose.

The calling sequence compiled for an external reference of the form
 CALL NAME (p₁,p₂...p_n) is:

JSX	NAME, R9
UJP	*+(n+1)
VFD	16/M, 16/p ₁
VFD	16/M, 16/p ₂
.	
.	
VFD	16/M, 16/p _n

Where:

NAME	Entry point of the program unit being referenced
M	Mode of p _i
	Bit 2 Set Bit address
	Bit 3 Set Char address
	Bit 4 Set Half-word address
	Bit 5 Set Word address
	Bit 6 Set Double word address
p _i *	Address of the i th parameter

The main program, the set of FORTRAN statements bounded by a PROGRAM statement and an END statement is entered initially by MPX. If the main program contains either ASCII I/O statements or STOP statements, a FORTRAN library routine with entry points Q8QENTRY and Q8QEXITS is provided to interface with MPX.

Subroutines, entered by the CALL statement or from COMPASS programs, use all registers. Functions return the function value in register RE (or RE-RF if the function is a type double precision). Both subroutines and functions require register X1 to be zero upon entrance. Refer to Section 7 for further explanation of subprogram relationships.

*Note that if p_i is type character, address field would be 14/0,18/p_i.

LIBRARY ROUTINES

A

<u>Library Function</u>	<u>Definition</u>	<u>Type of Argument</u>	<u>Type of Result</u>
ABS(a)	$ a $ (absolute value)**	Real	Real
AINT(a)	Truncation**	Real	Real
ALOG(a)	$\log_e(a)$	Real	Real
ALOG10(a)	$\log_{10}(a)$	Real	Real
AMAX0(a ₁ , a ₂ , ...)	$\max(a_1, a_2, \dots)$	Integer	Real
AMAX1(a ₁ , a ₂ , ...)	$\max(a_1, a_2, \dots)$	Real	Real
AMIN0(a ₁ , a ₂ , ...)	$\min(a_1, a_2, \dots)$	Integer	Real
AMIN1(a ₁ , a ₂ , ...)	$\min(a_1, a_2, \dots)$	Real	Real
AMOD(a ₁ , a ₂)	$a_1 \pmod{a_2}$ **	Real	Real
AND(a ₁ , a ₂)	$a_1 \wedge a_2$ **	Integer	Integer
ATAN(a)	arctan(a)	Real	Real
ATAN2(a ₁ , a ₂)	arctan (a ₁ /a ₂)	Real	Real
COS(a)	cos(a)**	Real	Real
DABS(a)	$ a $ **	Double	Double
DATAN(a)	arctan(a)	Double	Double
DATAN2(a ₁ , a ₂)	arctan(a ₁ /a ₂)	Double	Double

** Function is performed in-line (not a library routine).

<u>Library Function</u>	<u>Definition</u>	<u>Type of Argument</u>	<u>Type of Result</u>
DBLE(a)	Express single precision argument in double precision form **	Real	Double
DCOS(a)	cos(a) **	Double	Double
DDIM(a ₁ , a ₂)	a ₁ - min(a ₁ , a ₂) **	Double	Double
DEXP(a)	e ^a	Double	Double
DIM(a ₁ , a ₂)	a ₁ - min(a ₁ , a ₂) **	Real	Real
DINT(a)	Sign of a times largest integer ≤ a **	Real	Double
DFLOAT(a)	Convert from integer to double **	Integer	Double
DLOG(a)	log _e (a)	Double	Double
DLOG10(a)	log ₁₀ (a)	Double	Double
DMAX1(a ₁ , a ₂ , ...)	max(a ₁ , a ₂ , ...)	Double	Double
DMIN1(a ₁ , a ₂ , ...)	min(a ₁ , a ₂ , ...)	Double	Double
DMOD(a ₁ , a ₂)	a ₁ (mod a ₂) **	Double	Double
DSIGN(a ₁ , a ₂)	Sign of a ₂ times a ₁ **	Double	Double
DSIN(a)	sin(a) **	Double	Double
DSQRT(a)	√a	Double	Double
ENABLE%	Initialize fault indicators and enable arithmetic class interrupts	Not Applicable	Integer
EXP(a)	e ^a	Real	Real
FDATE(a)	Subroutine to obtain system data	Any Type	ASCII (2 words)
FLOAT(a)	Conversion from integer to real **	Integer	Real

