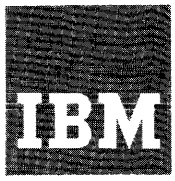
File No. 7090-25
Form C28-6274-2



Systems Reference Library

IBM 7090/7094 Programming Systems

FORTRAN IV Language

This publication contains a description of the 7090/7094 FORTRAN IV language that is processed by the FORTRAN IV Compiler, a 7090/7094 IBJOB Processor Component. FORTRAN is an automatic coding system designed primarily for scientific and engineering computations. The language closely resembles the language of mathematics, and includes various types of arithmetic, control, input/output, and specification statements.

PREFACE

This publication is intended to provide a reference to the language of 7090/7094 FORTRAN IV (IBFTC), which is a part of the 7090/7094 IBJOB Processor. The other parts of the IBJOB Processor are described in IBM 7090/7094 Programming Systems: IBJOB Processor, Form C28-6275, and MAP (Macro Assembly Program) Language, Form C28-6311.

7090/7094 FORTRAN IV (hereafter referred to as FORTRAN) is a language component of IBJOB. The basic concepts of a FORTRAN language are discussed in the FORTRAN general information manual, Form F28-8074-1. The level of information on the following pages is such that the reader must be thoroughly familiar with the information in the FORTRAN general information manual.

FORTRAN is under the control of the IBJOB Monitor. See the IBJOB Processor publication, referenced above, for a description of the minimum machine requirements for FORTRAN IV.

MINOR REVISION (February 1964)
This publication is a reprint of Form C28-6274-1, incorporating changes released in Technical Newsletter N28-0069. Form C28-6274-1 and Form N28-0069 are not obsolete.

Copies of this and other IBM publications can be obtained through IBM Branch Offices.
Address comments concerning the contents of this publication to:
IBM Corporation, Programming Systems Publications, Dept. D91, PO Box 390, Poughkeepsie, N.Y., 12602

Form C28-6274-2
Page Revised 6/10/64
by TNL N28-0107

CONTENTS

CHAPTER 1: GENERAL PROPERTIES OF A FORTRAN SOURCE PROGRAM 5
Writing the Source Program 5
Punching the Source Program 5
The FORTRAN Statements 5

CHAPTER 2: CONSTANTS, VARIABLES, SUBSCRIPTS, AND EXPRESSIONS 7
Constants 7
Variables 8
Subscripts 8
Expressions 9

CHAPTER 3: THE ARITHMETIC STATEMENT 11

CHAPTER 4: THE CONTROL STATEMENTS 12
The Unconditional GO TO Statement 12
The Computed GO TO Statement 12
The Assigned GO TO Statement 12
The ASSIGN Statement 12
The Arithmetic IF Statement 12
The Logical IF Statement 12
The DO Statement 13
The CONTINUE Statement 13
The PAUSE Statement 14
The END Statement 14
The STOP Statement 14
The RETURN Statement 14

CHAPTER 5: INPUT/OUTPUT STATEMENTS 15
List Specifications 15
Input/Output of Entire Arrays 16
FORMAT Statement 16
Numeric Fields 16
Complex Number Fields 17
Alphameric Fields 17
Logical Fields 18
Blank Fields - X-Conversion 18
Repetition of Field Format 18
Repetition of Groups 18
Scale Factors 18
Multiple-Record Formats 19
Carriage Control 19
FORMAT Statements Read In at Object Time 19
Data Input Referencing a FORMAT Statement 19
NAMELIST Statement 20
Data Input Referencing a NAMELIST Statement 20
The General Input/Output Statements 21
Input 21
Output 21.1
The Manipulative Input/Output Statements 21.1
Symbolic Input/Output Unit Designation 21.2

CHAPTER 6: SUBROUTINES, FUNCTIONS, AND SUBPROGRAM STATEMENTS 22
Naming Subroutines 22
Defining Subroutines 22
Arithmetic Statement Functions 22
Built-In Functions 22
FUNCTION Subprogram 24
SUBROUTINE Subprogram 24
Subprogram Names as Arguments 25
The CALL Statement 26
Subprograms Provided by FORTRAN 26
The BLOCK DATA Subprogram 27

CHAPTER 7: THE SPECIFICATION STATEMENTS 28
The DIMENSION Statement 28
The COMMON Statement 28
The EQUIVALENCE Statement 29
The TYPE Statements 30
The DATA Statement 30

APPENDIXES 32
Appendix A. Source Program Statements and Sequencing . . 32
Appendix B. Table of Source Program Characters 33
Appendix C. Differences Between FORTRAN II and FORTRAN IV 33
Appendix D. Limitations on Source Program Size 35
Appendix E. Optimization of Arithmetic Expressions . . . 37

INDEX 38

WRITING THE SOURCE PROGRAM

The statements of a FORTRAN source program are normally written on a standard FORTRAN coding sheet. An example of a FORTRAN source program is given in Figure 1-1; the purpose of this program is to determine the largest value attained by a set of numbers, A_i.

1. Columns 1-5 of the first line of a statement may contain a statement number that is less than 32,768, to identify the statement. Blanks and leading zeros are ignored in these columns.
2. Column 6 of the first line of a statement must be a blank or a zero.
3. Columns 7-72 contain the actual FORTRAN statement. Blanks are ignored except in an H field of a FORMAT or alphameric argument or a DATA statement.
4. A statement may be continued over as many as nineteen continuation cards. Any card with a non-blank, non-zero column 6 is considered a continuation card.
5. Cards with a C in column 1 are not processed by FORTRAN, and columns 2-72 may be used for comments.
6. Columns 73-80 are not processed by FORTRAN and may be used for identification.
7. The order of execution of the source statements is governed by the normal source program statement sequencing given in Appendix A.

PUNCHING THE SOURCE PROGRAM

The FORTRAN statements, prepared as above, are punched into the standard FORTRAN card in Figure 1-2 for input to the computer.

THE FORTRAN STATEMENTS

The FORTRAN statements may be divided into the following groups:

1. The arithmetic statement specifies a numerical or logical calculation.
2. The control statements govern the flow of control in the program.
3. The input/output statements provide the necessary input/output routines and the input/output format.
4. The subprogram statements enable the programmer to define and use subprograms.
5. The specification statements provide information about the constants and variables used in the program and provide information about storage allocation.

IBM

FORTRAN CODING FORM

Form X28-7327-3
Printed in U.S.A.

Program _____
Coded By _____
Checked By _____

Date _____
Page _____ of _____

Identification (73–80)

C FOR COMMENT

STATEMENT NUMBER (1–5)	Cont. (6)	FORTRAN STATEMENT (7–72)
C		PROGRAM FOR FINDING THE LARGEST VALUE
C		ATTAINED BY A SET OF NUMBERS
		DIMENSION A(999)
		READ (5,11) N,(A(I),I=1,N)
1		FORMAT (I3/(12F6.2))
		BIGA=A(1)
		DO 20 I=2,N
		IF (BIGA-A(I))10,20,20
10		BIGA=A(I)
20		CONTINUE
		WRITE (6,12) N,BIGA
2		FORMAT (2H1THE LARGEST OF THESE,I3,12H NUMBERS IS,F7.2)
		STOP
		END

Figure 1-1

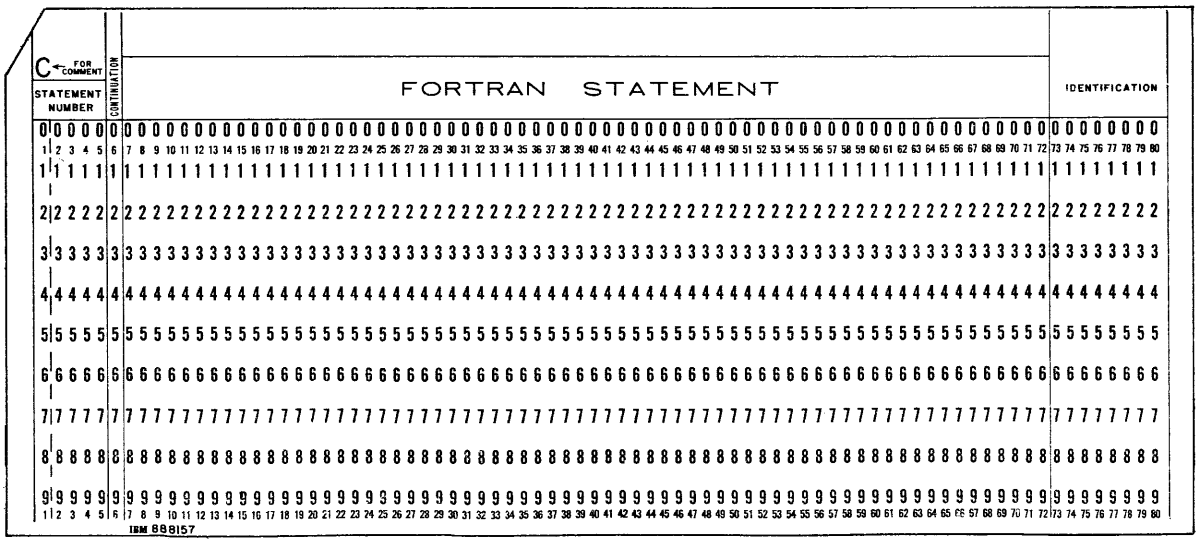


Figure 1-2

FORTRAN provides a means of expressing constants, variables, and a subscript notation for expressing one-, two-, and three-dimensional arrays of variables.

CONSTANTS

Five types of constants are permitted in a FORTRAN source program: integer or fixed point, real or single-precision floating point, double-precision floating point, complex, and logical.

Integer Constants

General Form
An integer constant consists of 1-11 decimal digits written without a decimal point.

Examples:

3
528
8085

An integer constant may be as large as $2^{35}-1$, except when used for the value of a subscript or as an index of a DO, in which case the value of the integer is computed modulo 2^{15}.

Real Constants

General Form
1. A real constant consists of 1-9 significant decimal digits written with a decimal point. 2. A real constant may be followed by a decimal exponent, which is written as the letter E followed by an integer constant. The field following the letter E must not be blank; it may be zero.

Examples:

21.
.203
8.0067
5.0 E3 (means 5.0×10^3, i.e., 5000.)
5.0 E-3 (means 5.0×10^{-3}, i.e., .005)

1. The magnitude of a real constant must be between the approximate limits of 10^{38} and 10^{-38}, or be zero.
2. A real constant has precision to 8 digits.

Double-Precision Constants

General Form
1. A double-precision constant consists of 1-17 significant decimal digits written with a decimal point. 2. If it is desired to specify a decimal exponent, or if the constant contains fewer than 10 digits, the letter D, followed by the exponent, must follow the number. The exponent is an integer constant. The field following the letter D must not be blank; it may be a zero.

Examples:

21.987538294
21.9D0
.203D0
5.0D3 (means 5.0×10^3, i.e., 5000.)
5.0D-3 (means 5.0×10^{-3}, i.e., .005)

1. The magnitude of a double-precision constant must lie between the approximate limits of 10^{-29} and 10^{38}, or be zero. Numbers between 10^{-29} and 10^{-38} may be used, but only eight digits are significant in this range.
2. Double-precision constants are floating point quantities that have precision to 16 digits.

Complex Constants

General Form
A complex constant consists of an ordered pair of real constants separated by a comma and enclosed in parentheses.

Examples:

(3.2, 1.86) is equal to 3.2+1.86i.
(2.1, 0.0) is equal to 2.1 +0.0i.
(5.0E3, 2.12) is equal to 5000. +2.12i.

1. The first real constant represents the real part of the complex number, and the second real constant represents the imaginary part of the complex number.
2. The parentheses are required regardless of the context in which the complex constant appears.
3. Each part of the complex constant may be preceded by a + or - sign.

Logical Constants

General Form
A logical constant may take either of the following forms: .TRUE. .FALSE.

VARIABLES

A variable is specified by its name and its type. There are five types of variables: integer, real, double-precision, complex, and logical.

Variable Names

General Form
A variable name consists of 1-6 alphameric characters, the first of which must be alphabetic.

Examples:

L5
JOB1
BETATS
COST
K

Subroutines are named in the same manner as variables (see "Naming Subroutines").

Variable Type Specification

The type of a real or integer, variable or function name may be specified in one of two ways: implicitly by name, or explicitly by a type statement (see the sections, "Type Statements" and "Naming Subroutines"). All other variables must have their type specified by a type statement.

Implicit Type Assignment

Implicit type assignment pertains only to integer and real, variable and function names, as follows:

1. If the first character of the symbol is I, J, K, L, M, or N, it is an integer name; e.g., MAX, JOB, IDIST, LESL.
2. If the first character of the symbol is not I, J, K, L, M, or N, it is a real name; e.g., ALPHA, BMAX, Q, WHIT.

SUBSCRIPTS

A variable may be made to represent any element of a one-, two-, or three-dimensional array of quantities by appending one, two, or three subscripts, respectively, to the variable name. The variable is then a subscripted variable. The subscripts are expressions of a special form whose value determines the member of the array to which reference is made.

Form of Subscripts

General Form
A subscript may take only one of the following forms, where v represents any unsigned, non-subscripted integer variable, and c and c' represent any unsigned integer constant: v c v+c v-c c*v c*v+c' or c*v-c'

Examples:

IMAS
J9
K2
N+3
8*IQUAN
5*L+7
4*M-3
7+2*K invalid
9+J invalid

Subscripted Variables

General Form
A subscripted variable consists of a variable name followed by parentheses enclosing one, two, or three subscripts that are separated by commas.

Examples:

A(I)
K(3)
BETA (8*J + 2, K-2, L)
MAX (I, J, K)

1. During execution, the subscript is evaluated so that the subscripted variable refers to a specific member of the array.
2. Each variable that appears in subscripted form must have the size of the array specified preceding the first appearance of the subscripted variable in an executable statement or DATA statement. This must be done by a DIMENSION or COMMON, with dimensions, statement.

