digital

decsystem10

mathematical languages handbook

second edition

fortran basic algol

decsystem10 handbook series

digital

decsystem10

mathematical languages handbook

second edition

Additional copies of this handbook may be ordered from:
Program Library, DEC, Maynard, Mass. 01754. Order code: DEC-10-KRZB-D.

decsystem10 handbook series

The material in this handbook is for information purposes and is subject to change without notice.

tab index

fortran

basic

algol

NOTICE

For the reader's convenience:

1) Consecutive page numbers have been added to the top center of each page in the handbook; these numbers have the form –nn..– (for example –25–) and are supplied in addition to the standard document numbers printed at the bottom center of each page.

2) The appropriate document name has been added to the top outside corner of each page of the handbook.

3) A global index comprised of the merged and alphabetized entries of all of the indexes which were previously part of the documents contained by the handbook is supplied at the end of the handbook. The global index replaces the individual document indexes.

4) The entries of the global index and the Table of Contents for each document reference the *consecutive* page numbers located at the top center of each page.

5) Black locator tabs are printed on the outside edge of the first ten pages of each document in the handbook. A tab locator page on which each set of tabs is identified by the name of the document which they represent is supplied at the front of the handbook.

DEC-10-AFDO-D

decsystem10

FORTRAN IV PROGRAMMER'S REFERENCE MANUAL

The information in this document reflects the software as of Version 26 of the FORTRAN Compiler and Version 32 of the run-time operating system (LIB40).

DIGITAL EQUIPMENT CORPORATION • MAYNARD, MASSACHUSETTS

1st Printing March 1967
2nd Printing (Rev) November 1967
3rd Printing (Rev) September 1968
4th Printing April 1969
5th Printing June 1969
6th Printing September 1969
7th Printing (Rev) February 1970
Update Pages October 1970
Update Pages February 1971
Update Pages October 1971
Update Pages May 1972

CONTENTS

CONTENTS (Cont)

CONTENTS (Cont)

CONTENTS (Cont)

ILLUSTRATIONS

TABLES

PREFACE

This is a reference manual describing the specific statements and features of the FORTRAN IV language for the PDP-10. It is written for the experienced FORTRAN programmer who is interested in writing and running FORTRAN IV programs alone or in conjunction with MACRO-10 programs in the single-user or time-sharing environment. Familiarity with the basic concepts of FORTRAN programming on the part of the user is assumed. PDP-10 FORTRAN IV conforms to the requirements of the USA Standard FORTRAN.

INTRODUCTION TO THE FORTRAN IV SYSTEM

The FORTRAN compiler translates source programs written in the FORTRAN IV language into the machine language of the PDP-10. This translated version of the FORTRAN program exists as a retrievable, relocatable binary file on some storage device. All relocatable binary filenames have the extension .REL if they reside on a directory-oriented device (disk or DECtape). Binary files may also be created by the MACRO-10 assembler (see Chapter 9)[1].

In order for the FORTRAN program to be processed, the Linking Loader must load the relocatable binary file into core memory. Also loaded are any relocatable binary files found in the FORTRAN library (LIB40) which are necessary for the program's execution. Within the FORTRAN source program, the library files may be called explicitly, such as SIN, in the statement

```
X = SIN(Y)
```

or implicitly, such as FLOUT., the floating-point to ASCII conversion routine, which is implied in the following statements.

```
      PRINT    3,X
3     FORMAT(1X,F4.2)
```

A FORTRAN main program and its FORTRAN and/or MACRO-10 subprograms may be compiled or assembled separately and then linked together by the Linking Loader at load time. The core image may then be saved on a storage device. When saved on a directory storage device, these files have the extension .SAV in a multiprogramming Monitor system and .SVE in a single-user Monitor system.

The Time-Sharing Monitors act as the interface between the user and the computer so that all users are protected from one another and appear to have system resources available to themselves. Several user programs are loaded into core at once and the Time-Sharing Monitors schedule each program to run for a certain length of time. All Monitors direct data flow between I/O devices and user programs, making the programs device independent, and overlap I/O operations concurrently with computations.

In a multiprogramming system, all jobs reside in core and the scheduler decides which of these jobs should run. In a swapping system, jobs can exist on an external storage device (usually disk) as well as in core. The scheduler

[1]For further information on the MACRO-10 assembler, see the MACRO-10 ASSEMBLER manual, DEC-10-AMZB-D.

decides not only which job is to run but also when a job is to be swapped out onto the disk or brought back into core.

The number of users that can be handled by a given size time-sharing configuration is further increased by using the reentrant user-programming capability. This means that a sequence of instructions may be entered by more than one user job at a time. Therefore, a single copy of a reentrant program may be shared by a number of users at the same time to increase system economy. The FORTRAN compiler and operating system are both reentrant.

SECTION I
The PDP-10 FORTRAN IV Language

The seven chapters of this section deal with the PDP-10 FORTRAN IV language. Included in these chapters are the language elements of FORTRAN IV and the five categories of FORTRAN IV statements (arithmetic, control, input/output, specification, and subprogram).

CHAPTER 1
INTRODUCTION TO THE FORTRAN LANGUAGE

The term FORTRAN IV (FORmula TRANslation) is used interchangeably to designate both the FORTRAN IV language and the FORTRAN IV translator or compiler. The FORTRAN IV language is composed of mathematical-form statements constructed in accordance with precisely formulated rules. FORTRAN IV programs consist of meaningful sequences of FORTRAN statements intended to direct the computer to perform the specified operations and computations.

The FORTRAN IV compiler is itself a computer program that examines FORTRAN IV statements and tells the computer how to translate the statements into machine language. The compiler runs in a minimum of 9K of core. The program written in FORTRAN IV language is called the source program. The resultant machine language program is called the object program. Digital's small FORTRAN compiler, which runs in 5.5K of core, is virtually identical to the larger compiler, except for differences explained in Appendix 2. Operating procedures and diagnostic messages for both compilers are explained in the PDP-10 System Users Guide (DEC-10-NGCC-D).

