Burroughs

B5700

Information
Processing Systems

FORTRAN COMPILER
REFERENCE MANUAL

Burroughs
B 5700
INFORMATION PROCESSING SYSTEMS

FORTRAN COMPILER
REFERENCE MANUAL

Burroughs Corporation
Detroit, Michigan 48232

$4.00

Printed in U.S. America 1-71 1051182

Burroughs Corporation believes the program described in this manual to be accurate and reliable, and much care has been taken in its preparation. However, the Corporation cannot accept any responsibility, financial or otherwise, for any consequences arising out of the use of this material. The information contained herein is subject to change. Revisions may be issued to advise of such changes and/or additions.

Correspondence regarding this document should be forwarded using the Remarks Form at the back of the manual, or may be addressed directly to Systems Documentation, Sales Technical Services, Burroughs Corporation, 6071 Second Avenue, Detroit, Michigan 48232.

TABLE OF CONTENTS

TABLE OF CONTENTS (cont)

TABLE OF CONTENTS (cont)

TABLE OF CONTENTS (cont)

SECTION TITLE PAGE

TABLE OF CONTENTS (cont)

LIST OF ILLUSTRATIONS

LIST OF ILLUSTRATIONS (cont)

LIST OF TABLES

INTRODUCTION

This manual provides a complete description of the Burroughs FORTRAN compiler language.*

The FORTRAN language is designed for writing programs for scientific and engineering applications. Statements can be written in the general format of mathematical notation, thus increasing the ease of solving formula oriented problems.

The Burroughs FORTRAN compiler operates under the control of the Master Control Program (MCP) and similarly, the object code produced by the compiler is executed under the control of the MCP. For a description of the MCP, reference should be made to the System Operation Manual.

The FORTRAN compiler language is based on USASI FORTRAN (refer to the publication: ASA X3.9-1966). Refer to appendix F of this manual for a listing of the constructs which differ from USASI FORTRAN.

* FORTRAN is an acronym for FORmula TRANslation and was originally developed for International Business Machine equipment.

SECTION 1
GENERAL PROPERTIES

GENERAL.

Normally, a FORTRAN source program is prepared on punched cards. These cards are of three general types: general program cards, comment cards, and dollar sign cards. These cards have certain column restrictions, and their format is referred to as "restricted field format." A free-field format, as used in time-sharing mode, is described in appendix J.

PROGRAM CARDS.

Program cards are used to contain FORTRAN statements under the following limitations (see figure 1-1):

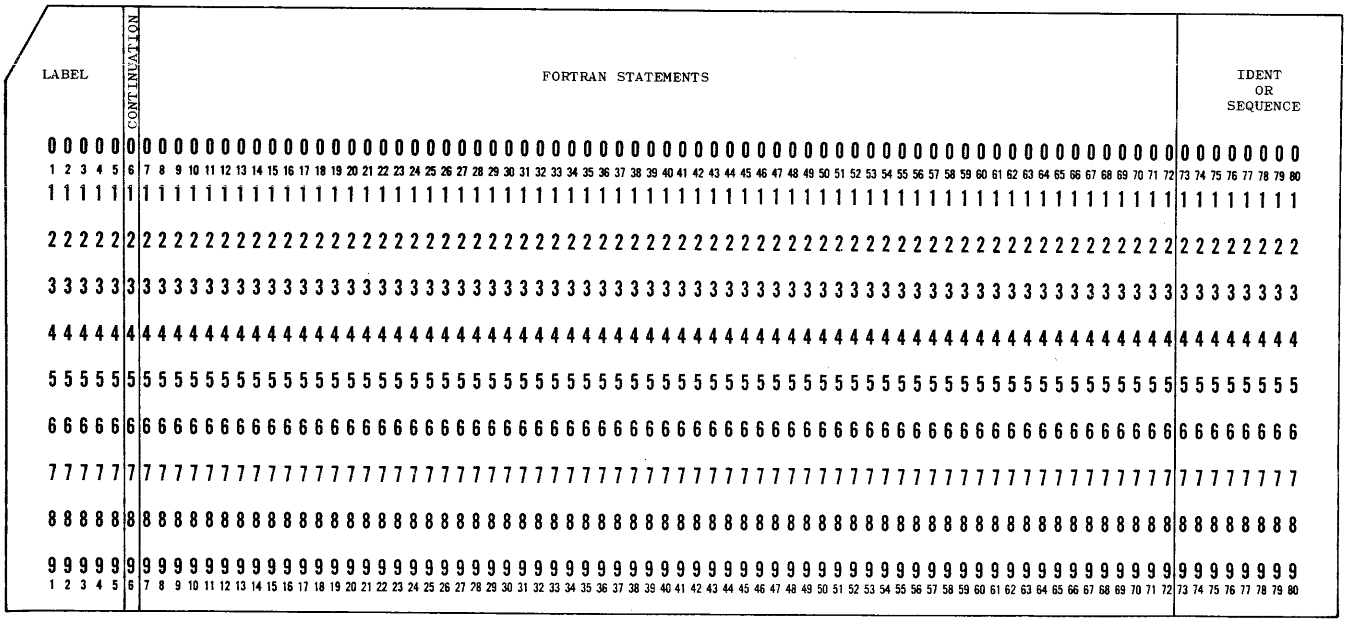


Figure 1-1. Program Card Layout

a. Columns 1-5. The label of a labeled statement consists of from one to five digits and must be placed in columns 1 through 5. The label may be placed anywhere within these columns; neither blanks nor leading zeros are significant in differentiating statement labels. All labels within a program unit must be distinct. The label field is ignored on all non-executable statements and continuation cards except for cards containing FORMAT statements.

b. Column 6. Column 6 of the initial card of a statement must be either blank or zero. Column 6 of a continuation card (any additional card after the initial card needed to contain the statement) must contain any character other than blank or zero. An unlimited number of continuation cards may follow an initial card. Continuation cards may not be labeled.

c. Blank characters are significant only in column 6 of a non-comment card, in a Hollerith constant, or in a Hollerith field specification. With these exceptions, blanks may be used or omitted without affecting the interpretation of a FORTRAN statement.

d. Column 7-72. Columns 7 through 72 contain the FORTRAN statement.

e. Columns 73-80. These columns are not interpreted by the compiler and may contain identification or sequencing information. This field is, however, analyzed when changes are merged with a source tape (see appendix C).

f. Two or more statements may be punched on the same physical card if they are separated by semicolons. If columns 1 through 5 of the card are interpreted as a label, the label corresponds to the first statement on the card. Subsequent statements on that card are considered unlabeled. The last statement must not end with a semi-colon.

g. A program unit must have an END statement as its last card. The END statement is used only to tell the compiler that it has reached the end of a program unit.

The END statement is a card with blanks in columns 1 through 6, the characters E, N, and D once each and in that order in columns 7 through 72, preceded by, interspersed with, or followed by blanks.

The END statement is not an executable statement. If a program attempts to execute an END statement, the program is terminated with an INVALID EOJ message.

COMMENT CARD.

Comment cards are not interpreted by the compiler, but their information does appear on the compilation listing for documentation purposes. Card punching limitations are as follows (see figure 1-2):

a. Column 1. A comment card must have the comment code, the letter C, in column 1.

b. Columns 2-72. Columns 2 through 72 may be used for comments.

c. Columns 73-80. These columns may contain identification or sequencing information.

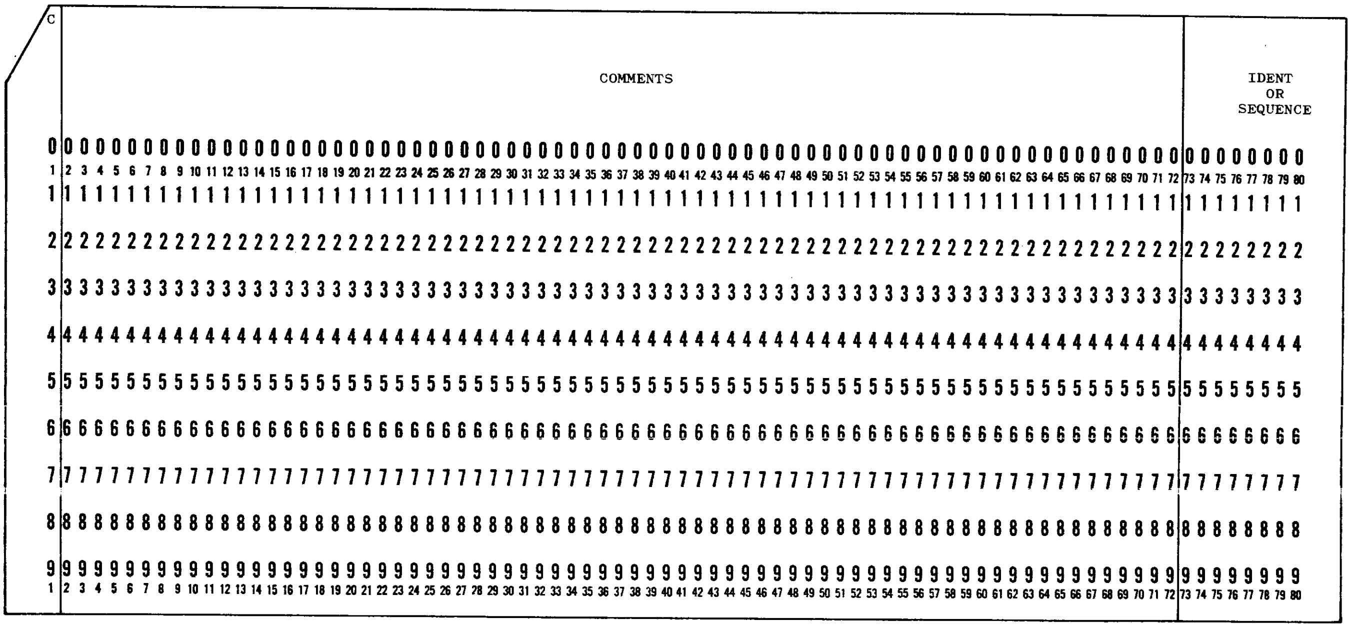


Figure 1-2. Comment Card

DOLLAR SIGN CARD.

A dollar sign card is used to specify certain compiler options (see appendix C). Dollar sign cards must not be interspersed between the continuation cards of a multi-card statement. Punching limitations of the dollar sign card are as follows (see figure 1-3):

a. Column 1. Column 1 of a dollar sign card must contain a $.

b. Columns 2-72. Columns 2 through 72 contain the compiler options desired.

c. Columns 73-80. These columns may contain identification or sequencing information.

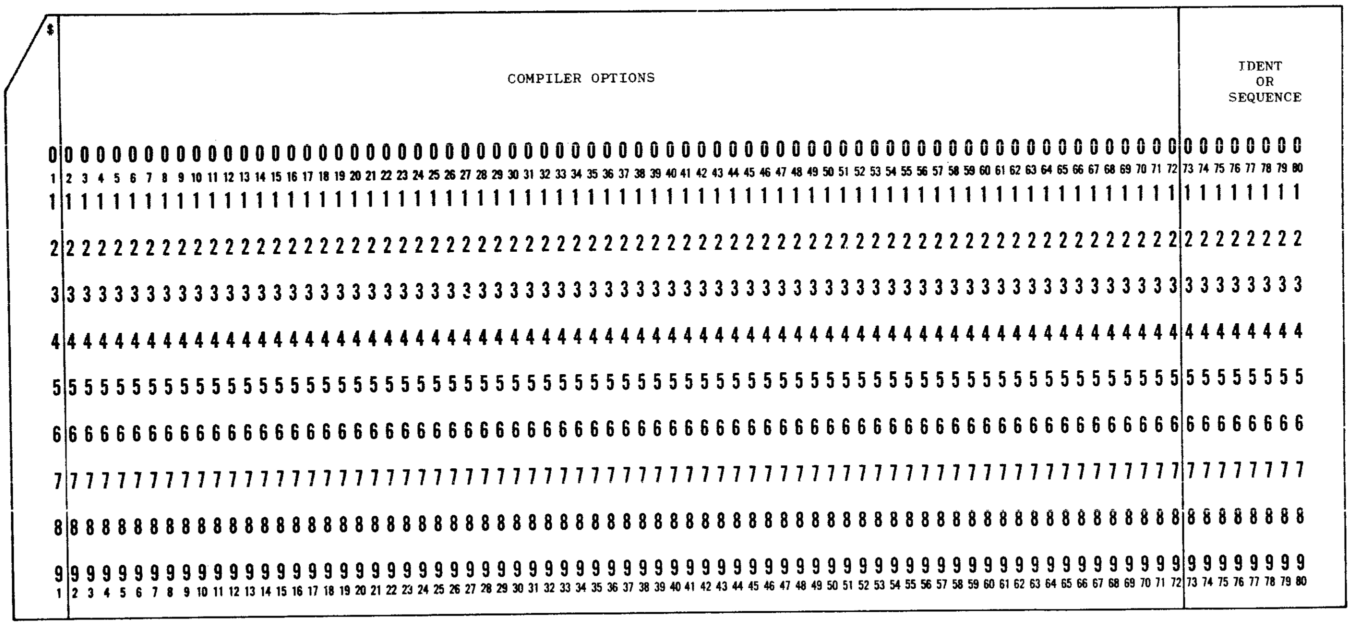


Figure 1-3. Dollar Sign Card

DECK SET-UP.

The arrangement of cards for use with the FORTRAN compiler is as follows:

```
? COMPILE CARD                                  )
? FORTRAN FILE LABEL EQUATION CARDS             )
? FORTRAN CONTROL CARDS                         )  See System
? OBJECT FILE LABEL EQUATION CARDS              )  Operation Manual
? OBJECT CONTROL CARDS                          )

? DATA CARD
  DOLLAR SIGN CARD          )
  FILE CARDS                )
      SOURCE DECK           )  See appendices
          or                )
      PATCH DECK            )
? END
```

The question mark (?) represents an invalid character, and must appear in column 1.

SECTION 2
CHARACTER SET, CONSTANTS, VARIABLES

CHARACTER SET.

The FORTRAN character set consists of digits, letters, and special characters.

DIGITS.

A digit is any one of the following ten characters: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. Digits will be in the decimal number system unless otherwise specified.

LETTERS.

A letter is any one of the following 26 characters: A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z.

SPECIAL CHARACTERS.

The special characters are divided into two categories, the USASI FORTRAN special characters and those added to Burroughs FORTRAN.

The USASI FORTRAN special characters are the following:

Character	Name
=	Equal Sign
+	Plus Sign
-	Minus Sign
*	Asterisk
/	Slash
(	Left Parenthesis
)	Right Parenthesis
,	Comma
.	Decimal Point
	Blank
$	Dollar Sign

B 5700 FORTRAN also recognizes the quote character (").

The following BCL* characters are recognized as alternatives to the standard FORTRAN character set:

FORTRAN Character	BCL Alternative
+	&
=	# or ←
(	%
)	[
*	x (BCL multiplication sign)
"	@ or :

The relational operators are represented in FORTRAN as two-letter mnemonics which are preceded and followed by a period. These operators and their BCL alternatives are:

FORTRAN Mnemonic	Meaning	BCL Alternative
.LT.	Less Than	<
.LE.	Less Than or Equal	≤
.NE.	Not Equal	≠
.GT.	Greater Than	>
.GE.	Greater Than or Equal	≥
.EQ.	Equal	=

Two remaining BCL characters serve special purposes. The semicolon (;) may be used to separate two or more statements on one physical card, and may also be used in a Hollerith field. The right bracket (]) is reserved for use by the compiler. Imbedded blanks have no meaning in a FORTRAN statement except in a Hollerith field. (see paragraph c, page 1-2).

CONSTANTS.

Six basic types of constants are allowed in the FORTRAN programming language: integer, real, double precision, complex, logical, and Hollerith.

* BCL is an abbreviation for Burroughs Common Language.

INTEGER CONSTANT.

An integer constant is formed by a string of decimal digits.

The general form is:

N
where $-549755813887 \leq N \leq +549755813887$

An integer constant is written without a decimal point or exponent.

If the range specified above is exceeded, the constant is interpreted as a double precision constant.

Examples:

12
-16729
3624138

REAL CONSTANT.

A real constant is a string of decimal digits with a decimal point and, optionally, an exponent.

The general form is:

M.NEX
where M and N are strings of decimal digits, only one of which may be blank; X is a signed or unsigned one or two-digit integer which is the exponent.

A real constant may be signed or unsigned.

An exponent is optional. If it is used, then a letter E follows the mantissa and precedes the exponent.

The exponent, if present, is interpreted such that 10^X is multiplied times the mantissa.

4.31359146672E68 ≥ ABS(R) ≥ 8.7581154021E-47, where R is a real constant, is the range within which a real constant may fall.

Examples:

```
56.9
.075
-253.
71.32E+02 (which represents 7132.0)
-71.32E-2 (which represents -.7132)
```

DOUBLE PRECISION CONSTANTS.

A double precision constant is of the same form as a real constant, except that its mantissa may contain up to 23 decimal digits and its exponent is preceded by a D instead of an E.

The general form is:

M.NDX
where M and N are strings of decimal digits, only one of which may be blank; X is a signed or unsigned one or two-digit integer which is the exponent.

The mantissa may contain up to 23 decimal digits. If more are used, then the mantissa is truncated to the 23 most-significant digits.

The range of a double precision constant is identical to that of a real constant.

A constant which does not have an exponent but which specifies more digits than a single precision value can maintain is interpreted as a double precision constant.

Examples:

```
12D-1
-5.36D+56
52D-07
.713D-17
```

USE OF SINGLE PRECISION CONSTANTS IN DOUBLE PRECISION STATEMENTS.

There can be a great difference in the internal machine representation of single or double precision constants when they are used in double precision statements. An example shows this situation best:

```
Given:   DOUBLE PRECISION D
       1 D = 3.14159265      (Single Precision Constant)
       2 D = 3.14159265D+00  (Double Precision Constant)
```

The internal representation of statement 1 takes 13 octal digits:

$$D_{internal} = 3.1103755236220000000000000_8$$

The internal representation of statement 2 takes 26 octal digits:

$$D_{internal} = 3.1103755236215236041736370_8$$

Thus, due to the very large (perhaps indefinitely large) number of octal digits required to represent the constant .14159265, there is a difference in the internal representations.

COMPLEX CONSTANT.

A complex constant, in the mathematical sense, is composed of a real part and an imaginary part.

The general form is:

(M,N)
where M is the real part and N is the imaginary part.

Each of the two components may be either a real constant or an integer constant.

Double precision components are not permitted.

Examples:

Complex Constant	Mathematical Interpretation
(5,64.2)	$5 + 64.2i$
(0,-1)	$-i$
(3.5E-2,75.9)	$.035 + 75.9i$

NOTE

$i = \sqrt{-1}$

LOGICAL CONSTANT.

A logical constant may be either true or false.

The general form is:

```
.TRUE.
.FALSE.
```

Examples:

```
.TRUE.
.FALSE.
```

HOLLERITH CONSTANT.

A Hollerith constant is a string of any valid FORTRAN characters.

The general form is:

wHs

where w is the width of the string and s is the string.

"s"

where s is the string.

Blanks appearing in the string must be included in the field width w when form wHs is used.

The string may contain any valid FORTRAN characters except the quote character (") and its alternatives: @ and :.

Strings are stored in memory, six characters per word. 36 bits

Although a word is capable of storing eight characters, only the six right-most character positions are used for storage. The two left-most character positions always contain zeros.

If a string does not contain a multiple of six characters, then in the last word used for storing the string the remaining characters are stored left-justified over a field of blanks.

Examples: (b represents blank)

```
2HbT
4H"C:"
"DOUT"
5HABCDE
```

VARIABLES.

There are two forms of variables: simple and subscripted. Each of these are, in turn, classified into five basic types: integer, real, double precision, complex, and logical.

SIMPLE VARIABLE.

A simple variable represents a single value.

The general form is:

> From one to six alphanumeric characters, the first of which must be alphabetic.

A variable name with a first character of I, J, K, L, M, or N implicitly types that variable as an integer variable. A variable name beginning with any other alphabetic character is implicitly typed as a real variable unless otherwise defined in a Type statement.

A variable of type DOUBLE PRECISION, COMPLEX, or LOGICAL must be declared as such in a Type statement.

Examples:

Integer Variables	Real Variables
IB2	A123
J12	TSUB2
KALPHA	ZSQD

SUBSCRIPTED VARIABLE.

A subscripted variable refers to a particular element of an array of the same name as the subscripted variable. (See section 6 for additional discussion.)

The general form is:

$$N(a_1,a_2,\ldots,a_n)$$

where N is the array name, $a_1,a_2,\ldots,a_n$ are arithmetic expressions which determine the values of the subscripts of the subscripted variable, and n is the number of subscripts declared in the declaration of the array N.

A subscripted variable is named and typed according to the same rules as a simple variable.

As specified by default or in the Type statement (see page 6-10), all elements of an array must be of the same type, i.e., if N(2) is integer, then N(3) must also be integer.

A subscript may be an integer or real arithmetic expression.

If a subscript is a real arithmetic expression, then it will be evaluated and converted to integer by rounding before being used as a subscript.

Subscripted variables must have their subscript bounds specified in a DIMENSION, Type, or COMMON statement prior to their first appearance in either an executable statement or in a DATA statement.

A subscript value, after any necessary conversion, must be greater than zero and may not exceed the bound specified for the array in the DIMENSION, Type, or COMMON statement in which it is declared.

Multi-dimensioned arrays are stored with the left-most subscript varying most rapidly and the right-most subscript varying least rapidly.

For example, the array A(2, 3, 4) would be stored:

```
        A(1, 1, 1)
        A(2, 1, 1)
        A(1, 2, 1)
        A(2, 2, 1)
        A(1, 3, 1)
        A(2, 3, 1)
        A(1, 1, 2)
        A(2, 1, 2)
        A(1, 2, 2)
        A(2, 2, 2)
        A(1, 3, 2)
        A(2, 3, 2)
        A(1, 1, 3)
        A(2, 1, 3)
        A(1, 2, 3)
        A(2, 2, 3)
        A(1, 3, 3)
        A(2, 3, 3)
        A(1, 1, 4)
        A(2, 1, 4)
        A(1, 2, 4)
        A(2, 2, 4)
        A(1, 3, 4)
        A(2, 3, 4)
```

Examples:

```
B(I)
GSUB(8*K+3,L)
DMIN(I,J,K)
ISUB(I,J,*K/L,C,B*D,F/G)
```

SECTION 3
EXPRESSIONS

GENERAL.

An expression is any constant, variable, or function reference, or combination of these separated by operators, commas, or parentheses. There are two types of expressions:

a. Arithmetic.

b. Logical.

ARITHMETIC EXPRESSION.

An arithmetic expression is a rule for computing a numerical value.

The general form is:

```
Any constant, variable, or function ref-
erence, or combination of these separated
by operators, commas, or parentheses.
```

An arithmetic expression may contain the following arithmetic operators:

Operator	Meaning
+	Addition
-	Subtraction
*	Multiplication
/	Division
**	Exponentiation
()	Grouping Operator Pair

Arithmetic expressions may be connected by arithmetic operators to form longer arithmetic expressions, provided no two operators appear in sequence and no arithmetic operator is assumed to be present. Examples of invalid arithmetic expressions are:

```
A++B
(A+2)(B+3)
```

Any arithmetic expression can be enclosed in parentheses.

All actual arguments of a function reference are evaluated before the function is evaluated.

Parentheses may be used in an arithmetic expression to denote the order in which operations are to be performed. Parentheses have first precedence in determining the order of evaluation and, when nested parentheses occur, evaluation proceeds from the innermost to outermost set.

The precedence order used in evaluating an arithmetic expression is as follows:

(highest)	Primary
	Exponentiation
	Multiplication and division
(lowest)	Addition and subtraction

where the precedence for successive operators of the same level is from left to right, e.g., A**B**C is evaluated as (A**B)**C.

For the operation A**B, the valid combinations and results are noted in table 3-1.

Table 3-1
Resultant Type for Operation A**B

Base A	Exponent B			
	Integer	Real	Double Precision	Complex
Integer	Integer	Real	Double Precision	Not permitted
Real	Real	Real	Double Precision	Not permitted
Double Precision	Double Precision	Double Precision	Double Precision	Not permitted
Complex	Complex	Complex	Complex (see NOTE)	Not permitted

NOTE

The double precision exponent is converted to real before exponentiation.

Any element may be combined with any other element through use of all the arithmetic operators except exponentiation. The resultant type is listed in table 3-2 for A OP B, where A and B are operands and OP is either +, -, *, or /.

Examples:

```
B
2.316
K + 1
(X + A(I,J,L) - SIN(Y(K)))
X - C + Y(I,L) * 16.397
```

Table 3-2

Combination of Elements

A	B			
	Integer	Real	Double Precision	Complex
Integer	Integer (see NOTE 1)	Real	Double Precision	Complex
Real	Real	Real	Double Precision	Complex
Double Precision	Double Precision	Double Precision	Double Precision	Complex (see NOTE 2)
Complex	Complex	Complex	Complex (see NOTE 2)	Complex

NOTE 1

Integer division yields a truncated result.

NOTE 2

The double precision element is converted to real before the operation.

LOGICAL EXPRESSION.

A logical expression is a rule for computing a logical value.

The general form is:

> Any constant, variable, or function reference, or combination of these separated by operators, logical operators, commas, or parentheses.

Logical quantities may be combined by logical operators to form logical expressions in a manner analogous to the combination of arithmetic quantities by arithmetic operators.

A logical quantity, of itself, may also constitute a logical expression.

A logical quantity may be:

a. Any logical variable.
b. Either of the logical constants .TRUE. or .FALSE.
c. Any logical function reference.
d. Any relation.

The logical operators are defined in table 3-3.

The precedence of operators in the evaluation of logical expressions is:

(highest)	Function reference
	** (Exponentiation)
	* and / (Multiplication and division)
	+ and - (Addition and subtraction)
	.LT., .LE., .EQ., .NE., .GT., .GE.
	.NOT.
	.AND.
(lowest)	.OR.

Parentheses may be used to alter the order of evaluation (just as in arithmetic expressions).

Table 3-3

Definitions of Logical Operators

Operator	Definition
.NOT.	The expression .NOT. P is .TRUE. when P is .FALSE. The expression .NOT. P is .FALSE. when P is .TRUE.
.AND.	The expression P .AND. Q is .TRUE. when both P and Q are .TRUE. It is .FALSE. if either P or Q is .FALSE. or both are .FALSE.
.OR.	The expression P .OR. Q is .TRUE. if either P or Q, or both, are .TRUE. It is .FALSE. if and only if both P and Q are .FALSE.