Arrangement of Arrays in Storage

Arrays are stored in column order in increasing storage locations, with the first of their subscripts varying most rapidly and the last varying least rapidly.

Example: The 2-dimensional array $A_{m,n}$ is stored as follows, from lowest core storage location to highest:

$A_{1,1}, A_{2,1}, \ldots, A_{m,1}, A_{1,2}, A_{2,2}, \ldots, A_{m,2}, \ldots, A_{m,n}$

EXPRESSIONS

The FORTRAN language includes two kinds of expressions: arithmetic and logical.

Arithmetic Expressions

An arithmetic expression consists of certain sequences of constants, subscripted and nonsubscripted variables, and arithmetic function references separated by arithmetic operation symbols, commas, and parentheses.

The arithmetic operation symbols +, -, *, /, ** denote addition, subtraction, multiplication, division, and exponentiation, respectively.

The following are the rules for constructing arithmetic expressions:

1. Figures 2-1 and 2-2 indicate which constants, variables, and functions may be combined by the arithmetic operators to form arithmetic expressions. Figure 2-1 gives the valid combinations with respect to the arithmetic operators +, -, *, and /. Figure 2-2 gives the valid combinations with respect to the arithmetic operator **. In these figures, Y indicates a valid combination and N indicates an invalid combination.

+, -, *, /	Real	Integer	Complex	Double-Precision	Logical
Real	Y	N	Y	Y	N
Integer	N	Y	N	N	N
Complex	Y	N	Y	N	N
Double-Precision	Y	N	N	Y	N
Logical	N	N	N	N	N

Figure 2-1

		Exponent				
	**	Real	Integer	Complex	Double-Precision	Logical
Base	Real	Y	Y	N	Y	N
	Integer	N	Y	N	N	N
	Complex	N	Y	N	N	N
	Double-Precision	Y	Y	N	Y	N
	Logical	N	N	N	N	N

Figure 2-2

2. A real constant, variable, or function name combined with a double-word quantity results in an expression with the type of the double-word quantity; e.g., a real variable plus a complex variable forms a complex expression.

3. Any expression may be enclosed in parentheses.

4. Expressions may be connected by the arithmetic operation symbols to form other expressions, provided that:

   a. No two operators appear in sequence, and
   b. No operation symbol is assumed to be present.

The expression A**B**C is not permitted; it must be written as either A**(B**C) or (A**B)**C, whichever is intended.

5. Preceding an expression by a + or - sign does not affect the type of the expression.

6. Hierarchy of Operations. Parentheses may be used in expressions to specify the order in which operations are to be computed. Where parentheses are omitted, the order is understood to be as follows (from innermost operations to outermost operations):

   a. Function Reference
   b. ** Exponentiation
   c. * and / Multiplication and Division
   d. + and - Addition and Subtraction

For example, the expression A+B/C-D**E*F-G will be taken to mean A+ (B/C) - (D^{E}*F)-G.

Logical Expressions

A logical expression consists of certain sequences of logical constants, logical variables, references to logical functions, and arithmetic expressions (except complex expressions) separated by logical operation symbols or relational operation symbols. A logical expression always has the value .TRUE. or .FALSE..

The logical operation symbols (where a and b are logical expressions) are:

Symbol	Definition
.NOT. a	This has the value .TRUE. only if a is .FALSE.; it has the value .FALSE. only if a is .TRUE.
a.AND.b	This has the value .TRUE. only if a and b are both .TRUE.; it has the value .FALSE. if either a or b is .FALSE.
a.OR.b	(Inclusive OR) This has the value .TRUE. if either a or b is .TRUE.; it has the value .FALSE. only if both a and b are .FALSE.

The logical operators NOT, AND, and OR must always be preceded and followed by a period.

The relational operation symbols are:

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

The relational operators must always be preceded and followed by a period.

The following are the rules for constructing logical expressions:

1. Figure 2-3 indicates which constants, variables, and functions may be combined by the relational operators to form a logical expression. In this figure, Y indicates a valid combination and N indicates an invalid combination.

The logical expression will have the value .TRUE. if the condition expressed by the relational operator is met; otherwise, the logical expression will have the value .FALSE..

2. A logical expression may consist of a single logical constant, a logical variable, or a reference to a logical function.

.GT.,.GE.,.LT., .LE.,.EQ.,.NE.	Real	Integer	Complex	Double-Precision	Logical
Real	Y	N	N	Y	N
Integer	N	Y	N	N	N
Complex	N	N	N	N	N
Double-Precision	Y	N	N	Y	N
Logical	N	N	N	N	N

Figure 2-3

3. The logical operator .NOT. must be followed by a logical expression, and the logical operators .AND. and .OR. must be preceded and followed by logical expressions to form more complex logical expressions.

4. Any logical expression may be enclosed in parentheses; however, the logical expression to which the .NOT. applies must be enclosed in parentheses if it contains two or more quantities.

5. Hierarchy of Operations. Parentheses may be used in logical expressions to specify the order in which operations are to be computed. Where parentheses are omitted, the order is understood to be as follows (from innermost operation to outermost operation):

Function Reference	
**	Exponentiation
* and /	Multiplication and Division
+ and -	Addition and Subtraction
.LT.,.LE.,.EQ.,.NE.,.GT.,.GE.	
.NOT.	
.AND.	
.OR.	

The arithmetic statement defines a numerical or logical calculation. A FORTRAN arithmetic statement closely resembles a conventional arithmetic formula; however, the equal sign of the FORTRAN statement specifies replacement rather than equivalence.

General Form
a=b where: 1. a is a real, integer, double-precision, complex, or logical, subscripted or non-subscripted variable, and 2. b is an expression.

Examples:

```
Q1 = K
A(I) = B(I) + ASIN(C(I))
V = .TRUE.
E = C.GT.D.AND.F.LE.G
```

Figure 3-1 indicates which type expressions may be equated to which type of variable in an arithmetic statement. In this figure, Y indicates a valid statement and N indicates an invalid statement.

Right side of equal sign

Left side of equal sign: variable \ expression	Real	Integer	Complex	Double-Precision	Logical
Real	Y	Y	N	Y	N
Integer	Y	Y	N	Y	N
Complex	N	N	Y	N	N
Double-Precision	Y	Y	N	Y	N
Logical	N	N	N	N	Y

Figure 3-1

In the following examples of arithmetic statements, I is an integer variable, A and B are real variables, C and D are double-precision variables, E and F are complex variables, and G, H, and P are logical variables.

A = B	Replace A by the current value of B.
I = B	Truncate B to an integer, convert it to an integer constant, and store it in I.
A = I	Convert I to a real variable and store it in A.
I = I + 1	Add 1 to I and store it in I.
A = 3*B	Not permitted. The expression is mixed for multiplication, i.e., it contains both a real variable and an integer constant.
A = B*C	Multiply B by C using double-precision arithmetic, and store the most significant part of the result as a real number in A.
E = F*(3.7,2.0)	Multiply F by 3.7 + 2.0i using complex arithmetic, and store the result in E as a complex number.
G = .TRUE.	Store the logical constant .TRUE. in G.
H = .NOT.G	If G is .TRUE., store the value .FALSE. in H; if G is .FALSE., store the value .TRUE. in H.
H = I.GE.A	Not permitted. An integer and a real variable may not be joined by a relational operator.
G = H.OR..NOT.P	(see table below)
G = 3..GT.B	G is .TRUE. if 3. is greater than B; G is .FALSE. otherwise.

H	P	~P	Hv~P
T	T	F	T
T	F	T	T
F	T	F	F
F	F	T	T

where: ~ implies .NOT., and v implies .OR.

The last two examples above illustrate the following rules:

1. Two logical operators may appear in sequence, not separated by a comma or parentheses, only if the second logical operator is .NOT. .
2. Two decimal points may appear in succession as in item 1 or when one belongs to a constant and the other to a relational operator.

CHAPTER 4: THE CONTROL STATEMENTS

The control statements enable the programmer to control and terminate the flow of his program.

The Unconditional GO TO Statement

General Form
GO TO n where: n is a statement number.

Example:

```
GO TO 25
```

This statement causes control to be transferred to the statement numbered n.

The Computed GO TO Statement

General Form
GO TO (n_1, n_2,...,n_m),i where: 1. n_1, n_2,...,n_m are statement numbers, and 2. i is a non-subscripted integer variable.

Example:

```
GO TO (30,45,50,9),K
```

This statement causes control to be transferred to the statement numbered $n_1, n_2, \ldots, n_m$ depending on whether the value of i is $1, 2, 3, .., m$, respectively, at the time of execution. Thus, in the example, if K is 3 at the time of execution, a transfer to the third statement in the list, i.e., statement 50, will occur.

The Assigned GO TO Statement

General Form
GO TO n, (n_1, n_2,...,n_m) where: 1. n is a non-subscripted integer variable appearing in a previously executed ASSIGN statement, and 2. $n_1, n_2, \ldots, n_m$ are statement numbers.

Example:

```
GO TO J, (17,12,19)
```

This statement causes control to be transferred to the statement number last assigned to n by an ASSIGN statement; $n_1, n_2, \ldots, n_m$ is a list of the m values that n may assume.

The ASSIGN Statement

General Form
ASSIGN i TO n where: 1. i is a statement number, and 2. n is a non-subscripted integer variable that appears in an assigned GO TO statement.

Examples:

```
ASSIGN 12 TO K
ASSIGN 37 TO JA
```

This statement causes a subsequent GO TO n, (n_1, $n_2, \ldots, n_m$) to transfer control to the statement numbered i, where i is one of the statement numbers included in the series $n_1, n_2, \ldots, n_m$.

The Arithmetic IF Statement

General Form
IF (a) n_1, n_2, n_3 where: 1. a is an arithmetic expression (not complex), and 2. n_1, n_2, n_3 are statement numbers.

Examples:

```
IF (A(J,K) - B) 10,4,30
IF (D*E+BRN) 9,9,15
```

This statement causes control to be transferred to the statement numbered n_1, n_2, or n_3 if the value of a is less than, equal to, or greater than zero, respectively.

The Logical IF Statement

General Form
IF (t) s where: 1. t is a logical expression, and 2. s is any executable statement except DO or another logical IF.

Examples:

```
IF (A.AND.B) F = SIN (R)
IF (16.GT.L) GO TO 24
IF (D.OR.X.LE.Y) GO TO (18,20),I
IF (Q) CALL SUB
```

1. If the logical expression t is true, statement s is executed. Control is then transferred to

the next sequential statement unless s is an arithmetic IF-or GO TO-type statement, in which case, control is transferred as indicated.

2. If t is false, control is transferred to the next sequential statement.
3. If t is true and s is a CALL statement, upon return from the subprogram control is transferred to the next sequential statement.

The DO Statement

General Form
DO n i = m_1, m_2, m_3 where: 1. n is a statement number, 2. i is a non-subscripted integer variable, and 3. m_1, m_2, m_3 are each either an unsigned integer constant or a non-subscripted integer variable; if m_3 is not stated, it is taken to be 1.

Examples:

DO 30 I = 1, M, 2

DO 24 I = 1, 10

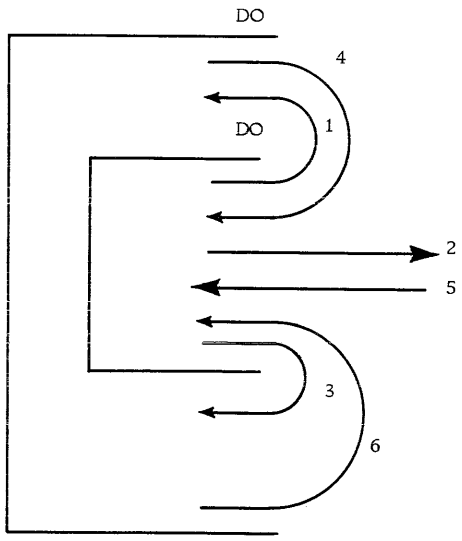
The DO statement is a command to execute repeatedly the statements that follow, up to and including the statement numbered n. The statements in the range of the DO are executed repeatedly with i equal to m_1, then i equal to $m_1 + m_3$, then i equal to $m_1 + 2m_3$, etc., until i is equal to the highest value in this sequence that does not exceed m_2. The statements in the range of the DO will be executed at least once.

1. The range of a DO is that set of statements that will be executed repeatedly; i.e., it is the sequence of consecutive statements immediately following the DO statement, up to and including the statement numbered n. After the last execution of the range, the DO is said to be satisfied.

2. The index of a DO is the integer variable i. Throughout the range of the DO, the index is available for computation, either as an ordinary integer variable or as the variable of a subscript. Upon exiting from a DO by satisfying the DO, the index i must be redefined before it is used in computation. Upon exiting from a DO by transferring out of the range of the DO, the index i is available for computation and is equal to the last value it attained.

3. DOs within DOs. Among the statements in the range of a DO may be other DO statements; such a configuration is called a nest of DOs. If the range of a DO includes another DO, then all of the statements in the range of the latter must also be in the range of the former.

4. Transfer of Control and DOs. Control may not be transferred into the range of a DO from outside its range. Thus, in the configuration below 1, 2, and 3 are permitted transfers, but 4, 5, and 6 are not.



5. Restrictions on Statements in the Range of a DO
   a. Any statement that redefines the index or any of the indexing parameters (m's) is not permitted in the range of a DO.
   b. The range of a DO cannot end with an arithmetic IF or GO TO-type statement or with a non-executable statement. The range of a DO may end with a logical IF, in which case, control is handled as follows: if the logical expression t is false, the DO is reiterated; if the logical expression t is true, statement s is executed and then the DO is reiterated. However, if t is true and s is an arithmetic IF or transfer type statement, control is transferred as indicated.

6. When a reference to a subprogram is executed in the range of a DO, care must be taken that the called subprogram does not alter the DO index or the indexing parameters.