** Function is performed in-line (not a library routine).

<u>Library Function</u>	<u>Definition</u>	<u>Type of Argument</u>	<u>Type of Result</u>
IABS(a)	a **	Integer	Integer
IDIM(a ₁ , a ₂)	a ₁ - min(a ₁ , a ₂) **	Integer	Integer
FTIME(a)	Subroutine to obtain system time	Any Type	ASCII (2 words)
IARCHK(I)	Determine if arithmetic overflow has occurred. Returns 1 if there is a fault, 2 if there is no fault	Integer variable	Integer
IDINT(a)	Sign of a times largest integer $\leq a $ **	Double	Integer
IDVCHK(I)	Determine if divide fault has occurred. Returns 1 if there is a fault, 2 if there is no fault	Integer variable	Integer
IFIX(a)	Conversion from real to integer **	Real	Integer
IFNCHK(I)	Determine if function fault has occurred. Returns 1 if there is a fault, 2 if there is no fault	Integer variable	Integer
INT(a)	Sign of a times largest integer $\leq a $ **	Real	Integer
IOVERFL(I)	Determine if exponent overflow has occurred. Returns 1 if there is a fault, 2 if there is no fault	Integer variable	Integer
ISHFT(a ₁ , a ₂)	Value is first argument shifted by second. If second argument is negative, shift is right; if positive, shift is left circular **	Any Type	Integer
ISIGN(a ₁ , a ₂)	Sign of a ₂ times a ₁ **	Integer	Integer
MAX0(a ₁ , a ₂ , ...)	max (a ₁ , a ₂ , ...)	Integer	Integer
MAX1(a ₁ , a ₂ , ...)	max(a ₁ , a ₂ , ...)	Real	Integer

** Function is performed in-line (not a library routine).

<u>Library Function</u>	<u>Definition</u>	<u>Type of Argument</u>	<u>Type of Result</u>
MIN0(a ₁ , a ₂ , ...)	min (a ₁ , a ₂ , ...)	Integer	Integer
MIN1(a ₁ , a ₂ , ...)	min(a ₁ , a ₂ , ...)	Real	Integer
MOD(a ₁ , a ₂)	a ₁ (mod a ₂)* **	Integer	Integer
NOT(a)	\bar{a} **	Integer	Integer
OR(a ₁ , a ₂)	a ₁ ∨ a ₂ **	Integer	Integer
SECOND(a)	System time in seconds	Any Type	Real
SIGN(a ₁ , a ₂)	Sign of a ₂ times a ₁ **	Real	Real
SIN(a)	sin(a) **	Real	Real
SNGL(a)	Obtain most significant part of double precision argument **	Double	Real
SQRT(a)/SQRTF(a)	\sqrt{a} **	Real	Real
TAN(a)	tan(a)	Real	Real
TANH(a)	tanh(a)	Real	Real
XOR(a ₁ , a ₂) or EOR(a ₁ , a ₂)	a ₁ ⊕ a ₂	Integer	Integer

*a₁ (mod a₂) is defined as a₁ - $\left[\frac{a_1}{a_2} \right] a_2$, where [X] is the integer whose magnitude does not exceed the magnitude of X and whose sign is the same as X.

** Function is performed in-line (not a library routine).

FTNWRITE

This subroutine is called as follows:

```
CALL FTNWRITE (P1, P2, P3, P4)
```

where P1 = First word/byte of data to be written (see P3 for word/byte)

P2 = Number of words/bytes to be written (see P3 for words/bytes)

P3 = Mode control (2 digits in hexadecimal format):

00 = ASCII record, word format

10 = ASCII record, byte format

20 = BINARY record, word format

P4 = Logical unit number.