1.1 LINE FORMAT

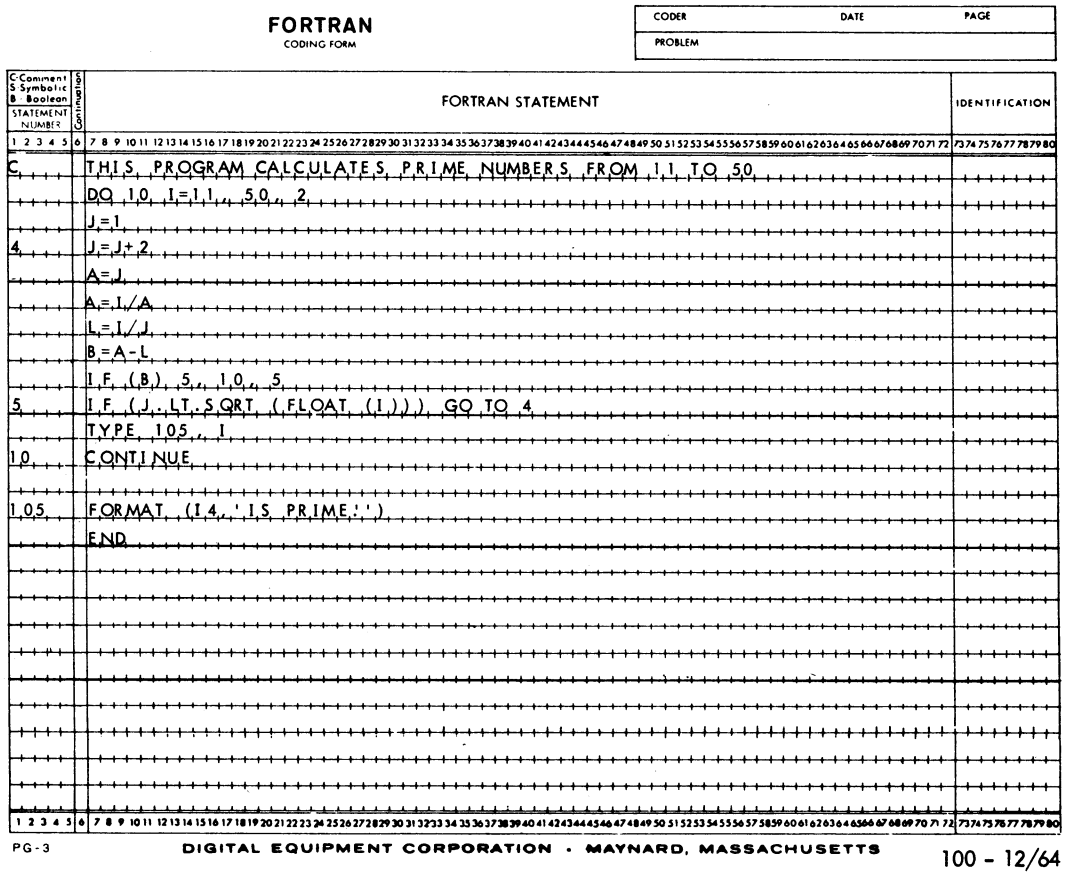
Each line of a FORTRAN program consists of three fields: statement number field, line continuation field, and statement field. A typical FORTRAN program is shown in Figure 1-1.

1.1.1 Statement Number Field

A statement number consists of from one to five digits in columns 1-5. Leading zeros and all blanks in this field are ignored. Statement numbers may be in any order and must be unique. Any statement referenced by another statement must have a statement number. For source programs prepared on a teletypewriter, a horizontal tab may be used to skip to the statement field with from 0 through 5 characters in the label field. This is the only place a tab is not treated as a space.

1.1.2 Line Continuation Field

If a FORTRAN statement is so large that it cannot conveniently fit into one statement field, the statement fields of up to 19 additional lines may be used to specify the complete statement. Any line which is not continued, or the first line of a sequence of continued lines, must have a blank or zero in column 6. Continuation lines must



FORTRAN
CODING FORM

CODER | DATE | PAGE
PROBLEM

C Comment S Symbolic B Boolean | STATEMENT NUMBER | Continuation | FORTRAN STATEMENT | IDENTIFICATION

```
C     THIS PROGRAM CALCULATES PRIME NUMBERS FROM 1 TO 50
      DO 10 I=1,,50,,2
      J=1
4     J=J+2
      A=J
      A=I/A
      L=I/J
      B=A-L
      IF (B) 5,,10,,5
5     IF (J.LT.SQRT (FLOAT (I))) GO TO 4
      TYPE 105,,I
10    CONTINUE

105   FORMAT (I4,' IS PRIME.')
      END
```

PG-3 DIGITAL EQUIPMENT CORPORATION · MAYNARD, MASSACHUSETTS 100 - 12/64

Figure 1-1 Typical FORTRAN Coding Form

have a character other than blank or zero in column 6. If a continuation line is desired when a TAB is used in the statement number field, a digit from 1 to 9 must immediately follow the TAB.

1.1.3 Statement Field

Any FORTRAN statement, as described in later sections, may appear in the statement field (columns 7-72). Except for alphanumeric data within a FORMAT statement, DATA statement, or literal constant, blanks (spaces) and TABS are ignored and may be used freely for appearance purposes. Thus the following statements are equivalent.

```
END (tab) FILE (tab) 2
END (space) FILE (space) 2
ENDFILE2
```

1.1.4 Comment Line

Any line that starts with one of the characters $ * / or the letter C in column 1 is interpreted as a line of comments. Comment lines are printed onto any listings requested but are otherwise ignored by the compiler. Columns 2-72 may be used in any format for comment purposes. A comment line must not immediately precede a continuation line.

As an aid for program debugging, the letter D in column 1 causes the line to be interpreted as a comment unless the /I switch appears in the command string. (Refer to Table 11-1 for Compile Switch options.) If the /I switch is present, the letter D in column 1 is interpreted as a space and the line is compiled as a program statement.

1.2 CHARACTER SET

The following characters are used in the FORTRAN IV language:

Blank	0	@	P
!	1	A	Q
"	2	B	R
#	3	C	S
$	4	D	T
%	5	E	U
&	6	F	V
'	7	G	W
(	8	H	X
)	9	I	Y
*	:	J	Z
+	;	K	↑
,	<	L	
-	=	M	
.	>	N	
/	?	O	

NOTE

ASCII characters greater than Z (132_8) are replaced by the error character "↑". See Chapter 12 for the internal representation of these characters.

CHAPTER 2
CONSTANTS, VARIABLES, AND EXPRESSIONS

The rules for defining constants and variables and for forming expressions are described in this chapter.

2.1 CONSTANTS

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

2.1.1 Integer Constants

An integer constant consists of from one to eleven decimal digits written without a decimal point. A negative constant must be preceded by a minus sign. A positive constant may be preceded by a plus sign.

Examples:
```
3
+10
-528
8085
```

An integer constant must fall within the range $-2^{35}+1$ to $2^{35}-1$. When used for the value of a subscript, the value of the integer constant is taken as modulo 2^{18}.

2.1.2 Real Constants

Real constants are written as a string of decimal digits including a decimal point. A real constant may consist of any number of digits but only the leftmost 9 digits appear in the compiled program. Real constants may be given a decimal scale factor by appending an E followed by a signed integer constant. The field following the letter E must not be blank, but may be zero.

Examples:
```
 15.
  0.0
   .579
-10.794
  5.0E3(i.e., 5000.)
  5.0E+3(i.e., 5000)
  5.0E-3(i.e., 0.005)
```

A real constant has precision to eight digits. The magnitude must lie approximately within the range 0.14×10^{-38} to 1.7×10^{38}. Real constants occupy one word of PDP-10 storage.

2.1.3 Double Precision Constants

A double precision constant is specified by a string of decimal digits, including a decimal point, which are followed by the letter D and a signed decimal scale factor. The field following the letter D must not be blank, but may be zero.