The CONTINUE Statement

General Form
CONTINUE

CONTINUE is a dummy statement that gives rise to no instructions in the object program. It is most frequently used as the last statement in the range of a DO to provide a transfer address for IF and GO TO statements that are intended to begin another repetition of the DO range.

The PAUSE Statement

General Form
PAUSE or PAUSE n where n is an unsigned octal integer constant of 1-5 digits.

Examples:

PAUSE

PAUSE 77777

The machine will halt with the octal number n in the address field of the Storage Register. If n is not specified, it is understood to be zero. Depressing the START key causes the program to resume execution of the object program with the next executable FORTRAN statement.

The END Statement

General Form
END

1. The END statement terminates compilation of a program.
2. The END statement must be the physically last statement of the program.

The STOP Statement

General Form
STOP

The STOP statement terminates the execution of any program by returning control to the Monitor. Execution of a program may also be terminated by a CALL to the EXIT and DUMP subroutines (see the section, "EXIT, DUMP, and PDUMP").

The RETURN Statement

General Form
RETURN

The normal exit from any subprogram is the RETURN statement, which returns control to the calling program. The RETURN statement is the logical end of the program; there may be any number of RETURN statements in the program.

CHAPTER 5: INPUT/OUTPUT STATEMENTS

The FORTRAN statements that specify transmission of information to or from I/O devices may be grouped as follows:

General I/O Statements: The statements READ and WRITE cause the transmission of a specified list of quantities between core storage and an input/output device. The statements PUNCH and PRINT cause information to be transmitted from core storage to the card punch and on-line printer, respectively.

Manipulative I/O Statements: Statements END FILE, REWIND, and BACKSPACE manipulate I/O devices.

Nonexecutable Statements: Either of two nonexecutable statements (the FORMAT statement or the NAMELIST statement) may be used with the general I/O statements. The FORMAT statement, which can be used with any general I/O statement, specifies the arrangement of data in the external input/output medium. If the FORMAT statement is referenced by a READ statement, the input data must meet the specifications described in the section "Data Input Referencing a FORMAT Statement." The NAMELIST statement specifies an input/output list of variables and arrays. Input/output of the values associated with the list is effected by reference to the list in a READ or WRITE statement. If the NAMELIST statement is referenced by a READ statement, the input data must meet the specifications described in the section "Data Input Referencing a NAMELIST Statement."

LIST SPECIFICATIONS

If transmission of arrays or variables is desired using a FORMAT statement, an ordered list of the quantities to be transmitted must be included in the general I/O statement. The order of the I/O list must be the same as the order in which the information exists in the input/output medium.

The following notes on the formation and meaning of an I/O list are most clearly understood by considering the following I/O list:

A, B(3), (C(I), D(I, K), I=1, 10),
((E(I, J), I=1, 10, 2), F(J, 3), J=1, K)

which implies that the information in the external I/O medium is arranged as follows:

A, B(3), C(1), D(1, K), C(2), D(2, K), . . . ,
C(10), D(10, K), E(1, 1), E(3, 1), . . . ,
E(9, 1), F(1, 3), E(1, 2), E(3, 2), . . . ,
E(9, 2), F(2, 3), . . . , F(K, 3)

1. An I/O list is a string of list items separated by commas. A list item may be:
   a. A subscripted or non-subscripted variable; or
   b. An implied DO.

An I/O list reads from left to right, with repetition of variables enclosed in parentheses.

2. A constant may appear in an I/O list only as a subscript or as an indexing parameter.

3. The execution of an I/O list is exactly that of a DO-loop, as though each left parenthesis (except subscripting parentheses) were a DO, with indexing given immediately before the matching right parenthesis, and with the DO range extending up to that indexing information. The order of the I/O list above may be considered equivalent to the following "program":

```
      A
      B(3)
      DO 5 I = 1,10   }
      C(I)            }   (C(I), D(I, K), I=1, 10)
5     D(I, K)
      DO 9 J=1, K     }
      DO 8 I=1, 10, 2 }   ((E(I, J), I=1, 10, 2),
8     E(I, J)         }   F(J, 3), J=1, K)
9     F(J, 3)         }
```

4. An implied DO is best defined by an example. In the I/O list above, the list item (C(I), D(I, K), I=1, 10) is an implied DO; it is evaluated as in the above program.

The range of an implied DO must be clearly defined by parentheses.

5. For a list of the form K, A(K), or K, (A(I), I=1, K), where the definition of an index or an indexing parameter appears earlier in the list of an input statement than its use, the indexing will be carried out with the newly read in value.

6. Any number of quantities may appear in a single list. However, each quantity must have the correct format as specified in a corresponding FORMAT statement. Essentially, it is the list which controls the quantity of data read. If more quantities are to be transmitted than are in the list, only the number of quantities specified in the list are transmitted, and remaining quantities are ignored. Conversely, if a list contains more quantities than are given on one BCD input record, more records are read; if a list contains more quantities than are given in one binary record, reading is terminated as an object program error and control is transferred to FXEM.

INPUT/OUTPUT OF ENTIRE ARRAYS

By referencing the NAMELIST statement, an entire array can be designated for transmission between core storage and an input/output medium. Input of an entire array using the NAMELIST statement is described in the section "Data Input Referencing a NAMELIST Statement"; output of an entire array using the NAMELIST statement is described in the section "Output." If the FORMAT statement is referenced and input/output of an entire array is desired, an abbreviated notation may be used in the list of the general I/O statement. Only the name of the array need be given and the indexing information may be omitted.

1. If A has previously been listed in a DIMENSION or COMMON (with dimensions) statement, the statement
   READ (5, 10)A
   is sufficient to read in all of the elements of the array A (see the section, "Input").
2. The elements read in by this notation are stored in accordance with the description of the arrangement of arrays in storage (see the section, "Arrangement of Arrays in Storage").
3. If A has not been previously dimensioned, only the first element will be read in.

FORMAT STATEMENT

The BCD input/output statements require, in addition to a list of quantities to be transmitted, reference to a FORMAT statement which describes the type of conversion to be performed between the internal machine language and the external notation for each quantity in the list.

General Form
FORMAT ($S_1, S_2, \ldots, S_n/S'_1, S'_2, \ldots, S'_n/\ldots$) where each field, S_i, is a format specification.

Example:
FORMAT (I2/ (E12.4, F10.2))

1. FORMAT statements are not executed; they may be placed anywhere in the source program. Each FORMAT statement must be given a statement number.

2. The FORMAT statement indicates, among other things, the maximum size of each record to be transmitted. In this connection, it must be remembered that the FORMAT statement is used in conjunction with the list of some particular input/output statement, except when a FORMAT statement consists entirely of alphameric fields. In all other cases, control in the object program switches back and forth between the list (which specifies whether data remains to be transmitted) and the FORMAT statement (which gives the specifications for transmission of that data).

3. Slashes are used to specify unit records, which must be one of the following:
   a. A tape record with a maximum length corresponding to the printed line of the off-line printer.
   b. A punched card, to be read on-line, with a maximum of 72 characters; a punched card, to be read off-line, with a maximum of 80 characters.
   c. A line to be printed on-line, with a maximum of 120 characters.

Thus, FORMAT (3F9.2, 2F10.4/8E14.5) would specify records in which the first, third, fifth, etc., have the format (3F9.2, 2F10.4), and the second, fourth, sixth, etc., have the format (8E14.5).

4. During input/output of data, the object program scans the FORMAT statement to which the relevant input/output statement refers. When a specification for a numerical field is found and list items remain to be transmitted, input/output takes place according to the specification, and scanning of the FORMAT statement resumes. If no items remain, transmission ceases and execution of that particular input/output statement is terminated. Thus, a decimal input/output operation will be brought to an end when there are no items remaining in the list.

Numeric Fields

Five types of conversion are available for numeric data:

Internal	Conversion Code	External
Floating point (double-precision)	D	Real with D exponent
Floating point	E	Real with E exponent
Floating point	F	Real without exponent
Integer	I	Integer
Integer	O	Octal Integer

These types of conversion are specified in the forms Dw.d, Ew.d, Fw.d, Iw, Ow, where:

1. D, E, F, I, and O represent the type of conversion;
2. w is an unsigned integer constant that represents the field width for converted data; this field width may be greater than required in order to provide spacing between numbers;

3. d is an unsigned integer or zero that represents the number of positions of the field that appear to the right of the decimal point.

For example, the statement FORMAT (I2,E12.4, O8, F10.4, D25.16) might cause the following line to be printed:

```
I2E12.4      O8      F10.4     D25.16

27b-0.9321Eb0257734276bbb-0.0076bb-0.7878977909500672Db03
```

where b indicates a blank space.

Notes on D-, E-, F-, I-, and O-Conversion

1. Specifications for successive fields are separated by commas.

2. No format specification that provides for more characters than permitted for a relevant input/output record should be given. Thus, a format for a BCD record to be printed off-line should not provide for more characters (including blanks) than the capabilities of the printer.

3. Information to be transmitted with O-conversion may be given any type of variable name; information to be transmitted with E-, and F-conversion must have real names; information to be transmitted with I-conversion must have integer names; information to be transmitted with D-conversion must have double-precision names.

4. The field width w, for D-, E-, and F-conversion, must include a space for the decimal point and a space for the sign. Thus, for D- and E-conversion, $w \geq d+7$, and for F-conversion, $w \geq d+3$.

5. The exponent, which may be used with D- and E-conversion, is the power of 10 to which the number must be raised to obtain its true value. The exponent is written with an E (for E-conversion) or D (for D-conversion) followed by a minus sign if the exponent is negative, or a plus sign or a blank if the exponent is positive, and then followed by two numbers which are the exponent. For example, the number .002 is equivalent to the number .2E-02.

6. For input under D-conversion, up to 17 decimal digits are converted and the result is stored so that the most significant part and the least significant part are in adjacent core storage locations.

For output under D-conversion, the two core storage words representing the double-precision quantity are considered one piece of data and converted as such.

7. If a number converted by I-conversion requires more spaces than are allowed by the field width w, the excess on the high-order side is lost. If the number requires fewer than w spaces, the leftmost spaces are filled with blanks. If the number is negative, the space preceding the leftmost digit will contain a minus sign if sufficient spaces have been reserved.

Complex Number Fields

Since a complex quantity consists of two separate and independent real numbers, a complex number is transmitted by two successive real number specifications or by one real number specification that is repeated.

See Figure 5-1 for an example of a FORMAT statement to transmit an array consisting of six complex numbers.

```
FORMAT (2E10.2, E8.3, 1PE9.4, E10.2, F8.4, 3(E10.2, F8.2))
```

Figure 5-1

Alphameric Fields

FORTRAN provides two ways by which alphameric information may be transmitted; both specifications result in storing the alphameric information internally in BCD.

1. The specification Aw causes w characters to be read into, or written from, a variable or array name.
2. The specification nH introduces alphameric information into a FORMAT statement.

The basic difference between A- and H-conversion is that information handled by A-conversion is given a variable name or array name and hence can be referred to by means of this name for processing and modification, whereas, information handled by H-conversion is not given a name and may not be referred to or manipulated in storage in any way.

A-Conversion

The variable name to be converted by A-conversion must conform to the normal rules for naming FORTRAN variables; it may be any type.

1. On input, nAw will be interpreted to mean that the next n successive fields of w characters each are to be stored as BCD information. If w is greater than 6, only the 6 rightmost characters will be significant. If w is less than 6, the characters will be left-adjusted, and the word filled out with blanks.
2. On output, nAw will be interpreted to mean that the next n successive fields of w characters each are to be the result of transmission from storage without conversion. If w exceeds 6, only 6 characters of output will be transmitted, preceded by w-6 blanks. If w is less than 6, the w leftmost characters of the word will be transmitted.

H-Conversion

The specification nH is followed in the FORMAT statement by n alphameric characters. For example,

31H THIS IS ALPHAMERIC INFORMATION

Note that blanks are considered alphameric characters and must be included as part of the count n. The effect of nH depends on whether it is used with input or output.

1. On input, n characters are extracted from the input record and replace the n characters included with the source program FORMAT specification..
2. On output, the n characters following the specification, or the characters that replaced them, are written as part of the output record.

See Figure 5-2 for an example of A- and H-conversion in a FORMAT statement.

The statement FORMAT (4HbXY=, F8.3, A8) might produce the following lines, where b indicates a blank character:

```
XY = b-93.210bbbbbbbb
XY = 9999.999bbOVFLOW
XY = bb28.768bbbbbbbb
```

Figure 5-2

Figure 5-2 assumes that there are steps in the source program that read the data OVFLOW, store this data in the word to be printed in the format A8 when overflow occurs, and store six blanks in the word when overflow does not occur.

Logical Fields

Logical variables may be read or written by means of the specification Lw, where L represents the logical type of conversion and w is an integer constant that represents the data field width.

1. On input, a value of either true or false will be stored if the first non-blank character in the field of w characters is a T or an F, respectively. If all the w characters are blank, a value of false will be stored.
2. On output, a value of true or false in storage will cause w minus 1 blanks, followed by a T or an F, respectively, to be written out.

Blank Fields - X-Conversion

The specification nX introduces n blank characters into an input/output record where $0 \leq n \leq 132$.

1. On input, nX causes n characters in the input record to be skipped, regardless of what they actually are.
2. On output, nX causes n blanks to be introduced into the output record.

Repetition of Field Format

It may be desired to print or read n successive fields in the same format within one record. This may be specified by giving n, an unsigned integer, before D, E, F, I, L, O, or A. Thus, the field specification 3E12.4 is the same as writing E12.4, E12.4, E12.4.

Repetition of Groups

A limited parenthetical expression is permitted in order to enable repetition of data fields according to certain format specifications within a longer FORMAT statement. Thus, FORMAT (2(F10.6,E10.2),I4) is equivalent to FORMAT (F10.6, E10.2, F10.6, E10.2, I4). Two levels of parentheses, in addition to the parentheses required by the FORMAT statement, are permitted. The second level of parentheses facilitates the transmission of complex quantities.