Examples:

If A and B are logical expressions, then each of the following is also a logical expression:

```
.NOT. B
A
(B)
A.OR.B
((B))
B .AND. A
```

RELATION.

A relation is a conditional logical expression.

The general form is:

A OP B
where A and B are arithmetic expressions and OP is a relational operator.

The relational operators and their meaning are noted in table 3-4.

Table 3-4

Relations and Meanings

Relation	Meaning
A_1 .GT. A_2	A_1 Greater Than A_2
A_1 .GE. A_2	A_1 Greater Than or Equal to A_2
A_1 .LT. A_2	A_1 Less Than A_2
A_1 .LE. A_2	A_1 Less Than or Equal to A_2
A_1 .NE. A_2	A_1 Not Equal to A_2
A_1 .EQ. A_2	A_1 Equal to A_2

NOTE

A_1 and A_2 may be of type INTEGER, REAL, or DOUBLE PRECISION. Neither may be of type COMPLEX.

Relations, when evaluated, may have one of two values, true or false.

Chains of relations are not permitted, e.g.,

```
A .LT. B .LT. C
```

A correct form would be:

```
A .LT. B .AND. B .LT. C
```

or

```
A .LT. B .AND. A .LT. C
```

whichever is intended.

Examples:

(A, B, Q, Z, E, F, X, G, H, and Y are arithmetic expressions.)

```
A .LT. B
A .LT. B .AND. Q .GT. Z
(E+F).NE.SIN(X).OR.(G-H).LT.ABS(Y)
A.LT.B.AND.(C.NE.D.OR.A.NE.D)
```

SECTION 4
ASSIGNMENT STATEMENTS

GENERAL.

There are three types of assignment statements:

a. Arithmetic assignment statement.

b. Logical assignment statement.

c. ASSIGN statement.

ARITHMETIC ASSIGNMENT STATEMENT.

The arithmetic assignment statement causes the value represented by an arithmetic expression appearing to the right of the assignment operator (=) to be assigned to the simple or subscripted variable appearing to the left of the assignment operator.

The general form is:

v = a.e.
where v represents a variable name, simple or subscripted, and a.e. represents an arithmetic expression.

The variable v cannot be of type LOGICAL.

The rules provided in table 4-1 apply for type and value assignment in arithmetic expressions.

Examples:

```
X = Y+Z
X(10) = A(5)+B(6)-(C/D)
JX = 342
X = 5.49
X(1) = B(1)+COS(A(1))
X(4) = D - C**2
X(I,J) = A(I,J)+B(J,I)
```

Table 4-1

Rules for Arithmetic Assignment Statement (v = a)

v	a	Rule
Integer	Integer	Assign.
Integer	Real	Truncate to an integer and assign.
Integer	Double Precision	Truncate to an integer and assign.
Integer	Complex	Not permitted.
Real	Integer	Assign.
Real	Real	Assign.
Real	Double Precision	Assign the most-significant part.
Real	Complex	Not permitted.
Double Precision	Integer	Extend to double precision and assign.
Double Precision	Real	Extend to double precision and assign.
Double Precision	Double Precision	Assign.
Double Precision	Complex	Not permitted.
Complex	Integer	Not permitted.
Complex	Real	Not permitted.
Complex	Double Precision	Not permitted.
Complex	Complex	Assign.

LOGICAL ASSIGNMENT STATEMENT.

The logical assignment statement causes the value represented by the logical expression appearing to the right of the assignment operator (=) to be assigned to the simple or subscripted variable of type LOGICAL appearing to the left of the replacement operator.

The general form is:

v = l.e.
where v is a simple or subscripted variable of type LOGICAL and l.e. represents a logical expression.

The variable v must be of type LOGICAL.

Examples:

(K, L, M, and N are logical variables.)

```
          K = A .OR. B
          L(J,5) = .TRUE.
          M = A .LT. B
          N = Q .GT. R .AND. Z .LT. P
```

ASSIGN STATEMENT.

The ASSIGN statement is used to initialize an assigned GO TO statement (see section 5).

The general form is:

ASSIGN n TO t
where n is a statement label referenced in an assigned GO TO statement, and t is a simple integer or real variable appearing in the same assigned GO TO statement.

The statement label n must be referenced in the assigned GO TO statement being initialized.

The variable t must be the same variable referenced in the assigned GO TO statement being initialized.

Example:

```
          ASSIGN 10 TO J
```

SECTION 5
CONTROL STATEMENTS

GENERAL.

Control statements are used to alter the normal flow of a program. They may transfer control to another part of the program, terminate computation, or control iterative processes. Control may be transferred to labeled executable statements only. There are 12 different control statements:

a. Unconditional GO TO statement.
b. Computed GO TO statement.
c. Assigned GO TO statement.
d. Arithmetic IF statement.
e. Logical IF statement.
f. DO statement.
g. CONTINUE statement.
h. PAUSE statement.
i. STOP or CALL EXIT statement.
j. RETURN statement.
k. CALL statement (not including CALL EXIT and CALL ZIP).
l. CALL ZIP statement.

UNCONDITIONAL GO TO STATEMENT.

Execution of this statement causes control to be transferred to a statement other than that sequentially following the unconditional GO TO statement.

The general form is:

GO TO n
where n is a statement label which exists within the same program unit.

A statement label n must be defined within the same program unit as the unconditional GO TO statement which references it.

The statement labeled n may appear before or after the unconditional GO TO statement referencing it.

Example:

```
         GO TO 31
         ...
         ...
      31 ...
```

COMPUTED GO TO STATEMENT.

Execution of this statement causes control to be transferred to one of several statements other than that sequentially following the computed GO TO statement.

The general form is:

GO TO $(n_1,n_2,\ldots,n_i)$, t
where $n_1,n_2,\ldots,n_i$ are statement labels and t is an arithmetic expression.

Control will be transferred to the statement label whose position in the list is equal to the value of the arithmetic expression t, i.e., n_t.

The statement labels $n_1,n_2,\ldots,n_i$ must exist in the same program unit as the computed GO TO statement.

The computed GO TO statement is valid for values of t such that $1 \leq t \leq i$, otherwise the program will be terminated with an INVALID INDEX.

The arithmetic expression t must be of type INTEGER or of type REAL.

If t is of type REAL, it will be evaluated and then rounded to an integer.

Example:

```
        K=4
        GO TO (50,40,30,20,10),K
```

Execution of these two statements will cause control to be transferred to statement 20.

ASSIGNED GO TO STATEMENT.

Execution of this statement causes control to be transferred to one of several alternative statements other than that sequentially following the assigned GO TO statement.

The general form is:

GO TO t, (n_1,n_2,...,n_i)

where t is a simple integer or real variable and n_1,n_2,...,n_i are statement labels.

Control will be transferred to the statement whose label has been ASSIGNed to t with an ASSIGN statement.

The values ASSIGNable to t are the actual statement labels appearing in the list n_1,n_2,...,n_i.

The variable t must be a simple integer or real variable.

If t has not been assigned a label appearing in the list, an INVALID INDEX termination of the program will result.

The statement labels n_1,n_2,...,n_i must appear in the same program unit as the ASSIGN statement and the ASSIGNed GO TO statement (see ASSIGN statement, section 4).

Example:

```
        ...
        ...
        ASSIGN 10 TO J
        GO TO J,(50,40,30,20,10)
```

Execution of these two statements will cause control to be transferred to statement 10.

ARITHMETIC IF STATEMENT.

Execution of the arithmetic IF statement causes an arithmetic expression to be evaluated and a different branch to be made depending upon whether the expression evaluated is negative, zero, or positive.

The general form is:

IF(a.e.)n_1;n_2;n_3
where a.e. is an arithmetic expression and n_1,n_2, and n_3 are statement labels.

Execution of the arithmetic IF statement causes control to be transferred to n_1;n_2, or n_3 if a.e. is less than, equal to, or greater than zero, respectively.

The arithmetic expression a.e. may not be complex.

Examples:

```
        IF(A-B) 1,2,3
        IF(X(I,J)-C*E) 43,51,96
```

LOGICAL IF STATEMENT.

Execution of the logical IF statement causes a logical expression to be evaluated and the sequence of execution of the program statements to be altered, depending upon whether the logical expression evaluated is true or false.

The general form is:

IF(l.e.) s
where l.e. is a logical expression and s is an executable FORTRAN statement.

The statement s may be any executable FORTRAN statement except a DO statement.

Execution of the logical IF statement results in the logical expression l.e. being evaluated. If l.e. is true, statement s is executed. If l.e. is false, then statement s is not executed, and control is transferred to the next sequential executable statement following the logical IF statement.

Examples:

X and Y are of type LOGICAL.

```
          IF(X .AND. Y) A = 3.1
          IF(A .LE. B .OR. I .EQ. 0) GO TO 5
```

DO STATEMENT.

The DO statement provides a means of controlling program loops.

The general form is:

DO m i=n_1,n_2,n_3

where m is a statement label, i is a variable, and n_1,n_2, and n_3 are arithmetic expressions.

Execution of a DO statement results in the following actions:

a. The control variable i is set to the initial value n_1.

b. All executable statements up to and including the terminal statement are executed.

c. The control variable i is incremented by n_3.

d. The value of the control variable i is compared to the terminal value n_2. If the terminal value has been exceeded, control is transferred to the first executable statement following the terminal statement. Otherwise, steps b through d are repeated until the control variable comparison is satisfied.

In the general form, the control variable i is a simple integer or real variable.

In the general form, m is the label of an executable statement terminating the DO loop.

In the general form, n_1, n_2, and n_3 are integer or real arithmetic expressions which are the initial, terminal, and incremental parameters, respectively, for the control variable i.

If not specified, n_3 is assumed to be 1.

If present, n_3 must be greater than zero.

In the general form, n_2 must be greater than n_1.

The DO statement is always executed once with its initial value.

The DO parameter is incremented and compared to n_2 prior to its use and may be modified by any statement within the range of the DO loop.

The control variable i is available for use by all statements within the DO loop, including the terminal statement, and may be modified as desired. The control variable i is available for computation when exiting from a DO loop by transferring outside the loop and not making a normal exit. When a normal exit is made from the DO loop, the control variable is undefined.

A DO statement may appear within a DO loop. This is defined as being a DO nest. However, all statements in the range of the latter DO loops must be within the range of the initial DO loop (see figure 5-1).

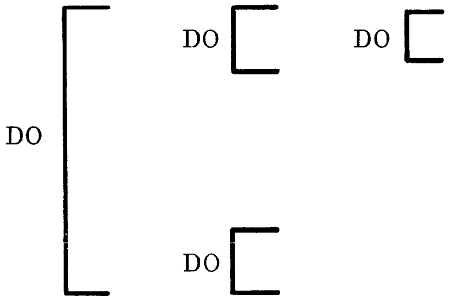


Figure 5-1. DO Nesting

Nested DO's may specify the same statement as their last statement m.

Any number of DO statements may be nested within the range of another DO statement.

There are no restrictions on transfer out of or into the range of a DO loop. If a transfer is made into the range of a DO loop, then the programmer is responsible for the appropriate assignment of a value to the control variable i. If no assignment for i is indicated, it is assumed to be zero.

When several DO statements share the same last statement m, the control variable i of the outermost DO statements is not reassigned and tested until each of the inner DO statements in its range is satisfied, starting with the innermost one. When the outermost DO statement is satisfied, a normal exit is made and control is transferred to the next executable statement following the range of the just-satisfied DO loop.

Examples:

```
          DO 10 I=2,200,4
          ...
          ...
          ...
      10  ...

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

CONTINUE STATEMENT.

The CONTINUE statement is considered a dummy statement because it causes no action in the execution of a program. It is frequently used as the terminal statement of a DO loop to provide a transfer point for an IF or GO TO statement.

The general form is:

```
CONTINUE
```

Example:

```
        DO 30 J=2,N
        B(J)=NM(J-1) + INC
        IF (N(J).LT. MAX) GO TO 30
        K=J-1
        GO TO 40
   30   CONTINUE
   40   ...
        ...
        ...
```

PAUSE STATEMENT.

The PAUSE statement is used to interrupt execution of a program when action is required of the computer operator.

The general form is:

PAUSE n
where n is an integer constant up to six digits long or is blank.

If the n is present, it is displayed to the operator at the time of interruption. Execution is resumed with the first executable statement immediately following the PAUSE statement after an OK message has been keyed in at the SPO (refer to System Operation Manual).

NOTE

SPO stands for SuPervisOry printer, and specifies the console typewriter.

Examples:

```
PAUSE

PAUSE 30
```

STOP OR CALL EXIT STATEMENT.

The STOP statement causes an immediate termination of the program. A STOP statement must appear prior to the END statement in the main program. If it is omitted, the program will be DSed with an INVALID EOJ.

The general form is:

STOP n
where n is an integer constant up to six digits long or is blank.

The CALL EXIT statement is equivalent to STOP.

The general form is:

```
CALL EXIT
```

Examples:

```
STOP

STOP 4

CALL EXIT
```

RETURN STATEMENT.

Execution of the RETURN statement causes control to be transferred from a subprogram to the calling program.

The general form is:

RETURN n
where n is an arithmetic expression or is blank.

Every subprogram must contain at least one RETURN statement, but more than one may appear in a subprogram.

If n is not blank, control returns to the point of reference and is used to select one of the formal parameters in the subroutine represented by an asterisk (nonstandard return). Control then returns to the statement specified by a corresponding actual parameter in the calling program (see nonstandard returns, section 8).

If n is blank, then control returns to the point of reference in the calling program unit.

CALL STATEMENT.

A subroutine is referenced by a CALL statement.

The general form is:

CALL N
CALL N(a_1,a_2,...,a_n)
where N is the name of the subroutine and a_1,a_2,...,a_n are the actual parameters.

The actual parameters which constitute the parameter list must agree in order, number, and type with the corresponding formal parameters in the program unit defining the subroutine subprogram.

If a formal parameter is real, then an integer actual parameter may be used.

For purposes of type agreement, a Hollerith constant is considered of type INTEGER.

An actual parameter in a subroutine reference may be one of the following:

a. A Hollerith constant.

b. A variable name.

c. A subscripted variable.

d. An array name.

e. An expression.

f. The name of a subprogram.

g. $label (see nonstandard returns, section 8)

Execution of a subroutine reference results in an association of actual parameters with all appearances of formal parameters in executable statements in the subroutine body, and in an association of actual parameters with variable dimensions in the subroutine, if any exist.

Following the above associations, control is transferred to the first executable statement in the subroutine body.

If an actual parameter is a subscripted variable with an arithmetic expression as a subscript, then, effectively, the arithmetic expression is evaluated, and the resulting subscripted variable is associated with the corresponding formal parameter in the subroutine.

If a formal parameter of a subroutine is an array name, the corresponding actual parameter must be an array name or an array element name.

Examples:

```
CALL FALL(X,Y,Z)

CALL KOST(A(I+J,2),B,"HEAD")
```

CALL ZIP STATEMENT.

The CALL ZIP statement is used to pass control and/or parameter card information to the MCP.

The general form is:

CALL ZIP(A)
where A is an array name.

The parameter A should be a real or integer array which is large enough to contain any information that will be placed into it 6 characters per word.

If A is larger than 27 words, only the first 27 words of A will be used.

The information contained in A must be in BCL format as it would appear on physical control/program cards.

The letters CC must be the first two characters contained in the array. Only one set of these letters may appear in the array A.

The information following the letters CC must appear as a single punched card except that the array A may contain up to 162 characters (27 words, 6 characters per word).

The information that would be contained on more than one physical control card may be put into the array, but a semicolon must be used to delimit the end of a logical card.

The last logical card must be:

```
        END.
```

The control information in A should pertain to only one compiler or object program.

After the CALL ZIP statement has been executed, the FORTRAN object program that executed the statement continues processing while the MCP examines the control information in the array A. If the MCP finds an error in this control information, an appropriate error message is typed on the SPO.

<u>Example</u>:

```
              DIMENSION E(12)
              READ(5,25,END=30) (E(I),I=1, 12)
        25    FORMAT(12A6)
        30    CALL ZIP(E)
              ...
              ...

Input:        1234...........      (card column)
              CC EXECUTE A/B; COMMON=20;END.
```

In the above example, the first program, after executing statement 30, will continue processing with the next executable statement. Meanwhile, the MCP will scan the information in array E and, if program A/B is on disk, will initiate the execution of A/B. The array is available for other uses immediately upon returning from ZIP. Thus, the original program and the program initiated by it will be running simultaneously but independently.

SECTION 6
DECLARATIVE STATEMENTS

GENERAL.

The declarative statements are non-executable statements used to supply variable and array information and storage allocation information. The seven different declarative statements are:

a. DIMENSION statement.

b. COMMON statement.

c. EQUIVALENCE statement.

d. Type statement.

e. EXTERNAL statement.

f. DATA statement.

g. Function statement (see page 8-1).

DIMENSION STATEMENT.

The DIMENSION statement provides a means for specifying a collection of values with a single name, and at the same time specifying to the compiler the structure which is imposed on the collection.

The general form is:

DIMENSION $a_1(i_1)$, $a_2(i_2)$, $a_3(i_3)$ . . . .

where each a is an array name and each i represents dimension information having the form of one or more subscript bounds separated by commas.

Each bound is an integer constant.

Variable names appearing with subscripts in the source program must have dimension information specified for them prior to their use.

Dimension information may be given in a DIMENSION, COMMON, or Type statement; however, the dimension information for a specific array name must appear only once in the program unit.

The magnitude of the values of the subscript bounds indicates the maximum values the subscripts may obtain in any reference to the array. The lower subscript bound is always one.

An array may have variables for its subscript bounds in a FUNCTION or SUBROUTINE subprogram only. In this case, the array name and all variables used as subscript bounds must appear as formal parameters in the subprogram. The actual values assumed by these variables are not determined until the subprogram is entered at execution time (see variable dimensions below).

No array may exceed 32767 words.

VARIABLE DIMENSIONS.

An array can be placed in a subprogram with variables used as dimensions instead of constants. The advantage to this is that a given subprogram can perform calculations on such a generally stated array with specific dimensions provided from any calling program. The specific dimensions must be provided in a DIMENSION statement in the calling program. The actual values assumed by these variables are not determined until the subprogram is entered at execution time.

The general form is:

DIMENSION $a_1(i_1)$, $a_2(i_2)$, $a_3(i_3)$,.....

where each a is an array name and each i is one or more subscript bounds separated by commas. Each bound is an integer variable.

Variables can be used as dimensions of an array in a FUNCTION or SUBROUTINE subprogram only.

The variables must appear in a DIMENSION statement of the subroutine.

The array name and all variables used as dimensions must appear as formal parameters in the initial FUNCTION, SUBROUTINE, or ENTRY statement.

Specific dimensions passed to the subprogram from the calling program must be identified in a DIMENSION statement of the calling program.

Specific variable size can be passed down through more than one level of a subprogram to a given subprogram using the variable as a dimension.

Example:

```
DIMENSION A(10,20)
...
...
...
I=5
J=7
CALL SUB(A,I,J)
...
...
...
END
SUBROUTINE SUB(B,K,L)
DIMENSION B(K,L)
...
...
...
END
```

COMMON STATEMENT.

The COMMON statement provides a means for sharing core storage between the main program and its subprograms, or among the subprograms. Information appearing in the storage area reserved by a COMMON statement is ordered in the sequence specified by the COMMON statement. The ordered information is relative to the beginning of a given COMMON block. They are two types of COMMON storage: labeled and unlabeled.

The general form is:

COMMON/x_1/a_1/x_2/a_2/. . . /x_n/a_n
where each a in the COMMON statement is a list containing any combination of variable names, array names, or dimensioned array names, and each x is a block name or is empty. If x_1 is empty, the first two slashes are optional.

Array names in a COMMON statement may have their dimensioning information appended to them. When arrays are dimensioned in a COMMON statement, they cannot be dimensioned in Type or DIMENSION statements.

COMMON area storage is assigned in the order of appearance of the elements within the COMMON block list.

Block names may be duplicated within a program unit, causing the associated elements from each COMMON block list having the same name to be cumulatively assigned to one block with the same name. The effect is the same as declaring the block name once and listing all elements for that block in the COMMON block list. This is also true for multiple unlabeled COMMON block lists within a given program unit.

Variables and array names may not be duplicated in COMMON statements.

COMMON elements may be assigned initial values through use of the BLOCK DATA subprogram.

The number and type of variables appearing in the COMMON block list and related EQUIVALENCE statements specify the length of the COMMON block.

All subscript bounds for any array which appears in a COMMON statement must be integer constants.

A COMMON block need not have the same size in each program unit in which it appears. However, if the size of a block is greater than 1023 words in any program unit, it must also be greater than 1023 words in the first program unit in which it appears.

No COMMON block may exceed 32767 words.

A double precision or complex variable in a COMMON block must be positioned such that the first of the two words containing the double precision or complex variable is always located at an odd-numbered word location in the block. As an example, assume the variables A, B, and C to be declared as follows:

```
DOUBLE PRECISION A
COMPLEX B
REAL C
```

COMMON could then be declared in either of the following ways:

```
COMMON A, B, C
COMMON B, A, C
```

In both of these declarations, A and B begin in odd-numbered locations.

	COMMON BLOCK
1	A_1
2	A_2
3	B_1
4	B_2
5	C

	COMMON BLOCK
1	B_1
2	B_2
3	A_1
4	A_2
5	C

However, it would be <u>incorrect</u> to declare COMMON as:

```
COMMON C, A, B
COMMON C, B, A
COMMON A, C, B
COMMON B, C, A
```

In the first two examples, both A and B would begin in even locations.

	COMMON BLOCK
1	C
2	A_1
3	A_2
4	B_1
5	B_2

	COMMON BLOCK
1	C
2	B_1
3	B_2
4	A_1
5	A_2

In the third example, A would be positioned correctly but B would not.

	COMMON BLOCK
1	A_1
2	A_2
3	C
4	B_1
5	B_2

In the final example, B would be positioned correctly but A would not.

	COMMON BLOCK
1	B_1
2	B_2
3	C
4	A_1
5	A_2

Labeled COMMON statements are specified by a COMMON block name, slashes, preceding the list of elements assigned to that labeled COMMON block. Termination of the list of elements assigned to a block is by:

a. Termination of the COMMON statement.

b. Introduction of a new block name.

c. Introduction of an unlabeled COMMON block.

COMMON block names are unique identifiers, however, a block identifier may be reused within the program unit to represent another type element. When a block name is present in a COMMON statement, it is embedded in slashes, e.g., /x/.

Blocks of labeled COMMON statements in different program units which have the same block name will occupy the same storage area.

Unlabeled COMMON statements are specified by a blank block name, e.g., / /, followed by the unlabeled COMMON block list. The two slashes may be omitted if they appear at the beginning of a COMMON statement list. Termination of an unlabeled COMMON block is accomplished by the introduction of a block name or termination of the COMMON statement. COMMON elements may not be used in DATA statements (see page 6-13).

Examples:

```
COMMON X,Y,Z
COMMON /Y/Q,R,S
COMMON / / K(5,5),L
COMMON A,B,C/S/D(10,10),E
COMMON /Y/Q,R,S/ /K(5,5),L
```

EQUIVALENCE STATEMENT.

By using the EQUIVALENCE statement, a storage location can be given more than one name. Variables or array elements not listed in an EQUIVALENCE statement have unique storage assignments.

The general form is:

EQUIVALENCE $(Q_1),(Q_2),(Q_3),.....(Q_n)$

where each Q is a list of two or more simple or subscripted variables or array names separated by commas.

Subscripts must be positive integer constants and must correspond in number to the declared number of dimensions of the array, or be single subscripted by equating the element position in the array to a single subscript. For an explanation of the latter, see table 6-1.

An array name without subscripts is considered as that identifier with a subscript of one.

Any number of equivalence lists may appear in an EQUIVALENCE statement.

Elements may be entered into COMMON blocks by setting them equivalent to an element appearing in a COMMON statement list. If the element is an array element, the whole array is brought into COMMON. This may extend the size of the COMMON block involved either at its beginning or at its end.

Example:

```
COMMON Z
DIMENSION Z(100),E(200)
EQUIVALENCE (E(200),Z(1))
...
...
END
SUBROUTINE SBT
COMMON X(100)
...
...
```

The above statements will allocate storage so that X will be equivalent to the first 100 locations of E, not Z. In other words, the EQUIVALENCE statement has displaced the origin of COMMON.

When two elements share storage because of their appearance in one or more EQUIVALENCE statements, only one may appear in a COMMON statement.

All subscript bounds for an array which appears in an EQUIVALENCE statement must be integer constants.

An EQUIVALENCE statement must precede any reference to the elements EQUIVALENCEd.

Table 6-1

EQUIVALENCing Multiple Subscripts To One Subscript

Number of Dimensions	Array Declarations	Array Element	Same Array Element With One Subscript	Maximum Single-Subscript Value
1	A(I)	A(i)	A(i)	I
2	A(I,J)	A(i,j)	A(i+Ix(j-1))	IxJ
3	A(I,J,K)	A(i,j,k)	A(i+Ix(j-1)+IxJx(k-1))	IxJxK
4	A(I,J,K,L)	A(i,j,k,l)	A(i+Ix(j-1)+IxJx(k-1)+IxJx Kx(1-1))	IxJxKxL

<u>Example:</u>

```
DIMENSION C (120)
DIMENSION B (4,5,6) Element referenced B (3,2,1)
EQUIVALENCE (B C)
B(3,2,1) ≡ C([ 3]+[4x(2-1)]+[4x5x(1-1)]) = C(7)
```

When two elements share storage because of their appearance in one or more EQUIVALENCE statements, only one may appear in a COMMON statement.

All subscript bounds for an array which appears in an EQUIVALENCE statement must be integer constants.

An EQUIVALENCE statement must precede any reference to the elements EQUIVALENCEd.

Example:

```
        DIMENSION A(10), B(5,5) D(3,3,3)
        EQUIVALENCE (A(3), B(5,4), D(1,1,1)), (A(1),E)
```

The above statements assign specific variable values to the same storage locations, as shown below, where each horizontal line is one memory location.

Variable Value

A(1)	B(3,4)	...	E
A(2)	B(4,4)	...	...
A(3)	B(5,4)	D(1,1,1)	...
A(4)	B(1,5)	D(2,1,1)	...
A(5)	B(2,5)	D(3,1,1)	...
A(6)	B(3,5)	D(1,2,1)	...
A(7)	B(4,5)	D(2,2,1)	...
A(8)	B(5,5)	D(3,2,1)	...
A(9)	...	D(1,3,1)	...
A(10)	...	D(2,3,1)	...

TYPE STATEMENT.

Type statements are used to declare the type of variables, array names, and function names. The general form is:

INTEGER type list
REAL type list
DOUBLE PRECISION type list
COMPLEX type list
LOGICAL type list

where a type list is composed of variable names, array names, or statement function names separated by commas. In addition, arrays may be dimensioned by appending the dimension information to the array name in one or more subscript positions.

When a subscripted variable is declared DOUBLE PRECISION or COMPLEX, the compiler will automatically assign two words of storage for each element of an array.

Implicit type assignment is overridden by Type statements.

A variable name must be typed prior to its use in an executable statement or DATA statement. If the first letter of a variable name isI, J, K, L, M, or N, it is implicitly declared of type INTEGER and need not appear in a Type statement. If a variable name begins with any other letter, it is implicitly declared of type REAL and need not appear in a Type statement.

Examples:

```
INTEGER X,Y,Z,A(10,10)
REAL H,I,J,K
LOGICAL ATEST, BTEST
```

EXTERNAL STATEMENT.

When an actual parameter list of a function or subroutine reference contains a function or subroutine name, that name must appear in an EXTERNAL statement.

The general form is:

EXTERNAL $n_1,n_2,......,n_n$
where the n's are the names of the functions or subroutines appearing in the parameter list of a function or subroutine reference.

The EXTERNAL statement appears in the calling program unit.

Example:

```
EXTERNAL SIN,COS
CALL SUBT (SIN,COS)
...
...
END
SUBROUTINE SUBT (A,B)
TANX=A(X)/B(X)
...
...
RETURN
END
```

DATA STATEMENT.

The DATA statement permits variables and arrays to be initialized to predetermined values.

The general form is:

DATA list$_1$/d$_1$,d$_2$,d$_3$,...d$_n$/,list$_2$/d$_1$,d$_2$,...d$_n$/,...

A list element may be an array name or a simple or subscripted variable name, where the subscripts must be integer constants. If more than one element of an array is to be initialized, an implied DO loop must be employed (see implied DO loop in section 7). The d$_i$ represents a constant, or has the form i*c, where i is a repeat count and c is a constant.

The constants may be any of the following:

a. Integer, real, or double precision constant.

b. Octal constants of the form Odd...d, i.e., the letter O followed by an optional sign, followed by 1 to 16 octal digits (not greater than O+3777777777777777 nor less than O-3777777777777777). If a minus sign is included, the mantissa sign bit [1:1] is set to 1 if it was previously O, or O if it was 1.

c. Logical constants. The quantities may be expressed as .TRUE.,.FALSE.,T, or F.

d. Hollerith constants.

A one-to-one correspondence must exist between the list elements and the constants.

The first time a subprogram is entered, all of the variables contained in all DATA statements within that subprogram are initialized. In succeeding entries to the subprogram, the DATA statements are ignored and the variables within the DATA statement assume the last value assigned to them at the time of the previous exit from the subprogram.

If a Hollerith constant is used, it must be considered a single value and must correspond to a single element, even though it may actually occupy several computer words. If it occupies more than one word, the list element must be an array name or an implied DO loop with enough array elements remaining in the array to contain the Hollerith constant (see Hollerith constant in section 2).

Elements in a COMMON block may appear in a DATA statement only in a BLOCK DATA statement (see BLOCK DATA in section 8).

Variables assigned quantities by a DATA statement may be assigned other values during execution.

When an array name without subscripts appears in the list, the entire array is initialized.

Subscripted variables appearing in a program must have their subscript bounds specified in a DIMENSION, COMMON, or Type statement prior to the first appearance of the subscripted variable in a DATA statement.

<u>Example</u>:

```
DIMENSION T(3),V(3)
INTEGER X,Y,Z,A(5,5)
REAL H,I,J,N(8),K
LOGICAL ATEST,BTEST
DATA X,Y,H/1,3,5.7/,I,J,ATEST/6.2,99.99,F/
DATA Z,A,K,BTEST,(N(I),I=1,8)/0,25*0,-99.,.TRUE.,8*77.77/
DATA T(5)/6HONEWRD/
DATA V/14HABCDEFGHIJKLMN/
```

SECTION 7
INPUT/OUTPUT

GENERAL.

The following areas of Input/Output (I/O) are covered in this section:

a. Input statements.

b. Output statements.

c. I/O lists.

d. Implied DO loop.

e. Action labels.

f. Auxiliary I/O statements.

g. FORMAT statement.

h. NAMELIST statement.

i. Tape and Disk I/O.

INPUT STATEMENTS.

In explanations presented in this manual section, the symbols u, r, f, k, and l have the following meanings unless otherwise specified:

u - file specifier or unit number. The file specifier is an arithmetic expression whose value identifies the file being used for input or output. The file specifier must conform to the restrictions of a subscript. Unless otherwise specified by a FILE card (see appendix B), it is assumed at object time that the file specifier designates a tape unit. The range of u must be $0 < u < 31$.

r - random record number. It is an arithmetic expression whose value represents a particular record within a random disk file.

f - format specifier. It may be the label of a FORMAT statement, an array identifier, or a NAMELIST identifier.

l - action label. It specifies a statement label to which a branch is made if a parity error or an End-of-File condition is encountered during execution of an input statement.

k - input/output list. It may be a blank or it may contain one or more variables and/or implied DO loops, in any combination.

Execution of any of the READ statements causes the next record to be read from the specified file. The information is scanned and converted as specified by the format specifier f if the statement is a formatted READ statement. The values are assigned to the elements specified by the list k. If the list is not specified, either a record is skipped or data is read into the locations in storage occupied by the FORMAT statement.

FORMATTED INPUT STATEMENTS.

Formatted input statements are always associated with a FORMAT statement, an array containing FORMAT specifications, or a NAMELIST.

The general form is:

1.	READ f,k
2.	READ(u,f) k
3.	READ(u,f,l) k
4.	READ(u=r,f) k
5.	READ(u=r,f,l) k

In all five forms, the input list may be empty (i.e., blank).

When the first form is used, the input will be assumed to be from a card, tape file labeled READER, or a terminal unit (see time-sharing appendix J).

When the second or third form is used, input will be assumed to be from a tape file labeled FILEn, where n is the value of u in the input statement, unless otherwise specified by a FILE card (see appendix B).

When the fourth or fifth form is used, then input must be from a random disk file. In this instance, a FILE card must be used.

In using the fourth or fifth form, the random record number r, when evaluated, must have a non-negative integer value (see random disk I/O, 7-51).

Examples:

```
READ 75
READ(8,BID),((I,J,A(I,J),J=6,9),I=1,5)
READ(UNIT,75)X,Z,A
READ(14,LISTA)
READ(6=5*X-3,25,END=101,ERR=77) ARRAY
```

For further information, see I/O lists, implied DO loop, action labels, FORMAT statement, NAMELIST statement, and tape and disk I/O in this section; FILE cards in appendix B.

UNFORMATTED INPUT STATEMENTS.

Unformatted input statements do not have a format specifier associated with them. Input must be from a tape or disk file which has been created with an unformatted output statement.

The general form is:

1.	READ(u) k
2.	READ(u,l) k
3.	READ(u=r) k
4.	READ(u=r,l) k

In all four forms, the input list k may be empty (i.e., blank).

If the list k is not specified, than a record is skipped.

The file used for input must have been previously created with a similar unformatted output statement if a list part is present.

When either of the first two forms is used, input must be from a tape or serial disk file.

When either of the last two forms is used, input must be from a random disk file (see random disk I/O, page 7-51).

Examples:

```
        READ(9) I,A,J,B,D
        READ(2*U,ERR=37) SAM
        READ(UNIT=10,END=99) FEAT,HAMER
```

For further information, see I/O lists, implied DO loop, action labels, tape and disk I/O in this section; FILE cards in appendix B.

OUTPUT STATEMENTS.

In explanations following, the symbols u, r, f, and k have the same meanings as outlined under input statements.

Execution of any of the output statements causes the next record in the output file to be created. The information is converted and positioned on output as specified by the format specifier f if the statement is a formatted output statement. If the list is not specified, either a record is skipped or data contained in the locations in storage occupied by the FORMAT statement is outputted.

FORMATTED OUTPUT STATEMENTS.

Formatted output statements are always associated with a FORMAT statement, an array containing FORMAT specifications, or a NAMELIST.

The general form is:

1.	PRINT f,k
2.	PUNCH f,k
3.	WRITE(u,f) k
4.	WRITE(u=r,f) k

In all four forms, the output list k may be empty (i.e., blank).

If the first form is used, output will be to a line printer file labeled PRINT or a terminal unit (see time-sharing appendix J).

If the second form is used, output will be to a card punch file labeled PUNCH.

When the third form is used, output will be to a tape which will be labeled FILEn, where n is the value of u in the output statement, unless otherwise specified by a FILE card.

When the fourth form is used, output should be to a random disk file. In this instance, a FILE card must be used.

In using the fourth form, the random record number r, when evaluated, must have a non-negative integer value (see random disk I/O, page 7-51).

Examples:

```
PRINT 95,V(J),K(4),ZEDD
PUNCH 55
WRITE(NO,A) ROW
WRITE(3=200-R,68) MATRIX
```

For further information, see I/O lists, implied DO loop, FORMAT statement, tape and disk I/O, and random disk I/O in section 7; FILE cards in appendix B.

UNFORMATTED OUTPUT STATEMENTS.

Unformatted output statements do not have a format specifier associated with them. Output must be to a tape or disk file.

The general form is:

1. WRITE(u) k
2. WRITE(u=r) k

In both forms, the output list k can't be empty (i.e., blank). See page 7-3.

When the first form is used, output must be to a tape or serial disk file.

When the second form is used, then output must be to a random disk file (see random disk I/O, page 7-51).

Examples:

```
WRITE(OUT) (X(K),K=I,J),XX
WRITE(11=REC) BOOL
```

For further information, see I/O lists, implied DO loop, tape and disk I/O, and random disk I/O in section 7; FILE cards in appendix B.

I/O LISTS.

An input list k in an input statement specifies the variables to which values are assigned on input. An output list k specifies the variables whose values are transmitted on output. The input and output lists are of the same form.

The general form is:

$k_1,k_2,\ldots,k_n$
where $k_1,k_2,\ldots,k_n$ are variables, array names, or implied DO loops, or any combination thereof.

An element k_i of an I/O list may be a simple variable, a subscripted variable, an array name without subscripts, or an implied DO loop. An I/O list k may consist of any combination of these elements.

An array name without subscripts in an I/O list is equivalent to inputting or outputting the entire array in the same order in which the elements are stored in memory, i.e., column-wise: with the left-most subscripts varying most rapidly.

Examples:

```
I,J,A,KP,B(I)
(A(INDEX),LP,INDEX=1,20),ZIP,ZAP
```

In addition, see implied DO loop and FORMAT statement in section 7.

IMPLIED DO LOOP.

An implied DO loop is used as an element in an I/O list to specify a repeated cycle of list elements.

The general form is:

1. $(L,i=n_1,n_2,n_3)$
2. $((L,i=n_1,n_2,n_3),j=m_1,m_2,m_3)$
where L is a list of I/O elements which may contain an implied DO loop, and i,n_1,n_2,n_3 and their counterparts j,m_1,m_2,m_3 are as defined for the DO statement.

Example:

```
PRINT 35,((I,B(I,J),I=1,3),J=6,7)
```

The output for the above statement would take the following form:

```
1  B(1,6)
2  B(2,6)
3  B(3,6)
1  B(1,7)
2  B(2,7)
3  B(3,7)
```

where the subscripted B's represent the values of those elements.

For further information, see DO statement, section 5; I/O list, section 7.

ACTION LABELS.

The formatted and unformatted input statements can be extended to programmatically recover from either End-of-File conditions or non-recoverable parity conditions, or both, through use of action labels.

The general form is:

1. $ERR=n_1$
2. $END=n_2$
3. $ERR=n_1,END=n_2$
4. $END=n_2,ERR=n_1$
where n_1 and n_2 are statement labels.

When an attempt is made to read a record which has a parity error from which the operating system cannot recover, control will be transferred to the statement labeled n_1.

When an attempt is made to read an End-of-File, control will be transferred to the statement labeled n_2.

The program will be terminated immediately by the operating system if either of the above conditions occurs and the associated label is not specified in the input statement being executed.

An End-of-File condition can occur under the following circumstances:

a. Attempting to read a card with an invalid character in column one.

b. Attempting to read an End-of-File record on tape.

c. Attempting to read a record from an area of disk which has not been written.

d. Attempting to read a record beyond the furthest record written on disk.

Examples:

```
READ(3,END=99)  (See page 7-3).
READ(6=R,35,ERR=70) A
READ(11,85,END=77,ERR=78) J,S,V
```

For further information, see input statements and tape and disk I/O, section 7.

AUXILIARY I/O STATEMENTS.

There are six types of auxiliary I/O statements:

a. REWIND statement.

b. BACKSPACE statement.

c. ENDFILE statement.

d. CLOSE statement.

e. LOCK statement.

f. PURGE statement.

REWIND STATEMENT.

The REWIND statement causes the pointer for the specified tape or disk file to be reset to the beginning of the file or, in the case of multi-file tape, to Beginning-of-Tape.

The general form is:

```
REWIND u
```

Execution of the REWIND statement causes the file u to be positioned to its initial point.

If the last reference to the file u was a WRITE statement, then an End-of-File record is written prior to positioning the file to its initial point.

The REWIND statement is undefined for other than tape or disk files.

Examples:

```
REWIND 5

REWIND UNIT
```

BACKSPACE STATEMENT

The BACKSPACE statement causes the file pointer to be returned to the preceding program record. For example: If the pointer in file u were positioned at record n, the execution of this statement would position the pointer to record (n-1).

The general form is:

```
BACKSPACE u
```

If the last reference prior to a BACKSPACE or REWIND instruction to a file is a WRITE statement, the file cannot be read beyond the record associated with the last WRITE statement.

Examples:

```
        BACKSPACE 8
        BACKSPACE N
```

The execution of this statement has no effect on the program when file u is positioned at it's initial point.

ENDFILE STATEMENT.

The ENDFILE statement causes an End-of-File record to be written on the specified file and the file to be closed.

The general form is:

```
ENDFILE u
```

The ENDFILE statement is undefined for anything other than a tape file.

When an ENDFILE statement follows a WRITE statement on the same file u, then an End-of-File record is written and the tape is positioned such that the next record written will follow the End-of-File record.

When an ENDFILE statement follows a READ statement on the same file u, then the tape is positioned to the beginning of the next file on the tape.

When an ENDFILE statement follows a BACKSPACE statement on the same file u, then the tape is positioned to the beginning of the file u.

When an ENDFILE statement follows REWIND or another ENDFILE statement on the same file u, then the ENDFILE statement is ignored.

Examples:

```
        ENDFILE TF1
        ENDFILE 7
```

CLOSE STATEMENT.

The CLOSE statement causes the referenced file to be closed.

The general form is:

```
CLOSE u
```

On a card output file, a card containing an ending label is punched, and the card punch is released to the system.

On a line printer file, the printer is skipped to channel 1, an ending label is printed, the printer is again skipped to channel 1, and the printer is released to the system.

On a labeled tape output file, a tape mark and ending label are written after the last block on tape, and the tape is released to the system.

On an unlabeled tape output file, a tape mark is written after the last block on tape, and the tape is released to the system.

Examples:

```
CLOSE 19

CLOSE N
```

LOCK STATEMENT.

The LOCK statement causes the referenced file to be closed.

The general form is:

```
LOCK u
```

If the file is tape, then it is rewound and a system message is written to notify the operator to remove the reel and save it.

If the file is not a disk file, then the unit is made inaccessible to the system until the operator resets it manually.

Examples:

```
LOCK NUT

LOCK 7
```

PURGE STATEMENT.

The PURGE statement causes the referenced file to be closed, purged, and released to the system.

The general form is:

PURGE u

Examples:

```
PURGE TAPE

PURGE 8
```

FORMAT STATEMENT.

The FORMAT statement specifies what type of conversion is to be performed on data from external representation to internal machine representation or vice-versa.

The general form is:

n FORMAT(f_1,f_2,...,f_n)
where n is a statement label and f_1,f_2,...,f_n are format specifications.

The FORMAT statement is non-executable.

The FORMAT statement is always associated with one or more formatted input and/or output statements.

The commas separating the format specifications may be replaced with one or more slashes. However, slashes in a FORMAT statement are used for record control.

Each FORMAT specification must agree in type with the corresponding variable in the list of the associated I/O statement.

When inputting data under a numeric format specification (I, F, E, D, G, O), leading blanks are not significant and imbedded blanks are interpreted as zeros.

Plus signs are optional on input and may be omitted.

When inputting data under a real format specification (F, E, G, D), a decimal point appearing in the input field overrides the decimal point placement specified.

Any entire blank fields read in under a numeric format specification (I, F, E, D, G) which are outputted with no action being performed on them between inputting and outputting will appear in the output field as negative zeros.

In the following FORMAT discussions, the symbols w, d, b, and s will have these meanings:

w - total input or output field width, a positive unsigned integer.

d.- number of decimal places, a non-negative unsigned integer.

b - blank.

s - a string of any valid FORTRAN characters.

INTEGER CONVERSION ON INPUT USING Iw.
The integer format specification Iw on input causes the value of the integer datum in the input field to be assigned to the corresponding integer variable in the input list.

The general form is:

$$\boxed{\text{Iw}}$$

The integer datum must be in the form of an integer constant right-justified in the input field.

Examples:

Input Field	Specification	Internal Value
567	I3	+567
bb-329	I6	-329
-bbbb27	I7	-27
27bbb	I5	+27000
b-bb234	I6	-234

INTEGER CONVERSION ON OUTPUT USING Iw.

The integer format specification Iw on output causes the value of the corresponding integer variable in the output list to be written on the specified output file.

The general form is:

$\boxed{\text{Iw}}$

The integer is placed right-justified in the output field over a field of blanks.

The plus sign is omitted for positive numbers.

If the size of the integer exceeds the specified field width w, the output field will be filled with asterisks.

Examples:

Internal Value	Specification	Output Field
+23	I4	bb23
-79	I4	b-79
+67486	I5	67486
-67486	I5	*****
+978	I1	*
0	I3	bb0

REAL CONVERSION ON INPUT USING Fw.d.

The real format specification Fw.d on input causes the value of the real datum in the input field to be assigned to the corresponding real variable in the input list.

The general form is:

$$\boxed{\text{Fw.d}}$$

If there is no decimal point in the input field, then a decimal point is inserted d places from the right side. Embedded or trailing blank columns are interpreted as zero.

The field width w must be greater than or equal to the specified number of decimal places d. An input datum optionally may have an exponent (see real conversion on input using Ew.d).

Examples:

Input Field	Specification	Internal Value
36725931	F8.4	+3672.5931
3.672593	F8.4	+3.672593
-367259	F8.4	-367259
-3672.E2	F8.4	-367200
367259E2	F8.4	+3672.59
3.672E-1	F8.4	+3672
367259	F6.6	+0.367259
b-b3456	F7.2	-34.56
b2032b	F6.0	+20320

REAL CONVERSION ON OUTPUT USING Fw.d.

The real format specification Fw.d on output causes the value of the corresponding real variable in the output list to be written on the specified output file.

The general form is:

$$\boxed{\text{Fw.d}}$$

The real number is placed, right-justified and rounded to d decimal places, in the output field over a field of blanks.

The plus sign is omitted for positive numbers.

If the size of the number exceeds the specified field width w, then the output field will be filled with asterisks. A safe rule to use is:

$$(w - d) \geq 3$$

Examples:

Internal Value	Specification	Output Field
+36.7929	F7.3	b36.793
+36.7934	F9.3	bbb36.793
-0.0316	F6.3	-0.032
0.0	F6.4	0.0000
0.0	F6.2	bb0.00
+579.645	F4.2	****
+579.645	F6.2	579.65
-579.645	F6.2	******

REAL CONVERSION ON INPUT USING Ew.d.

The real format specification Ew.d on input causes the value of the real datum in the input field to be assigned to the corresponding real variable in the input list.

The general form is:

$$\boxed{\text{Ew.d}}$$

If there is no decimal point in the input field, then a decimal point is inserted d places from either the right side of the input field or from the E denoting the exponent, if there is one.

The field width w must be greater than or equal to the specified number of decimal places d.

An input datum may or may not have an exponent.

Embedded or trailing blank columns are interpreted as zero, and can cause problems. When blank columns are located on the right end of a field they are interpreted as zero and large errors can occur. For example: If the value $4.527x10^4$ were punched as 4.527E4b, it would be stored internally as $4.527x10^{40}$.

It is advisable to always punch E-field exponents as far to the right as possible.

Examples:

Input Field	Specification	Internal Value
bbbbbb25046	E11.4	+2.5046
bbbbb25.046	E11.4	+25.046
-bb25046E-3	E11.4	-0.0025046
bb250.46E-3	E11.4	+0.25046
b-b25.04678	E11.4	-25.04678
bbb4.527E1b	E11.4	+45270000000.

REAL CONVERSION ON OUTPUT USING Ew.d.

The real format specification Ew.d on output causes the value of the corresponding real variable in the output list to be written on the specified output file.

The general form is:

$$\boxed{\text{Ew.d}}$$

The real number is placed right-justified and rounded to a d-digit mantissa, together with a four-place exponent field, in the output field over a field of blanks. Note that with the Ew.d format specification, d takes on a slightly different interpretation since no significant digits are written to the left of the decimal point in the output field. The plus sign is omitted for positive numbers. If the following rule is violated, the output field will be filled with asterisks:

$$(w - d) \geq 6$$

If a scale factor n is used, then it will control the decimal normalization between the number part and the exponent part as follows:

a. If $n \leq 0$, then $|n|$ zeros will be placed immediately to the right of the decimal point with $(d-|n|)$ significant digits following the zeros.

b. If $n \geq 0$, then n significant digits will be placed to the left of the decimal point and $(d-n+1)$ significant digits will be placed to the right of the decimal point.

Examples:

Internal Value	Specification	Output Field
+36.7929	E12.5	bb.36793Eb02
-36.7929	E11.5	-.36793Eb02
-36.7924	E10.5	**********
+36.7929	-2PE12.5	bb.00368Eb04
+36.7929	+2PE12.5	36.79290Eb00

DOUBLE PRECISION CONVERSION ON INPUT USING Dw.d.

The double precision format specification Dw.d on input causes the value of the real datum in the output field to be assigned to the corresponding variable of type DOUBLE PRECISION in the input list.

The general form is:

```
Dw.d
```

Aside from the fact that a double precision value is stored in two words, and that the exponent in the input field is preceded by a D rather than an E, the double precision format specification Dw.d behaves in the same manner as Ew.d.

DOUBLE PRECISION CONVERSION ON OUTPUT USING Dw.d.

The double precision format specification Dw.d on output causes the value of the corresponding double precision variable in the output list to be written on the specified output file.

The general form is:

```
Dw.d
```

The double precision format specification Dw.d is identical to Ew.d, with the following exceptions:

a. The value associated with it is stored in two machine words.

b. The variable name associated with the value must be of type DOUBLE PRECISION.

c. The exponent part of the output contains a D rather than an E.

REAL CONVERSION ON INPUT USING Gw.d.

The real format specification Gw.d on input is identical to Fw.d.

The general form is:

$$\boxed{\text{Gw.d}}$$

REAL CONVERSION ON OUTPUT USING Gw.d.

The real format specification Gw.d on output causes the value of the corresponding real variable in the output list to be written on the specified output file.

The general form is:

$$\boxed{\text{Gw.d}}$$

The representation in the output field is a fraction of the magnitude of the real number being outputted.

If N is the magnitude of the number being outputted, then table 7-1 shows how the number will appear in the output field.

If a scale factor is used, then it will have no effect on output conversion unless the magnitude of the number being written is outside the range which permits effective use of F conversion.

Examples:

Internal Value	Specification	Output Field
+10.	G12.5	bb10.000
+1000.	G12.5	bb1000.0
+100000.	G12.5	bb.10000Eb06
+1000000.	G12.5	bb.10000Eb07

Table 7-1

Datum Conversion

Magnitude of Datum	Equivalent Conversion Effected
$0.1 \le N < 1$	F(w-4). d, 4X
$1 \le N < 10$	F(w-4). (d-1), 4X
.	.
.	.
.	.
$10^{d-2} \le 10^{d-1}$	F(w-4). 1, 4X
$10^{d-1} \le 10^{d}$	F(w-4). 0, 4X
Otherwise	Ew.d

OCTAL CONVERSION ON INPUT USING Ow.

The octal format specification Ow on input causes the value of the octal datum in the input field to be assigned to the corresponding variable in the input list.

The general form is:

$$\boxed{\text{Ow}}$$

If the datum is less than 16 octal digits long, then it is right-justified and stored in a machine word. The maximum octal constant which can be read is 3777777777777777.

Octal constants can be signed or unsigned. The plus sign (+) and the ampersand (&) sign are skipped over and the minus (-) sign causes bit 1 of the input element to be complemented with 1. Leading, imbedded, and trailing blanks are treated as zeros. An execution time type error is emitted if the sign occurs more than once or occurs imbedded within the octal constants, or if the magnitude of the octal constant exceeds Ø3777777777777777.

Examples:

Input Field	Specification	Internal Value
16	Ø2	0000000000000016
1777777777777777	Ø16	1777777777777777
-16	Ø3	2000000000000016

OCTAL CONVERSION ON OUTPUT USING Ow.

The octal format specification Ow on output causes the octal value of the corresponding variable in the output list to be written on the specified output file.

The general form is:

$$\boxed{\text{Ow}}$$

The octal value is placed right-justified in the output field over a field of blanks.

Examples:

Internal Value	Specification	Output Field
0000376754320017	Ø3	017
0000376754320017	Ø10	6754320017 Truncated
0000376754320017	Ø16	0000376754320017

LOGICAL CONVERSION ON INPUT USING Lw.

The logical format specification Lw on input causes the value of the logical datum in the input field to be assigned to the corresponding variable of type LOGICAL in the input list.

The general form is:

$$\boxed{\text{Lw}}$$

The input field width w must be greater than or equal to one. There may be leading blanks. Normally, the first character encountered in the field exclusive of leading blanks is either T or F, for true or false, respectively. Any characters following the T or F will be ignored. If the first non-blank character is a T, then the variable will be assigned a value of true, otherwise it is false.

Examples:

Input Field	Specification	Internal Value
T	L1	TRUE
bbF	L3	FALSE
bbbTRU	L6	TRUE

LOGICAL CONVERSION ON OUTPUT USING Lw.

The logical format specification Lw on output causes the logical value of the corresponding variable of type LOGICAL in the output list to be written on the specified output file.

The general form is:

$$\boxed{\text{Lw}}$$

The logical value is placed right-justified in the output field over a field of blanks as a T or F, for true or false, respectively.

Examples:

Internal Value	Specification	Output Field
FALSE	L1	F
FALSE	L3	bbF
TRUE	L2	bT

ALPHANUMERIC CONVERSION ON INPUT USING Aw.

The alphanumeric format specification Aw on input causes the character string of width w in the input field to be assigned to the corresponding variable in the input list.

The general form is:

$$\boxed{\text{Aw}}$$

The variable may be real or integer. The field width w should never exceed six. If it does, then the right-most six characters in the string are stored and the rest are ignored. If w is less than six, then the string is stored left-justified with (6-w) trailing blanks.

Examples:

Input Field	Specification	Internal Value
ABCDEFGHIJK	A3	ABCbbb
ABCDEFGHIJK	A6	ABCDEF
ABCDEFGHIJK	A11	FGHIJK

ALPHANUMERIC CONVERSION ON OUTPUT USING Aw.

The alphanumeric format specification Aw on output causes the character string assigned to the corresponding variable in the output list to be written on the specified output file.

The general form is:

$$\boxed{\text{Aw}}$$

The string is placed right-justified in the output field over a field of blanks.

Examples:

Internal Value	Specification	Output Field
ABCbbb	A3	ABC
ABCbbb	A5	ABCbb
ABCbbb	A9	bbbABCbbb

INPUTTING A CHARACTER STRING USING wHs.

The Hollerith field specification wHs on input causes the character string of width w in the input field to replace the character string s of the Hollerith field specification in a FORMAT statement.

The general form is:

$$\boxed{\text{wHs}}$$

The Hollerith field specification on input may be used to read in page headings which are to be printed on output, but which may vary in content from one run to another.

Example:

```
              READ 15
           15 FORMAT(2X,9HDUMMYbbbb)
              PRINT 15
                                  1 1
    Input:    1 2 3 4 5 6 7 8 9 0 1   (card column)
              X Y b A b S A M P L E

    Output:   b b A b S A M P L E
```

Note that in the printed output, although 2X has been specified, only one blank is printed since the first blank is a carriage control character (see carriage control, page 7-28).

OUTPUTTING A CHARACTER STRING USING wHs.

The Hollerith field specification wHs on output causes the character string s of width w of the Hollerith field in a FORMAT statement to be written on the specified output file.

The general form is:

$$\boxed{\text{wHs}}$$

The string s remains unchanged.

Example:

```
              PUNCH
           95 FORMAT(12HbBURROUGHSbb)
                                  1 1 1
    Output:   1 2 3 4 5 6 7 8 9 0 1 2   (card column)
              b B U R R O U G H S b b
```

INPUTTING A CHARACTER STRING USING "s".

The literal string specification "s" on input is identical in operation to the Hollerith field specification wHs.

The general form is:

"s"

Example:

```
             READ 15
          15 FORMAT(2X,"DUMMYbbbb")
             PRINT 15

                                 1 1
   Input:    1 2 3 4 5 6 7 8 9 0 1  (card column)
             X Y b A b S A M P L E

   Output:   b b A b S A M P L E
```

OUTPUTTING A CHARACTER STRING USING "s".

The literal string specification "s" on output is identical (except for ",!,@) in operation to the Hollerith field specification wHs.

The general form is:

"s"

SKIPPING CHARACTERS USING nX.

The format editing specification nX on input or on output will cause n characters to be skipped in the respective input or output field.

The general form is:

nX

EDITING USING Tn.

The format editing specification Tn is used to transfer data to or from a specified position n within a record. The use of Tn in a format list will cause the next item of data transferred in the corresponding I/O list to be transferred to or from the position indicated by the letter n.

Example:

```
     WRITE (6,1) A,B,C
   1 FORMAT (F4.1, T10, F7.2, T25, E12.4)
```

In the above example, the data referenced by the variable B will be positioned in the tenth position of the record when it is written, and the data referenced by the variable C will be positioned in the 25th position of the record.

The general form is:

Tn

SCALE FACTOR nP.

A scale factor is defined for use with the F, E, G, and D format specifications, and is of the form:

nP

where the scale factor n is a signed integer constant.

When FORMAT control is initiated, a scale factor of zero is automatically established and applies until a scale factor is encountered in the FORMAT statement. Once a scale factor is encountered, it applies to all subsequently encountered F, E, G, and D conversions until another different scale factor or the end of the FORMAT statement is encountered.

SCALE FACTOR ON INPUT.

For F, E, G, and D format specifications on input, where the input datum does not have an exponent, the input datum is multiplied by 10^{-n}, where n is the scale factor. For example, the datum 573.19

read with a format of 2PF6.2 would be stored internally as 5.7319. If the input datum contains an exponent, the scale factor has no effect.

SCALE FACTOR ON OUTPUT.

For F, E, and D format specification on output, when the output datum does not have an exponent, the output datum is multiplied by 10^n, where n is the scale factor. For example, the number stored internally as 5.7319 and written with a format of 2PF6.2 would have the external value of 573.19. If the output datum contains an exponent, the datum is multiplied by 10^n and the exponent is reduced by n. Therefore, the value is not changed. For example, the number stored internally as 5.7319E+02 and written with a format of 1PE11.3 would have the external value of 57.319E+01.

For the G format specification on output, the effect of the scale factor is suspended unless the magnitude of the datum being outputted is outside the range that permits effective use of F conversion. If the use of E conversion is required, then the scale factor has the same effect as when using the E format specification on output.

For further information, see real conversion on page 7-16 using Ew.d.

FORMAT SPECIFICATION IN AN ARRAY.

Any of the formatted input/output statements may contain an array name in place of a FORMAT statement label. At the time the input/output statement containing the array reference is executed, the array must contain the equivalent of a FORMAT statement, with the first non-blank character being a left parenthesis. Any characters in the array following the final right parenthesis of the FORMAT statement in the array are ignored.

Example:

```
          DIMENSION FORM (5),INFO(6)
          READ(5,75) FORM
       75 FORMAT(5A6)
```

```
                READ(20,FORM) Q,R,(INFO(I),I=1,6)
                ...
                ...
    Input:      (F6.2,3X,E15.8,6I3)bbbbbbbbbb
```

CARRIAGE CONTROL.

When a line printer is used for output, the first character of each line of print controls the spacing of the printer carriage. The control characters are:

Character	Action
Blank	One space before printing.
Zero	Double space before printing.
1-9	Skip to channel 1-9 of the carriage control tape before printing.
Plus sign	No advance before printing.

Examples:

```
        25 FORMAT(1H0,E12.6,A5)
```

Causes the carriage to double space before printing.

```
        35 FORMAT(6H+TITLE)
```

Provides no carriage advance before printing.

```
        45 FORMAT(3X,6I5)
```

Causes the carriage to single space before printing.

```
        55 FORMAT(1H1,"TITLE")
```

Causes the printer to page eject and print TITLE.

USE OF SLASH (/).

A slash in a FORMAT statement is used to indicate the end of a record. On input, any remaining characters in the current record are ignored when a slash is encountered in the FORMAT statement. On output, the current record is terminated and any subsequent output is placed in the next record. Multiple slashes may be used to skip several records on input or create several blank records on output.

REPEAT SPECIFICATIONS.

Repetition of any format specification except nX, wHs, "s", or Tn is accomplished by preceding it with a positive integer constant called the repeat count. If the I/O list warrants it, the specified conversion will be interpreted repetitively up to the specified number of times. If a scale factor is included, then it must precede the repeat count.

Repetition of a group of format specifications is accomplished by enclosing them within parentheses and preceding the left parenthesis with a positive integer constant called the group repeat count, which indicates the number of times to interpret the enclosed groupings. If a group repeat count is not given, then the group is repeated until the I/O list is exhausted. Grouping with parentheses may be continued to any desired level.

Example:

```
        85 FORMAT(3E16.6,5(F10.5,I3,4A2))
```

FORMAT AND I/O LIST INTERACTION.

The execution of a formatted I/O statement initiates format control. If there is an I/O list, then at least one format specification other than wHs, "s", nX, or Tn must exist in the FORMAT statement referenced.

When a formatted input statement is executed, one record is initially read. No other records are read unless otherwise specified by the FORMAT statement. The I/O list associated with a FORMAT statement may not require more data of a record that it contains.

When a formatted output statement is executed, writing of a new record occurs each time the FORMAT statement referenced so specifies. Terminating execution of a formatted output statement causes the current record to be written. A slash also causes the record to be written.

Except for the effects of repeat counts, the FORMAT statement is interpreted from left to right.

To each I, F, E, G, D, O, A, or L format specification there corresponds one element in the I/O list. A list element of type COMPLEX is considered, for purposes of I/O conversion, as two list elements of type REAL. Thus, there must be two format specifications (or a format specification preceded by a repeat count) for every list element of type COMPLEX.

There is no corresponding I/O list element for any wHs, "s", Tn, or nX format specification. The information indicated by the wHs and "s" is inputted or outputted directly to or from the FORMAT statement.

If, under format control, the right-most right parenthesis of the FORMAT statement is encountered and the I/O list is still not exhausted, then format control reverts to the last previously encountered left parenthesis. If a group repeat count precedes this left parenthesis, then it also takes effect.

If, during execution of a formatted I/O statement, the I/O list is exhausted but the right-most right parenthesis of the specified FORMAT statement has not been encountered, then execution of the I/O statement is complete. This action, of itself, has no effect on the scale factor.

NAMELIST STATEMENT.

The NAMELIST statement associates an I/O list with a unique identifier. This identifier may not be used for any other purpose in the program unit in which it occurs. Only variable and array identifiers may be used as NAMELIST elements. These identifiers may not be formal parameters.

The general form is:

NAMELIST/N1/$a_1,a_2,...,a_n$/N2/$b_1,b_2,...,b_n$
where N1 and N2 are NAMELIST identifiers and $a_1,...,a_n$ and $b_1,...,b_n$ are variable or array names.

INPUT USING NAMELIST.

Input using NAMELIST is accomplished by executing a formatted READ statement which has as its format specifier f a NAMELIST identifier which has previously been declared in the same program unit. No input list k is allowed in the READ statement.

The input data file is free format except for the first two characters of the first record. The first character is ignored and the second must be a dollar sign ($). The NAMELIST identifier designated in the associated READ statement must follow the dollar sign, with one or more blanks following the NAMELIST identifier.

Following the NAMELIST identifier and blank(s) are placed, in free format, the variables assigned to the NAMELIST and the values which are being assigned to them. These may take any one of three forms, or any combination thereof:

a. V = N, where V is a simple or subscripted variable assigned to the NAMELIST identifier, and N is the value being assigned to the variable V.

b. B(i) = m_i, m_{i+1},...,m_n, where B is a previously DIMENSIONed array of size n, i is an integer constant designating an element of the array B (i is less than or equal to n), and m_i,...,m_n are the values being assigned to the array elements B(i) through B(n) and are either constants or are of the form i*c, where i is a repeat count and c is a constant. Values must be assigned to all elements of the array from B(i) through B(n).

c. A = $m_1,m_2,...,m_n$, where A is a previously DIMENSIONed array and $m_1,m_2,...,m_n$ are the values being assigned to the entire array A, and are either constants or are of the form i*c, where i is a repeat count and c is a constant.

If the first record is other than that specified above, then additional records are read until the required record is found or the End-of-File is encountered.

The READ statement will be terminated when a second dollar sign is encountered in the data file. Anything following the dollar sign within the record is ignored. Trailing blanks are interpreted as zeros. An assignment of the form V=E cannot be divided between two data cards. That is, V on one card, and E on another.

Example:

```
                        DIMENSION A(4,4),M(10),N(20)
                        NAMELIST/NAMEA/A,D,K,M,N,X/NAMEB/M,N,X
                        READ NAMEA
                        ...
                        ...
First input card:       12345678.....    (card column)
                        b$NAMEA D=7.1,N(4)=2.9,5.7,1.5,X=2.5,
Second input card:      12345678.....    (card column)
                        A(2,3)=15.9,M=2,1,3*6,4*74$
```

OUTPUT USING NAMELIST.

Output using NAMELIST is accomplished by executing a formatted output statement which has as its format specifier f a NAMELIST identifier which has previously been declared in the same program unit. No output list k is allowed in the output statement.

Output records produced by using NAMELIST may be read by a READ with NAMELIST statement, and are therefore of the same general format as that specified for input to a READ with NAMELIST statement.

Example:

```
        DIMENSION A(4,4),M(10),N(20)
        NAMELIST/NAMEA/A,D,K,M,N,X/NAMEB/M,N,X
        ...
        WRITE(6,NAMEA)
        . . .
        WRITE(6,NAMEB)
        . . .
```

TAPE AND DISK I/O.

Tape and disk file unformatted output statements, BLOCKING and BUFFERing options, serial and random disk I/O information are available in this section of the document.

UNFORMATTED OUTPUT.

When a tape file is written using unformatted output statements, it is formatted in the manner illustrated by figure 7-1.

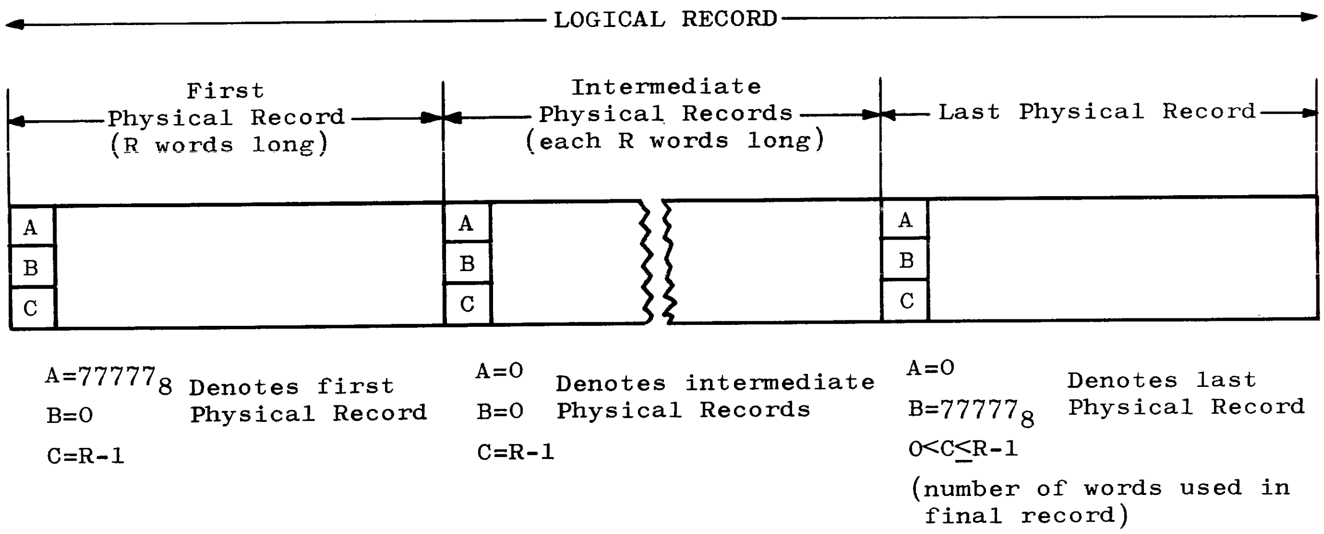

LOGICAL RECORD
First Physical Record (R words long)
Intermediate Physical Records (each R words long)
Last Physical Record
A
B
C
A
B
C
A
B
C
$A=77777_8$ Denotes first Physical Record
B=0
C=R-1
A=0 Denotes intermediate Physical Records
B=0
C=R-1
A=0 Denotes last Physical Record
$B=77777_8$
$0<C\leq R-1$
(number of words used in final record)

Figure 7-1. File Format Using Unformatted Output Statements

NOTE

Numbers with subscript 8 are octal.

In figure 7-1, R represents the number of words per record declared by the RECORD option of the FILE card. The logical record size is determined by the number of items in the I/O list. A, B, and C are partial words that comprise the control, or first word of each physical record.

A = [3:15]
B = [18:15]
C = [33:15]

RECORD CONTROL WORDS. Each physical record in a tape file contains a control word. The control word is always the first word of each record, and is classified into four types.

a. Type 1 - 077777|77777|$XXXXX_8$

   This type of control word indicates that the entire logical record follows the control word in the next $XXXXX_8$ words of the physical record.

b. Type 2 - 077777|00000|$XXXXX_8$

   This control word indicates that the first $XXXXX_8$ words of the logical record are contained in the physical record associated with the control word. The remainder of the logical record is contained in the physical record or records that follow.

c. Type 3 - 000000|00000|$XXXXX_8$

   This control word indicates that the physical record associated with it is an intermediate record and contains $XXXXX_8$ words of the logical record. This means that the first part of the logical record is contained in a preceding physical record or records, and that the remainder is contained in the physical record or records that follow.

d. Type 4 - 000000|77777|XXXXX$_8$

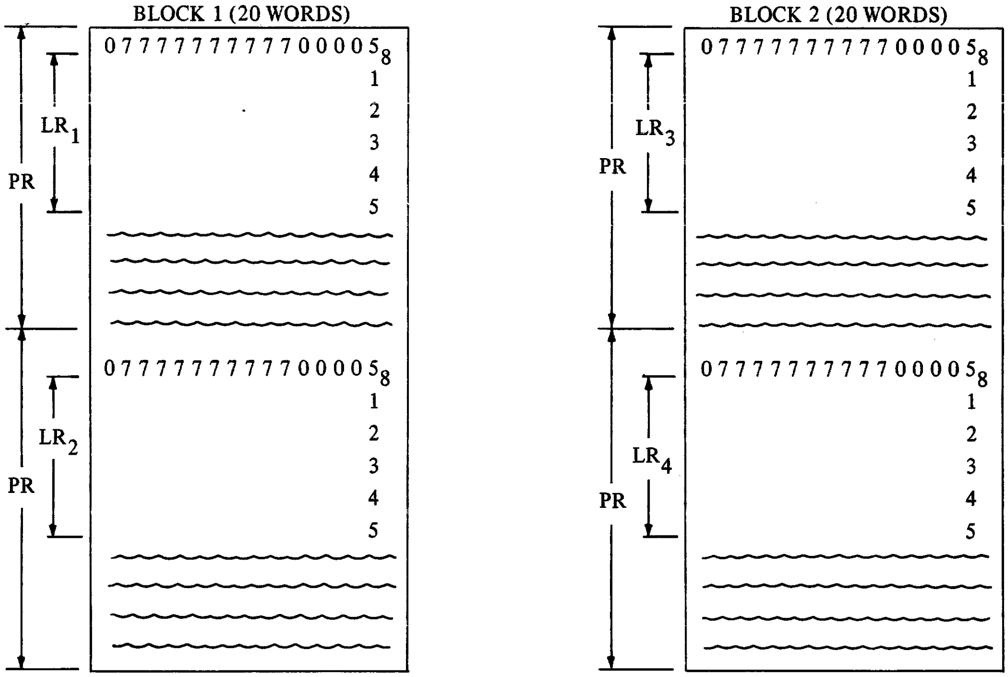
This control word indicates that the associated physical record contains the last XXXXX$_8$ words of the logical record.

Examples of Unformatted Tape Output.

The following examples illustrate unformatted tape output. PR indicates the number of words in a given physical record, and LR the number of words in a given logical record.

Example 1 PR = 10, LR = 5, BLOCKING = 2.

The tape output for a program writing a five word logical record (1, 2,3,4,5) four times is illustrated in the following manner.

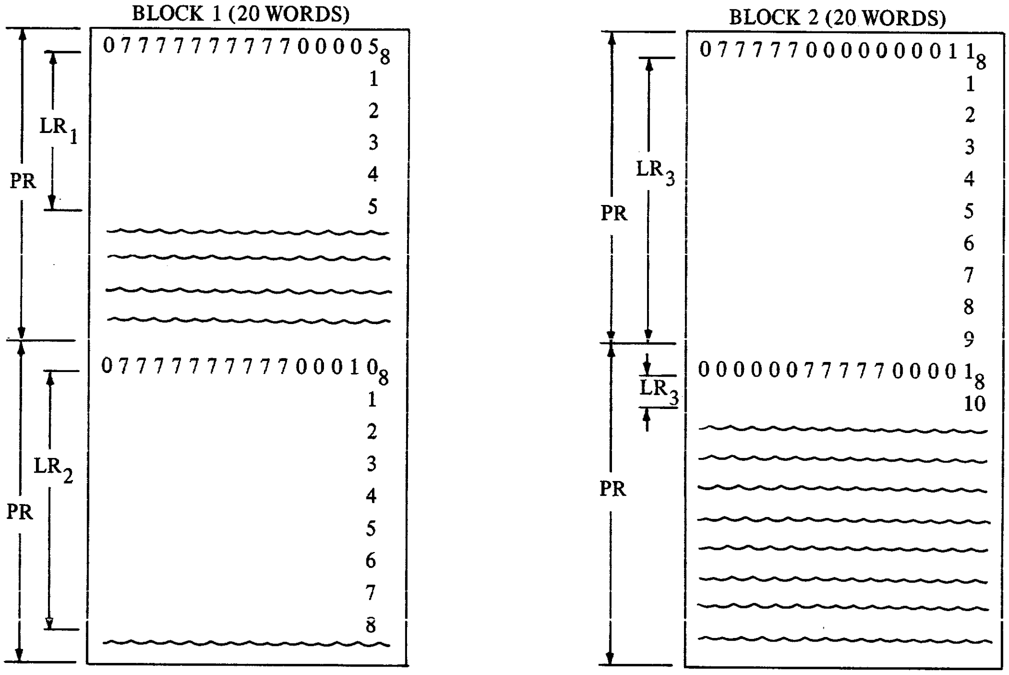


The block size of a tape file is determined by multiplying the BLOCKING factor by the physical record length. The block size in example 1 is 2 x 10 = 20 words.

The BLOCKING factor is the number of physical records in a block. When the size of the logical record is less than the physical record as shown in example 1, then the first LR words after the control word of the physical record comprises the logical record. The remaining words in the physical record are invalid.

<u>Example</u> 2 PR = 10, LR = 5, 8, 10, BLOCKING = 2.

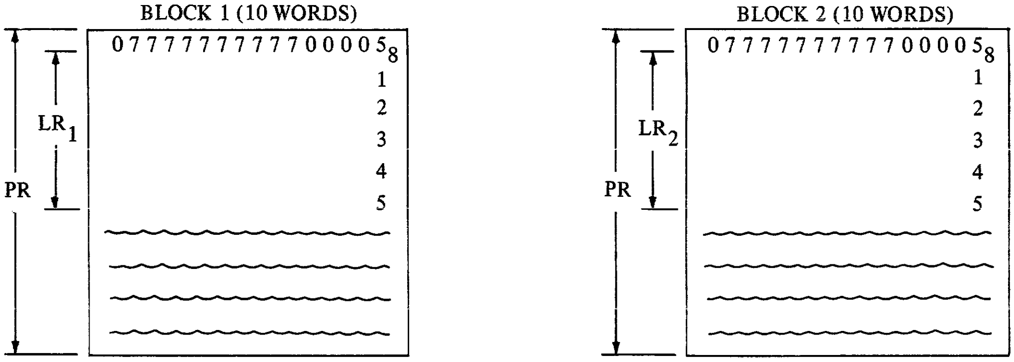
The tape output of a program that writes three variable logical records (1,2,3,4,5), (1,2,...,8), and (1,2,...,10) is illustrated in the following manner.



Note that the control word uses one word of the physical record, and that the ten word logical record (1,2,..., 10) could not be completely contained in the first physical record of block two.

Example 3 PR = 10, LR = 5, UNBLOCKED.

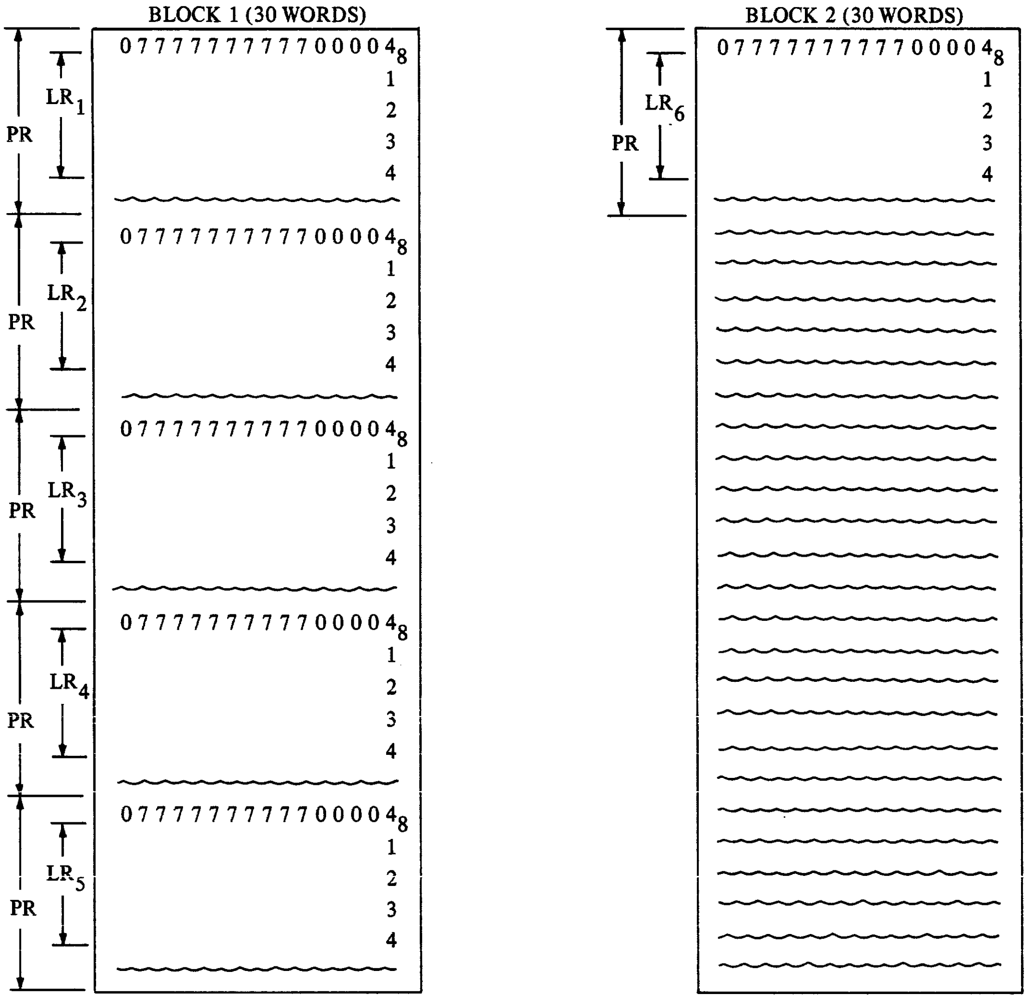
The tape output of a program that writes a five word logical record (1,2,3,4,5) two times to an unblocked tape file is illustrated in the following manner.



An unblocked tape file implies, and is implied by a blocking factor of one.

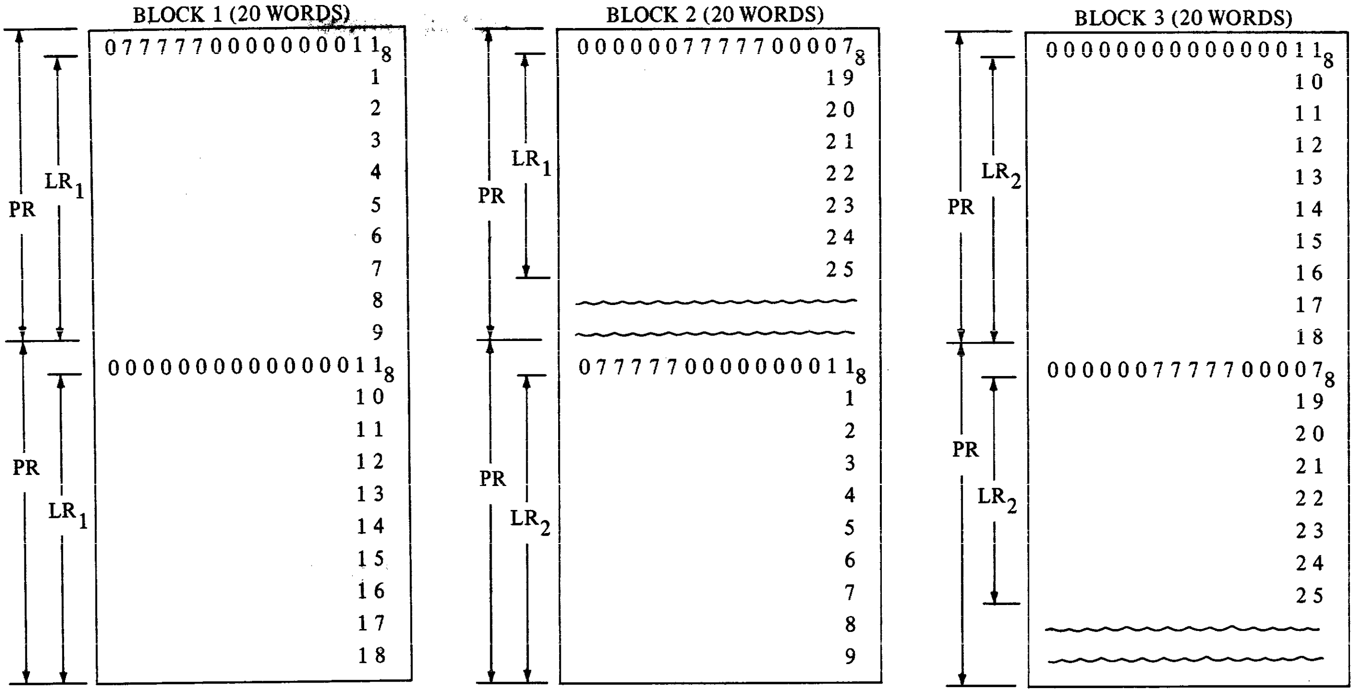
Example 4 PR = 6, LR = 4, BLOCKING = 5.

The tape output for a program that writes a four word logical record (1,2,3,4) six times is illustrated as the following.



<u>Example</u> 5 PR = 10, LR = 25, BLOCKING = 2.

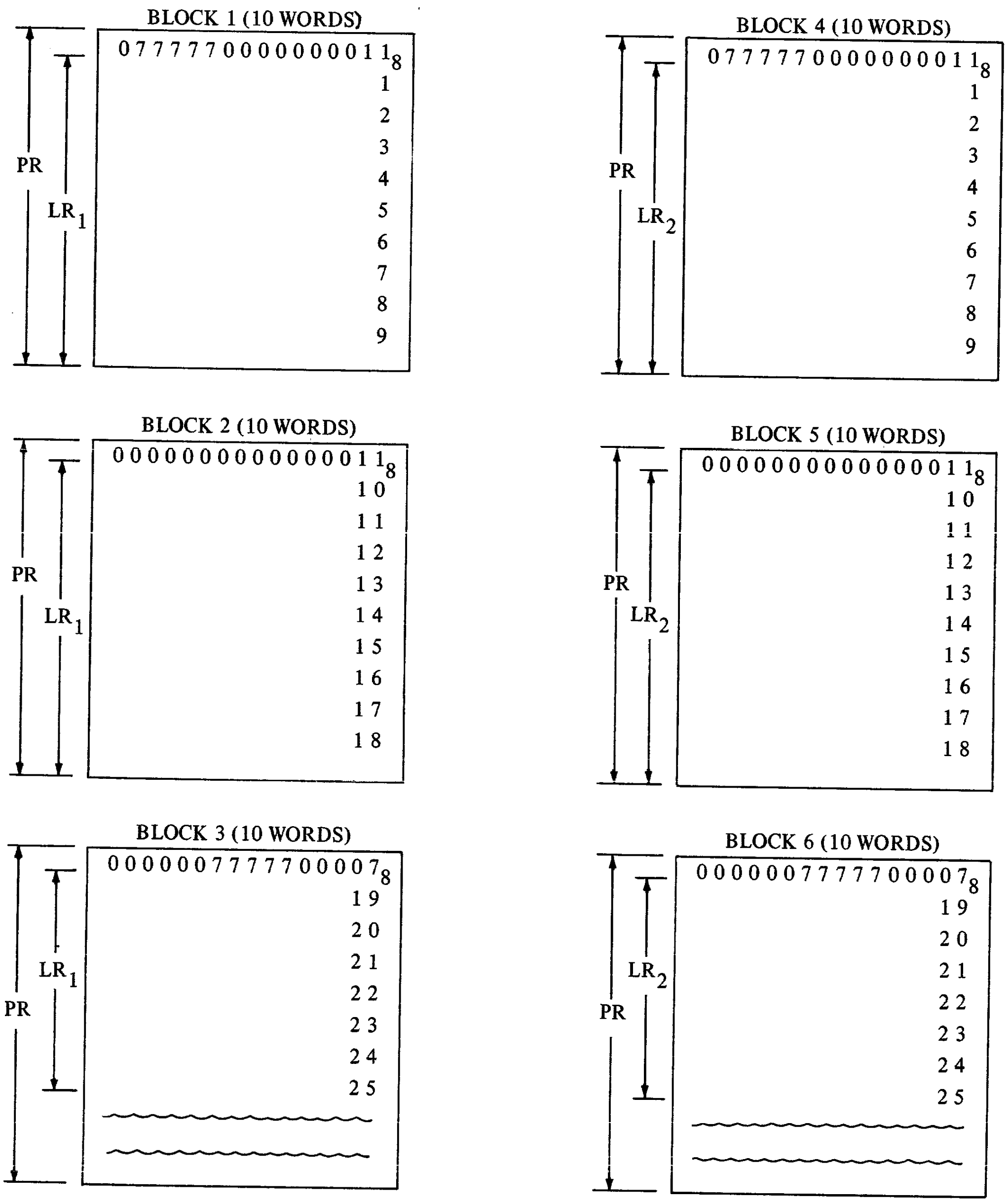
The tape output for a program that writes a 25 word logical record (1, 2, 3, . . . , 25) two times is illustrated in the following manner.



If the logical record is greater than or equal to the physical record, then the first (PR-1) words of the logical record will be contained in the first associated physical record.

Example 6   PR = 10, LR = 25, UNBLOCKED.

The format of the tape output for a program that writes two 25-word logical records (1, 2, 3, . . . ,25) to an unblocked file is illustrated as follows.



UNFORMATTED DISK OUTPUT

The disk is allocated into 30 word segments, and the longest record that can be written to disk is 1023 words.

Efficient use of the disk capability can be achieved by using a blocking factor that will either be 30 words (240 characters), or an integral multiple thereof.

The compiler generates syntax errors for disk files when PR is greater than 1023 words, or the block size (PR x BLOCKING factor) is greater than 1890 words (63 disk segments).

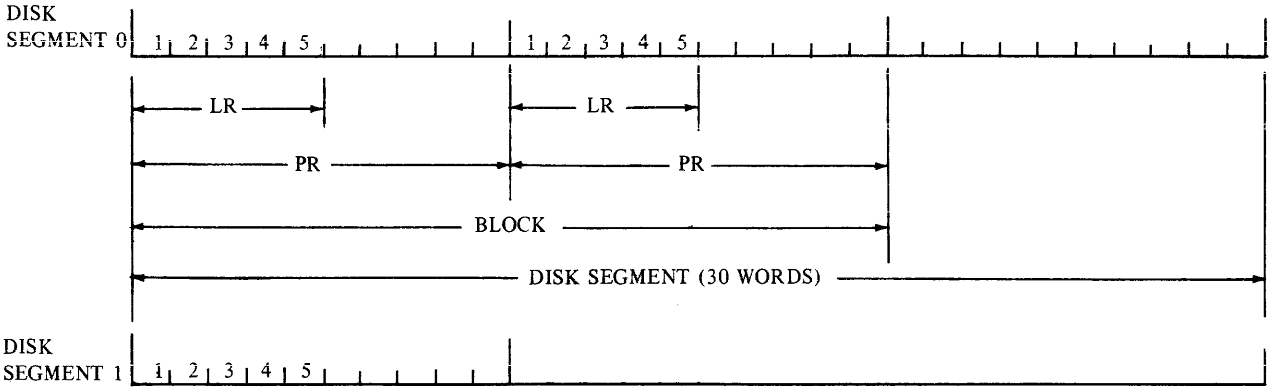
When writing to disk, each block will start at the beginning of the next disk segment.

The size of the logical record is determined by the number of items in the I/O list of the WRITE statement. For example: In the statement WRITE (6) A, B, C, D, E where A, B, C, D, E are simple variables, the logical record size is five.

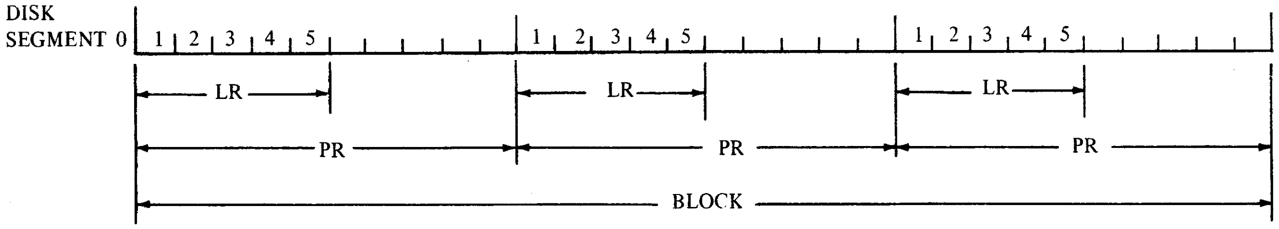
The following examples illustrate some of the various disk unformatted output formats. PR indicates the number of words in a given physical record, and LR the number of words in a given logical record.

Example 1   PR = 10, LR = 5, BLOCKING = 2.

The disk output for a program writing a five word logical record (1, 2, 3, 4, 5) three times is illustrated in the following manner.



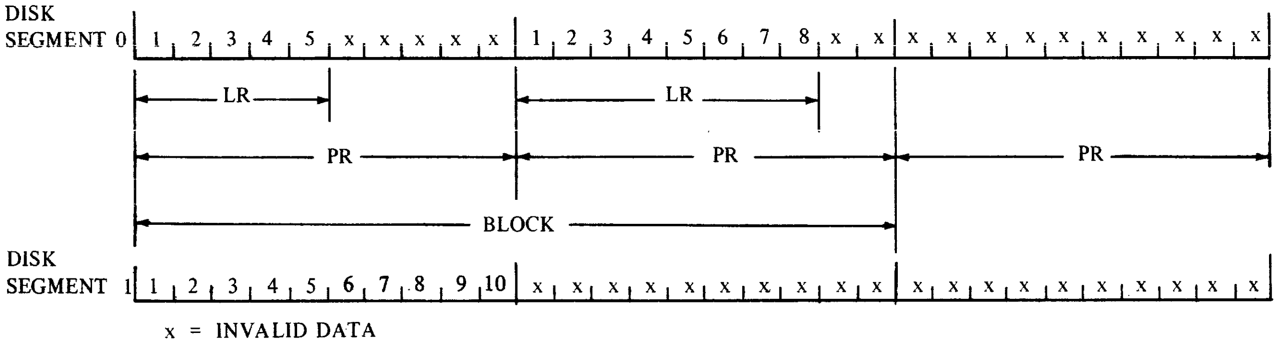
It would be more efficient to use a blocking factor of three.



For maximum efficiency let PR=5, and the BLOCKING factor=6. The substitution of these values will create six 5-word logical records on one disk segment.

Example 2 PR = 10, LR = 5, 8, 10, BLOCKING = 2.

A program that writes three logical records (1, 2, 3, 4, 5), (1, 2, 3, 4, 5, 6, 7, 8) and (1, 2, 3, 4, 5, 6, 7, 8, 9, 10) is illustrated in the following manner.



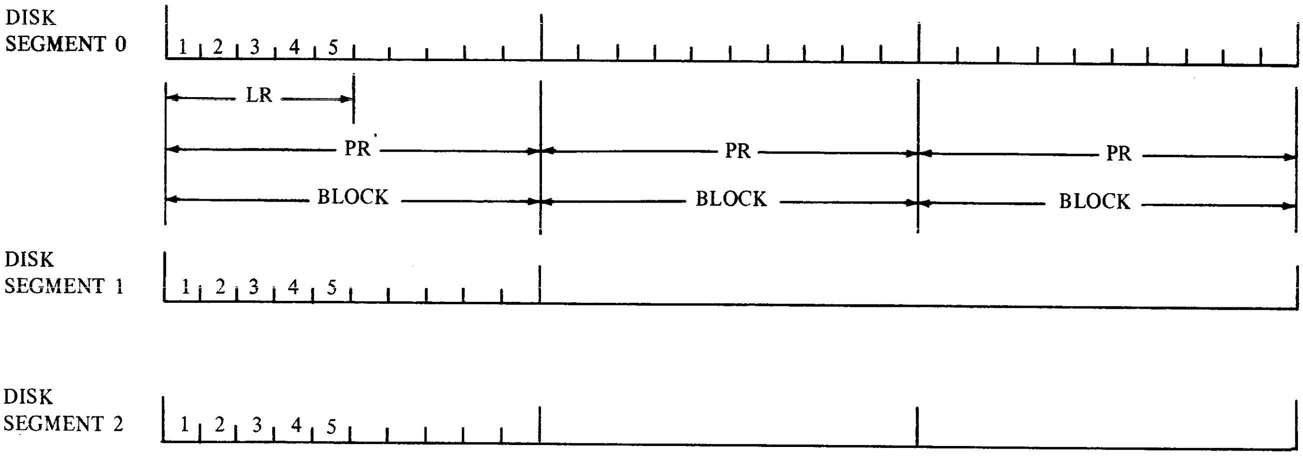
When the disk file is read, both valid and invalid data is processed by the compiler.

The block size of an unblocked disk file is the same size as the physical record; i.e., the blocking factor assumes a value of one.

If the blocking factor is not specified, and the LR less than or equal to the PR, the physical record that contains the logical record is written one per block.

Example 3 PR = 10, LR = 5, UNBLOCKED.

The disk output for a program writing a five word logical record (1, 2, 3, 4, 5) three times is illustrated in the following manner.



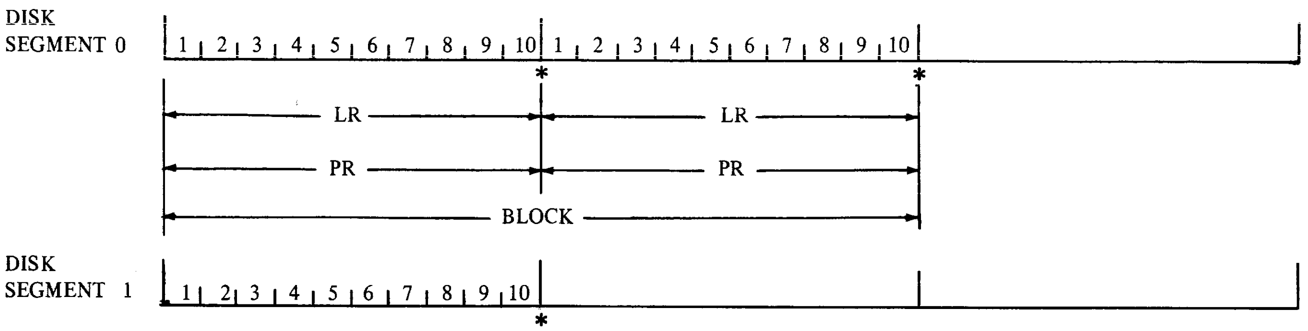
NOTE

Each block starts at a new disk segment.

The length of a logical record cannot exceed the length of a physical record. When this condition exists, the remaining (LR-PR) words of the logical record are lost.

Example 4 PR = 10, LR = 25, BLOCKING = 2.

The disk output for a program writing a 25 word logical record (1, 2, 3, ..., 24, 25) three times is illustrated in the following manner.

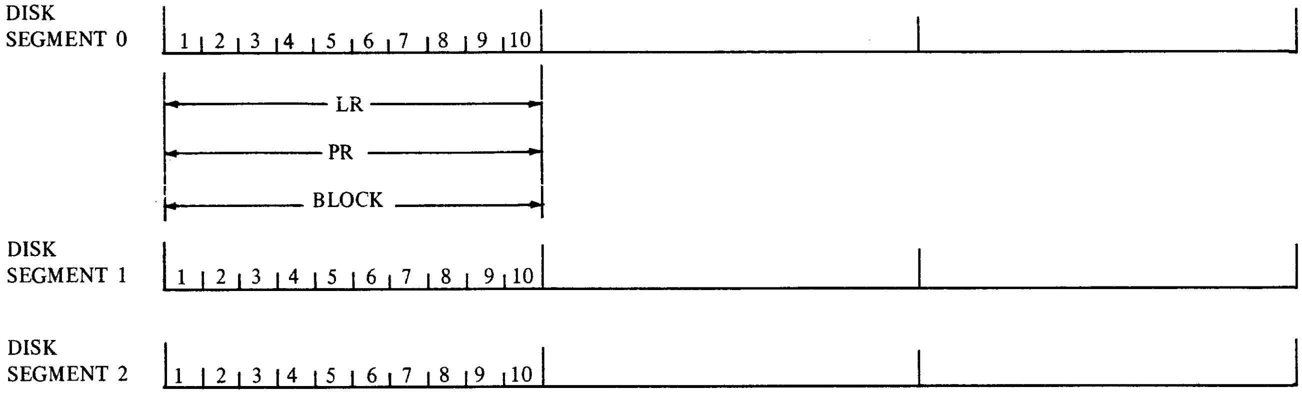


Each logical record is written to the end of the physical record as indicated by the asterisk (*), and records 11 through 25 are lost.

When the blocking factor is not specified, and the LR is greater than the PR, the physical record will contain the first PR words of the logical record. The size of the physical record is the same as the size of the block, and each block starts at the beginning of the next disk segment when the files are unblocked.

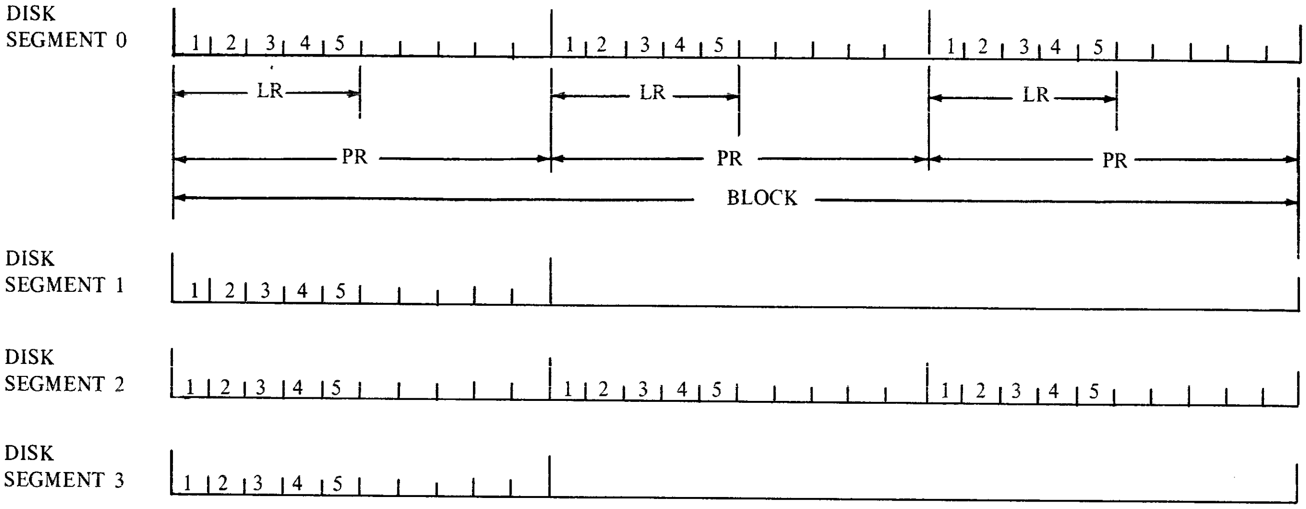
Example 5   PR = 10, LR = 25, UNBLOCKED

The disk output for a program writing an unblocked 25-word logical record (1, 2, 3, . . . , 24, 25) three times is illustrated in the following manner.



Example 6   PR = 10, LR = 5, BLOCKING =4

The disk output for a program that writes a five word logical record (1, 2, 3, 4, 5) eight times is illustrated in the following manner.



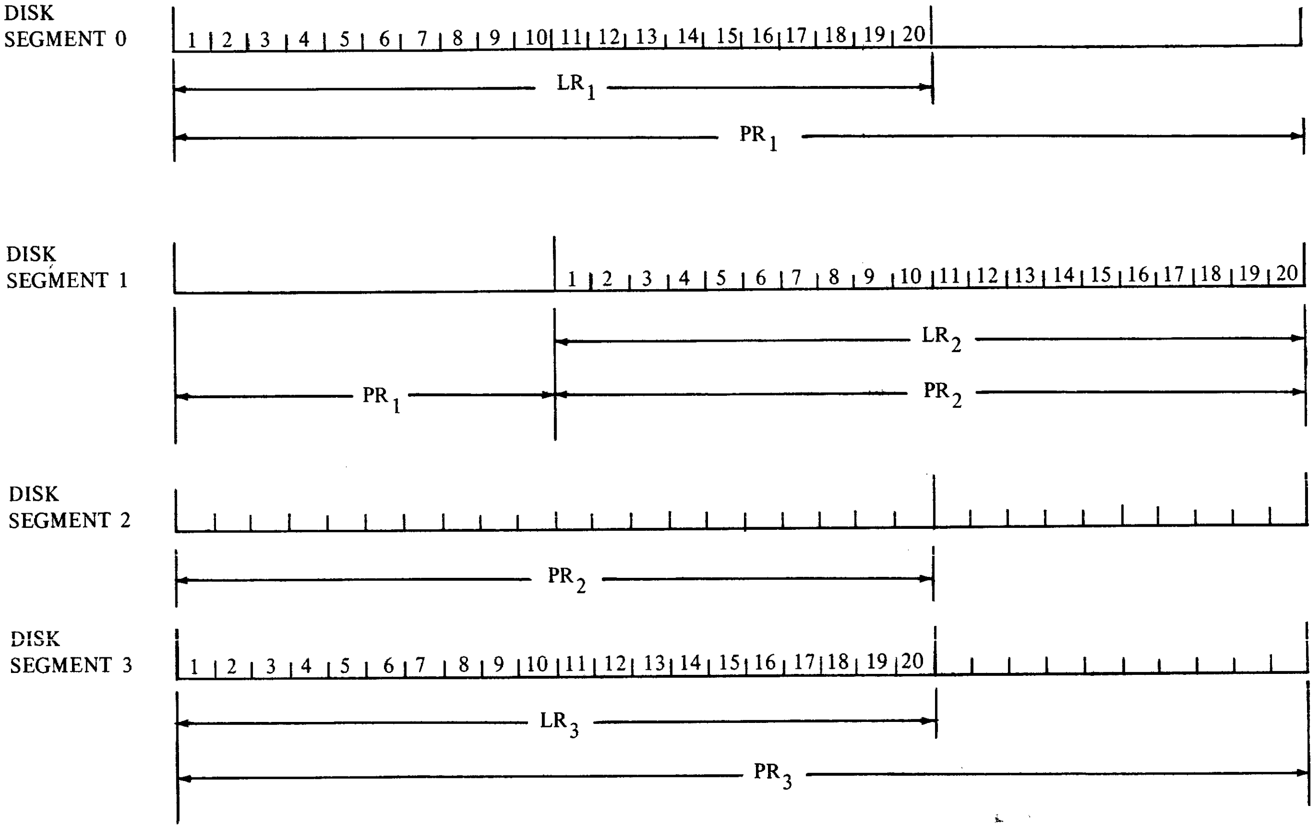
If a blocking factor of six had been specified, six records could be contained in two disk segments.

However, maximum efficiency could be achieved by letting PR=5, along with a blocking factor of 6. Using this case, all six records could be contained within one disk segment.

When the block size is 80, each block will use three disk segments, and each block will start at the beginning of the next disk segment.

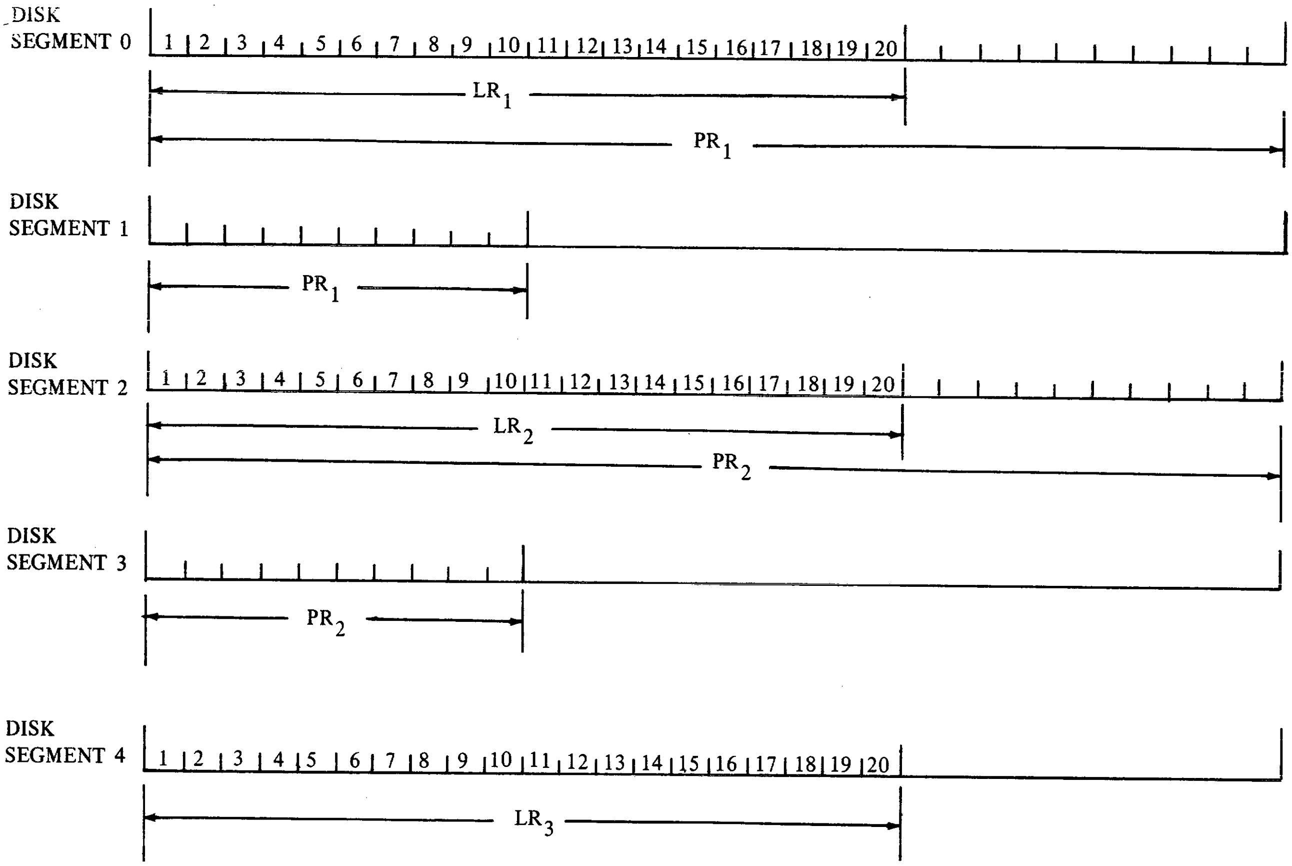
Example 7   PR = 40, LR = 20, BLOCKING =2

The disk output from a program that writes a logical record (1, 2, 3, . . . , 20) three times is illustrated in the following manner.



Example 8   PR = 40, LR = 20, UNBLOCKED

The disk output for a program that writes a logical record (1, 2, 3, . . . , 19, 20) three times to an unblocked disk file is illustrated in the following manner.



Since the blocking factor is undeclared, the physical record size also becomes the block size. In the above example, each block requires two disk segments.

BLOCKING.

The BLOCKING option on the FILE card provides the capability of packing more than one record into a physical block.

There are two advantages in blocking files:

a. Faster I/O speeds can be obtained since many records can be brought into or out of internal storage in a single access, thus giving a faster access time per record.

b. More efficient packing of data can be obtained. For example, an 80-character record written on disk unblocked would waste 160 characters. This is because the smallest addressable area on disk is the segment which contains 240 characters. By specifying three records per block (BLOCKING = 3), 100% utilization of disk can be obtained. Another example is tape blocking. By writing longer blocks, the amount of tape space wasted by inter-record-gaps would be reduced.

For most efficient utilization of disk, the blocking should be such that the block size should be 30 words (240 characters) or some integer multiple thereof.

The block size for disk should not exceed 1890 words.

The blocking of records during WRITE and unblocking during READ is handled automatically by the operating system.

BUFFERING.

The FILE card has an option whereby the number of buffers assigned to a file can be specified (two are assigned by default).

The number of buffers that should be specified for a given file depends on the characteristics of the file.

A file from which a record is accessed infrequently should have only one buffer. Specifying more wastes internal storage space.

A file that is accessed frequently should have two buffers. While data is being processed into or from one buffer, I/O can be in progress on the other buffer.

A file that is accessed N times between long processing loops should have N buffers. Since the operating system always tries to keep the buffers full for input and empty for output, the N buffers could be processed without having to wait for any actual I/O. Then, during the long processing loops, the operating system can do the required actual I/O operations.

DISK I/O.

The assignment of a file to disk requires the use of a FILE card (see appendix B).

If the AREA option is specified, the first reference to the file will cause the MCP to set up a directory indicating the amount of disk specified. In making the actual allocation of disk, the MCP will subdivide the file into 20 areas, each area containing 1/20 of the file. Actual allocation of disk space for each area occurs only when a WRITE statement references a record in that area.

If the FILE option card does not specify the AREA option, the first reference to the file expects the file to exist on disk.

To create a permanent file on disk, it is necessary to lock the file by either:

a. Specifying the LOCK and SAVE options on the FILE card, or

b. Executing a LOCK statement on the specified file before the program comes to an end, and specifying the SAVE option on the FILE card.

Each record on disk is addressed by its relative location in the file; the first record is record 0 (zero). The MCP, in order to compute an actual disk address from a record address, requires that each record be of fixed length. This record length is either 17 words (by default) or the record size specified in the FILE card. Attempting to write or read a logical record where the amount of data specified by the list exceeds the amount of data in the record will result in program termination. For the situation where a logical record is written which is smaller than the record size, the contents of that portion of the record left unfilled is undefined. Attempting to read this undefined data should be avoided; program termination can occur.

Associated with each file on disk is an End-of-File pointer. Each WRITE operation updates this pointer so that its value is always that of the highest record written.

SERIAL DISK I/O. Serial Disk I/O is selected by specifying the SERIAL option on the FILE card.

The operating system keeps an internal record pointer to control serial disk I/O. This pointer is set to -1 initially. Each READ or WRITE statement counts the pointer up by 1, then uses it as the relative record address to read or write.

The random access forms of the READ or WRITE I/O statement can be used for a file specified serial (see random disk I/O). When used, the internal record pointer is set equal to the address specified in the I/O statement rather than being counted by 1. Using the random access forms of the I/O statement on files specified as serial, although allowed, is slower than when the file is specified as random.

The results of mixing or alternating serial disk READ and WRITE statements without intervening REWINDs are not defined.

REWIND sets the internal record pointer to -1.

RANDOM DISK I/O. Random Disk I/O is selected by specifying RANDOM on the FILE card. Associated with random access is a special form of file identifier in the READ and WRITE I/O statements as follows:

u = r
where r is the relative address of the record to be accessed.

The rules for the form of r are the same as for an array subscript. The record specified by r will be the record accessed.

The internal record pointer is always set from r in the I/O statements. If the serial forms of the I/O statements are used with a file specified random, the internal record pointer is not changed. This results in several consecutive serial I/O statements accessing the same record repeatedly.

Mixing READ and WRITE statements is allowed in any sequence.

REWIND sets the internal record pointer at zero.

<u>Examples</u>:

```
READ(1=I....)
WRITE(6=(A+B-C)...
```

SECTION 8
SUBPROGRAMS

GENERAL.

A subprogram is a program unit, a self-contained and independent routine, which may be referenced by the main program and by other subprograms. There are three types of subprograms.

a. FUNCTION subprograms.
b. SUBROUTINE subprograms.
c. BLOCK DATA subprograms.

FUNCTIONS.

In mathematics, if the value of one quantity is dependent on the value or values of another quantity, then it is said to be a function of the other quantity. The first quantity is called the function and the other quantities are called the arguments. For example, in

arctan(x)

arctan is the function and x is the argument.

Functions may be divided into three categories:

a. Statement functions.
b. Intrinsic functions.
c. External functions.

STATEMENT FUNCTIONS.

A statement function is declared within the program unit in which it is referenced. It is defined by a single statement similar in form to an arithmetic or logical assignment statement.

The general form is:

$$f(x_1,x_2,...,x_n)=e$$

where f is the statement function name, $x_1,x_2,...,x_n$ are the dummy arguments, and e is an expression.

The rules for naming a function subprogram are the same as those for naming a variable (see section 2). The dummy arguments may be simple or subscripted variables. They represent values which are passed to the function subprogram and are used in the expression e in order to evaluate the function f. The dummy arguments are undefined outside of the statement function and may be redefined within the program unit. Together, f and e must conform to the rules for arithmetic or logical assignment statements.

Aside from the dummy arguments, the expression e may contain:

a. Variables used in the program unit.
b. Intrinsic function references.
c. References to previously defined statement functions.
d. External function references.

A statement function must be declared before the first executable statement.

A statement function is referenced in the same manner as a FUNCTION subprogram is referenced.

The name of a statement function must not appear in an EXTERNAL statement, nor as a variable name or an array name in the same program unit.

Example:

```
      DIMENSION A(10)
      LOGICAL STAFUN,Y,Z
      STAFUN(N)=X .LT. SIN(A(N))
      READ 25,X,Y,(A(I),I=1,10)
   25 FORMAT(F8.2,L2,10F7.2)
      DO 50 J=1,10
      Z=Y .AND. STAFUN(J)
      ...
   50 ...
```

INTRINSIC FUNCTIONS.

The intrinsic functions are those functions made available to a FORTRAN object program by the operating system. The names, types, and definitions of the intrinsic functions are predefined, so they need only be referenced in order to be used.

An intrinsic function name may be redefined within a program unit. However, if it has been redefined, then that intrinsic function will no longer be recognized by the compiler, but its identifier will be used as it has been redefined.

Also, the user may redefine the meaning of an intrinsic for an entire program by providing a FUNCTION subprogram of the same name. However, the type of the function and the number and type of its parameters must be the same as the original intrinsic.

Example:

```
REAL FUNCTION SIN(Y)
SIN=COS(Y)
RETURN
END
```

An intrinsic function is referenced by using it as a primary in an arithmetic or logical expression. The actual parameters which constitute the parameter list must agree in type, number, and order with the specifications in table 8-1, and may be any expression of the specified type. (For an explanation of actual parameters, see CALL statement, section 6.) When a real parameter is specified, however, an integer parameter may be used.

Execution of an intrinsic function reference results in the passing of the actual parameter values to the corresponding formal parameters of the intrinsic function and an evaluation of the intrinsic. The resultant value is then assigned to the intrinsic function identifier and thereby passed back to the intrinsic function reference.

Examples:

```
IBIG=MAXO(I,J,K,LEST)
TANGE=SIN(X+Y)/COS(A-B)
```

EXTERNAL FUNCTIONS.

An external function is a program unit which has as its first statement a FUNCTION statement.

The general form is:

t FUNCTION f(a_1,a_2,...,a_n)

where:

a. t is either INTEGER, REAL, DOUBLE PRECISION, LOGICAL, COMPLEX, or empty.

b. f is the symbolic name of the function being defined.

c. a_1,...,a_n are formal parameters which may be either a variable name, an array name, a SUBROUTINE or FUNCTION name.

An external function is normally referenced by another program unit. However, Burroughs FORTRAN permits an external function to reference itself, i.e., recurse.

The construction of external functions is subject to the following conditions:

a. The function name must be used as a variable within the function subprogram to the left of the replacement operator (=) in an assignment statement at least once. Its value at the time of execution of any RETURN statement within the function subprogram is the value of the function.

b. The name of the function must not appear in any non-executable statement in the function subprogram, except for the FUNCTION statement.

c. The symbolic names of the formal parameters may not appear in an EQUIVALENCE, COMMON, or DATA statement in the function subprogram.

d. The function subprogram may define or redefine one or more of its parameters to effectively return results in addition to the value of the function.

e. The function subprogram may contain any statements except SUBROUTINE, another FUNCTION statement, or BLOCK DATA.

f. The function subprogram must contain at least one RETURN statement.

g. An END statement must be the last statement of the subprogram body.

Example:

```
      FUNCTION EVAL(U,V)
      IF(U .LT. V) GO TO 1
      EVAL=V/U
      RETURN
    1 EVAL=U/V
      RETURN
      END
```

REFERENCING EXTERNAL FUNCTIONS.

An external function is referenced by using it as a primary in an arithmetic or logical expression. The actual parameters, which constitute the parameter list, must agree in order, number, and type with the corresponding formal parameters in the defining program unit. If a formal parameter is real, an integer actual parameter may be used. An actual parameter in an external function reference must be one of the following:

a. A Hollerith constant.

b. A variable name.

c. An array element name.

d. An array name.

e. An arithmetic or logical expression.

f. The name of a function or subroutine.

If an actual parameter is a function name (external or intrinsic) or a subroutine name, then the corresponding formal parameter must be used as a function name or a subroutine name, respectively.

If an actual parameter corresponds to a formal parameter that is defined or redefined in the referenced subprogram, the actual parameter must be a variable name, an array element name, or an array name. Execution of an external function reference, as described in the foregoing, results in an association of actual parameter with all appearances of corresponding formal parameters in the executable statements of the subprogram, and in an association of actual parameters with variable dimensions, if present, in the subprogram. Following these associations, execution of the first executable statement of the subprogram body is undertaken.

An actual parameter which is an array element name containing variables in the subscript could in every case be replaced by the same parameter with a constant subscript containing the same values as would be derived by computing the variable subscript just before association of parameters takes place.

If a formal parameter of an external function is an array name, the corresponding actual parameter must be an array name or array element name.

Example:

```
        TOTAL=EVAL(P,X)+CPS(Y)
```

SUBROUTINE.

A subroutine is defined externally to the program unit that references it. A subroutine defined by a FORTRAN statement headed by a SUBROUTINE statement is called a subroutine subprogram.

DEFINING SUBROUTINE SUBPROGRAMS.

The SUBROUTINE statement is one of the forms:

SUBROUTINE N

SUBROUTINE N $(a_1, a_2, \ldots a_n)$

where:

a. The letter N is the symbolic name of the subroutine to be defined.

b. The a's are formal parameters which may be either a variable name, an array name, a function or subroutine name, or an asterisk (*).

The construction of subroutine subprograms is subject to the following restrictions:

a. The symbolic names of the formal parameters may not appear in an EQUIVALENCE, COMMON, NAMELIST, or DATA statement in the subprogram.

b. The subroutine subprogram may define or redefine one or more of its parameters in order to effectively return results.

c. The subroutine subprogram may contain any statements except FUNCTION, another SUBROUTINE statement, or BLOCK DATA.

d. The subroutine subprogram must contain at least one RETURN, STOP, or CALL EXIT statement.

e. An END must be physically the last statement.

In Burroughs FORTRAN, a subroutine may call itself, i.e., recurse.

Example:

```
SUBROUTINE FALL(T,V,S)
G=32.172
S=G*T**2/2
V=G*T
RETURN
END
```

PASSING ARRAY DATA TO A SUBROUTINE. The first subscript of an array varries most rapidly, and the last subscript the least rapidly. An array A dimensioned 3 x 3 is linearly layed-out as follows:

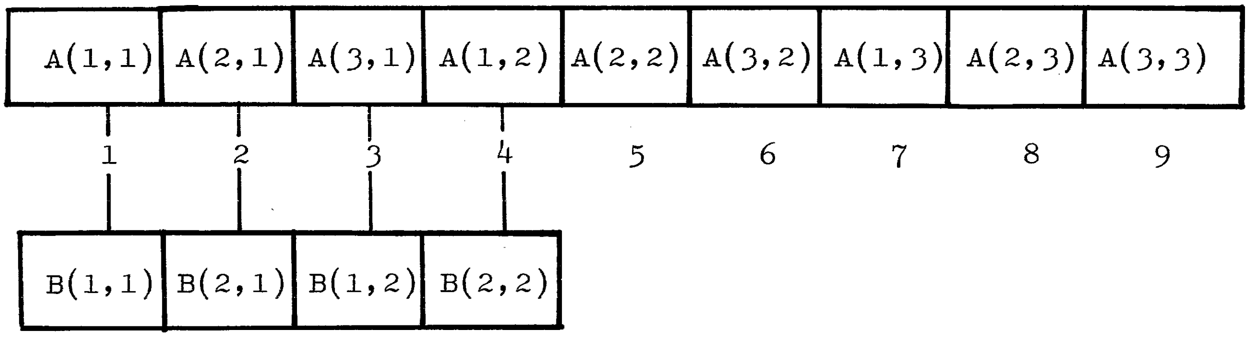
A(1,1)	A(2,1)	A(3,1)	A(1,2)	A(2,2)	A(3,2)	A(1,3)	A(2,3)	A(3,3)

<u>Example</u>:

```
      DIMENSION A (3,3)
      DO 10 I=1,3
      DO 10 J=1,3
      K= K+1
      A(I,J) = K
   10 CONTINUE
      M=2
      CALL SUB (A,M)
      STOP
      END
      SUBROUTINE SUB (B,N)
      DIMENSION B (N,N)
      RETURN
      END
```

In this example, the main program assigns values to the A array and variable M, and then passes the data to the subroutine SUB. The actual parameters A and M corresponds to the formal parameters B and N of the subroutine. Note: The B array is dimensioned 2x2; whereas, A is dimensioned 3x3.

The following is an example of the mapping sequence set-up between the four elements of B, and the first four elements of A.



The mapping is always set-up so that the n elements of the subroutine array will correspond to the first n elements of the array passed to it in the calling program.

When an array is passed to a subroutine, the size of the corresponding subroutine array cannot exceed the size of the array being passed by the calling program. If this condition exists an "INVALID INDEX" message is printed at the time of program execution.

NONSTANDARD RETURNS FROM SUBROUTINES.

If a subroutine contains one or more nonstandard return statements (has the term RETURN n), the formal parameter list must contain one asterisk (*) for each return number. The actual parameter list of the referencing program unit must then have a dollar sign ($) followed by a label in the corresponding position.

Example:

```
      Calling Program                Called Program

      ...                            SUBROUTINE XYZ (U,V,*,*)
      ...                            ...
      ...                            ...
      CALL XYZ (A,B,$10,$15)         IF (EXP) 1,2,3
    5 ...                          1 RETURN
      ...                          2 RETURN 1
      ...                          3 RETURN 2
      ...                            END
   10 ...
      ...
      ...
      ...
   15 ...
      ...
      ...
      END
```

In the above example, if the value of EXP is negative, control will be returned to the referencing program at the statement labeled 5; if the value of EXP is zero, control will be returned at label 10; and if the value of EXP is positive, control will be returned at label 15.

MULTIPLE ENTRY POINTS INTO A SUBPROGRAM.

For a normal entry into a subroutine subprogram, a CALL statement that refers to the subroutine is used. A normal entry into a FUNCTION subprogram is made by a reference to the function name in an arithmetic expression. Entry is made at the first executable statement following the SUBROUTINE or FUNCTION statement.

A subprogram can also be entered by way of a CALL statement or a function reference that refers to the name in an ENTRY statement in the subprogram. The entry is made at the first executable statement following the ENTRY statement.

ENTRY statements are non-executable. Therefore, they do not affect control sequencing during normal execution of a subprogram. The type, order, and number of parameters need not agree between the SUBROUTINE or FUNCTION statement and the ENTRY statement, nor do the ENTRY statements have to agree among themselves. However, each CALL or function reference must agree in type, order, and number with the SUBROUTINE, FUNCTION, or ENTRY statement that it refers to.

The ENTRY statement in the called subprogram is one of the forms:

ENTRY N

ENTRY N($a_1,a_2,...a_n$)

where:

a. N is the symbolic name of an entry point.

b. The a's are formal parameters which may be either a variable name, an array name, a subroutine or function name, or an asterisk (*).

Example:

```
      Calling Program               Called Program
      ...                           SUBROUTINE SUB(U,V,W,X)
      ...                           ...
    5 CALL SUB(A,B,C,D)             ...
      ...                        10 ...
      ...                           ...
   10 CALL ENT1                     ENTRY ENT1
      ...                           GO TO 10
      ...                           ...
   15 CALL ENT2(G,H)                ...
      ...                           ENTRY ENT2(G,H)
      ...                           ...
      END                           ...
                                    END
```

In the above example, execution of statement 5 causes entry into SUB, starting with the first executable statement of the subroutine. Execution of statements 10 and 15 also causes entry into the called program, starting with the first executable statement following the ENTRY ENT1 and ENTRY ENT2(G,H) statements respectively.

The following are additional rules for entry points:

a. An ENTRY name may appear in an EXTERNAL statement in the same manner as a FUNCTION or SUBROUTINE name.

b. ENTRY statements may appear only in subprograms.

c. Entry into a subprogram initializes all references in the entire called subprogram from items in the parameter list of the CALL or function reference.

d. If an adjustable array name or any of its variable dimensions appear in a parameter list for a FUNCTION, SUBROUTINE, or ENTRY statement, that array name and all its variable dimensions must appear in that parameter list.

e. If an array is passed as a parameter to a SUBROUTINE or FUNCTION and is also used by a section of the program entered through an ENTRY statement, then the array name must appear in the parameter list of the ENTRY statement.

f. In a FUNCTION subprogram, only the FUNCTION name may be used as the variable to carry a result back to the calling program. The ENTRY name may not be used for this purpose.

g. An ENTRY name defined in a subroutine subprogram, if referenced, must be referenced by a CALL statement. Similarly, an entry defined in a function subprogram, if referenced, must be referenced as a function.

BLOCK DATA.

Further use of the DATA statement is in the BLOCK DATA subprogram. It is used to enter data into COMMON blocks; however, the following must be observed:

a. There may be no executable statements in a BLOCK DATA subprogram. The first statement of the subprogram must be BLOCK DATA.

b. The subprogram may contain only Type, EQUIVALENCE, DATA, DIMENSION, and COMMON statements.

c. All elements of a COMMON BLOCK must appear in the COMMON statement list even though some do not appear in the DATA statement list.

d. More than one COMMON block may be initialized by a single BLOCK DATA subprogram.

e. There may be as many BLOCK DATA subprograms as desired in a program, but any block identifier may occur in only one BLOCK DATA subprogram.

Example:

```
BLOCK DATA
COMMON /TEST/ K, L, S/ AATWO/ B,C
DIMENSION C(10)
DATA L, S/ 1, 3.5/, C/ 10*16.2/
END
```

Table 8-1

Resulting Actions of an Intrinsic Function

Function	Definition	Number of Arguments	Symbolic Name	Type of Argument	Type of Function
Absolute Value	$\lvert a \rvert$	1	ABS	Real	Real
			IABS	Integer	Integer
			DABS	Double	Double
			CABS	Complex	Real
Truncation	Sign of a times largest integer $\leq \lvert a \rvert$	1	AINT	Real	Real
			INT	Real	Integer
			IDINT	Double	Integer
Remaindering*	$a_1 \pmod{a_2}$	2	AMOD	Real	Real
			MOD	Integer	Integer
			DMOD	Double	Double
Choosing Largest Value	Max $(a_1,a_2 \ldots)$	≥ 2	AMAX0	Integer	Real
			AMAX1	Real	Real
			MAX0	Integer	Integer
			MAX1	Real	Integer
			DMAX1	Double	Double
Choosing Smallest Value	Min $(a_1,a_2 \ldots)$	≥ 2	AMIN0	Integer	Real
			AMIN1	Real	Real
			MIN0	Integer	Integer
			MIN1	Real	Integer
			DMIN1	Double	Double
Float	Conversion from integer to real	1	FLOAT	Integer	Real

*Note: The functions MOD, AMOD and DMOD (a_1,a_2) are defined as $a_1 - [a_1/a_2]*a_2$, where [a] denotes the integral part of a.

Table 8-1 (cont)

Resulting Actions of an Intrinsic Function

Function	Definition	Number of Arguments	Symbolic Name	Type of Argument	Type of Function
Fix	Conversion from real to integer	1	IFIX	Real	Integer
Transfer of sign	sign of a_2 times $\|a_1\|$	2	SIGN ISIGN DSIGN	Real Integer Double	Real Integer Double
Positive Difference	a_1-Min (a_1,a_2)	2	DIM IDIM	Real Integer	Real Integer
Obtain Most Significant Part of Double Precision Argument		1	SNGL	Double	Real
Express Single Precision Argument in Double Precision Form		1	DBLE	Real	Double
Obtain Real Part		1	REAL	Complex	Real
Obtain Imaginary Part		1	AIMAG	Complex	Real
Create Complex	$C = a_1 + ia_2$	2	CMPLX	Real	Complex
Complex Conjugate	$C = X - iY$	1	CONJG	Complex	Complex
Exponential	e^a	1 1 1	EXP DEXP CEXP	Real Double Complex	Real Double Complex

Table 8-1 (cont)

Resulting Actions of an Intrinsic Function

Function	Definition	Number of Arguments	Symbolic Name	Type of Argument	Type of Function
Natural Logarithm	$\log_e$ (a)	1	ALOG	Real	Real
		1	DLOG	Double	Double
		1	CLOG	Complex	Complex
Common Logarithm	$\log_{10}$ (a)	1	ALOG10	Real	Real
		1	DLOG10	Double	Double
Trigometric Sine	sin (a)	1	SIN	Real	Real
		1	DSIN	Double	Double
		1	CSIN	Complex	Complex
Trigometric Cosine	cos (a)	1	COS	Real	Real
		1	DCOS	Double	Double
		1	CCOS	Complex	Complex
Arctangent	arctan (a)	1	ATAN	Real	Real
		1	DATAN	Double	Double
Arctangent	arctan (a_1/a_2)	2	ATAN2	Real	Real
		2	DATAN2	Double	Double
Square Root	$(a)^{1/2}$	1	SQRT	Real	Real
		1	DSQRT	Double	Double
		1	CSQRT	Complex	Complex
Hyperbolic Tangent	tanh (a)	1	TANH	Real	Real
Trigometric Tangent	tan (a)	1	TAN	Real	Real
Trigometric Cotangent	cot (a)	1	COTAN	Real	Real

NOTE: Where applicable, trigonometric functions must be in radians.

Table 8-1 (cont)

Resulting Actions of an Intrinsic Function

Function	Definition	Number of Arguments	Symbolic Name	Type of Argument	Type of Function
Arcsine	arcsine (a)	1	ARSIN	Real	Real
Arccosine	arccosine (a)	1	ARCOS	Real	Real
Hyperbolic Sine	sinh (a)	1	SINH	Real	Real
Hyperbolic Cosine	cosh (a)	1	COSH	Real	Real
Error Function	error function (a)	1	ERF	Real	Real
Gamma Function	gamma (a)	1	GAMMA	Real	Real
Log Gamma Function	log gamma (a)	1	ALGAMA	Real	Real
47-bit Logical AND		2	AND	Real	Real
47-bit Logical OR		2	OR	Real	Real
47-bit Logical COMPLEMENT		1	COMPL	Real	Real
47-bit Logical EQUIVALENCE		2	EQUIV	Real	Real
Concatenation		5	CONCAT	Real Integer	Real
Time	a = 0, date in form OYYDDD (right-adjusted)	1	TIME	Integer	Alpha when a=0

Table 8-1 (cont)

Resulting Actions of an Intrinsic Function

Function	Definition	Number of Arguments	Symbolic Name	Type of Argument	Type of Function
Time (cont)	a = 1, time of day in sixtieths of a second (based on 24-hour clock) a = 2, elapsed processor time of program since its start in sixtieths of a second. a = 3, elapsed I/O time of program since its start in sixtieths of a second. a = 4, value of 6-bit machine timer.	1			Integer when a=1,2,3,4.

APPENDIX A
GLOSSARY

ACTUAL PARAMETERS. Those parameters in the parameter list of a subroutine call or function reference. In contrast to FORMAL PARAMETERS.

ALPHANUMERIC. Contraction of alphabetic and numeric, signifying the alphabetic and numeric characters.

ASSIGNMENT OPERATOR. In FORTRAN, the equal sign (=).

BCL. An acronym for Burroughs Common Language.

EXECUTABLE PROGRAM. A program that can be used as a self-contained computing procedure. It consists minimally of one main program. It may consist of one main program and any number of subprograms.

EXECUTABLE STATEMENT. A non-declarative statement which is executed at object time. In contrast to NON-EXECUTABLE STATEMENT.

EXPONENT. That part of a real (floating-point) number which determines the decimal point placement in the mantissa.

EXPRESSION. Any constant, variable, or function reference, or any combination of these separated by operators, commas, or parentheses.

FIXED-POINT. An arithmetic notation in which the decimal point is not present and is assumed to be on the extreme right of a number. In contrast to FLOATING-POINT.

FLOATING-POINT. An arithmetic notation in which the position of the decimal point does not remain fixed with respect to one end of the numerals. In contrast to FIXED-POINT.

FORMAL PARAMETERS. Those parameters in the parameter list of a subroutine or function declaration. In contrast to ACTUAL PARAMETERS.

MAIN PROGRAM. A set of statements and comments not containing a FUNCTION, SUBROUTINE, or BLOCK DATA statement.

MANTISSA. That part of a real (floating-point) number which contains the significant digits.

MCP. An acronym for the Master Control Program, the executive system.

NON-EXECUTABLE STATEMENT. A declaration which, at compile time, provides the compiler with a description of data. It is not executed at object time. In contrast to EXECUTABLE STATEMENT.

PRIMARY. An arithmetic expression enclosed in parentheses, a constant, a variable reference, an array element reference, or a function reference.

PROGRAM UNIT. Refers to either a main program or subprogram.

PRT. An acronym for Program Reference Table. An area in memory for the storage of operands, references to arrays, references to segments of a program, and references to files. Permits programs to be independent of the actual memory locations occupied by data and parts of the program.

REFERENCE. A term used with special meaning to indicate an identification of:

a. A datum, implying that the current value of the datum will be made available during the execution of the statement containing the reference.

b. A procedure, implying that the actions specified by the procedure will be made available upon reference.

SPO. An acronym for SuPervisOry Printer, the system console typewriter.

SUBPROGRAM. A set of statements and comments headed by a FUNCTION, SUBROUTINE, or BLOCK DATA statement.

APPENDIX B
FILE CARDS

FILE cards are optional since all the parameters used in declaring a file have default values. The various I/O statements and their default descriptions when a FILE card is not used are listed in table B-1. Time-Sharing FORTRAN default conditions are described in appendix J.

Table B-1
File Default Descriptions

I/O Statement	File Name	Blocking	Mode	Peripheral
READ f,k*	READER	10 Word Buffer, 80 Characters	Alpha	Tape or Card Reader
READ(u,f) k READ(u) k	FILEi (where i is the value of u)	17 Word Buffer, 132 Characters, (size of logical record is unlimited)	Binary	Tape
WRITE(u,f) k WRITE(u) k	FILEi (where i is the value of u)	17 Word Buffer 132 Characters, (size of logical record is unlimited)	Binary	Tape
PRINT f,k*	PRINT	17 Word Buffer, 132 Characters	Alpha	Line Printer
PUNCH f,k*	PUNCH NOTE In all cases, the multi-file name is empty.	10 Word Buffer, 80 Characters	Alpha	Card Punch

*FILE cards cannot be used for these I/O statements.

FILE CARD FORMAT

FILE cards are free format, with the exception of card columns 1-6:

```
1 2 3 4 5 6  (card column)
F I L E b b
```

where a blank is denoted by b.

Columns 73 through 80 are used for a sequence number or identification only, and will be ignored by the compiler except when merging a card and tape file at compile time.

Following the two blanks, the following information must be inserted in free format:

N = FID

or

N = MFID/FID

where N is an unsigned integer constant representing the logical unit number. It is the value of u in READ(u,f)k and WRITE(u,f)k. MFID is the multi-file identification, and FID is the file identification. If MFID is not included, then it is assumed to be seven zeros. For further information, reference should be made to the System Operation Manual.

The following is a list of options which may be included on the FILE card. They may be in free format, but they must come in the order in which they are given below:

a. ,UNIT = t

where t is one of the following:

PRINT
PRINTER
READER
PUNCH
DISK
TAPE
REMOTE

TAPE is the default option for UNIT.

b. ,UNLABELED

LABELED is the default option if unlabeled is not specified (tape only).

c. ,ALPHA
BINARY is the default option if ALPHA is not specified (tape only).

d. ,SAVE = n
where n is an unsigned integer whose value cannot exceed 999. It is the save factor, in days (see System Operation Manual). The default save factor is zero.

e. ,LOCK
When this option is used, the MCP will close and lock a disk file when the program creating it has gone to End-of-Job (see tape and disk I/O, section 7). In the case of a disk file, if a CLOSE statement or END FILE statement appears for the designated file, the LOCK will be overridden and the file will be released.

f. , SERIAL
, RANDOM
This option specifies the access mode for disk files only. The default option is SERIAL.

g. ,AREA = n
where n is an unsigned integer constant which denotes the amount of area on disk (in number of records) to reserve for this file (see tape and disk I/O, section 7). If a file is to be opened for input this option must <u>not</u> be present.

h. ,BLOCKING = n
where n is an unsigned integer which represents the number of logical records per physical block. The default option is unblocked files, i.e., a blocking factor of one.

i. ,RECORD = n
where n is an unsigned integer which represents the size (in words) of a logical record. The default option is 17 (see table B-1).

j. ,BUFFER = n

where n is an unsigned integer which represents the number of buffers. The default option is 2 (see tape and disk I/O, section 7).

If the FILE option extends across more than one card, then the next card must be flagged as a FORTRAN continuation card with a character other than a blank or zero in card column 6.

In the I/O statements READ(u,f)k and WRITE(u,f)k, if u is not an integer constant, then its value at run time must correspond to a logical unit number declared on a FILE card.

If the option

UNIT = t

is used to declare the file as a line printer, card punch, or card reader, then the remaining default descriptions used for this file are designated in table B-1.

APPENDIX C
DOLLAR SIGN CARDS

A dollar sign card is optional and is used to indicate to the compiler that certain options are to be used at compile time. The format of a dollar sign card is:

Card Column	Contents
1	$
2-72	Options in free field format.
73-80	Card number or blank.

The dollar sign card may be placed:

a. Immediately after the MCP control cards used for compilation and immediately before the first FORTRAN FILE card or FORTRAN source or patch card if no FILE cards are used (see section 1).

b. Anywhere else in the source or patch deck with a proper sequence number in order to change options at some point in compilation, e.g., to list only a part of the compiled source program. Dollar sign cards may not, however, be interspersed with the continuation cards of a multi-card statement.

Example:

```
                                  Sequence
                                  Number
$CARD                             00000100
      ...                         ...
      ...                         ...
      ...                         ...
      A=B+C                       00009000
$CARD LIST                        00009100
      X=SQRT(Y**2+Z**2)           00009200
      PAR=TAN(X/A)                00009300
      ...                         ...
      V=SIN(X+Y-Z)                00012200
$CARD                             00012300
      ...                         ...
```

Only cards 00009200 through 00012200 will be listed on the file LINE.

Each dollar sign card causes all previous options to be reset, with the exceptions: $REMOTE, $ONSITE, $SEQXEQ, $SEQ, $NOSEQ, and $TIME. If no dollar sign card is included with the source deck, then the CARD and LIST options are assumed (see appendix J).

The various options available are as follows:

TAPE or CARD

a. One of these, but not both, should be the first option on the dollar sign card immediately following the dollar sign.

b. CARD indicates to the compiler that the source program input is entirely from the file labeled CARD.

c. TAPE indicates to the compiler that the source program input is from the file named TAPE and labeled FORSYM and that change or patch cards may be inputted from the file labeled CARD. If a change or patch card file is used, then it is merged into the source program from the file named TAPE and labeled FORSYM as a function of the sequence number in columns 73-80. If a listing is obtained, then the source statements from the TAPE file will have a T following the sequence number, and the source statements being merged from the CARD file will have an R following the sequence number on the compiled source listing. The merging process uses the system alphanumeric collating sequence (see appendix G).

d. If the first word on a dollar control card is not CARD or TAPE, then the dollar control card input mode (either card or tape) is set to the mode of the previous dollar control card. The initial mode is card.

LIST

a. If present, then a compiled source listing of the source program will be made on the file LINE, including any change or patch cards.

b. Segment and address information will also be listed with the source program.

SGL or SINGLE.

a. If present, then a single-spaced compiled source listing of the source program will be made on the file line. If LIST is also present, the SGL or SINGLE takes precedence.

NEW or NEW TAPE

a. If present, a new source tape file labeled FORSYM is created which includes all change or patch cards and FILE cards, but does not include dollar sign cards.

PRT

a. If present, then a listing of the source program will be made on the file LINE, including any change or patch cards, and at the end of each program unit listing, a listing of PRT* and stack assignments for each local identifier within that program unit will be made.

b. At the end of the entire program, PRT assignments for all global names will be listed.

c. If PRT is specified, then LIST is automatically evoked.

DEBUGN

a. If present, then the actual machine code emitted by the compiler is also listed on the file LINE together with octal values of constants and format of PRT entries.

b. If DEBUGN is specified, then PRT and LIST are automatically evoked.

TRACE

a. If present, then information is listed on the file LINE which indicates how the FORTRAN compiler is analyzing the syntax of the source program.

* PRT is an abbreviation for Program Reference Table (see the System Operation Manual).

b. TRACE should be used only in extreme cases because of the great volume of output produced.

c. If TRACE is specified, the LIST, PRT, and DEBUGN options are automatically evoked.

SEQ f s i

a. If present, the listing on file LINE and the new source program on the file NEWTAPE, labeled FORSYM (if NEW or NEW TAPE is specified), will be resequenced.

b. The specifications following SEQ have the following interpretations:

f - the sequence number of the first card of the source program.

s - any special character, usually plus (+) or comma (,).

i - increment. If i=0, or i is not a number, then an increment of 1000 is used.

c. The SEQ option, if used, must be the last option on the dollar sign card.

NOSEQ

a. If present, will cause the SEQ option to be turned off.

HOL

a. If the source cards are punched in IBM code and the HOL option is not used, then the listing of the source program produced by the compiler will be in IBM card codes, e.g., (will be printed as %, = will be printed as #, etc. However, the compiler will properly interpret the source program and compile it.

b. If the source cards are punched in IBM card code and the HOL option is used, then all characters will be converted to BCL before printing on the file LINE.

c. If the source cards are punched in IBM/360 card code, then the HOL option must be used to convert the source program to BCL.

d. The HOL option will translate all IBM or IBM/360 cards to BCL including strings and Hollerith constants. This option also causes the object program produced by the compiler to automatically convert into BCL data read with an A format specification and data read into Hollerith strings.

e. The use of the HOL option will slow compilation speed. For repeated compilations from large source programs, it would be advantageous to use the NEW TAPE option with HOL on the first compilation. Thereafter, compilations may be made without the HOL option from the generated source tape.

TIME

a. If present and if the LIST option is not present, then the source program will not be listed, but at the end of the compilation, compilation information will be listed on the file LINE.

CHECK

a. If present, the number of sequence errors detected in the source file is printed next to the number of syntax errors. Also, under the $CHECK option, a sequence error for a source record will cause the card image and a warning message to be written to the file LINE:

SEQUENCE ERROR "n" < "p"

where n is the sequence number of the card image and p is the sequence number of the previous card image. The system alphanumeric collating sequence is used (see appendix G).

VOID n

a. If present, VOID must be the only option on the dollar sign card. This option is used only when merging a CARD and TAPE file.

b. If present, and if n is blank, the record on the TAPE file with the same sequence number (in columns 73-80) as the $VOID card will be ignored by the compiler, will not be listed on the file LINE, and will not be inserted in the file NEWTAPE, if the NEW option has been specified previously.

c. If present, and if n is not blank, n must be the sequence number of a record existing on the TAPE file and, in addition, the $VOID card must have a sequence number in columns 73-80. The records on the TAPE file, starting with the record which has the same sequence number as the $VOID card (columns 73-80), will be ignored up to but <u>not</u> including the record on the TAPE file with the sequence number n. These records will be ignored by the compiler, not listed on the file LINE, and not inserted in the file NEWTAPE, if the NEW option has been specified previously.

d. If the first word on a dollar control card is not VOID, all dollar control card options are set to OFF. Only those options invoked by the current dollar control card are set to ON.

NOTE

See appendix J, Time-Sharing FORTRAN, for additional option information.

APPENDIX D

COMPILE TIME ERROR MESSAGES

ERROR NUMBER	ERROR MESSAGE
000	SYNTAX ERROR
001	MISSING OPERATOR OR PUNCTUATION
002	CONFLICTING COMMON AND/OR EQUIVALENCE ALLOCATION
003	MISSING RIGHT PARENTHESIS
004	ENTRY STMT ILLEGAL IN MAIN PGM OR BLOCK DATA
005	MISSING END STATEMENT
006	ARITHMETIC EXPRESSION REQUIRED
007	LOGICAL EXPRESSION REQUIRED
008	TOO MANY LEFT PARENTHESES
009	TOO MANY RIGHT PARENTHESES
010	FORMAL PARAMETER ILLEGAL IN COMMON
011	FORMAL PARAMETER ILLEGAL IN EQUIVALENCE
012	THIS STATEMENT ILLEGAL IN BLOCK DATA SUBPROGRAM
013	INFO ARRAY OVERFLOW
014	IMPROPER DO NEST
015	DO LABEL PREVIOUSLY DEFINED
016	UNRECOGNIZED STATEMENT TYPE
017	ILLEGAL DO STATEMENT
018	FORMAT STATEMENT MUST HAVE LABEL
019	UNDEFINED LABEL
020	MULTIPLE DEFINITION
021	ILLEGAL IDENTIFIER CLASS IN THIS CONTEXT
022	UNPAIRED QUOTES IN FORMAT
023	NOT ENOUGH SUBSCRIPTS
024	TOO MANY SUBSCRIPTS
025	FUNCTION OR SUBROUTINE PREVIOUSLY DEFINED
026	FORMAL PARAMETER MULTIPLY DEFINED IN HEADING
027	ILLEGAL USE OF NAMELIST
028	NUMBER OF PARAMETERS INCONSISTENT
029	CANNOT BRANCH TO FORMAT STATEMENT
030	SUBROUTINE OR FUNCTION NOT DEFINED IN PROGRAM

APPENDIX D (cont)
COMPILE TIME ERROR MESSAGES

ERROR NUMBER	ERROR MESSAGES
031	IDENTIFIER ALREADY GIVEN TYPE
032	ILLEGAL FORMAT SYNTAX
033	INCORRECT USE OF FILE
034	INCONSISTENT USE OF IDENTIFIER
035	ARRAY IDENTIFIER EXPECTED
036	EXPRESSION VALUE REQUIRED
037	ILLEGAL FILE CARD SYNTAX
038	ILLEGAL CONTROL ELEMENT
039	DECLARATION MUST PRECEDE FIRST REFERENCE
040	INCONSISTENT USE OF LABEL AS PARAMETER
041	NO. OF PARAMS. DISAGREES WITH PREV. REFERENCE
042	ILLEGAL USE OF FORMAL PARAMETER
043	ERROR IN HOLLERITH LITERAL CHARACTER COUNT
044	ILLEGAL USE OF FORMAL PARAMETER
045	TOO MANY SEGMENTS IN SOURCE PROGRAM
046	TOO MANY PRT ASSIGNMENTS IN SOURCE PROGRAM
047	LAST BLOCK DECLARATION HAD LESS THAN 1024 WORDS
048	ILLEGAL I/O LIST ELEMENT
049	LEFT SIDE MUST BE SIMPLE OR SUBSCRIPTED VARIABLE
050	VARIABLE EXPECTED
051	ILLEGAL USE OF .OR.
052	ILLEGAL USE OF .AND.
053	ILLEGAL USE OF .NOT.
054	ILLEGAL USE OF RELATIONAL OPERATOR
055	ILLEGAL MIXED TYPES
056	ILLEGAL EXPRESSION STRUCTURE
057	ILLEGAL PARAMETER
058	RECORD BLOCK GREATER THAN 1023
059	TOO MANY OPTIONAL FILES
060	FILE CARDS MUST PRECEDE SOURCE DECK
061	BINARY WRITE STATEMENT HAS NO LIST

APPENDIX D (cont)

COMPILE TIME ERROR MESSAGES

ERROR NUMBER	ERROR MESSAGES
062	UNDEFINED FORMAT NUMBER
063	ILLEGAL EXPONENT IN CONSTANT
064	ILLEGAL CONSTANT IN DATA STATEMENT
065	MAIN PROGRAM MISSING
066	PARAMETER MUST BE ARRAY IDENTIFIER
067	PARAMETER MUST BE EXPRESSION
068	PARAMETER MUST BE LABEL
069	PARAMETER MUST BE FUNCTION IDENTIFIER
070	PARAMETER MUST BE FUNCTION OR SUBROUTINE ID
071	PARAMETER MUST BE SUBROUTINE IDENTIFIER
072	PARAMETER MUST BE ARRAY IDENTIFIER OR EXPRESSION
073	ARITHMETIC - LOGICAL CONFLICT ON STORE
074	ARRAYID MUST BE SUBSCRIPTED IN THIS CONTEXT
075	MORE THAN ONE MAIN PROGRAM
076	ONLY COMMON ELEMENTS PERMITTED
077	TOO MANY FILES
078	FORMAT OR NAMELIST TOO LONG
079	FORMAL PARAMETER MUST BE ARRAY IDENTIFIER
080	FORMAL PARAMETER MUST BE SIMPLE VARIABLE
081	FORMAL PARAMETER MUST BE FUNCTION IDENTIFIER
082	FORMAL PARAMETER MUST BE SUBROUTINE IDENTIFIER
083	FORMAL PARAMETER MUST BE FUNCTION OR SUBROUTINE
084	DO OR IMPLIED DO INDEX MUST BE INTEGER OR REAL
085	ILLEGAL COMPLEX CONSTANT
086	ILLEGAL MIXED TYPE STORE
087	CONSTANT EXCEEDS HARDWARE LIMITS
088	PARAMETER TYPE CONFLICTS WITH PREVIOUS USE
089	COMPLEX EXPRESSION ILLEGAL IN IF STATEMENT
090	COMPLEX EXPRESSION ILLEGAL IN RELATION
091	TOO MANY FORMATS REFERENCED BUT NOT YET FOUND
092	VARIABLE ARRAY BOUND MUST BE FORMAL VARIABLE

APPENDIX D (cont)
COMPILE TIME ERROR MESSAGES

ERROR NUMBER	ERROR MESSAGES
093	ARRAY BOUND MUST HAVE INTEGER OR REAL TYPE
094	COMMA OR RIGHT PARENTHESIS EXPECTED
095	ARRAY ALREADY GIVEN BOUNDS
096	ONLY FORMAL ARRAYS MUST BE GIVEN VARIABLE BOUNDS
097	MISSING LEFT PARENTHESIS IN IMPLIED DO
098	SUBSCRIPT MUST BE INTEGER OR REAL
099	ARRAY SIZE CANNOT EXCEED 32767 WORDS
100	COMMON OR EQUIV BLOCK CANNOT EXCEED 32767 WORDS
101	THIS STATEMENT ILLEGAL IN LOGICAL IF
102	REAL OR INTEGER TYPE REQUIRED
103	ARRAY BOUND INFORMATION REQUIRED
104	REPLACEMENT OPERATOR EXPECTED
105	IDENTIFIER EXPECTED
106	LEFT PARENTHESIS EXPECTED
107	ILLEGAL FORMAL PARAMETER
108	RIGHT PARENTHESIS EXPECTED
109	STATEMENT NUMBER EXPECTED
110	SLASH EXPECTED
111	ENTRY STATEMENT CANNOT START PROGRAM UNIT
112	ARRAY MUST BE DIMENSIONED PRIOR TO EQUIV STMT
113	INTEGER CONSTANT EXPECTED
114	COMMA EXPECTED
115	SLASH OR END OF STATEMENT EXPECTED
116	FORMAT, ARRAY OR NAMELIST EXPECTED
117	END OF STATEMENT EXPECTED
118	IO STATEMENT WITH NAMELIST CANNOT HAVE IO LIST
119	COMMA OR END OF STATEMENT EXPECTED
120	STRING TOO LONG
121	MISSING QUOTE AT END OF STRING
122	ILLEGAL ARRAY BOUND
123	TOO MANY HANGING BRANCHES

APPENDIX D (cont)

COMPILE TIME ERROR MESSAGES

ERROR NUMBER	ERROR MESSAGES
124	TOO MANY COMMON OR EQUIVALENCE ELEMENTS
125	ASTERISK EXPECTED
126	COMMA OR SLASH EXPECTED
127	DATA SET TOO LARGE
128	TOO MANY ENTRY STATEMENTS IN THIS SUBPROGRAM
129	DECIMAL WIDTH EXCEEDS FIELD WIDTH
130	UNSPECIFIED FIELD WIDTH
131	UNSPECIFIED SCALE FACTOR
132	ILLEGAL FORMAT CHARACTER
133	UNSPECIFIED DECIMAL FIELD
134	DECIMAL FIELD ILLEGAL FOR THIS SPECIFIER
135	ILLEGAL LABEL
136	UNDEFINED NAMELIST
137	MULTIPLY DEFINED ACTION LABELS
138	TOO MANY NESTED DO STATEMENTS
139	STMT FUNCTION ID AND EXPRESSION DISAGREE IN TYPE
140	ILLEGAL USE OF STATEMENT FUNCTION
141	UNRECOGNIZED CONSTRUCT
142	RETURN, STOP OR CALL EXIT REQUIRED IN SUBPROGRAM
143	FORMAT NUMBER USED PREVIOUSLY AS LABEL
144	LABEL USED PREVIOUSLY AS FORMAT NUMBER
145	NON-STANDARD RETURN REQUIRES LABEL PARAMETERS
146	DOUBLE OR COMPLEX REQUIRES EVEN OFFSET
147	FORMAT PARAMETER ILLEGAL IN DATA STATEMENT
148	TOO MANY LOCAL VARIABLES IN SOURCE PROGRAM
149	A TSS SOURCE LINE MUST HAVE LESS THAN 67 COL
150	A TSS HOL OR QUOTED STRING MUST BE ON 1 LINE
151	THIS CONSTRUCT IS ILLEGAL IN TSS FORTRAN
152	ILLEGAL FILE CARD PARAMETER VALUE
-	SEQUENCE ERROR "n" < "p", where n is the sequence number of the card image and p is the sequence number of the previous card image.

APPENDIX E

OBJECT TIME ERROR TERMINATION MESSAGES

The following object time error termination messages may be generated by FORTRAN compiled programs:

ARG .GT. MAX f (where f is CSIN or CCOS)

Imaginary component exceeds 158.

DATA STMT ERR

a. Too much or too little data for list.

b. Complex, double, or logical list element must correspond with complex, double, or logical data.

DIV BY ZERO ⟨job specifier⟩, ⟨terminal reference⟩

An object program performed a Divide operation using a zero denominator; processing of the subject program was discontinued.

⟨job specifier⟩ = ⟨mix index⟩ DS-ED

Processing of an object program was discontinued before End-of-Job; the EOJ option was set.

⟨compiler name⟩ / ⟨program identifier⟩ = ⟨mix index⟩ DS-ED

Compilation was discontinued before the compiler reached End-of-Job; the TYPE EOJ option was set.

EOF NO LABEL ⟨file designator⟩ : ⟨job specifier⟩, ⟨terminal reference⟩

An object program has reached the end of the designated file and has not specified what is to be done; processing of the program was discontinued.

EXPON OVRFLW ⟨job specifier⟩, ⟨terminal reference⟩

An object program has performed an operation which caused an exponent overflow to occur; processing of the program was discontinued.

FLAG BIT ⟨job specifier⟩, ⟨terminal reference⟩

An object program has performed an operation which caused a word with a flag bit of 1 to be accessed as if it were an operand; processing of the program was discontinued.

FRMT ERROR

a. Illegal character in format.

b. Unrecognizable format specification.

c. Required numeric field is not numeric.

d. Field width greater than 63.

e. Format specifies record longer than buffer.

INTGR OVRFLW ⟨job specifier⟩, ⟨terminal reference⟩

An object program performed an operation which caused an integer overflow to occur; processing of the program was discontinued.

INVALD ADRSS ⟨job specifier⟩, ⟨terminal reference⟩

An object program performed an operation which addressed a memory location in an absent memory module or an address less than 00512; processing of the program was discontinued.

INVALID ARG CONCAT

See appendix H.

INVALID EOJ

A STOP, CALL EXIT, or transfer of control statement was missing from the mainline program, and an attempt was made to execute an END statement.

LIST SIZE ERROR

The number of elements in the list of a READ statement exceeds the number of data items in the logical record.

NEGATIVE BASE XTOI

A**B, where A is negative and B is not an integer.

NEGTV ARGMNT LN ⟨program specifier⟩ ⟨terminal reference⟩

A negative argument has been passed to the intrinsic which computes the natural logarithm.

NEGTV ARGMNT SQRT ⟨program specifier⟩ ⟨terminal reference⟩

A negative argument has been passed to the SQRT intrinsic.

NMLST ERR

This error message can be generated during input only.

a. Illegal subscript on data card.
b. Too many or too few subscripts.
c. Illegal character encountered.
d. = missing.
e. , or * missing after a constant.
f. Repeat count not an integer constant.

OPRTR DS-ED ⟨job specifier⟩, ⟨terminal reference⟩

The system operator caused processing of a program to be discontinued through use of a DS message.

SELECT ERROR ⟨file designator⟩ : ⟨job specifier⟩, ⟨terminal reference⟩

An object program performed an invalid operation on the designated file, e.g., rewinding a card reader. Processing of the program was discontinued.

STACK OVRFLW ⟨job specifier⟩, ⟨terminal reference⟩

The operations performed by an object program have caused its stack to overflow its limit; processing of the program has been discontinued.

TYPE ERR

a. Exponent part in data contains non-digit after D, E, +, or - (input only).

b. The data read in using an I specification in a FORMAT statement is either:

1) Greater than the maximum integer allowed (549755813887).
2) Double, real, or alpha.

c. The list element using a D specification in a FORMAT statement is not double precision (output only).

d. The list element using an E, F, or G specification in a FORMAT statement is logical, integer, or double precision (output only).

e. The list element using an L specification in a FORMAT statement is not logical (output only).

ZERO ARGMNT LN ⟨program specifier⟩ ⟨terminal reference⟩

An argument of zero has been passed to the intrinsic which computes the natural logarithm.

ZERO MODULUS DMOD

DMOD(A,B), where B = 0.

ZIP ERROR - IGNORED

This message is typed if a program performs a generalized ZIP statement, but provides control information containing an error. Occurrence of this message signifies that the error was present and that all control information following and including the error was ignored.

NOTE

For further information, refer to the System Operation Manual.

APPENDIX F

BURROUGHS VERSUS USASI FORTRAN, EXTENSIONS AND DIFFERENCES

Those extensions and differences listed below are based on a comparison of Burroughs FORTRAN and USASI FORTRAN, as specified in the document ASA X3.9-1966.

EXTENSIONS PERMITTED IN BURROUGHS FORTRAN.

The following extensions are permitted in Burroughs FORTRAN.

a. More than one statement per card is allowed.

b. The character set includes the quote sign (").

c. The relational operators $<$, $\leq$, $\neq$, $>$, $\geq$ are allowed in place of their FORTRAN mnemonics.

d. Hollerith constants may be used in assignment statements.

e. Theoretically, there is no limit to the number of dimensions which can be declared for an array.

f. A subscript may be any integer or real arithmetic expression.

g. In the statement:

   GO TO i,(k_1,k_2,...,k_n)

   i may be an integer or real variable.

h. In the statement:

   GO TO(k_1,k_2,...,k_n),i

   i may be an integer or real arithmetic expression.

i. In the statement:

   IF(l.e.) s

   s may be any executable statement except a DO statement.

j. The terminal statement of a DO loop may be any executable statement, with any implications involved assumed to be understood by the programmer.

k. In the statement:

   DO m i=n_1,n_2,n_3

   i may be an integer or real simple variable.

   n_1,n_2,n_3 may be integer or real arithmetic expressions.

   n_1 and n_2 do not have to be greater than zero.

   i,n_1,n_2, and n_3 may be redefined within the range of the DO statement, with any implications involved assumed to be understood by the programmer.

l. CLOSE u; LOCK u; PURGE u.

m. In I/O and AUXILIARY I/O statements, u may be an arithmetic expression.

n. NAMELIST and NAMELIST I/O.

o. READ f,k; PUNCH f,k; PRINT f,k.

p. The intrinsics: TAN, COTAN, ARSIN, ARCOS, SINH, COSH, ERF, GAMMA, ALGAMA, AND, OR, COMPL, EQUIV, CONCAT, TIME, DIV, MOD.

q. Random disk I/O.

r. Action labels.

s. Non-standard returns from subroutines.

t. Multiple entry points to subprograms.

u. Each of the two components of a complex constant may be either real or integer.

v. Hollerith constants and literals may be enclosed in quotes.

w. The format specifications Ow and Tn.

x. The ability to have a subroutine recurse (call itself).

y. Labeled common blocks do not need to be the same length in all subprograms.

z. Variables can be preassigned values in both labeled common blocks and the unlabeled common region through the use of the BLOCK DATA subprogram.

DIFFERENCE FROM USASI FORTRAN.

In the statements STOP n and PAUSE n, n is blank or an integer constant of up to six digits.

APPENDIX G

COLLATING SEQUENCE

CODES

CHAR.	INTERNAL CODE			BCL CODE		CARD CODE	
	BA	8421	OCTAL CODE	BA	8421	ZONE	NUM.
Blank	11	0000	60	01	0000	-	-
.	01	1010	32	11	1011	12	8 - 3
[	01	1011	33	11	1100	12	8 - 4
(	01	1101	35	11	1101	12	8 - 5
<	01	1110	36	11	1110	12	8 - 6
←	01	1111	37	11	1111	12	8 - 7
&	01	1100	34	11	0000	12	-
$	10	1010	52	10	1011	11	8 - 3
*	10	1011	53	10	1100	11	8 - 4
)	10	1101	55	10	1101	11	8 - 5
;	10	1110	56	10	1110	11	8 - 6
≤	10	1111	57	10	1111	11	8 - 7
-	10	1100	54	10	0000	11	-
/	11	0001	61	01	0001	0	1
,	11	1010	72	01	1011	0	8 - 3
%	11	1011	73	01	1100	0	8 - 4
=	11	1101	75	01	1101	0	8 - 5
]	11	1110	76	01	1110	0	8 - 6
"	11	1111	77	01	1111	0	8 - 7
#	00	1010	12	00	1011	-	8 - 3
@	00	1011	13	00	1100	-	8 - 4
:	00	1101	15	00	1101	-	8 - 5
>	00	1110	16	00	1110	-	8 - 6
≥	00	1111	17	00	1111	-	8 - 7
+	01	0000	20	11	1010	12	0
A	01	0001	21	11	0001	12	1
B	01	0010	22	11	0010	12	2
C	01	0011	23	11	0011	12	3
D	01	0100	24	11	0100	12	4
E	01	0101	25	11	0101	12	5
F	01	0110	26	11	0110	12	6
G	01	0111	27	11	0111	12	7

LOW ← COLLATING SEQUENCE → HIGH

APPENDIC G (cont)
COLLATING SEQUENCE

CHAR.	INTERNAL CODE			BCL CODE		CARD CODE	
	BA	8421	OCTAL CODE	BA	8421	ZONE	NUM.
H	01	1000	30	11	1000	12	8
I	01	1001	31	11	1001	12	9
×	10	0000	40	10	1010	11	0
J	10	0001	41	10	0001	11	1
K	10	0010	42	10	0010	11	2
L	10	0011	43	10	0011	11	3
M	10	0100	44	10	0100	11	4
N	10	0101	45	10	0101	11	5
O	10	0110	46	10	0110	11	6
P	10	0111	47	10	0111	11	7
Q	10	1000	50	10	1000	11	8
R	10	1001	51	10	1001	11	9
≠	11	1100	74	01	1010	0	8 - 2
S	11	0010	62	01	0010	0	2
T	11	0011	63	01	0011	0	3
U	11	0100	64	01	0100	0	4
V	11	0101	65	01	0101	0	5
W	11	0110	66	01	0110	0	6
X	11	0111	67	01	0111	0	7
Y	11	1000	70	01	1000	0	8
Z	11	1001	71	01	1001	0	9
0	00	0000	00	00	1010	-	0
1	00	0001	01	00	0001	-	1
2	00	0010	02	00	0010	-	2
3	00	0011	03	00	0011	-	3
4	00	0100	04	00	0100	-	4
5	00	0101	05	00	0101	-	5
6	00	0110	06	00	0110	-	6
7	00	0111	07	00	0111	-	7
8	00	1000	10	00	1000	-	8
9	00	1001	11	00	1001	-	9
?	00	1100	14	00	0000	ALL OTHER CARD CODES	

LOW ↕ HIGH

APPENDIX H
BIT-MANIPULATION INTRINSICS

The FORTRAN compiler provides five intrinsics for use in bit manipulation and masking. It is assumed that the programmer who makes use of these intrinsics has a prior working knowledge of the system. All five of these intrinsics permit access to all but the left-most bit, bit zero, of a word.

AND.

This intrinsic logically ANDs bit numbers 1 through 47 of its two arguments. The arguments remain unchanged.

The general form is:

AND(A,B)
where A and B are real arithmetic expressions.

Examples:

A subscript of 8 indicates an octal number.