Examples:
```
24.671325982134D0
3.6D2 (i.e., 360.)
3.6D-2 (i.e., .036)
3.0D0
```

Double precision constants have precision to 16 digits. The magnitude of a double precision constant must lie approximately between 0.14×10^{-38} and 1.7×10^{38}. Double-precision constants occupy two words of PDP-10 storage.

2.1.4 Octal Constants

A number preceded by a double quote represents an octal constant. An octal constant may appear in an arithmetic or logical expression or a DATA statement. Only the digits 0-7 may be used and only the last twelve digits are significant. A minus sign may precede the octal number, in which case the number is negated. A maximum of 12 octal digits are stored in each 36-bit word.

Examples:
```
"7777
"-31563
```

2.1.5 Complex Constants

FORTRAN IV provides for direct operations on complex numbers. Complex constants are written as an ordered pair of real constants separated by a comma and enclosed in parentheses.

Examples:
```
(.70712, -.70712)
(8.763E3,2.297)
```

The first constant of the pair represents the real part of the complex number, and the second constant represents the imaginary part. The real and imaginary parts may each be signed. The enclosing parentheses are part of the constant and always appear, regardless of context. Each part is internally represented by one single-precision floating point word. They occupy consecutive locations of PDP-10 storage.

FORTRAN IV arithmetic operations on complex numbers, unlike normal arithmetic operations, must be of the form:

$$A \pm B = a_1 \pm b_1 + i(a_2 \pm b_2)$$

$$A * B = (a_1 b_1 - a_2 b_2) + i(a_2 b_1 + a_1 b_2)$$

$$A / B = \frac{(a_1 b_1 + a_2 b_2)}{b_1^2 + b_2^2} + i \frac{(a_2 b_1 - a_1 b_2)}{b_1^2 + b_2^2}$$

where $A = a_1 + ia_2$, $B = b_1 + ib_2$, and $i = \sqrt{-1.}$

2.1.6 Logical Constants

The two logical constants, .TRUE. and .FALSE., have the internal values -1 and 0, respectively. The enclosing periods are part of the constant and always appear.

Logical constants may be entered in DATA or input statements as signed octal integers (-1 and 0). Logical quantities may be operated on in either arithmetic or logical statements. Only the sign is tested to determine the truth value of a logical variable.

2.1.7 Literal Constants

A literal constant may be in either of two forms:

a. A string of alphanumeric and/or special characters enclosed in single quotes; two adjacent single quotes within the constant are treated as one single quote.

b. A string of characters in the form

$$nHx_1x_2\ldots x_n$$

where $x_1x_2\ldots x_n$ is the literal constant, and n is the number of characters following the H.

Literal constants may be entered in DATA statements or input statements as a string of up to 5 7-bit ASCII characters per variable (10 characters if the variable is double-precision or complex). Literal constants may be operated on in either arithmetic or logical statements.

NOTE

Literal constants used as subprogram arguments will have a zero word as an end-of-string indicator.

Examples:
```
CALL SUB ('LITERAL CONSTANT')
'DONT''T'
5HDON'T
A = 'FIVE' + 42
B = (5HABCDE .AND. ''376)/2
```

2.2 VARIABLES

A variable is a quantity whose value may change during the execution of a program. Variables are specified by name and type. The name of a variable consists of one or more alphanumeric characters, the first one of which must be alphabetic. Only the first six characters are interpreted as defining the variable name. The type of variable (integer, real, logical, double precision, or complex) may be specified explicitly by a type declaration statement or implicitly by the IMPLICIT statement. If the variable is not specified in this manner, then a first letter of I, J, K, L, M or N indicates a fixed point (integer) variable; any other first letter indicates a floating-point (real) variable. Variables of any type may be either scalar or array variables. When used in a subscript or as an index to a DO Statement, the value of the integer variable is taken as modulo 2^{18}.

2.2.1 Scalar Variables

A scalar variable represents a single quantity.

Examples:
```
A
G2
POPULATION
```

2.2.2 Array Variables

An array variable represents a single element of an n dimensional array of quantities. The variable is denoted by the array name followed by a subscript list enclosed in parentheses. The subscript list is a sequence of integer expressions, separated by commas. The expressions may be of any form or type providing they are explicitly changed to type integer when each is completely evaluated. Each expression represents a subscript, and the values of the expressions determine the array element referred to. For example, the row vector A_i would be represented by the subscripted variable A(J), and the element, in the second column of the first row of the square matrix A, would be represented by A(1,2). Arrays may have any number of dimensions.

Examples:
```
Y(1)
STATION (K)
A (3* K+2, I, J-1)
```

The three arrays above (Y, STATION, and A) would have to be dimensioned by a DIMENSION, COMMON, or type declaration statement prior to their first appearance in an executable statement or in a DATA or NAMELIST statement. (Array dimensioning is discussed in Chapter 6).

1-Dimensional Array A(10)

A(1)	A(2)	A(3)	A(4)	A(5)	A(6)	A(7)	A(8)	A(9)	A(10)

CONSECUTIVE STORAGE LOCATIONS ——————————

2-Dimensional Array B(5,5)

1	B(1,1)	6	B(1,2)	11	B(1,3)	16	B(1,4)	21	B(1,5)
2	B(2,1)	7	B(2,2)	12	B(2,3)	17	B(2,4)	22	B(2,5)
3	B(3,1)	8	B(3,2)	13	B(3,3)	18	B(3,4)	23	B(3,5)
4	B(4,1)	9	B(4,2)	14	B(4,3)	19	B(4,4)	24	B(4,5)
5	B(5,1)	10	B(5,2)	15	B(5,3)	20	B(5,4)	21	B(5,5)

B(3,1) IS THE THIRD STORAGE WORD IN SEQUENCE
B(3,4) IS THE EIGHTEENTH STORAGE WORD IN SEQUENCE

3-Dimensional Array C(5,5,5)

										101	C(1,1,5)	106	C(1,2,5)	111	C(1,3,5)	116	C(1,4,5)	121	C(1,5,5)
										102	C(2,1,5)	107	C(2,2,5)	112	C(2,3,5)	117	C(2,4,5)	122	C(2,5,5)
						76	C(1,1,4)	81	C(1,2,4)	86	C(1,3,4)	91	C(1,4,4)	96	C(1,5,4)	118	C(3,4,5)	123	C(3,5,5)
						77	C(2,1,4)	82	C(2,2,4)	87	C(2,3,4)	92	C(2,4,4)	97	C(2,5,4)	119	C(4,4,5)	124	C(4,5,5)
				51	C(1,1,3)	56	C(1,2,3)	61	C(1,3,3)	66	C(1,4,3)	71	C(1,5,3)	98	C(3,5,4)	120	C(5,4,5)	125	C(5,5,5)
				52	C(2,1,3)	57	C(2,2,3)	62	C(2,3,3)	67	C(2,4,3)	72	C(2,5,3)	99	C(4,5,4)				
		26	C(1,1,2)	31	C(1,2,2)	36	C(1,3,2)	41	C(1,4,2)	46	C(1,5,2)	73	C(3,5,3)	100	C(5,5,4)				
		27	C(2,1,2)	32	C(2,2,2)	37	C(2,3,2)	42	C(2,4,2)	47	C(2,5,2)	74	C(4,5,3)						
1	C(1,1,1)	6	C(1,2,1)	11	C(1,3,1)	16	C(1,4,1)	21	C(1,5,1)	48	C(3,5,2)	75	C(5,5,3)						
2	C(2,1,1)	7	C(2,2,1)	12	C(2,3,1)	17	C(2,4,1)	22	C(2,5,1)	49	C(4,5,2)								
3	C(3,1,1)	8	C(3,2,1)	13	C(3,3,1)	18	C(3,4,1)	23	C(3,5,1)	50	C(5,5,2)								
4	C(4,1,1)	9	C(4,2,1)	14	C(4,3,1)	19	C(4,4,1)	24	C(4,5,1)										
5	C(5,1,1)	10	C(5,2,1)	15	C(5,3,1)	20	C(5,4,1)	25	C(5,5,1)										