Scale Factors

To permit more general use of D-, E-, and F-conversion, a scale factor followed by the letter P may precede the specification. The magnitude of the scale factor must be between -8 and +8, inclusive. The scale factor is defined for input as follows:

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

The scale factor is defined for output as follows:

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

For input, scale factors have effect only on F-conversion. For example, if input data is in the form xx.xxxx and it is desired to use it internally in the form .xxxxxx, then the FORMAT specification to effect this change is 2PF7.4. For output, scale factors may be used with D-, E-, and F-conversion.

For example, the statement FORMAT (I2, 3F11.3) might give the following printed line:

27bbbb-93.209bbbbb-0.008bbbbbb0.554

But the statement FORMAT (I2, 1P3F11.3) used with the same data would give the following line:

27bbb-932.094bbbbb-0.076bbbbbb5.536

Whereas, the statement FORMAT (I2, -1P3F11.3) would give the following line:

27bbbbb-9.321bbbbb-0.001bbbbbb0.055

A positive scale factor used for output with D- and E-conversion increases the number and decreases the exponent. Thus, FORMAT (I2, 1P3E12.4) would produce with the same data:

27b-9.3209Eb01b-7.5804E-03bb5.5536E-01

The scale factor is assumed to be zero if no other value has been given. However, once a value has been given, it will hold for all D-, E-, and F-conversions following the scale factor within the same FORMAT statement. This applies to both single-record formats and multiple-record formats (see below). Once the scale factor has been given, a subsequent scale factor of zero in the same FORMAT statement must be specified by 0P. Scale factors have no effect on I- and O-conversion.

Multiple-Record Formats

To deal with a block of more than one line of print, a FORMAT specification may have several different one-line formats separated by a slash, /, to indicate the beginning of a new blank line. Thus, FORMAT (3F9.2,2F10.4/8E14.5) would specify a multi-line block of print in which lines 1, 3, 5, ... have format (3F9.2,2F10.4), and lines 2, 4, 6, ... have format (8E14.5).

If a multiple-line format is desired in which the first two lines are to be printed according to a special format and all remaining lines according to another format, the last line-specification should be enclosed in a second pair of parentheses; e.g., FORMAT (I2,3E12.4/2F10.3,3F9.4/ (10F12.4)). If data items remain to be transmitted after the format specification has been completely "used," the format repeats from the last previous parenthesis which is a zero or a first level parenthesis. For example, consider the FORMAT statement:

```
FORMAT (3E10.3, (I2,2 (F12.4)), D28.17)
        0       1   2    21       0
```

The parentheses labeled 0 are 0 level parentheses; those labeled 1 are first level parentheses; and, those labeled 2 are second level parentheses. If more items in the list are to be transmitted after the format statement has been completely used, the FORMAT repeats from the last first level left parenthesis; i.e., the parenthesis preceding I2.

As these examples show, both the slash and the right parenthesis of the FORMAT statement indicate a termination of a record.

Blank lines may be introduced into a multi-line FORMAT statement by listing consecutive slashes. When n+1 consecutive slashes appear at the end of the FORMAT, they are treated as follows: for input, n+1 records are skipped; for output, n blank lines are written. When n+1 consecutive slashes appear in the middle of the FORMAT, n records will be skipped for both input and output.

Carriage Control

The WRITE (i,n) list statement prepares a BCD tape that can be used to obtain off-line printed output. The PRINT n, list statement prints on-line during execution. The off-line printer may be set manually to operate in one of three modes: single space, double space, and program control. Under program control, the first character of each BCD record, the control character, controls spacing of the printer; this first character is not printed. The control characters and their effects, both on-line and off-line, are:

Character	Effect	Off-line	On-line
Blank	Single space	before printing	after printing
0	Double space	before printing	after printing
1	Eject	before printing	before printing followed by a single space after printing

Program control is usually obtained by beginning a FORMAT specification, for a BCD record, with 1H followed by the desired control character.

FORMAT Statements Read In at Object Time

FORTRAN accepts a variable FORMAT address. This permits specifying a FORMAT for an I/O list at object time.

```
      DIMENSION FMT (12)
1     FORMAT (12A6)
      READ (5,1) (FMT(I), I=1, 12)
      READ (5,FMT) A,B,(C(I), I=1,5)
```

Figure 5-3

In Figure 5-3, A, B, and the array C are converted and stored according to the FORMAT specifications read into the array FMT at object time.

1. The name of the variable FORMAT specification must appear in a DIMENSION statement, even if the array size is only 1.
2. The format read in at object time must take the same form as a source program FORMAT statement, except that the word FORMAT is omitted, i.e., the variable format begins with a left parenthesis.

Data Input Referencing a FORMAT Statement

Data input to the object program is punched into cards according to the following specifications:

1. The data must correspond in order, type, and field with the field specifications in the FORMAT statement. Punching begins in card column 1.
2. Plus signs may be omitted or indicated by a +. Minus signs are indicated by an 11-punch.
3. Blanks in numeric fields are regarded as zeros.
4. Numbers for E- and F-conversion may contain any number of digits, but only the high-order 8 digits of precision will be retained. For D-conversion, the high-order 16 digits of precision will be retained. In both cases, the number is rounded to 8 or 16 digits of accuracy, as applicable.

To permit economy in punching, certain relaxations in input data format are permitted.

1. Numbers for D- and E-conversion need not have four columns devoted to the exponent field. The start of the exponent field must be marked by a D or E or, if that is omitted, by a + or - (not a blank). Thus, E2, E+2, +2, +02, and D+02 are all permissible exponent fields.
2. Numbers for D-, E-, and F-conversion need not have their decimal point punched; the format specification will supply it. For example, the number -09321+2 with the specification E12.4 will be treated as though the decimal point had been punched between the 0 and the 9. If the decimal point is punched in the card, its position overrides the position indicated in the FORMAT specification.

NAMELIST STATEMENT

The NAMELIST statement and modified forms of the READ and WRITE statements provide for reading, writing, and converting data without the use of an input/output list in the I/O statement and without a reference to a FORMAT statement.

General Form
NAMELIST /X/A,B,...C/Y/D,E,...,F/Z/G,H,...,I where: X,Y,Z,... are NAMELIST names, and A,B,C,D,... are variable or array names.

NAMELIST /NAM1/ A,B,I,J,L/NAM2/A,C,J,K

In the preceding example, the arrays A,I, and L and the variables B and J belong to the NAMELIST name, NAM1, and the array A and the variables C, J, and K belong to the NAMELIST name, NAM2.

Each list that is mentioned in the NAMELIST statement is given a NAMELIST name. Only the NAMELIST name is needed in an input/output statement to reference that list thereafter in the program. The following rules apply to assigning and using a NAMELIST name:

1. A NAMELIST name consists of 1-6 alphameric characters; the first character must be alphabetic.
2. A NAMELIST name is enclosed in slashes. The field of entries belonging to a NAMELIST name ends either with a new NAMELIST name enclosed in slashes or with the end of the NAMELIST statement.
3. A variable name or any array name may belong to one or more NAMELIST names.
4. A NAMELIST name must not be the same as any other name in the program.
5. A NAMELIST name may be defined only once by its appearance in a NAMELIST statement. After it has been defined in the NAMELIST statement, the NAMELIST name may appear only in READ or WRITE statements thereafter in the program.
6. A NAMELIST statement must precede any appearance of a NAMELIST name in the program.
7. A variable that belongs to a NAMELIST statement cannot be used as an argument of a subroutine.
8. If a NAMELIST statement contains a dimensioned variable, the DIMENSION statement defining the variable must precede the NAMELIST statement.

Data Input Referencing a NAMELIST Statement

When a READ statement references a NAMELIST name, the designated input device is prepared and input of data is begun. The first character on all input data records is always ignored. The first input data record is searched for a $ as the second character, immediately followed by the NAMELIST name, immediately followed by one or more blank characters. If the search fails, additional records are examined consecutively until there is a successful match. When a successful match is made of the NAMELIST name on a data record and the NAMELIST name referenced in a READ statement, data items are converted and placed in storage. Any combination of three types of data items may be used in a data record. The data items must be separated by commas; however, use of a comma following the last item is optional. If more than one record is needed for input data, the last item of each record must be a constant followed by a comma. The end of a group of data is signaled by a $ either in the same data record as the NAMELIST name or anywhere in any succeeding records except in the first character position.

The form that data items may take is:

1. Variable name = constant where variable name may be an array element name or a simple variable name. Subscripts must be integer constants.

2. Array name = set of constants (separated by commas) in which k*constant may be included to represent k constants (k must be an unsigned integer). The number of constants must be equal to the number of elements in the array.

3. Subscripted variable = set of constants (separated by commas) in which k*constant may be included to represent k constants (k must be an unsigned integer). A data item of this form results in the set of constants being placed in consecutive array elements, starting with the element designated by the subscripted variable. The number of constants given cannot exceed the number of elements in the array that are included between the given element and the last element in the array, inclusive.

Constants used in the data items may take any of the following forms:

a. integers
b. real numbers
c. double-precision numbers
d. complex numbers, which must be written in the usual form, (C1, C2), where C1 and C2 are real numbers
e. logical constants, which must be written as T or . TRUE., and F or . FALSE.

Logical and complex constants may be associated only with logical and complex variables, respectively. The other types of constants may be associated with integer, real, or double-precision variables and are converted in accordance with the type of variable. Blanks must not be embedded in a constant or repeat constant field, but may be used freely elsewhere within a data record.

Any selected set of variable or array names belonging to the NAMELIST name which is referenced by the READ statement may be used as specified in the preceding description of data items. Names that are made equivalent to these names may not be used unless they also belong to the NAMELIST name.
Example:

```
                    Col
                    2
First Data Card     $NAM1  I(2,3) =5, J=4.2, B=4,
Second Data Card    A(3) = 7, 6.4, L = 2,3,8*4.3 $
```

If this data is input to be used with the NAMELIST statement previously illustrated and with a READ statement, the following actions take place. The input unit designated in the READ statement is prepared and the first record is read. The record is searched for a $ in column 2, immediately followed by the NAMELIST name, NAM1. Since the search is successful, data items are converted and placed in core storage.

The integer constant 5 is placed in I(2,3), the real constant 4.2 is converted to an integer and placed in J, and the integer constant 4 is converted to real and placed in B. Since no data items remain in the record, the next input record is read. The integer constant 7 is converted to real and placed in A(3), and the real constant 6.4 is placed in the next consecutive location of the array, A(4). Since L is an array name not followed by a subscript, the entire array is filled with the succeeding constants. Therefore, the integer constants 2 and 3 are placed in L(1) and L(2), respectively, and the real constant 4.3 is converted to an integer and placed in L(3), L(4), ..., L(10). The $ signals termination of the input for the READ operation.

THE GENERAL INPUT/OUTPUT STATEMENTS

Input

The READ statement designates input. The following table gives the forms of the READ statement, where i, an unsigned integer constant or an integer variable, is a reference to an input device, n is a FORMAT statement number, and X is a NAMELIST name.

Type of Input	General Form
Cards on-line	READ n, list
BCD record	READ (i,n) list
Binary record	READ (i) list
BCD records	READ (i,X)

Examples:

```
READ 10, (A(I),I=1,5)
READ (5,10) A,B,(D(J),J=1,10)
READ (N,10) K,DC(J)
READ (3) (A(J),J=1,10)
READ (N) (A(J), J=1,10)
READ (5,NAM1)
```

1. The READ n, list statement causes cards to be read from the card reader.
2. The READ (i,n) list statement causes BCD information to be read from symbolic input device i (except the card reader).
3. The READ (i) list statement causes binary information to be read from symbolic input device i (except the card reader).
4. The READ (i, X) statement causes BCD information relating to variables and arrays associated with the NAMELIST name X to be read from symbolic input device i (except the card reader).
5. Under the first two forms of the READ statement, successive records are read until the entire I/O list has been satisfied: i.e., all data items have been read, converted, and stored in the locations specified by the I/O list.

Under the form READ (i) list, a record is read completely only if the list specifies as many words as the record contains. Binary records to be read

in by a FORTRAN program should be written by a FORTRAN program or should be in the proper binary record format as follows:

Consider a logical record as being any sequence of binary words to be read by any one input statement. This logical record must be divided into physical records, each of which is a maximum of 256_{10} words long. Of course, if a logical record consists of fewer than 256_{10} words, it will comprise only one physical record. The first word of each physical record is a "signal" word that is not part of the list. This word contains zero for all but the last physical record of a logical record. The first word of the last physical record contains a number designating the number of physical records in this logical record.

Output

The PRINT and PUNCH statements designate on-line printing and punching of data and require both a reference to a FORMAT statement and an output list as part of the statement. All other output is designated by a WRITE statement which also references either a FORMAT statement or a NAMELIST statement. The following table gives the forms of the output statements, where i, an unsigned integer constant or an integer variable, is a reference to an output device, n is a FORMAT statement number, and X is a NAMELIST name.

Type of Output	General Form
Cards on-line	PUNCH n, list
Print on-line	PRINT n, list
BCD Record	WRITE (i, n) list
BCD Record	WRITE (i, X)
Binary Record	WRITE (i) list

Examples:

```
PUNCH 20, (A(J), J=1,6)
PRINT 2, (A(J), J=1,6)
WRITE (6,10) A,B,(C(J),J=1,10)
WRITE (N,11) K,D(J)
WRITE (2) (A(J), J=1, 10)
WRITE (M) A,B,C
WRITE (6,NAM1)
```

1. The PUNCH statement causes alphameric cards to be punched on-line.
2. The PRINT statement causes data to be output on the on-line printer.
3. The WRITE (i,n) list statement causes BCD information to be written on symbolic output device i.
4. The WRITE (i) list statement causes binary information to be written on symbolic output device i.
5. The WRITE (i, X) statement causes the names and values of all variables and arrays that belong to the NAMELIST name, X, to be written on symbolic output device i.