```
Y=AND(S,T)
```

S	T	Y
3777777777777777_8	1111111111111111_8	1111111111111111_8
1234567023456701_8	3210765432107654_8	1210565022006600_8

OR.

This intrinsic logically ORs bit numbers 1 through 47 of its two arguments. The arguments remain unchanged.

The general form is:

OR(A,B)
where A and B are real arithmetic expressions.

Examples:

A subscript of 8 indicates an octal number.

```
        Y=OR(S,T)
```

S	T	Y
3777777777777777_8	1111111111111111_8	3777777777777777_8
1234567012345670_8	3210765432107654_8	3234767432347674_8

COMPLEMENT.

This intrinsic returns the logical COMPLEMENT of its argument. The argument remains unchanged.

The general form is:

COMPL(A)
where A is an arithmetic expression.

Examples:

A subscript of 8 indicates an octal number.

```
        Y=COMPL(S)
```

S	Y
3777777777777777_8	0000000000000000_8
1234567012345670_8	2543210765432107_8

EQUIVALENCE.

This intrinsic logically EQUIVALENCEs its two arguments. The arguments remain unchanged.

The general form is:

EQUIV(A,B)
where A and B are real arithmetic expressions.

Examples:

Y = EQUIV(S,T)

S	T	Y
3777777777777777_8	1111111111111111_8	1111111111111111_8
1234567012345670_8	3210765432107654_8	1753575357535753_8

CONCAT.

The FORTRAN intrinsic CONCAT provides general bit-wise partial-word manipulation. CONCAT is a REAL FUNCTION of the form:

CONCAT (A,B,S1,S2,N)

where:

a. A and B are integer or real arithmetic expressions;

b. S1, S2, and N are integer arithmetic expressions;

c. $S1 > 0$;

d. $S2 > 0$;

e. $N > 0$;

f. $S1 + N \leq 48$;

g. $S2 + N \leq 48$.

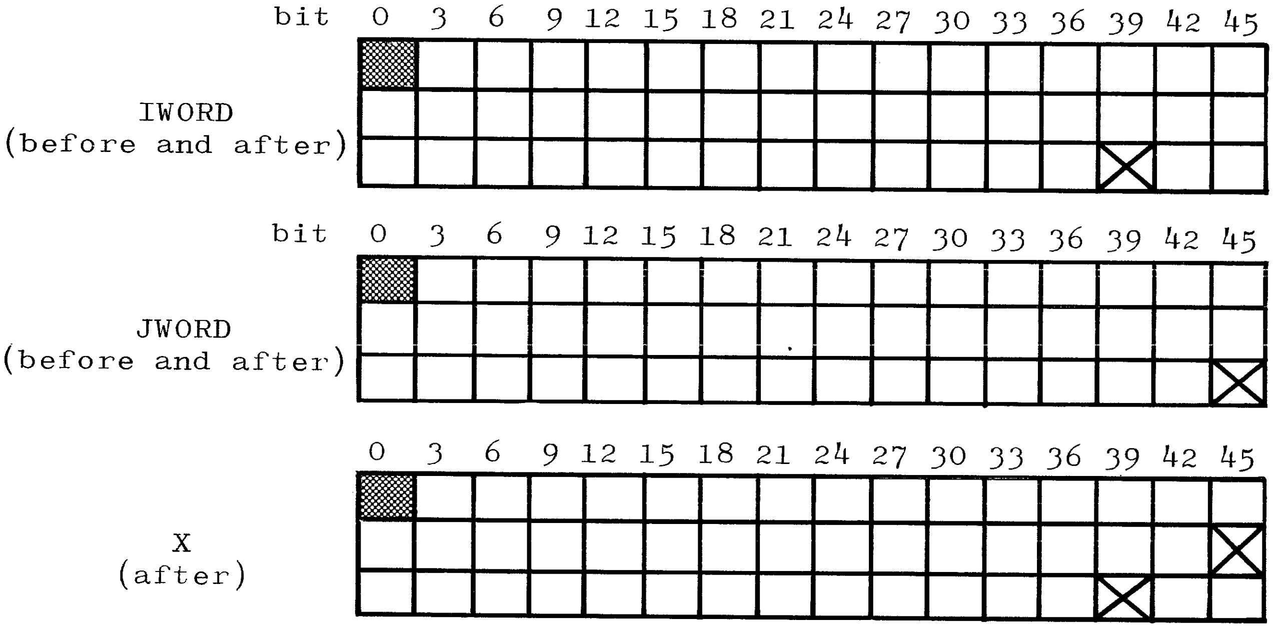
If any one of conditions (c) through (g) is not true, the object program will be discontinued with an INVALID ARG CONCAT message.

When this function is called, first bit S2 of B is transferred to bit S1 of A, then bit S2 + 1 of B is transferred to bit S1 + 1 of A, and so forth, until N bits have been transferred. In other words, starting with bit S2 of B and moving to the right, each successive bit is transferred to A, starting with bit S1 of A, until N bits have been transferred. A, B, S1, S2, and N remain unchanged after the operation unless they are to the left of the replacement operator (=) in the statement referencing CONCAT.

Although there are 48 bits in a word, numbered 0 through 47, bit number 0 cannot be accessed by using CONCAT.

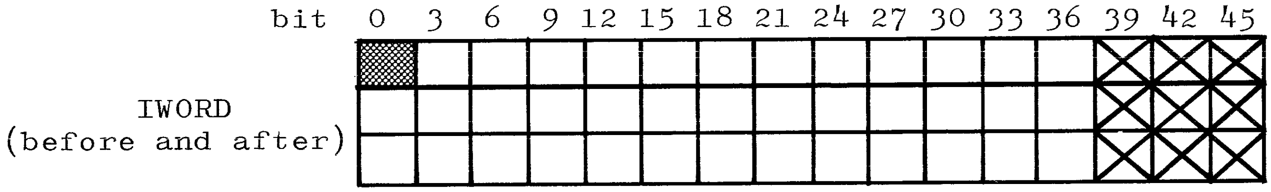
Example 1:

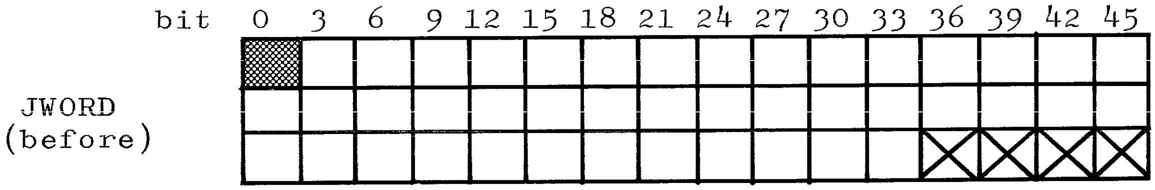
```
...
...
...
IWORD=64
JWORD=1
IBIT=46
JBIT=47
N=1
X=CONCAT(IWORD,JWORD,IBIT,JBIT,N)
...
...
...
```

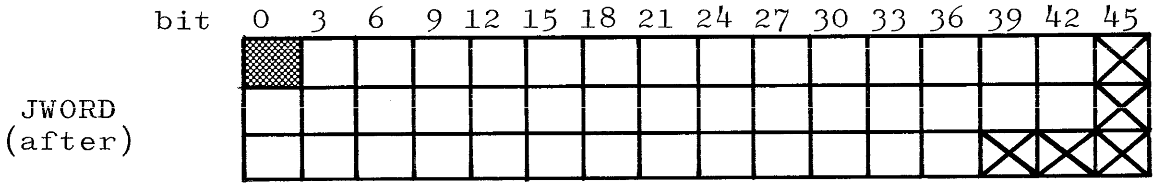


Example 2:

```
...
...
...
DATA IWORD,JWORD,IBIT,JBIT,N/Ø777,Ø1111,39,36,6/
...
...
JWORD=CONCAT(IWORD,JWORD,IBIT,JBIT,N)
```

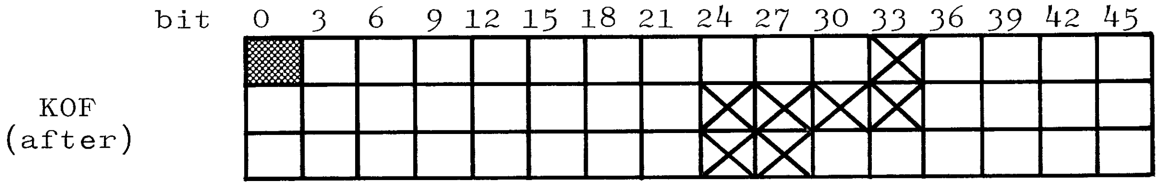






Example 3:

```
...
...
...
KOF=CONCAT(0,1750,24,36,12)
...
...
```



Example 4:

```
INTEGER BIT
...
...
A=24.0E+0
BIT=42
A=CONCAT(A,12,38,BIT,6)
...
...
...
```

bit	0	3	6	9	12	15	18	21	24	27	30	33	36	39	42	45
A (before)	▨															
															☒	
															☒	

bit	0	3	6	9	12	15	18	21	24	27	30	33	36	39	42	45
A (after)	▨															
														☒		
														☒	☒	

APPENDIX I

PRT CONTENTS OF A FORTRAN OBJECT PROGRAM

R + 0	EEEEEEEE
1	Used by .LABEL.
2	500000000
3	FPB
4	SD
5	BC
6	AIT
7	MSCW
10	INCW
11	COM/PRL
12	R + 0, Stack
13	OWNARRAY description
14	ALGOL WRITE
15	ALGOL READ
16	ALGOL FILE CONTROL
17	0
20	BLOCKCTR
21	JUNK
22	BASENSIZE
23	LISTRTN
24	CLASN
25	HOLTOG
26	Powers of Ten
27	21 word ARRAY for any formatted output and for use by ZIP
30	ERR
31	SQRT
32	ARSIN
33	EXP
34	SIN
35	ALOG
36	TAN
37	ATAN
40	GAMMA
41	DATAN
42	DCOS
43	DSIN
44	ATAN2
45	CABS
46	DMOD
47	DEXP
50	DSQRT

APPENDIX J
TIME-SHARING FORTRAN

This appendix describes capabilities of FORTRAN compilers equipped for time-sharing.

For Source Programs compiled/executed from the card reader:

$TSSEDIT Option.
This option causes the compiler to print diagnostic messages whenever a source statement contains a construct(s) which would elicit a syntax error if the source statement were compiled from the terminal. These constructs are:

a. The PAUSE statement.

b. Formal subprograms.

c. Hollerith or quoted strings extended from one line to another.

Also, the ZIP construct is not reserved in Time-Sharing FORTRAN.

If the $NEW TAPE option accompanies the $TSSEDIT option, the format of the new symbolic file will be the "REMOTE FREE FIELD" format (see page J-4). This Source File Editing option, when used with a disk file label-equate, allows the user to automatically prepare a card or tape source program or subprogram for remote terminal usage.

$SEQXEQ Option.
A program compiled under this option will cause Run-Time terminating errors (e.g., Div by Zero, Invalid Index) to reference the sequence number of the source line containing the terminating construct, instead of its segment/address. Note that this option has meaning only if it precedes the first executable statement. It may not be reset.

File Unit REMOTE.
See page J-2 for a description of the File Unit REMOTE.

If in an object program an I/O statement using a REMOTE File Unit is executed, the program will be DR-ED with a DCTU NOT ASSIGNED message. The REMOTE Unit designation has legitimate meaning only for programs executed from the terminal. (Of course, the REMOTE unit designation may be used in source programs compiled to the library from a card reader.)

For Source Programs compiled/executed from the terminal:

$TSSEDIT Option.
This option causes the compiler to consider the format of the source file to be the ordinary restricted field format. The option is treated as any other $OPTION, and must be renewed on each $ Control Card (if renewal is desired). This option may be thought of as TURNING OFF the Remote Free-Field format. Note, however, that the source line may not exceed 66 columns.

$ERRMES Option.
Ordinarily, compilation syntax errors elicit the message "ERR#nnn @ mmmmmmmm: nnnnnn", where nnn is the error number, mmmmmmmm is the line number (see $SEQXEQ Option, page J-1), the nnnnnn is the erroneous construct; use of this option will cause the compiler to print a description of the error next to nnnnnn.

$LIST Option.
This is the familiar $LIST option. However, unlike card/tape input compilations, this option is initially set to OFF when compiling from a terminal. Use of the option from a terminal provides a line printer listing of the compilation, and each listed line of terminal file will be flagged with a D in column 82.

File Unit REMOTE.
Use in a source program of a file card containing UNIT=REMOTE will cause relevant I/O statements to reference the terminal from which the source program was compiled and/or executed. No buffer or blocking information is required and, if provided, it will be ignored.

New Meaning for the PRINT Statement and the READ Statement Without an Input Unit.

Ordinarily, the PRINT statement references the line printer, and the READ statement without a unit designator references a file labeled READER. In Time-Sharing FORTRAN, however, source programs compiled from the terminal will by default have their PRINT and READ statements reference the terminal from which the associated object program is executed. If this effect is undesired, use of the new option $ONSITE at compile time restores the statement references to their ordinary meaning.

Use of the new option $REMOTE causes source programs compiled from the card reader to have their PRINT and READ statements reference the terminal from which the associated object program is executed.

Note that using $REMOTE from the terminal merely re-initiates the (unordinary) default file references, and that using $ONSITE from the card reader merely re-initiates the ordinary file reference.

Also, note that these two new $ options have meaning only if they precede the first executable statement, and they will be ignored if placed anywhere else.

General Observations.
FORTRAN programs written, compiled, and executed from the terminal will normally:

a. Be written in REMOTE FREE-FIELD format (see page J-4).

b. Not contain constructs indicated on page J-1 under $TSSEDIT option).

c. Have I/O statements with a Unit Number n, where n is defined in a file card of the form:

```
FILEbbn=FILEN,UNIT=REMOTE
```

or use the PRINT and READ statements (which by default reference the user's terminal).

d. Have their compilation and execution errors printed on the terminal and referencing a line number.

e. Not produce any line printer compiler output.

FORTRAN REMOTE FREE-FIELD FORMAT (Time-Sharing).
Ordinarily, FORTRAN retains special meaning for columns 1, 1-5, 6, and 7-72. However, in Time-Sharing FORTRAN, these special meanings are abandoned, and the following conventions are used (assuming that $TSSEDIT is not used, see page J-2). Note that column 1 refers to the first column following the sequence number.

a. Continuation cards contain a minus (-) in column 1, and the card text starts in the first non-blank column or in column 7, whichever comes first.

b. Comment cards contain a C in column 1, a minus (-) in column 2, and the comment starts in column 3.

c. Labels may be a maximum of five columns long and may contain imbedded blanks (imbedded blanks do not contribute to the value of the label but do count in the five column limitation). A non-blank non-numeric character, or the seventh column after the start of the label, ends the label and starts the card text. A label may be separated from the sequence number by any number of blanks. For example, (b represents blank):

```
SEQ#
1000bbb...bb1A=B
2000bbb...bb1b2b3bA=B
3000bbb...bb1b3bbbbA=B
4000bbb...bb12345A=B
```

In the above examples, the underlined character is the first character of the card text, and the labels have, respectively, the values 1, 123, 13, and 12345.

d. File cards must start in column 1, therefore, the word FILEbb starting in column 1 is reserved in REMOTE FREE-FIELD format.

e. Dollar-sign control cards contain a $ in column 1 and control information in columns 2 through 72.

f. For all other cards, the card text starts with the first non-blank character.

g. Only 66 columns of card text (see paragraphs a, c, and f above) are allowed. Additional text will elicit syntax error #149 (A TSS SOURCE LINE MUST HAVE LESS THAN 67 COLS.).

FORTRAN LIMITED FREE-FIELD READ.

The characteristics of a limited free-field READ in the compiler is as follows:

a. The free-field READ is indicated by a slash (/), and replaces the format statement number in the READ statement.

b. All blanks in the data field are ignored.

c. Only numerical or logical statements are acceptable to the compiler.

d. Commas (,) are used as delimiters.

e. "TRUE" or 1, and "FALSE" or 0 represent that particular condition in a logical data field.

f. Double precision is permitted, but only the most significant half is transmitted. This means that double precision is treated as a single precision constant.

g. If a parity branch appears in a READ statement, a type error will cause that branch to be taken.

h. The free-field READ condition is terminated when either a list is exhausted, or an End-of-File is encountered.

Examples:

```
READ /, A, B, C, D
5.4, bb3.572, .95E+03, 5.39E-04

LOGICAL X,Y
READ /, C, D, X, Y
4.9, .039, "TRUE", 1, 5

LOGICAL A, B, C, D
READ (5, /) A, B, C, D
0, "TRUE", "FALSE", 1

READ(1,/, END=50, ERR=6) C, D
5.4E+06, .095
```

INDEX

INDEX (cont)

INDEX (cont)

INDEX (cont)

BURROUGHS CORPORATION
DATA PROCESSING PUBLICATIONS
REMARKS FORM

TITLE: B 5700 Information Processing Systems
FORTRAN Compiler
Reference Manual

FORM: 1051182
DATE: 1-71

CHECK TYPE OF SUGGESTION:

□ ADDITION □ DELETION □ REVISION □ ERROR

GENERAL COMMENTS AND/OR SUGGESTIONS FOR IMPROVEMENT OF PUBLICATION:

FROM: NAME ____________________ DATE ____________
TITLE ____________________
COMPANY ____________________
ADDRESS ____________________
____

cut along ’ted line

STAPLE

FOLD DOWN SECOND FOLD DOWN

Postage
Will Be Paid
by
Addressee

No
Postage Stamp
Necessary
If Mailed in the
United States

BUSINESS REPLY MAIL
First Class Permit No. 817, Detroit, Mich. 48232

Burroughs Corporation
6071 Second Avenue
Detroit, Michigan 48232

attn: Sales Technical Services
Systems Documentation

FOLD UP FIRST FOLD UP

Wherever There's
Business There's

Burroughs

1-71

Printed in U.S. America