C(1,3,2) is the 36th storage word in sequence.

C(1,1,5) is the 101st storage word in sequence.

Figure 2-1 Array Storage

Arrays are stored in increasing storage locations with the first subscript varying most rapidly and the last subscript varying least rapidly. For example, the 2-dimensional array B(I,J) is stored in the following order: B (1,1), B (2,1),..., B (I,1),B (1,2),B (2,2),...,B (I,2),...,B (I,J).

2.3 EXPRESSIONS

Expressions may be either numeric or logical. To evaluate an expression, the object program performs the calculations specified by the quantities and operators within the expression.

2.3.1 Numeric Expressions

A numeric expression is a sequence of constants, variables, and function references separated by numeric operators and parentheses in accordance with mathematical convention and the rules given below.

The numeric operators are +, -, *, /, **, denoting, respectively, addition, subtraction, multiplication, division, and exponentiation.

In addition to the basic numeric operators, function references are also provided to facilitate the evaluation of functions such as sine, cosine, and square root. A function is a subprogram which acts upon one or more quantities, called arguments, to produce a single quantity called the function value. Function references are denoted by the identifier, which names the function (such as SIN, COS, etc.), followed by an argument list enclosed in parentheses:

identifier(argument, argument, ..., argument)

At least one argument must be present. An argument may be an expression, an array identifier, a subprogram identifier, or an alphanumeric string.

Function type is given by the type of the identifier which names the function. The type of the function is independent of the types of its arguments. (See Chapter 7, Section 7.4.1.1.)

A numeric expression may consist of a single element (constant, variable, or function reference):

```
2.71828
Z(N)
TAN(THETA)
```

Compound numeric expressions may be formed by using numeric operations to combine basic elements:

```
X+3.
TOTAL/A
TAN(PI*M)
(X+3.) -(TOTAL/A) * TAN (PI*M)
```

Compound numeric expressions must be constructed according to the following rules:

a. With respect to the numeric operators +, -, *, /, any type of quantity (logical, octal, integer, real, double precision, complex or literal) may be combined with any other, with one exception: a complex quantity cannot be combined with a double precision quantity.

The resultant type of the combination of any two types may be found in Table 2-1. The conversions between data types will occur as follows:

(1) A literal constant will be combined with any integer constant as an integer and with a real or double word as a real or double word quantity. (Double word refers to both double precision and complex.)

(2) An integer quantity (constant, variable, or function reference) combined with a real or double word quantity results in an expression of the type real or double word respectively; e.g., an integer variable plus a complex variable will result in a complex subexpression. The integer is converted to floating point and then added to the real part of the complex number. The imaginary part is unchanged.

(3) A real quantity (constant, variable, or function reference) combined with a double word quantity results in an expression that is of the same type as the double word quantity.

(4) A logical or octal quantity is combined with an integer, real, or double word quantity as if it were an integer quantity in the integer case, or a real quantity in the real or double word case (i.e., no conversion takes place).

b. Any numeric expression may be enclosed in parentheses and considered to be a basic element.

```
(X+Y)/2
(ZETA)
(COS(SIN(PI*M)+X))
```

Table 2-1
Types of Resultant Subexpressions

+,-,*,/		Type of Quantity				
		Real	Integer	Complex	Double Precision	Logical, Octal, or Literal
Type of Quantity	Real	Real	Real	Complex	Double Precision	Real
	Integer	Real	Integer	Complex	Double Precision	Integer
	Complex	Complex	Complex	Complex	Not Allowed	Complex
	Double Precision	Double Precision	Double Precision	Not Allowed	Double Precision	Double Precision
	Logical, Octal, or Literal	Real	Integer	Complex	Double Precision	Logical, Octal, or Literal

c. Numeric expressions which are preceded by a + or – sign are also numeric expressions:

```
+X
-(ALPHA*BETA)
-SQRT(-GAMMA)
```

d. If the precedence of numeric operations is not given explicitly by parentheses, it is understood to be the following (in order of decreasing precedence):

Operator	Explanation
**	numeric exponentiation
*and/	numeric multiplication and division
+and-	numeric addition and subtraction

In the case of operations of equal hierarchy, the calculation is performed from left to right.

e. No two numeric operators may appear in sequence. For instance:

```
X*-Y
```

is improper. Use of parentheses yields the correct form:

```
X*(-Y)
```

By use of the foregoing rules, all permissible numeric expressions may be formed. As an example of a typical numeric expression using numeric operators and a function reference, the expression for one of the roots of the general quadratic equation:

$$\frac{-b+\sqrt{b^2-4ac}}{2a}$$

would be coded as:

```
(-B+SQRT(B**2-4.*A*C))/(2.*A)
```

2.3.2 Logical Expressions

A logical expression consists of constants, variables, function references, and arithmetic expressions, separated by logical operators or relational operators. Logical expressions are provided in FORTRAN IV to permit the implementation of various forms of symbolic logic. Logical masks may be represented by using octal constants. The result of a logical expression has the logical value TRUE (negative) or FALSE (positive or zero) and therefore, only uses one word.

2.3.2.1 Logical Operators - The logical operators, which include the enclosing periods and their definitions, are as follows, where P and Q are expressions:

.NOT.P	Has the value .TRUE. only if P is .FALSE., and has the value .FALSE. only if P is .TRUE.
P.AND.Q	Has the value .TRUE. only if P and Q are both .TRUE., and has the value .FALSE. if either P or Q is .FALSE.
P.OR.Q	(Inclusive OR) Has the value .TRUE. if either P or Q is .TRUE., and has the value .FALSE. only if both P and Q are .FALSE.
P.XOR.Q	(Exclusive OR) Has the value .TRUE. if either P or Q but not both are .TRUE., and has the value .FALSE. otherwise.
P.EQV.Q	(Equivalence) Has the value .TRUE. if P and Q are both .TRUE. or both .FALSE., and has the value .FALSE. otherwise.