The PUNCH, PRINT, and WRITE (i,n) statements cause successive records to be written in accordance with the FORMAT statement until the list has been satisfied. The WRITE (i) list statement causes the writing of one logical record consisting of all the words specified in the list.

When a WRITE statement references a NAMELIST name, the values and names of all variables and arrays belonging to the NAMELIST name are written, each according to its type. A complete array is written out by columns. The output data is written such that:

1. The fields for the data are large enough to contain all the significant digits.
2. The output can be read by an input statement referencing the NAMELIST name.

THE MANIPULATIVE INPUT/OUTPUT STATEMENTS

The statements END FILE, REWIND, and BACKSPACE manipulate I/O devices. In the following table, i, an unsigned integer constant or integer variable, is a reference to an input/output device.

General Form
END FILE i REWIND i BACKSPACE i

Examples:

```
END FILE 3
END FILE N
REWIND 3
REWIND N
BACKSPACE 3
```

1. The END FILE i statement causes an end-of-file mark to be written on symbolic tape i.
2. The REWIND i statement causes symbolic tape unit i to be rewound.
3. A request to write an end of file or rewind system files SYSIN1, SYSOU1, and SYSPP1, corresponding in the standard FORTRAN I/O Library to symbolic units 5, 6, and 7, will be ignored.
4. The BACKSPACE i statement causes tape i to be backspaced one physical record if i refers to an I/O device in the BCD mode, or it causes tape i to be backspaced one logical record if i refers to an I/O device in the binary mode. A request to backspace SYSOU1

corresponding in the standard FORTRAN I/O Library to symbolic unit 6, will be ignored.

SYMBOLIC INPUT/OUTPUT UNIT DESIGNATION

Input/output devices are always referred to symbolically in FORTRAN input/output statements.

1. Object program input/output operates through the Minimum IOCS Buffering package. The correspondence between the symbolic unit reference and the actual physical unit is established in the initialization of IOCS.
2. The standard FORTRAN I/O configuration allows for symbolic tape units 1 through 8. The normal unit designation for BCD input statements is 5; for BCD output statements it is 6.

FORTRAN Tape Units	Mode	Function
1	Binary	Input or Output
2	Binary	Input or Output
3	Binary	Input or Output
4	Binary	Input or Output
5	BCD	Input
6	BCD	Output
7	Binary	Output
8	BCD	Input or Output

The symbolic unit references may be changed by each installation in accordance with its own needs. See the reference manual IBM 7090/7094 Operating Systems: Basic Monitor (IBSYS), Form C28-6248, and the reference manual IBM 7090/7094 Programming Systems, IBJOB Processor, Form C28-6275.

CHAPTER 6: SUBROUTINES, FUNCTIONS, AND SUBPROGRAM STATEMENTS

There are four classes of subroutines in FORTRAN: Arithmetic Statement Functions, Built-In Functions, FUNCTION subprograms, and SUBROUTINE subprograms. The major differences among the four classes of subroutines are as follows:

1. The first three classes may be grouped as functions; they differ from the SUBROUTINE subprogram in the following respects:
   a. The functions are always single-valued (that is, they return a single result); the SUBROUTINE subprogram may return more than one value.
   b. A function is referenced by an arithmetic expression containing its name; a SUBROUTINE subprogram is referenced by a CALL statement.
2. The built-in function is an open subroutine; i.e., a subroutine that is incorporated into the object program each time it is referred to in the source program. The three other FORTRAN subroutines are closed; i.e., they appear only once in the object program.

NAMING SUBROUTINES

All four classes of subroutines are named in the same manner as a FORTRAN variable (see the section, "Variables").

1. A subroutine name consists of 1 - 6 alphameric characters, the first of which must be alphabetic.
2. The type of the function, which determines the type of the result, may be defined as follows:
   a. The type of an arithmetic statement function may be indicated by the name of the function or by placing the name in a type statement.
   b. The type of a FUNCTION subprogram may be indicated by the name of the function (if it is real or integer) or by writing the type (REAL, INTEGER, COMPLEX, DOUBLE PRECISION, LOGICAL) preceding the word FUNCTION. In the latter case, the type, implied by name, is overridden.
   c. The type of a built-in function is indicated within the FORTRAN Processor and need not appear in a type statement (see columns 5 and 6 of Figure 6-1).
3. The type of a SUBROUTINE subprogram is unimportant and need not be defined, since the type of results returned is dependent only on the type of the variable names in the dummy argument list.

DEFINING SUBROUTINES

The method of defining each class of subroutines is discussed below.

Arithmetic Statement Functions

Arithmetic statement functions are defined by a single arithmetic statement and apply only to the source program containing the definition.

General Form
a = b where: 1. the letter a is a function name followed by parentheses enclosing its arguments, which must be distinct, non-subscripted variables, separated by commas. 2. the letter b is an expression that does not involve subscripted variables. Any arithmetic statement function appearing in b must have been previously defined.

Examples:

```
FIRST (X) = A*X+B
JOB (X, B) = C*X+B
THIRD F(D) = FIRST (E)/D
MAX (A, I) = A**I-B - C
LOGFCT (A, C) = A**2.GE. C/D
```

1. As many as desired of the variables appearing in b may be stated in a as the arguments of the function. Since the arguments are dummy variables, their names, which indicate the type of the variable, may be the same as names appearing elsewhere in the program of the same type.
2. Those variables included in b that are not stated as arguments are the parameters of the function. They are ordinary variables.
3. All arithmetic statement function definitions must precede the first executable statement of the source program.
4. The type of each argument must be defined preceding its use in the arithmetic statement function definition.

Built-In Functions

Built-in functions are pre-defined, open subroutines that exist within the FORTRAN Processor. A list of all the available built-in functions is given in Figure 6-1.

Function	Definition	Number of Arguments	Name	Type of Argument	Type of Function
Absolute value	$\|Arg\|$	1	ABS	Real	Real
			IABS	Integer	Integer
Truncation	Sign of Arg times largest integer $\leq \|Arg\|$	1	AINT	Real	Real
			INT	Real	Integer
Remaindering (see note below)	Arg_1 (mod Arg_2)	2	AMOD	Real	Real
			MOD	Integer	Integer
Choosing largest value	Max(Arg_1, Arg_2,...)	≥ 2	AMAX0	Integer	Real
			AMAX1	Real	Real
			MAX0	Integer	Integer
			MAX1	Real	Integer
Choosing smallest value	Min(Arg_1, Arg_2,...)	≥ 2	AMIN0	Integer	Real
			AMIN1	Real	Real
			MIN0	Integer	Integer
			MIN1	Real	Integer
Float	Conversion from integer to real	1	FLOAT	Integer	Real
Fix	Conversion from real to integer	1	IFIX	Real	Integer
Transfer of sign	Sign of Arg_2 times $\|Arg_1\|$	2	SIGN	Real	Real
			ISIGN	Integer	Integer
Positive difference	Arg_1 - Min (Arg_1, Arg_2)	2	DIM	Real	Real
			IDIM	Integer	Integer
Obtain most significant part of double-precision argument		1	SNGL	Double	Real
Obtain real part of complex argument		1	REAL	Complex	Real
Obtain imaginary part of complex argument		1	AIMAG	Complex	Real
Absolute value	$\|Arg\|$	1	DABS	Double	Double
Truncation	Sign of Arg times largest integer $\leq \|Arg\|$	1	IDINT	Double	Integer
Choosing largest value	Max (Arg_1, Arg_2,...)	≥ 2	DMAX1	Double	Double
Choosing smallest value	Min (Arg_1, Arg_2,...)	≥ 2	DMIN1	Double	Double
Transfer of sign	Sign of Arg_2 times $\|Arg_1\|$	2	DSIGN	Double	Double
Express single-precision argument in double-precision form	D=(Arg, 0)	1	DBLE	Real	Double
Express two real arguments in complex form	$C=Arg_1+iArg_2$	2	CMPLX	Real	Complex
Obtain conjugate of a complex argument	For Arg=X+iY, C=X-iY	1	CONJG	Complex	Complex

NOTE: The function MOD (Arg_1, Arg_2) is defined as $Arg_1 - \left[Arg_1/Arg_2\right] Arg_2$, where $[x]$ is the integral part of x

Figure 6-1. Built-In Functions

FUNCTION Subprogram

FUNCTION subprograms are defined by a special FORTRAN Source language program.

General Form
FUNCTION name $(a_1, a_2, \ldots, a_n)$ REAL FUNCTION name $(a_1, a_2, \ldots, a_n)$ INTEGER FUNCTION name $(a_1, a_2, \ldots, a_n)$ DOUBLE PRECISION FUNCTION name $(a_1, a_2, \ldots, a_n)$ COMPLEX FUNCTION name $(a_1, a_2, \ldots, a_n)$ LOGICAL FUNCTION name $(a_1, a_2, \ldots, a_n)$ where: 1. name is the symbolic name of a single-valued function, 2. the arguments $a_1, a_2, \ldots, a_n$, of which there must be at least one, are non-subscripted variable names or the dummy name of a SUBROUTINE or FUNCTION subprogram, and 3. the type of the function may be explicitly stated preceding the word FUNCTION.

Examples:

```
FUNCTION ARCSIN(RADIAN)
REAL FUNCTION ROOT (A,B,C)
INTEGER FUNCTION CONST(ING, SG)
DOUBLE PRECISION FUNCTION DBLPRE(R,S,T)
COMPLEX FUNCTION CCOT(ABI)
LOGICAL FUNCTION IFTRU (D, E, F)
```

1. The FUNCTION statement must be the first statement of a FUNCTION subprogram.
2. The name of the function must appear at least once as a variable on the left side of an arithmetic statement or in an input statement.

For example:

```
FUNCTION CALC (A, B)
.
.
.
CALC=Z+B
.
.
.
RETURN
```

   By this means the output value of the function is returned to the calling program.

3. The arguments may be considered dummy variable names that are replaced at the time of execution by the actual arguments supplied in the function reference in the calling program. The actual arguments must correspond in number, order, and type with the dummy arguments.
4. When a dummy argument is an array name, a DIMENSION or COMMON (with dimensions) statement must appear in the FUNCTION subprogram; also, the corresponding actual argument must be a dimensioned array name.
5. None of the dummy arguments may appear in an EQUIVALENCE statement in the FUNCTION subprogram.
6. The FUNCTION subprogram must be logically terminated by a RETURN statement (see the section, "The RETURN Statement").
7. The FUNCTION subprogram may contain any FORTRAN statements except SUBROUTINE or another FUNCTION statement.
8. The arguments of a FUNCTION subprogram may be any of the following:
   a. Any type of constant.
   b. Any type of subscripted or non-subscripted variable.
   c. An arithmetic or a logical expression.
   d. The name of a FUNCTION or SUBROUTINE subprogram.
9. A FUNCTION subprogram is referenced by using its name as an operand in an arithmetic expression.

   Those FUNCTION subprograms that are available with FORTRAN are given in Figure 6-2.

SUBROUTINE Subprogram

SUBROUTINE subprograms are defined by a special FORTRAN source language program.

General Form
SUBROUTINE name $(a_1, a_2, \ldots, a_n)$ where: 1. name is the symbolic name of a subprogram; and 2. each argument, if any, is a non-subscripted variable name or the dummy name of a SUBROUTINE or FUNCTION subprogram.

Examples:

```
SUBROUTINE MATMPY (A,N,M,B,L,J)
SUBROUTINE QDRTIC (B,A,C, ROOT1,ROOT2)
```

1. The SUBROUTINE statement must be the first statement of a SUBROUTINE subprogram.
2. The SUBROUTINE subprogram may use one or more of its arguments to return output. The arguments so used must appear on the left side of an arithmetic statement or in an input list within the subprogram.
3. The arguments may be considered dummy variable names that are replaced at the time

of execution by the actual arguments supplied in the CALL statement, which refers to the SUBROUTINE subprogram. The actual arguments must correspond in number, order, and type with the dummy arguments.

4. When a dummy argument is an array name, a DIMENSION or COMMON (with dimensions) statement must appear in the SUBROUTINE subprogram; also, the corresponding actual argument in the CALL statement must be a dimensioned, array name.

5. None of the dummy arguments may appear in an EQUIVALENCE statement in the SUBROUTINE subprogram.

6. The SUBROUTINE subprogram must be logically terminated by a RETURN statement.

7. The SUBROUTINE subprogram may contain any FORTRAN statements except FUNCTION or another SUBROUTINE statement.

Subprogram Names as Arguments

FUNCTION and SUBROUTINE subprogram names may be the actual arguments of subprograms. In order to distinguish these subprogram names from ordinary variables when they appear in an argument list, they must appear in an EXTERNAL statement.

```
EXTERNAL SIN
CALL SUBR (A, SIN, B)
```

Function	Definition	Number of Arguments	Name	Type of Argument	Type of Function
Exponential	$_{e}Arg$	1	EXP	Real	Real
Natural logarithm	$\log_e(Arg)$	1	ALOG	Real	Real
Common logarithm	$\log_{10}(Arg)$	1	ALOG10	Real	Real
Arctangent	arctan (Arg)	1	ATAN	Real	Real
	arctan (Arg_1/ Arg_2)	2	ATAN2	Real	Real
Trigonometric sine	sin (Arg)	1	SIN	Real	Real
Trigonometric cosine	cos (Arg)	1	COS	Real	Real
Hyperbolic tangent	tanh (Arg)	1	TANH	Real	Real
Square root	$(Arg)^{1/2}$	1	SQRT	Real	Real
Remaindering	Arg_1 (mod Arg_2)	2	DMOD	Double	Double
Exponential	$_{e}Arg$	1	DEXP	Double	Double
Natural logarithm	$\log_e$ (Arg)	1	DLOG	Double	Double
Common logarithm	$\log_{10}$ (Arg)	1	DLOG10	Double	Double
Arctangent	arctan (Arg)	1	DATAN	Double	Double
	arctan (Arg_1/ Arg_2)	2	DATAN2	Double	Double
Trigonometric sine	sin (Arg)	1	DSIN	Double	Double
Trigonometric cosine	cos (Arg)	1	DCOS	Double	Double
Square root	$(Arg)^{1/2}$	1	DSQRT	Double	Double
Absolute value	For Arg=X+iY C= $(X^2+Y^2)^{1/2}$	1	CABS	Complex	Complex
Exponential	$_{e}Arg$	1	CEXP	Complex	Complex
Natural logarithm	$\log_e(Arg)$	1	CLOG	Complex	Complex
Trigonometric sine	sin (Arg)	1	CSIN	Complex	Complex
Trigonometric cosine	cos(Arg)	1	CCOS	Complex	Complex
Square root	$(Arg)^{1/2}$	1	CSQRT	Complex	Complex

Figure 6-2. Mathematical Subroutines

THE CALL STATEMENT

The CALL statement is used to refer to a SUBROUTINE subprogram.