NOTE

Caution must be exercised when calling this subroutine that the FORTRAN user does not intermix with FORTRAN I/O statements without checking for completion status.

This subroutine will load the address of P1 and the values of P2 through P4 into registers and perform a monitor call to MPX routine WRITLU. Upon return of control, this subroutine will return to caller.

GROUP 4 FTNINT4

FTNPARM

This subroutine is called as follows:

```
CALL FTNPARM (P1, P2, P3)
```

where P1 = Address of area where data from PARM is to be saved

P2 = This value plus PARM will be first word retrieved from PARM

P3 = Number of words to be retrieved from PARM starting at word specified by P2.

This subroutine will load the appropriate parameters, P1 through P3 into registers. The subroutine will retrieve the locations within PARM specified by P2 and P3 and store the contents of these locations in the area specified by P1. Upon completion of this function, control will be returned to the calling program.

Q8QMOVE

This subroutine is called as follows:

```
CALL Q8QMOVE(P1, P2, P3)
```

Where:

P1 = buffer first word/character address of from area.

P2 = buffer first word/character address of to area

P3 = number of elements to move

NOTE:

P1, P2, P3 must not be logical

The subroutine will move P3 data items from P1 to P2. P3 assumes the mode of P1.

The MOVE subroutine is provided primarily to allow for the fastest possible movement of relatively large blocks of central memory. It employs the move machine instructions which were fetch and store one word (32-bits) of physical memory at a time directly from the originating block to the receiving block. Caution must be observed when the originating and receiving blocks overlap in physical memory. Review the following examples to see the effects of the move. Example 4 shows the effects of overlapped buffers.

Examples:

```
(1)  INTEGER A,B  
      DIMENSION A (100),B(100)  
      CALL Q8QMOVE(A(1),B(1),100)
```

This example will copy array A into array B

(2) CHARACTER B
 INTEGER A
 DIMENSION A (100),B(500)
 CALL Q8QMOVE(A(1),B(1),100)

This example will copy 100 elements from array A into character array B. The first 400 elements of array B will be filled. No check is made for bounds.

(3) CHARACTER A
 INTEGER B
 DIMENSION A (100),B(100)
 Call Q8QMOVE (A (1),B(1),100)

This example will copy 100 elements of A into B filling the first 25 elements of B

(4) INTEGER B
 CHARACTER A
 DIMENSION A(100)
 DIMENSION B(100)
 EQUIVALENCE (A(3),B(4))
 CALL Q8QMOVE (A(1),B(1),100)
 CALL Q8QMOVE (B(1),A(1),10)

The first call will cause a left shift of array A by 10 characters. The second call will cause a right shift of array B by 10 characters, repeating the first 10 characters every 10 characters.

	overlapped arrays				data before 1st move				data after 1st move			
B(1)					0	1	2	3	A	B	C	D
B(2)					4	5	6	7	E	F	G	H
B(3)			A(1)	A(2)	8	9	A	B	I	J	K	L
B(4)	A(3)	A(4)	A(5)	A(6)	C	D	E	F	M	N	O	P
B(5)	A(7)	A(8)	A(9)	A(10)	G	H	I	J	Q	R	S	T
B(6)	A(11)	A(12)	A(13)	A(14)	K	L	M	N	U	V	W	X

data after
2nd MOVE

A	B	C	D
E	F	G	H
I	J	A	B
C	D	E	F
G	H	I	J
A	B	C	D

Output Parameter:

K = Contents of character from PARM

Routine: IAPAA

Description: Allows passing of multiple cells to/from user PARM.

Calling Sequence:

CALL IAPAA (IA, IB, IE)

Input Parameters:

IA = Location where the multiple cells are to be stored.

IB = Beginning cell to be passed to/from PARM

IE = End cell to be passed to/from PARM. Maximum value of IE is 50.