Logical expressions are evaluated by combining the full word values of P and Q (only the high-order part if P and Q are double precision, only the real part if P and Q are complex) using the appropriate logical operator. The result is TRUE if it is arithmetically negative and FALSE if it is arithmetically positive or zero.

Logical operators may be used to form new variables, for example,

```
X = Y.AND.Z
E = E.XOR."400000000000
```

2.3.2.2 Relational Operators - The relational operators are as follows:

Operator	Relation
.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 enclosing periods are part of the operator and must be present.

Mixed expressions involving integer, real, and double precision types may be combined with relationals. The value of such an expression will be .TRUE. (-1) or .FALSE. (0).

The relational operators .EQ. and .NE. may also be used with COMPLEX expressions. (Double word quantities are equal if the corresponding parts are equal.)

A logical expression may consist of a single element (constant, variable, function reference, or relation):

```
.TRUE.
X.GE.3.14159
```

Single elements may be combined through use of logical operators to form compound logical expressions, such as:

```
TVAL.AND.INDEX
BOOL(M).OR.K.EQ.LIMIT
```

Any logical expression may be enclosed in parentheses and regarded as an element:

```
(T.XOR.S).AND.(R.EQV.Q)
CALL PARITY ((2.GT.Y.OR.X.GE.Y).AND.NEVER)
```

Any logical expression may be preceded by the unary operator .NOT. as in:

```
.NOT.T
.NOT.X+7.GT.Y+Z
BOOL(K).AND..NOT.(TVAL.OR.R)
```

No two logical operators may appear in sequence, except in the case where .NOT. appears as the second of two logical operators, as in the example above. Two decimal points may appear in sequence, as in the example above, or when one belongs to an operator and the other to a constant.

When the precedence of operators is not given explicitly by parentheses, it is understood to be as follows (in order of decreasing precedence):

```
**
*,/
+,-
.GT.,.GE.,.LT.,.LE.,.EQ.,.NE.
.NOT.
.AND.
.OR.
.EQV., .XOR.
```

For example, the logical expression

```
.NOT.ZETA**2+Y*MASS.GT.K-2.OR.PARITY.AND.X.EQ.Y
```

is interpreted as

```
(.NOT.(((ZETA**2)+(Y*MASS)).GT.(K-2))).OR.(PARITY.AND.(X.EQ.Y))
```

CHAPTER 3
THE ARITHMETIC STATEMENT

3.1 GENERAL DESCRIPTION

One of the key features of FORTRAN IV is the ease with which arithmetic computations can be coded. Computations to be performed by FORTRAN IV are indicated by arithmetic statements, which have the general form:

```
A=B
```

where A is a variable, B is an expression, and = is a replacement operator. The arithmetic statement causes the FORTRAN IV object program to evaluate the expression B and assign the resultant value to the variable A. Note that the = sign signifies replacement, not equality. Thus, expressions of the form:

```
A=A+B and
A=A*B
```

are quite meaningful and indicate that the value of the variable A is to be replaced by the result of the expression to the right of the = sign.

Examples:
```
Y=1*Y
P=.TRUE.
X(N)=N*ZETA(ALPHA*M/PI)+(1.,-1.)
```

Table 3-1 indicates which type of expression may be equated to each type of variable in an arithmetic statement. D indicates that the assignment is performed directly (no conversion of any sort is done); R indicates that only the real part of the variable is set to the value of the expression (the imaginary part is set to zero); C means that the expression is converted to the type of the variable; and H means that only the high-order portion of evaluated expression is assigned to the variable.

The expression value is made to agree in type with the assignment variable before replacement occurs. For example, in the statement:

```
THETA=W*(ABETA+E)
```

if THETA is an integer and the expression is real, the expression value is truncated to an integer before assignment to THETA.

Table 3-1
Allowed Assignment Statements

Variable	Expression				
	Real	Integer	Complex	Double Precision	Logical, Octal, or Literal Constant
Real	D	C	R,D	H,D	D
Integer	C	D	R,C	H,C	D
Complex	D,R,I	C,R,I	D	H,D,R,I	D,R,I
Double Precision	D,H,L	C,H,L	R,D,H,L	D	D,H,L
Logical	D	D	R,D	H,D	D

D - Direct Replacement

C - Conversion between integer and floating point

R - Real only

I - Set imaginary part to 0

H - High order only

L - Set low order part to 0

CHAPTER 4
CONTROL STATEMENTS

FORTRAN compiled programs normally execute statements sequentially in the order in which they were presented to the compiler. However, the following control statements are available to alter the normal sequence of statement execution: GO TO, IF, DO, PAUSE, STOP, END, CALL, RETURN. CALL and RETURN are used to enter and return from subroutines.

4.1 GO TO STATEMENT

The GO TO statement has three forms: unconditional, computed, and assigned.

4.1.1 Unconditional GO TO Statements

Unconditional GO TO statements are of the form:

GO TO n

where n is the number of an executable statement. Control is transferred to the statement numbered n. An unconditional GO TO statement may appear anywhere in the source program, except as the terminal statement of a DO loop.

4.1.2 Computed GO TO Statements

Computed GO TO statements have the form:

GO TO $(n_1,n_2,\ldots,n_k),i$

where $n_1,n_2,\ldots,n_k$ are statement numbers, and i is an integer expression.

This statement transfers control to the statement numbered $n_1,n_2,\ldots,n_k$ if i has the value 1, 2, ...,k, respectively. If i exceeds the size of the list of statement numbers or is less than one, execution will proceed to the next executable statement. Any number of statement numbers may appear in the list. There is no restriction on other uses for the integer variable i in the program.

In the example

```
GO TO (20,10,5),K
```

the variable K acts as a switch, causing a transfer to statement 20 if K=1, to statement 10 if K=2, or to statement 5 if K=3.

A computed GO TO statement may appear anywhere in the source program, except as the terminal statement of a DO loop.

4.1.3 Assigned GO TO Statement

Assigned GO TO statements have two equivalent forms:

```
GO TO k
```

and

```
GO TO k, (n1,n2,n3,...)
```

where k is a variable or array element and n_1, n_2, ...n_k are statement numbers. Any number of statement numbers may appear in the list. Both forms of the assigned GO TO have the effect of transferring control to the statement whose number is currently associated with the variable k. The second form of the assigned GO TO statement passes control to the next executable statement if k is not associated with one of the statement numbers in the list. This association is established through the use of the ASSIGN statement, the general form of which is:

```
ASSIGN i TO k
```

where i is a statement number and k is a variable or array element. If more than one ASSIGN statement refers to the same integer variable name, the value assigned by the last executed statement is the current value.

Examples:

```
ASSIGN 21 TO INT          ASSIGN 1000 TO INT
        :                         :
GO TO INT                 GO TO INT, (2,21,1000,310)
```

An assigned GO TO statement may appear anywhere in the source program, except as the terminal statement of a DO loop.

4.2 IF STATEMENT

IF statements have two forms in FORTRAN IV: numerical and logical.