General Form
CALL SUBR ($a_1, a_2, \ldots, a_n$) where: 1. SUBR is the name of a SUBROUTINE subprogram, and 2. $a_1, a_2, \ldots, a_n$ are the n arguments.

Examples:

```
CALL MATMPY (X,5,10,Y,7,2)
CALL QDRTIC (9.732,Q/4.536,R-S**2.0,X1,
X2)
```

The CALL statement transfers control to the subprogram and presents it with the actual arguments. The arguments may be any of the following:

1. Any type of constant.
2. Any type of subscripted or non-subscripted variable.
3. An arithmetic or a logical expression.
4. Alphameric characters. Such arguments must be preceded by nH, where n is the count of characters included in the argument, e.g., 9HEND POINT. Note that blank spaces and special characters are considered in the character count when used in alphameric fields.
5. The name of a FUNCTION or SUBROUTINE subprogram.

The arguments presented by the CALL statement must agree in number, order, type, and array size (except as explained under the DIMENSION statement) with the corresponding arguments in the SUBROUTINE statement of the called subprogram.

SUBPROGRAMS PROVIDED BY FORTRAN

FORTRAN includes several commonly used subroutines that are available to the programmer. The mathematical subroutines that are provided are defined as FUNCTION subprograms; the subroutines provided to test the status of the machine indicators (the sense switches and the sense lights) are defined as SUBROUTINE subprograms. In addition, FORTRAN includes the SUBROUTINE subprograms EXIT, DUMP, and PDUMP. EXIT terminates job execution; DUMP dumps core storage and then terminates job execution; PDUMP dumps core storage and then continues execution.

Mathematical Subroutines

FORTRAN provides various commonly used mathematical subroutines; these are defined as FUNCTION subprograms. The names of all of these subprograms are automatically typed by the FORTRAN IV Compiler; therefore, they need not appear in Type statements. Variables used as arguments of mathematical subroutines must be typed, either explicity or implicitly, in accordance with the function in which they appear. The mathematical subroutines are listed in Figure 6-2.

Machine Indicator Tests

In the following list of machine indicator test subroutines, assume that i is an integer expression and that j is an integer variable. These subroutines are referenced by CALL statements.

SLITE (i): If i = 0, all sense lights will be turned off. If i =1, 2, 3, or 4, the corresponding sense light will be turned on.

SLITET (i, j): Sense light i will be tested and turned off. The variable j will be set to 1 if i was on, or j will be set to 2 if i was off.

SSWTCH (i, j): Sense switch i is tested and j is set to 1 if i was down and j is set to 2 if i was up.

OVERFL (j): j is set to 1 if a floating point overflow condition exists, or j is set to 2 if no overflow condition exists. The machine is left in a no overflow condition.

DVCHK (j): If the divide check indicator is on, j is set to 1 and the divide check indicator is turned off; if the divide check indicator is off, j is set to 2.

EXIT, DUMP, and PDUMP

EXIT

A CALL to the EXIT subprogram terminates the execution of any program by returning control to the Monitor.

DUMP

A CALL to the DUMP subprogram by the statement

CALL DUMP ($A_1, B_1, F_1, \ldots, A_n, B_n, F_n$)

causes the indicated limits of core storage to be dumped and execution to be terminated by returning control to the Monitor.

1. A and B are variable data names that indicate the limits of core storage to be dumped; either A or B may represent upper or lower limits.

2. F_i is an integer indicating the dump format desired:
   F = 0 dump in octal
   1 dump as real
   2 dump as integer
   3 dump in octal with mnemonics
3. If no arguments are given, all of core storage is dumped in octal.
4. If the last argument F_n is omitted, it is assumed to be equal to 0 and the dump will be octal.

PDUMP

A CALL to the PDUMP subprogram by the statement

CALL PDUMP ($A_1, B_1, F_1, \ldots, A_n, B_n, F_n$)

causes the indicated limits of core storage to be dumped and execution to be continued. The PDUMP arguments are the same as the DUMP arguments.

THE BLOCK DATA SUBPROGRAM

In order to enter data into a COMMON block during compilation, a special subprogram must be written. This special subprogram contains only the DATA, COMMON, DIMENSION, and Type statements associated with the data being defined. Data may be entered into labeled, but not unlabeled, COMMON by the BLOCK DATA subprogram.

General Form
BLOCK DATA

1. The BLOCK DATA subprogram may not contain any executable statements.
2. The first statement of this subprogram must be the BLOCK DATA statement.
3. All elements of a COMMON block must be listed in the COMMON statement even though they do not all appear in the DATA statement; for example, the variable A in the COMMON statement in Figure 6-3 does not appear in the DATA statement.
4. Data may be entered into more than one COMMON block in a single BLOCK DATA subprogram.

```
BLOCK DATA
COMMON /ELN/C,A,B/RMG/Z,Y
DIMENSION B(4),Z(3)
DOUBLE PRECISION Z
COMPLEX C
DATA (B(I),I=1,4)/1.1,1.2,2*1.3/,C/(2.4,3.769)/,
    Z(1)/7.6498085D0/
END
```

Figure 6-3

The specification statements perform the following functions: they provide information about storage allocation and about the constants and variables used in the program.

The DIMENSION Statement

General Form

DIMENSION $v_1(i_1), v_2(i_2), v_3(i_3), \ldots$
where:
1. each v_m is a subscripted variable, and
2. each i_n is composed of 1, 2, or 3 unsigned integer constants and/or integer variables (i may be a variable only when the DIMENSION statement appears in a subprogram).

Examples:

```
DIMENSION A(10), B(5,15), C(L,M)
DIMENSION S(10), K(5,5,5), G(100)
```

The DIMENSION statement provides the information necessary to allocate storage for arrays in the object program. The DIMENSION statement defines the maximum size of arrays.

1. Each variable that appears in subscripted form in the source program must appear in a DIMENSION statement contained in the source program; however, if the dimension information for a variable is included in a COMMON or a Type statement in the source program, it must not be included in a DIMENSION statement.
2. The DIMENSION statement must precede the first appearance of each subscripted variable in an executable or DATA statement for which it specified the size.
3. A single DIMENSION statement may specify the dimensions of any number of arrays.
4. Dimensions specified in a COMMON or a Type statement are subject to all the rules for the DIMENSION statement.

Adjustable Dimensions

The name of an array and the constants that are its dimensions may be passed as arguments in a subprogram call. In this way, a subprogram may perform calculations on arrays whose size is not determined until the subprogram is called. Figure 7-1 illustrates the use of adjustable dimensions.

1. Variables may be used as dimensions of an array only in the DIMENSION statement of a FUNCTION or SUBROUTINE subprogram. For any such array, the array name and all the variables used as dimensions must appear as arguments in the FUNCTION or SUBROUTINE statement.

```
SUBROUTINE MAYMY (...,R,L,M,...)
        .
        .
        .
DIMENSION...,R(L,M),...
        .
        .
        .
DO 100 I=1,L
```

Figure 7-1

2. The adjustable dimensions may not be altered within the subprogram.
3. The absolute dimensions must be specified in a DIMENSION statement of the calling program.
4. The calling program passes the specific dimensions to the subprogram. These specific dimensions are those that appear in the DIMENSION statement of the calling program. Variable dimension size may be passed through more than one level of subprogram.

The COMMON Statement

General Form

COMMON a,b,c,.../r/d,e,f,.../s/g,h,...
where:
1. a,b,... are variables that may be dimensioned, and
2. /r/ ,/s/,... are variables that are block names.

Examples:

```
COMMON A,B,C/X/Q,R/YY/M,P,Q
COMMON /Z/G,H,J//D,F
```

Variables, including array names, appearing in a COMMON statement are assigned locations relative to the beginning of a particular common block. This COMMON area may be shared by a program and its subprograms.

1. If the variables appearing in a COMMON statement contain dimension information, they must not appear in a DIMENSION statement or in a Type statement that contains dimension information.
2. The locations in the COMMON area are assigned in the sequence in which the variables appear in the COMMON statement, beginning with the first COMMON statement of the program.
3. Elements placed in COMMON may be placed in separate blocks. These separate blocks may share space in core storage at object time. Blocks

are given names and those with the same name occupy the same space.

4. COMMON Block Names. The symbolic name of a block, which is 1-6 alphameric characters the first of which is alphabetic, precedes the variable names belonging to the block. The block name is always embedded in slashes, e.g., /BB/. It must not be the same as the name of any other subroutine which is part of the same job. There are two types of COMMON blocks: blank and labeled.

   a. Blank COMMON is indicated either by omitting the block name if it appears at the beginning of the COMMON statement, or by preceding the blank COMMON variable by two consecutive slashes.
   b. Labeled COMMON is indicated by preceding the labeled COMMON variables by the block name embedded in slashes.

5. The field of entries pertaining to a block name ends with a new block name or with the end of the COMMON statement or with a blank COMMON designation.

6. Block name entries are cumulative throughout the program. For example, the COMMON statements

```
COMMON A,B,C /R/D,E/S/F
COMMON G,H/R/I/S/P
```

have the same effect as the statement

```
COMMON A,B,C,G,H/R/D,E,I/S/F,P
```

7. Blank COMMON may be any length. Labeled COMMON must conform to the following size requirement: all COMMON blocks of a given name must have the same length in all the programs that are executed together.

8. Variables brought into a COMMON block through EQUIVALENCE statements may increase the size of the block (see the section, "The Equivalence Statement").

9. Two variables in COMMON may not be made equivalent to each other, directly or indirectly.

10. A double-word variable in COMMON must be placed such that its high-order part is an even number of words away from the first element in COMMON.

The EQUIVALENCE Statement

General Form
EQUIVALENCE (a,b,c,...),(d,e,f,...),... where: a,b,c,d,e,f,... are variables that may be subscripted; these subscripts must be integer constants. The number of subscripts appended to a variable must be equal to the number of dimensions of the variable.

Examples:

```
DIMENSION B(5), C(10,10), D(5,10,15)
EQUIVALENCE (A,B(1), C(5,4)), (D(1,4,3),E)
```

The EQUIVALENCE statement controls the allocation of data storage by causing two or more variables to share the same core storage location.

1. An EQUIVALENCE statement may be placed anywhere in the source program. Each pair of parentheses in the statement list encloses the names of two or more variables that are to be stored in the same location during execution of the object program; any number of equivalences (i.e., sets of parentheses) may be given.

2. In the preceding example, the EQUIVALENCE statement indicates that A, and the B and C arrays are to be assigned storage locations so that the elements A, B(1), and C(5,4) are to occupy the same location. In addition, it also specifies that D(1,4,3) and E are to share the same location.

3. Quantities or arrays that are not mentioned in an EQUIVALENCE statement will be assigned unique locations.

4. Locations can be shared only among variables, not among constants.

5. The sharing of storage locations requires a knowledge of which FORTRAN statements will cause a new value to be stored in a location. There are four such statements:

   a. Execution of an arithmetic statement stores a new value in the variable on the left side of the equal sign.
   b. Execution of an ASSIGN i TO n statement stores a new value in n.
   c. Execution of a DO statement or an implied DO in an input/output list sometimes stores a new indexing value.
   d. Execution of a READ statement stores new values in the variables mentioned in the input list.

6. Variables brought into a COMMON block through EQUIVALENCE statements may increase the size of the block indicated by the COMMON statements, as in the following example:

```
COMMON /X/A,B,C
DIMENSION D(3)
EQUIVALENCE (B,D(1))
```

The layout of core storage indicated by this example (extending from the lowest location of the block to the highest location of the block) is:

```
A
B, D(1)
C, D(2)
   D(3)
```

7. Since arrays must be stored in consecutive forward locations, a variable may not be made equivalent to an element of an array in such a way as to cause

the array to extend beyond the beginning of the COMMON block. For example, the following coding is invalid:

```
COMMON /X/A,B,C
DIMENSION D(3)
EQUIVALENCE (B,D(3))
```

because it would force D(1) to precede A, as follows:

```
    D(1)
A,  D(2)
B,  D(3)
C
```

8. The rule for making double-word variables equivalent to single-word variables is:

In COMMON, the effect of the EQUIVALENCE statements must be such that the high-order word of any double-word variable is an even number of locations away from the start of the COMMON block.

In non-COMMON, the effect of the EQUIVALENCE statements must be such that the high-order word of any double-word variable is an even number of words away from the start of any other double-word variable linked to it through EQUIVALENCE statements.