Routine: LOCF

Description: Provides the address of a variable instead of the data itself. LOCF actually modifies the caller's code to load the address, hence LOCF is entered one time per occurrence.

Calling Sequence:

K = LOCF (VAR)

Input Parameter:

VAR = Variable for which address is desired.

Output Parameter:

K = Address of VAR

To illustrate the use of the above routines, an example is provided.

PROGRAM EXAMPLE

```
.  
.   
.   
EXTERNAL STATGC  
.   
.   
.   
DIMENSION BLOCK(2)  
SCRATCH COMMON/GLOBAL/A(4096)  
.   
.   
.   
DATA BLOCK/4HBLOC,4HKONE/  
.   
.   
.   
C  
C STATUS GLOBAL COMMON BLOCKONE  
C  
ISTATUS = IESR (STATGC, BLOCK(1), BLOCK(2), LOCF(A))  
ISIZE = IPAW(2)
```

Routine: IBDB

Description: Generate JSX call to specified blocker/deblocker routine.

Calling Sequence:

CALL IBDB (TYPE, P1, P2, P3, P4)

K = IBDB (TYPE, P1, P2, P3, P4)

Input Parameters:

TYPE = blocker/deblocker routine to execute (must be declared in an
EXTERNAL statement)

P1-P4 = Parameters one thru four, dependent on blocker/deblocker routine.

Output Parameter:

K = Contents of Parm+0, from blocker/deblocker call.

GLOBAL COMMON DECLARATIONS

A special reserved scratch common block name (GLOBAL) is to be used by the programmer to signal the beginning of global common block declarations. Scratch common blocks encountered by the loader before the occurrence of the common block name GLOBAL constitute local scratch common. All subsequent scratch common blocks including GLOBAL will start on a page boundary and comprise global common.

An example of FORTRAN coding is shown below.

```
SCRATCH COMMON/A(100), B(100)
SCRATCH COMMON/BLOCK 1/C(4096)
SCRATCH COMMON/GLOBAL/D(2048), E(2048)
SCRATCH COMMON/BLOCK 2/F(8192)
```

The example generates two pages of local scratch common and three pages of global scratch common. Global common would start at logical address 2000. Global common blocks could be mapped into addresses 2000 through 4FFF₁₆ and referenced by the arrays D, E, and F.

The following ESRs support the GLOBAL common feature.

```
GETGC
STATGC
RETGC
```

Refer to the MPX/OS Reference Manual, Section 4.1, Executive Service Requests, for the usage of the above ESRs.



COMMENT SHEET

MANUAL TITLE MP-60 Computer System FORTRAN Reference Manual

PUBLICATION NO. 14061100 REVISION G

FROM: NAME: _____
BUSINESS ADDRESS: _____

COMMENTS:

This form is not intended to be used as an order blank. Your evaluation of this manual will be welcomed by Control Data Corporation. Any errors, suggested additions or deletions, or general comments may be made below. Please include page number references and fill in publication revision level as shown by the last entry on the Record of Revision page at the front of the manual. Customer engineers are urged to use the TAR.

CUT ALONG LINE

PRINTED IN U.S.A.

AA3419 REV. 11/69

NO POSTAGE STAMP NECESSARY IF MAILED IN U. S. A.

FOLD ON DOTTED LINES AND STAPLE

STAPLE

STAPLE

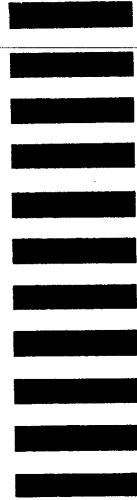
FOLD

FOLD

FIRST CLASS
PERMIT NO. 8241
MINNEAPOLIS, MINN.

BUSINESS REPLY MAIL
NO POSTAGE STAMP NECESSARY IF MAILED IN U.S.A.

POSTAGE WILL BE PAID BY
CONTROL DATA CORPORATION
3101 East 80th Street, HQG 305
Box 609
Minneapolis, Minnesota 55440



CUT ALONG LINE

FOLD

FOLD