4.2.1 Numerical IF Statements

Numerical IF statements are of the form:

IF (expression) n_1,n_2,n_3

where n_1,n_2,n_3 are statement numbers. This statement transfers control to the statement numbered n_1,n_2,n_3 if the value of the numeric expression is less than, equal to, or greater than zero, respectively. All three statement numbers must be present. The expression may not be complex.

Examples:
```
IF (ETA) 4,7,12
IF (KAPPA-L (10)) 20,14,14
```

4.2.2 Logical IF Statements

Logical IF statements have the form:

IF (expression)S

where S is a complete statement. The expression must be logical. S may be any executable statement other than a DO statement or another logical IF statement (see Chapter 2, Section 2.3.2). If the value of the expression is .FALSE. (positive or zero), control passes to the next sequential statement. If value of the expression is .TRUE. (negative), statement S is executed. After execution of S, control passes to the next sequential statement unless S is a numerical IF statement or a GO TO statement; in these cases, control is transferred as indicated. If the expression is .TRUE. (negative) and S is a CALL statement, control is transferred to the next sequential statement upon return from the subroutine.

Numbers are present in the logical expression:

```
IF (B)Y=X*SIN(Z)
W=Y**2
```

If the value of B is .TRUE., the statements Y=X*SIN(Z) and W=Y**2 are executed in that order. If the value of B is .FALSE., the statement Y=X*SIN(Z) is not executed.

Examples:
```
IF (T.OR.S)X=Y+1
IF (Z.GT.X(K)) CALL SWITCH (S,Y)
IF (K.EQ.INDEX) GO TO 15
```

NOTE

Care should be taken in testing floating point numbers for equality in IF statements as rounding may cause unexpected results.

4.3 DO STATEMENT

The DO statement simplifies the coding of iterative procedures. DO statements are of the form:

DO n i=m_1,m_2,m_3

where n is a statement number, i is a nonsubscripted integer variable, and m_1,m_2,m_3 are any integer expressions. If m_3 is not specified, it is understood to be 1.

The DO statement causes the statements which follow, up to and including the statement numbered n, to be executed repeatedly. This group of statements is called the range of the DO statement. The integer variable i of the DO statement is called the index. The values of m_1,m_2, and m_3 are called, respectively, the initial, limit, and increment values of the index.

A zero increment (m_3) is not allowed. The increment m_3 may be negative if $m_1 \geq m_2$. If $m_1 \leq m_2$, the increment m_3 must be positive. The index variable can assume legal values only if $(m_2-m_i)*m_3 \geq 0$. (m_i is the current value of the index variable m_1.)

Examples:	Form	Restriction
	DO 10 I=1,5,2	
	DO 10 I=5,1,-1	
	DO 10 I=J,K,5	$J \leq K$
	DO 10 I=J,K,-5	$J \geq K$
	DO 10 L=I,J,-K	$I \leq J, K<0$ or $I \geq J, K>0$
	DO 10 L=I,J,K	$I \leq J, K>0$ or $I \geq J, K>0$

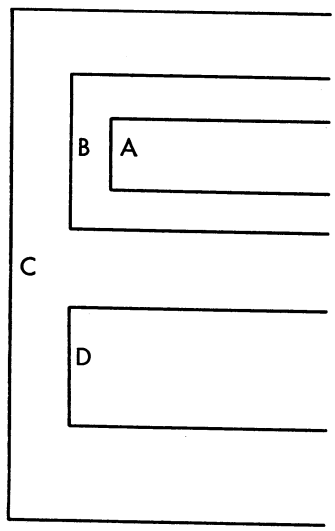
Initially, the statements of the range are executed with the initial value assigned to the index. This initial execution is always performed, regardless of the values of the limit and increment. After each execution of the range, the increment value is added to the value of the index and the result is compared with the limit value. If the value of the index is not greater than the limit value, the range is executed again using the new value of the index. When the increment value is negative, another execution will be performed if the new value of the index is not less than the limit value.

After the last execution of the range, control passes to the statement immediately following the range. This exit from the range is called the normal exit. Exit may also be accomplished by a transfer from within the range.

The range of a DO statement may include other DO statements, provided that the range of each contained DO statement is entirely within the range of the containing DO statement. When one DO loop is completely contained in another, it is said to be nested. DO loops can be nested to any depth. A transfer into the range of a DO statement from outside the range is not allowed.

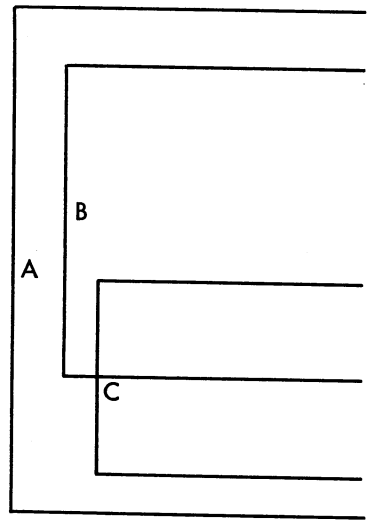
More than one DO loop within a nest of DO loops can end on the same statement. This terminal statement is considered to belong to the innermost DO loop that ends on the terminal statement. The statement label of such a terminal statement cannot be used in any GO TO or arithmetic IF statements except those that occur within the DO loop to which the terminal statement belongs.

Valid DO Loop Nest



Control must not pass from within loop A or loop B into loop D, or from loop D into loop A or loop B.

Invalid DO Loop Nest



Loop C is not fully within the range of loop B even though it is within the range of loop A.

Figure 4-1 Nested DO Loops

Within the range of a DO statement, the index is available for use as an ordinary variable. After a transfer from within the range, the index retains its current value and is available for use as a variable. The value of the index variable becomes undefined when the DO loop it controls is satisfied. The values of the initial, limit, and increment variables for the index and the index of the DO loop, may not be altered within the range of the DO statement.

The range of a DO statement must not end with a GO TO type statement or a numerical IF statement. If an assigned GO TO statement is in the range of a DO loop, all the statements to which it may transfer must be either in the range of the DO loop or all must be outside the range. A logical IF statement is allowed as the last statement of the range. In this case, control is transferred as follows. The range is considered ended when, and if, control would normally pass to the statement following the entire logical IF statement.

As an example, consider the sequences:

```
  DO 5 K = 1,4
5 IF(X(K).GT.Y(K))Y(K) = X(K)
6 ...
```

Statement 5 is executed four times whether the statement Y(K) = X(K) is executed or not. Statement 6 is not executed until statement 5 has been executed four times.

Examples:
```
DO 22 L = 1,30
DO 45 K = 2,LIMIT,-3
DO 7 X = T,MAX,L
```

4.4 CONTINUE STATEMENT

The CONTINUE statement has the form:

```
CONTINUE
```

This statement is a dummy statement, used primarily as a target for transfers, particularly as the last statement in the range of a DO statement. For example, in the sequence:

```
      DO 7 K = START,END
        :
      IF (X(K))22,13,7
        :
7     CONTINUE
```

a positive value of X(K) begins another execution of the range. The CONTINUE provides a target address for the IF statement and ends the range of the DO statement.