9. Two variables in one COMMON block or in two different COMMON blocks may not be made equivalent.

The Type Statements

The type of a variable or function may be specified by means of one of the six Type statements:

General Form

INTEGER $a(i_1)$, $b(i_2)$, $c(i_3)$,...
REAL $a(i_1)$, $b(i_2)$, $c(i_3)$,...
DOUBLE PRECISION $a(i_1)$, $b(i_2)$, $c(i_3)$,...
COMPLEX $a(i_1)$, $b(i_2)$, $c(i_3)$,...
LOGICAL $a(i_1)$, $b(i_2)$, $c(i_3)$,...
EXTERNAL a,b,c,...

where:

1. a,b,c,... are variable or function names appearing within the program.
2. each i_n is composed of 1, 2, or 3 integer constants and/or integer variables. Subscripts may only be appended to variable names appearing within the program, not function names.

Examples:

```
INTEGER BIXF, X, QF, LSL
REAL IMIN, LOG, GRN, KLW
DOUBLE PRECISION Q, J, DSIN
EXTERNAL SIN, MATMPY, INVTRY
INTEGER A(10,10), B
COMPLEX C(4,5,3), D
```

The variable or function names following the type (INTEGER, REAL, etc.) in the Type statement are defined to be of that type, and remain that type throughout the program; the type may not be changed.

Note that LSL and GRN need not appear in their respective Type statements, since their type is implied by their first characters. Note also that DSIN need not appear in its statement if it is used as a function in the program, since mathematical subroutines are automatically typed by the FORTRAN IV Compiler.

1. The appearance of a name in any Type statement, except EXTERNAL, overrides the implicit type assignment.
2. Variables that appear in EXTERNAL statements are subprogram names that are arguments of other subprograms.
3. A name may appear in two Type statements only if one of them is EXTERNAL.
4. The Type statement must precede the first use of a name in any executable statement or DATA statement in the program.
5. A name declared to be of a given type may assume only the values of a constant of the same type.
6. The EXTERNAL statement may not be used to dimension variables.
7. Any variable that is dimensioned by a Type statement may not be dimensioned elsewhere, i.e., it may not appear in a DIMENSION statement or in a COMMON statement that contains dimension information.

The DATA Statement

Data may be compiled into the object program by means of the DATA statement.

General Form

DATA list/$d_1,d_2,\ldots,d_n$/,list/d_1,d_2,k*$d_3,\ldots,d_m$/,...

where:

1. list contains the names of the variables being defined,
2. d is the information literal, and
3. k is an integer constant.

Examples:

```
DATA R,Q/14.2,3HEND/,Z/O777777700001/
DATA (B(I),C(I), I=1,40,2)/2.0,3.0,38*100.0/
LOGICAL LA, LB, LC, LD
DATA LA, LB, LC, LD/F,.TRUE.,.FALSE.,T/
```

1. List. Subscripted variables may appear in the list. Where a subscript symbol is used, it must be under control of DO-implying parentheses and associated parameters. Subscripts not so controlled must be integer constants. The DO-defining parameters must be integer constants.

2. k. k may appear before a d-field to indicate that the field is to be repeated k times. k must be

followed by an * to separate it from the field to be repeated.

3. d. The d-literals may take any of the four following forms:

   a. Integer, real, double-precision, and complex constants.
   b. Alphameric characters. The alphameric field is written as nH followed by n alphameric characters. Each group of six alphameric characters forms a word. If n is not a multiple of six, the remaining characters are left justified in the word, and the word is filled out with BCD blanks. Blanks are significant in alphameric fields.
   c. Octal digits. The octal field is written as O, followed by 1-12 signed or unsigned octal digits.
   d. Logical constants. The logical field may be written as either .TRUE., .FALSE., T, or F.

4. There must be a one-to-one correspondence between the list items and the data literals. Each data literal (integer constant, real constant, alphameric constant, complex constant, logical constant, double-precision constant, or octal constant) corresponds to one element (undimensioned variable or subscripted array reference). Caution: If it is desired to define 16 alphameric characters, say 16HDATAbTObBEbREADb starting at G(1), then G must be dimensioned to cover at least three locations and the entire literal corresponds to G(1).

5. The BLOCK DATA subprogram (see page 27), which includes a DATA statement, compiles data into the common area of the program.

6. When DATA defined variables are redefined during execution, these variables will assume their new values regardless of the DATA statement.

7. Where data is to be compiled into an entire array, the name of the array (with indexing information omitted) can be placed in the list. The number of data literals must be equal to the size of the array.

   For example, the statements

   ```
   DIMENSION B(25)
   DATA A,B,C/24*4.0,3.0,2.0,1.0/
   ```

define the values of A, B(1),..., B(23) to be 4.0, and the values of B(24), B(25), and C to be 3.0, 2.0, and 1.0, respectively.

8. The DATA statement may not be used to enter data into unlabeled COMMON.

APPENDIXES

APPENDIX A: SOURCE PROGRAM STATEMENTS AND SEQUENCING

The order in which the source program statements of a FORTRAN program are executed follows these rules:

1. Control originates at the first executable statement. The specification statements and the FORMAT, FUNCTION, SUBROUTINE, NAMELIST, and BLOCK DATA statement, are non-executable; in questions of sequencing, they can be ignored.
2. If control was with statement S, then control will pass to the statement indicated by the normal sequencing properties of S. If, however, S is the last statement in the range of one or more DOs that are not yet satisfied, then the normal sequencing of S is ignored and DO-sequencing occurs.

Every executable statement in a FORTRAN source program, except the first, should have some path of control leading to it.

The normal sequencing properties of each FORTRAN statement follow.

Statement	Normal Sequencing
a = b	Next executable statement
ASSIGN i TO n	Next executable statement
BACKSPACE i	Next executable statement
BLOCK DATA	Non-executable
CALL	First statement of called subprogram
COMMON	Non-executable
COMPLEX	Non-executable
CONTINUE	Next executable statement
DATA	Non-executable
DIMENSION	Non-executable
DO	DO-sequencing, then the next executable statement
DOUBLE PRECISION	Non-executable
END	Terminates program
END FILE	Next executable statement
EQUIVALENCE	Non-executable
EXTERNAL	Non-executable
FORMAT	Non-executable
FUNCTION	Non-executable
GO TO n	Statement n
GO TO n, $(n_1, n_2, \ldots, n_m)$	Statement last assigned to n
GO TO $(n_1, n_2, \ldots, n_m)$, i	Statement n_i
IF (a) n_1, n_2, n_3	Statement n_1, n_2, n_3 if a < 0, a=0, or if a > 0, respectively
IF (t) s	Statement s or next executable statement if t is true or false, respectively
INTEGER	Non-executable
LOGICAL	Non-executable
NAMELIST	Non-executable
PAUSE	Next executable statement
PRINT	Next executable statement
PUNCH	Next executable statement
READ	Next executable statement
REAL	Non-executable
RETURN	The first statement, or part of a statement, following the reference to this program
REWIND	Next executable statement
STOP	Terminates the program
SUBROUTINE	Non-executable
WRITE	Next executable statement

APPENDIX B: TABLE OF SOURCE PROGRAM CHARACTERS

Character	Card	BCD Tape	Storage	Character	Card	BCD Tape	Storage	Character	Card	BCD Tape	Storage	Character	Card	BCD Tape	Storage
					12				11				0		
1	1	01	01	A	1	61	21	J	1	41	41	/	1	21	61
					12				11				0		
2	2	02	02	B	2	62	22	K	2	42	42	S	2	22	62
					12				11				0		
3	3	03	03	C	3	63	23	L	3	43	43	T	3	23	63
					12				11				0		
4	4	04	04	D	4	64	24	M	4	44	44	U	4	24	64
					12				11				0		
5	5	05	05	E	5	65	25	N	5	45	45	V	5	25	65
					12				11				0		
6	6	06	06	F	6	66	26	O	6	46	46	W	6	26	66
					12				11				0		
7	7	07	07	G	7	67	27	P	7	47	47	X	7	27	67
					12				11				0		
8	8	10	10	H	8	70	30	Q	8	50	50	Y	8	30	70
					12				11				0		
9	9	11	11	I	9	71	31	R	9	51	51	Z	9	31	71
blank	blank	20	60	+	12	60	20	-	11	40	40	0	0	12	00
					12				11				0		
=	8-3	13	13	.	8-3	73	33	$	8-3	53	53	,	8-3	33	73
					12				11				0		
'	8-4	14	14	)	8-4	74	34	*	8-4	54	54	(	8-4	34	74

The characters $ and ' (8-4) can be used in FORTRAN only as alphameric text in an H field.

APPENDIX C: DIFFERENCES BETWEEN FORTRAN II and FORTRAN IV

1. All language items distinguished by a column 1 modal punch, except B, in FORTRAN II have been incorporated into FORTRAN IV by the Type statements as follows:

FORTRAN II Modal Punch	FORTRAN IV Type Statement (See the section, "The Type Statements")
Double-precision arithmetic - D	DOUBLE PRECISION
Complex arithmetic - I	COMPLEX
F-card - F	EXTERNAL

The DATA statement may be used to enter octal constants into a FORTRAN IV program; the programmer may write subroutines, using these constants, to handle Boolean arithmetic.

2. Function Naming.
   a. Where the initial character of a function name is used to denote the type as floating point (real) or fixed point (integer) in FORTRAN II, incompatibilities may arise. In FORTRAN IV, this difficulty is handled by the TYPE statements REAL and INTEGER, which define a variable name or function name as floating point or fixed point, respectively (see the section, "The Type Statements").
   b. The number of characters in an open, a closed, or an arithmetic statement function name in FORTRAN II is 4 through 7, ending in F; whereas, in FORTRAN IV, the number of characters is 1 through 6 and the final F has no meaning. In both cases, the first character of the function name must be alphabetic (see chapter 6).
   c. Built-in and arithmetic statement functions are not identified by a terminal F in FORTRAN IV; they are named in FORTRAN IV as described in b. above. The FORTRAN II library function is a FORTRAN IV FUNCTION subprogram which is internal to the Processor.
3. COMMON and EQUIVALENCE.
   a. In FORTRAN IV, EQUIVALENCE does not effect the ordering within COMMON, and it does not create a gap in COMMON storage; the only effect it can have on a COMMON block is to make its size greater than that indicated by the COMMON statements of the program (see the section, "The COMMON Statement").
   b. The FORTRAN IV COMMON statement may contain dimension information.

4. In FORTRAN IV, if an explicit type is given to a variable name which is used throughout the program as an ordinary variable and also as a dummy argument of an arithmetic statement function, the explicit type applies in both contexts.

5. Implicit multiplication, which occurs in FORTRAN II as a by-product of the arithmetic translator techniques, is not permitted in FORTRAN IV. Thus, the following combinations are not permitted in FORTRAN IV:

K( )
( )V
( )K

where V is a variable, K is a constant, and ( ) is any arithmetic expression within parentheses.

6. The FORTRAN II statements in column 1 are changed to the FORTRAN IV statements in column 2.

FORTRAN II Statement	FORTRAN IV Statement	
IF ACCUMULATOR OVERFLOW n, n_2	CALL OVERFL (j)	
IF QUOTIENT OVERFLOW n_1, n_2	CALL OVERFL (j)	
IF DIVIDE CHECK n, n_2	CALL DVCHK (j)	
IF (SENSE SWITCH i) n_1, n_2	CALL SSWTCH (i, j)	
SENSE LIGHT i	CALL SLITE (i)	
IF (SENSE LIGHT i) n_1, n_2	CALL SLITET (i, j)	
READ TAPE i, list	READ (i) list	Binary record
READ INPUT TAPE i, n, list	READ (i, n) list	BCD record
WRITE TAPE i, list	WRITE (i) list	Binary record
WRITE OUTPUT TAPE i, n, list	WRITE (i, n) list	BCD record

The FREQUENCY, READ DRUM, WRITE DRUM statements of FORTRAN II are not part of the FORTRAN IV language.

7. Additional FORTRAN IV statements.
   a. DATA (see the section, "The DATA Statement").
   b. BLOCK DATA (see the section, "The Block Data Subprogram").
   c. LOGICAL, an additional Type statement which defines variables to be used in logical computation (see the section "The Type Statement").

8. Differences in Output Produced by FORTRAN IV and FORTRAN II.

Programmers will find that the output produced by a source program in FORTRAN IV may not be the same as that provided by the identical program in FORTRAN II. Where differences do occur, they are attributed exclusively to the following differences which exist between FORTRAN IV and FORTRAN II.

   a. The logarithm subroutine of FORTRAN IV employs a new algorithm which yields more accurate results for most arguments than does the logarithm subroutine of FORTRAN II.
   b. Floating point constants which are written into the source program are converted by FORTRAN IV by a somewhat different algorithm than that used by FORTRAN II. The result is that more significance tends to be preserved and a more accurate conversion is achieved by FORTRAN IV as compared to its predecessor.
   c. The mathematical subroutines in FORTRAN IV are assembled by MAP, and those in FORTRAN II are assembled by FAP. The conversion routines in MAP provide more precise conversions for constants than do those in FAP. As a consequence, FORTRAN IV tends to produce more precise results than FORTRAN II for those subroutines which use the same algorithm (and its associated constants). The SIN/

COS subroutine is a very good example of this effect.

d. The order in which a sequence of multiplications (or of multiplications and divisions) is executed by the object program in FORTRAN IV may be different from that in FORTRAN II. If such a difference in ordering should occur, neither method may be considered superior to the other from the standpoint of computational accuracy.

APPENDIX D: LIMITATIONS ON SOURCE PROGRAM SIZE

In translating a source program into an object program, FORTRAN forms and utilizes various tables containing information about the source program. These tables are divided into two categories.

First Category Tables

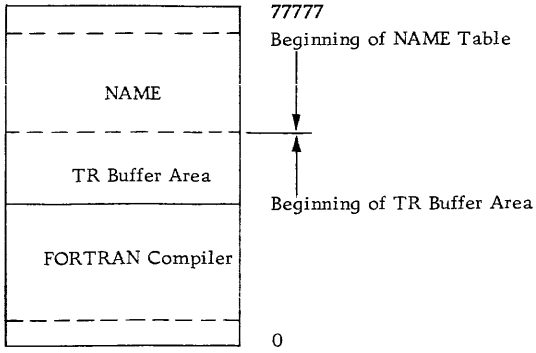
The first category of tables consists of those tables for which space is allocated only when a table is actually required.