4.5 PAUSE STATEMENT

The PAUSE statement enables the program to incorporate operator activity into the sequence of automatic events. The PAUSE statement assumes one of three forms:

```
PAUSE
PAUSE n
PAUSE 'xxxxx'
```

where n is an unsigned string of six or less octal digits, and 'xxxxx' is a literal message.

Execution of the PAUSE statement causes the message or the octal digits, if any, to be typed on the user's teletypewriter. Program execution may be resumed (at the next executable FORTRAN statement) from the console by typing "G," followed by a carriage return. Program execution may be terminated by typing "X," followed by carriage return.

Example:
```
PAUSE 167
PAUSE 'NOW IS THE TIME'
```

4.6 STOP STATEMENT

The STOP statement has the forms:

```
STOP      or
STOP n
```

where n is an unsigned string of one to five octal digits.

The STOP statement terminates the program and returns control to the monitor system. (Termination of a program may also be accomplished by a CALL to the EXIT or DUMP subroutines.) Use of the STOP statement implies a call to the EXIT subroutine.

4.7 END STATEMENT

The END statement has the form:

```
END
```

The END statement informs the compiler to terminate compilation and must be the physically last statement of the program. The END statement implies a STOP statement in a main program or a RETURN statement in a subroutine or a function. The END statement is implied by an end-of-file.

CHAPTER 5
DATA TRANSMISSION STATEMENTS

Data transmission statements are used to control the transfer of data between computer memory and either peripheral devices or other locations in computer memory. These statements are also used to specify the format of the output data. Data transmission statements are divided into the following four categories.

a. Nonexecutable statements that enable conversions between internal form data within core memory and external form data (FORMAT), or specify lists of arrays and variables for input/output transfer (NAMELIST).

b. Statements that specify transmission of data between computer memory and I/O devices: READ, WRITE, PRINT, PUNCH, TYPE, ACCEPT.

c. Statements that control magnetic tape unit mechanisms: REWIND, BACKSPACE, END FILE, UNLOAD, SKIP RECORD.

d. Statements that specify transmission of data between series of locations in memory: ENCODE, DECODE.

5.1 NONEXECUTABLE STATEMENTS

The FORMAT statement enables the user to specify the form and arrangement of data on the selected external medium. The NAMELIST statement provides for conversion and input/output transmission of data without reference to a FORMAT statement.

5.1.1 FORMAT Statement

FORMAT statements may be used with any appropriate input/output medium or ENCODE/DECODE statement. FORMAT statements are of the form:

$$n \text{ FORMAT } (S_1,S_2,\ldots S_n/S_1^1,S_2^1,\ldots,S_n^1/\ldots)$$

where n is a statement number, and each S is a data field specification.

FORMAT statements may be placed anywhere in the source program. Unless the FORMAT statement contains only alphanumeric data for direct input/output transmission, it will be used in conjunction with the list of a data transmission statement.

Slashes are used to specify unit records, which must be one of the following:

a. A tape or disk record with a maximum length corresponding to a line buffer (135 ASCII characters).

b. A punched card with a maximum of 80 characters.

c. A printed line with a maximum of 72 characters for a Teletype® and either 120 or 132 characters for the line printer.

During transmission of data, the object program scans the designated FORMAT statement. If a specification for a numeric field is present (see Section 5.2.1 of this chapter) and the data transmission statement contains items remaining to be transmitted, transmission takes place according to the specifications. This process ceases and execution of the data transmission statement is terminated as soon as all specified items have been transmitted. Thus, the FORMAT statement may contain specifications for more items than are specified by the data transmission statement. Conversely, the FORMAT statement may contain specifications for fewer items than are specified by the data transmission statement.

The following types of field specifications may appear in a FORMAT statement: numeric, numeric with scale factors, logical, alphanumeric. The FORMAT statement also provides for handling multiple record formats, formats stored as data, carriage control, skipping characters, blank insertion, and repetition. If an input list requires more characters than the input device supplies for a given unit record, blanks are supplied.

5.1.1.1 Numeric Fields - Numeric field specification codes designate the type of conversion to be performed. These codes and the corresponding internal and external forms of the numbers are listed in Table 5-2.

The conversions are specified by the forms:

1.	Dw.d	
2.	Ew.d	
3.	Fw.d	
4.	Iw	
5.	Ow	
6.	Gw.d	(for real or double precision)
	Gw	(for integer or logical)
	Gw.d,Gw.d	(for complex)

respectively. The letter D, E, F, I, O, or G designates the conversion type; w is an integer specifying the field width, which may be greater than required to provide for blank columns between numbers; d is an integer specifying the number of decimal places to the right of the decimal point or, for G conversion, the number of significant digits. (For D, E, F, and G input, the position of the decimal point in the external field takes precedence over the value of d in the format.)

® Teletype is a registered trademark of Teletype Corporation.

For example,

FORMAT (I5,F10.2,D18.10)

could be used to output the line,

bbb32bbbb-17.60bbb.5962547681D+03

on the output listing.

The G format is the general format code that is used to transmit real, double precision, integer, logical, or complex data. The rules for input depend on the type specification of the corresponding variable in the data list. The form of the output conversion also depends on the individual variable except in the case of real and double-precision data. In these cases the form of the output conversion is a function of the magnitude of the data being converted. The following table shows the magnitude of the external data, M, and the resulting method of conversion.

Table 5-1
Magnitude of Internal Data

Magnitude of Data	Resulting Conversion
$0.1 \leq M < 1$	F(w-4).d, 4x
$1 \leq M < 10$	F(w-4).(d-1), 4x
.	.
.	.
$10^{d-2} \leq M < 10^{d-1}$	F(w-4). 1, 4x
$10^{d-1} \leq M < 10^{d}$	F(w-4). 0, 4x
All others	Ew.d

The field width w should always be large enough to include spaces for the decimal point, sign, and exponent. In all numeric field conversions if w is not large enough to accommodate the converted number, the excess digits on the left will be lost; if the number is less than w spaces in length, the number is right-adjusted in the field.

Table 5-2
Numeric Field Codes

Conversion Code	Internal Form	External Form
D	Binary floating point double-precision	Decimal floating point with D exponent
E	Binary floating point	Decimal floating point with E exponent
F	Binary floating point	Decimal fixed point
I	Binary integer	Decimal integer
O	Binary integer	Octal integer
G	One of the following: single precision binary floating point, binary integer, binary logical, or binary complex	Single precision decimal floating point integer, logical (T or F), or complex (two decimal floating point numbers), depending upon the internal form

5.1.1.2 Numeric Fields with Scale Factors - Scale factors may be specified for D, E, F, and G conversions. A scale factor is written nP where P is the identifying character and n is a signed or unsigned integer that specifies the scale factor.