Storage of First Category Tables

At such time as a first category table is required, a buffer is assigned by the Table Routine in the TR buffer area, which begins immediately after the FORTRAN compiler in core storage and extends toward the top of storage. The top of the TR buffer area is determined by the NAME table.

The NAME table is the only first category table that is not stored in the TR buffer area. Rather, the NAME table storage begins near the end of core storage and extends downward toward the TR buffer area.

The following map of core storage indicates the location and direction of the TR buffer area and the NAME table.



Essentially, the TR buffers and the NAME table share the same space but start filling the space from opposite ends. When they meet an overflow occurs.

Overflow of TR Buffer Area and NAME Table Storage

It is unlikely that the TR buffers and the NAME table would overflow each other; however, if they did, a failure to compile would result.

If such an overflow occurs, it would be possible to recompile the program only after reducing the size of the TR buffers or by modifying the program so that excessive table entries will be eliminated.

First Category Tables

The following tables belong to the first category. With the following description of what causes entries to be made in the tables, the programmer should be able to reduce the size of tables if an overflow occurs.

The following comments and definitions apply to the tables:

1. The phrase "literal appearance" means that each item must be counted each time it appears.
2. The BETA, LAMBDA, ASFD, and RANGE2 tables are used for one statement only and are released immediately upon completion of processing of that statement.
3. In computing the table size of BETA, LAMBDA, and RANGE2, all subscripts are ignored.
4. A subexpression is any term or expression within parentheses; a function argument is a subexpression.
5. Each of the FORTRAN operators is given a value when computing the size of the BETA and LAMBDA tables. The operator .NOT. is ignored.

Operator	Value
.OR.	1
.AND.	2
Relational Operators	3
+, -	4
*, /	5
**	6
Function Reference, ⊕	7

The symbol ⊕ for a function reference is used as follows: MAX (X, Y) in a FORTRAN statement is interpreted by the compiler as if it had been written MAX ⊕ X ⊕ Y; of course, the programmer does not use the function symbol ⊕ .

6. [x] is the greatest integer less than or equal to x.

ASFD: A one word entry is made for each argument of an arithmetic statement function definition.

BETA Table: The size of the BETA table is computed by the following procedure:

1. Ignoring all subexpressions:
   a. Scan the right side of the arithmetic statement for the first operator; add the value of this operator to a counter.
   b. Scan for the next operator; if the value of this operator is less than or equal to the value of the immediately preceding operator, continue scanning; if the value of this operator is greater than the immediately preceding operator, add the difference to the counter; continue scanning and totaling until the entire expression has been processed.
2. For each subexpression not an argument, add to the counter the difference between 7 and the value of the operator which immediately precedes the left parenthesis of the subexpression.
3. For each subexpression, previously ignored, perform the processing described under item 1. above, keeping a continuous total in the counter.
4. The value in the counter is the table size.

BRADD Table: The number of entries is equal to the number of entries in the BRANCH table.

BRANCH Table: An entry is made for each statement that may result in a transfer of control. The number of words in each entry is found in the following table:

Statement	Number of Words in Entry
Logical IF	1
Arithmetic IF	3
Unconditional GO TO	1
Assigned GO TO	number of statement numbers in list
Computed GO TO	number of statement numbers in list

COMMON Table: A one word entry is made for each literal appearance of variables in a COMMON statement. A one word entry is also made for each specification of blank or labeled COMMON.

CONST Table: An entry is made for each literal appearance of a constant. The number of words in each entry is $2+\left\lceil\frac{n-1}{6}\right\rceil$, where n is the number of non-blank characters used for the representation of the constant in the program.

DDO Table: A two word entry is made for each implied DO in a DATA statement.

DELTA Table: An entry is made for each assigned and computed GO TO. The number of words in each entry is $1+\lceil n/2\rceil$, where n is the number of statement numbers in the list.

DIM Table: An entry is made for each assignment of a dimension to an array. The number of words in each entry is $1+\left\lceil\frac{n}{2}\right\rceil$, where n is the number of dimensions.

DLIST Table: An entry is made for each DATA statement. The entry size is n+d+v+2d where n is the number of lists in the DATA statement, v is the number of variables, and d is the number of implied DOs.

DLIT Table: A one word entry is made for each list in a DATA statement. An entry is also made for each literal (constant) appearing in a DATA statement. The size of this latter entry is $2+\left\lceil\frac{n-1}{6}\right\rceil$, where n is the number of non-blank characters in the literal.

EQUIT Table: A one word entry is made for each literal appearance of variables in an EQUIVALENCE statement.

FORMAT Table: An entry is made for each FORMAT statement. The number of words in each entry is $2+\left\lceil\frac{n-1}{6}\right\rceil$, where n is the number of characters following the word FORMAT and blanks are retained in an alphameric field, but deleted elsewhere.

FORTAG Table: A two word entry is made for each literal appearance of a subscripted variable in any statement except a DATA statement.

FORVAL Table: A one word entry is made for each literal appearance of a non-subscripted integer variable on the left side of an arithmetic statement, in an I/O list, in COMMON statements, and in the argument list of a SUBROUTINE subprogram.

INCDO Table: A one word entry is made for each left parenthesis that is not used for subscripts in a DATA statement. This table lasts for one statement only.

LAMBDA Table: The size of the LAMBDA table is computed by the following procedure.

1. Set a counter equal to the number of constants, variables, and functions in the arithmetic expression.
2. Ignoring all subexpressions, scan the expression for the first operator: if the next operator has the same value as the immediately

preceding operator, continue scanning; if the value of the next operator is less than the immediately preceding operator, add one to the counter and continue scanning; if the value of the next operator is greater than the immediately preceding operator, add the difference in values to the counter and continue scanning and totaling until the entire expression has been processed.

3. For each subexpression, perform the processing described under item 2, keeping a cumulative value in the counter.
4. The value in the counter is the table size.

NAME Table: A two word entry is made for each variable, function name, and subroutine name (other than the name of the program being compiled) in the source program.

RANGE2 Table: A one word entry is made for each function argument that is an expression.

SUBARG Table: An entry is made for each FUNCTION or SUBROUTINE subprogram. The number of words in each entry is $1 + \lceil n/2 \rceil$, where n is the number of subprogram arguments.

TAU Table: A one word entry is made for each appearance of a unique subscript combination. Subscript combinations that are actually the same as others are considered unique if there is a difference in the size of the dimensions of the subscripts.

TDO Table: A three word entry is made for each DO or implied DO.

TDOB Table: A one word entry is made for each DO or implied DO.

TEIFNO Table: A one word entry is made for each executable statement which has a statement number.

Second Category Tables

The second category of tables consists of those tables for which space is specifically allocated in the Compiler program. The space allocated for each of these tables remains assigned even though the table may not be required.

It is unlikely that a compilation failure will result from an overflow of one of these tables.

For a complete description of these tables and their limitations, see the FORTRAN section of the 7090/7094 IBJOB Processor publication.

APPENDIX E: OPTIMIZATION OF ARITHMETIC EXPRESSIONS

In order to optimize the object program, a sequence of operations on the multiply-divide (*, /) level may be reordered. The reordering tends to an alternation of the multiply and divide operations. It occurs where all elements of the expression are real type. This is done on the assumption that mathematically equivalent expressions are computationally equivalent.

Where the multiply-divide expression involves mixed real and complex types, the operations on the real types occur first and are alternated.

Where the order of operations is considered significant, the programmer may use nested parentheses in the expression.

There are two distinct ways a given floating point constant may enter computation within a FORTRAN program. It may be

1. written into the FORTRAN source program, or
2. read in as data during execution.

In the first version of the 7090/7094 FORTRAN Compiler (IBFTC) and its Library (IBLIB), there may be a difference in the low-order binary bit of the same constant arising from these two sources.

INDEX

Addition 9, 10
Adjustable dimensions 28
Alphameric fields 17
Arithmetic expression 9
Arithmetic GO TO 13, 32
Arithmetic IF 12, 13, 32
Arithmetic operations 9
Arithmetic operators 9
Arithmetic statement 5, 11
Arithmetic statement function 22
Arrays 7, 16, 28, 29
  input/output of 16
  storage arrangement 8
ASFD Table 36
ASSIGN 12, 32
Assigned GO TO 12, 32
ASSIGN TO 12, 32

BACKSPACE 15, 21.1, 32
BETA Table 36
Blank COMMON 29
Blank Fields 18
Blanks 5
BLOCK DATA 27, 32
BRADD Table 36
BRANCH Table 36
Built-in function 22, 23

CALL 14, 26, 32
Carriage control 19
Coding sheet 5, 6
Comments card 5
COMMON 8, 28, 32, 36
COMPLEX 30, 32
Complex constants 7
Complex number fields 17
Complex variables 8
Computed GO TO 12, 32
Constants 7, 9
  complex 7
  double-precision 7
  integer 7
  logical 7
  real 7
CONST Table 36
Continuation card 5
CONTINUE 13, 32
Control statements 5, 12, 13, 14
Conversion of data 16, 17, 18, 19, 20
  A-Conversion 17
  D-Conversion 16, 17, 18, 19, 20
  E-Conversion 16, 17, 18, 19, 20
  F-Conversion 16, 17, 18, 19, 20
  H-Conversion 18
  I-Conversion 16, 17
  O-Conversion 16, 17
  X-Conversion 18

DATA 5, 8, 30, 31, 32
Data conversion 16, 17, 18, 19, 20
  alphameric 17
  blank 18
  numeric 16
Data fields 16, 17, 18, 19
  alphameric 17
  blank 18
  complex numbers 17
  logical 18
  numeric 16
Data literal 30, 31
DDO Table 36
Defining subroutines 22
DELTA Table 36
DIMENSION 8, 28, 32
DIM Table 36
D-literals 9, 10
DLIST Table 36
DLIT Table 36
DO 13, 32
  indexing 13
  loops 13
  nested 13
  range of 13
  restrictions 13
  subprograms 13
DOUBLE PRECISION 30, 32
  constants 7
  variables 7
DUMP Subprogram 14, 26

END 14, 32
END FILE 15, 21.1, 32
EQUIT Table 36
EQUIVALENCE 29, 32
EXIT subprogram 14, 26
Exponentiation 9, 10
Expressions 7, 9
  arithmetic 9
  logical 9
EXTERNAL 25, 30, 32

Field format 18
First category table 35
FORMAT 5, 15, 16-20, 32
FORMAT Table 36
FORTAG Table 36
FORTRAN II 33, 34
FORTRAN IV
  coding sheet 5
  source program 5
  statement 5
FORTRAN statements 5, 6
  arithmetic 5
  control 5
  input/output 5
  subprogram 5
  specification 5
FORVAL Table 36
Functions name 9
Functions 10, 20
  arithmetic statement 22
  built-in 22
FUNCTION subprogram 22, 24

General input/output statements 15, 21
GO TO 12, 32

H-Conversion 5
Hierarchy of operations 9, 10
H-field 5

Identification 5
IF 12, 13, 32
  arithmetic 12
  logical 12, 13
Implicit variable 8
INCDO Table 36
Indexing DOs 13
Integer constants 7
Integer variables 8
Information literal 30
Input data 19, 20
Input/output devices 21
Input/output of arrays 12, 20
Input/output statements 5, 15-21, 32
  general 15, 21, 32
  manipulative 15, 21.1, 32
  non-executable 15, 16, 32
Input/output unit designation 21.2
INTEGER 30, 32

LAMBDA Table 36
Limitations of size 35
List specification 15
LOGICAL 30, 32
Logical constant 9, 10
Logical expression 9, 10
Logical fields 9
Logical function 10
Logical IF 12, 32
Logical operations 9
Logical operators 9, 10
Logical variable 8, 9, 10

Machine indicator tests 26
Manipulative input/output 15, 20
Mathematical subroutines 25, 26
Multiple record format 19
Multiplication 9, 10

NAME 35
NAMELIST 15, 20, 21, 32
NAME Table 37
Naming subroutines 22
Nested DO loops 13
Non-executable input/output statements 15, 16, 32
Non-subscripted variable 9
Normal sequencing 32
Numeric fields 16

Operators 9
Order of execution 32

Parenthesis in expressions 9
PAUSE 14, 32
PDUMP subprogram 14, 27
PUNCH 15, 21.1, 32
PRINT 15, 19, 21.1, 32

Range of DO 13
RANGE2 Table 37
READ 15, 21, 32
REAL 30, 32
Real Constants 7
Real variables 8
Relational operations 10
Relational operators 10
Repetition of fields 18
Restriction on DO 13
RETURN 14, 32
REWIND 15, 21.1, 32

Scale factor 18
Second category table 37
Sequencing 32
Source program coding 5, 6
Source program size 35
Specification statement 5, 28-31
START key 14
Statements 5, 6
STOP 14, 32
SUBARG Table 37
Subprograms 22
Subprogram statement 5
Subroutine 22
SUBROUTINE Subprogram 22, 24, 32
Subscripted variables 8, 9
Subscripts 7, 8
Subtraction 9, 10
Symbolic input/output units 21

Table Routine 35
TAU Table 37
TDO Table 37
TDOB Table 37
TEIFNO Table 37
Type statements 30

Unconditional GO TO 12, 32

Variable name 8
Variables 7, 8, 9, 10
  subscripted 8, 9
  non-subscripted 9
Variable type specification 8

WRITE 15, 19, 21.1, 32

Printed in U.S.A. C28-6274-2

IBM

International Business Machines Corporation
Data Processing Division
112 East Post Road, White Plains, N.Y., 10601