For F type conversions (or G type, if the external field is decimal fixed point), the scale factor specifies a power of ten so that

$$\text{external number} = (\text{internal number}) * 10^{(\text{scale factor})}$$

For D, E, and G (external field not decimal fixed point) conversions, the scale factor multiplies the number by a power of ten, but the exponent is changed accordingly leaving the number unchanged except in form. For example, if the statement:

```
FORMAT (F8.3,E16.5)
```

corresponds to the line

```
bb26.451bbbb-0.41321E-01
```

then the statement

```
FORMAT (-1PF8.3,2PE16.5)
```

might correspond to the line

bbb2.645bbb-41.32157E-03

In input operations, F type (and G type, if the external field is decimal fixed point) conversions are the only types affected by scale factors.

When no scale factor is specified, it is understood to be zero. However, once a scale factor is specified, it holds for all subsequent D, E, F, and G type conversions within the same format unless another scale factor is encountered. The scale factor is reset to zero by specifying a scale factor of zero. Scale factors have no effect on I and O type conversions.

5.1.1.3 Logical Fields - Logical data can be transmitted in a manner similar to numeric data by use of the specification:

Lw

where L is the control character and w is an integer specifying the field width. The data is transmitted as the value of a logical variable in the input/output list.

If on input, the first nonblank character in the data field is T or F, the value of the logical variable will be stored as true or false, respectively. If the entire data field is blank or empty, a value of false will be stored.

On output, w minus 1 blanks followed by T or F will be output if the value of the logical variable is true or false, respectively.

5.1.1.4 Variable Field Width - The D, E, F, G, I, and O conversion types may appear in a FORMAT statement without the specification of the field width (w) or the number of places after the decimal point (d). In the case of input, omitting the w implies that the numeric field is delimited by any character which would otherwise be illegal in the field, in addition to the characters -, +, ., E, D, and blank provided they follow the numeric field. For example, input according to the format

10 FORMAT(2I,F,E,O)

might appear on the input medium as

-10,3/15.621-.0016E-10,777.

In this case, commas delimit the numeric fields, blanks may also be used as field delimiters. On output, omitting the w has the following effect:

Format	Becomes
D	D25.16
E	E15.7
F	F15.7
G	G15.7 or G25.16
I	I15
O	O15

5.1.1.5 Alphanumeric Fields - Alphanumeric data can be transmitted in a manner similar to numeric data by use of the form Aw, where A is the control character and w is the number of characters in the field. The alphanumeric characters are transmitted as the value of a variable in an input/output list. The variable may be of any type. For the sequence:

```
READ 5,V
5 FORMAT (A4)
```

causes four characters to be read and placed in memory as the value of the variable V.

Although w may have any value, the number of characters transmitted is limited by the maximum number of characters which can be stored in the space allotted for the variable. This maximum depends upon the variable type. For a double precision variable the maximum is ten characters; for all other variables, the maximum is five characters. If w exceeds the maximum, the leftmost characters are lost on input and replaced with blanks on output. If, on input, w is less than the maximum, blanks are filled in to the right of the given characters until the maximum is reached. If, on output, w is less than the maximum, the leftmost w characters are transmitted to the external medium. Since for complex variables each word requires a separate field specification, the maximum value for w is 5. For example,

```
COMPLEX C
ACCEPT 1, C
1 FORMAT (2A5)
```

could be used to transmit ten alphanumeric characters into complex variable C.

5.1.1.6 Alphanumeric Data Within Format Statements - Alphanumeric data may be transmitted directly into or from the format statement by two different methods: H-conversion, or the use of single quotes.

In H-conversion, the alphanumeric string is specified by the form nH. H is the control character and n is the number of characters in the string counting blanks. For example, the format in the statement below can be used to print PROGRAM COMPLETE on the output listing.

```
FORMAT (17H PROGRAM COMPLETE)
```

The statement

```
FORMAT (16HPROGRAM COMPLETE)
```

causes ROGRAM COMPLETE to be printed.

Referring to this format in a READ statement would cause the 17 characters to be replaced with a new string of characters.

The same effect is achieved by merely enclosing the alphanumeric data in quotes. The result is the same as in H-conversion; on input, the characters between the quotes are replaced by input characters, and, on output, the characters between the quotes (including blanks) are written as part of the output data. A quote character within the data is represented by two successive quote marks. For example, referring to:

```
FORMAT (' DON''T')
```

with an output statement would cause DON'T to be printed. Referring to

```
FORMAT ('DON''T')
```

causes ON'T to be printed. The first character referenced by the FORMAT statement for output is interpreted as a carriage control character (see 5.1.1.13). TAB characters in FORMAT statements are converted to single blanks at runtime by the FORTRAN operating system.

5.1.1.7 Mixed Fields - An alphanumeric format field may be placed among other fields of the format. For example, the statement:

```
FORMAT (I5,7H FORCE=F10.5)
```

can be used to output the line:

```
bbb22bFORCE=bb17.68901
```

The separating comma may be omitted after an alphanumeric format field, as shown above.

5.1.1.8 Complex Fields - Complex quantities are transmitted as two independent real quantities. The format specification consists of two successive real specifications or one repeated real specification. For instance, the statement:

```
FORMAT (2E15.4,2(F8.3,F8.5))
```

could be used in the transmission of three complex quantities.

5.1.1.9 Repetition of Field Specifications - Repetition of a field specification may be specified by preceding the control character D, E, F, I, O, G, L, or A by an unsigned integer giving the number of repetitions desired. For example:

```
FORMAT (2E12.4,3I5)
```

is equivalent to:

```
FORMAT (E12.4,E12.4,I5,I5,I5)
```

5.1.1.10 Repetition of Groups - A group of field specifications may be repeated by enclosing the group in parentheses and preceding the whole with the repetition number. For example:

```
FORMAT (2I8,2(E15.5,2F8.3))
```

is equivalent to:

```
FORMAT (2I8,E15.5,2F8.3,E15.5,2F8.3)
```

5.1.1.11 Multiple Record Formats - To handle a group of input/output records where different records have different field specifications, a slash is used to indicate a new record. For example, the statement:

```
FORMAT (3O8/I5,2F8.4)
```

is equivalent to

```
FORMAT (3O8)
```

for the first record and

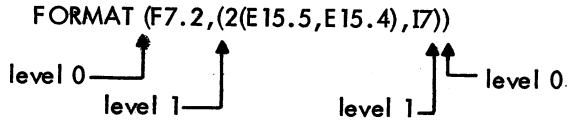
```
FORMAT (I5,2F8.4)
```

for the second record.

The separating comma may be omitted when a slash is used. When n slashes appear at the end or beginning of a format, n blank records may be written on output or records skipped on input. When n slashes appear in the middle of a format, n-1 blank records are written or n-1 records skipped.

Both the slash and the closing parenthesis at the end of the format indicate the termination of a record. If the list of an input/output statement dictates that transmission of data is to continue after the closing parenthesis of the format is reached, the format is repeated starting with that group repeat specification terminated by the last right parenthesis of level one or level zero if no level one group exists.

Thus, the statement



causes the format

F7.2,2(E15.5,E15.4),I7

to be used on the first record, and the format

2(E15.5,E15.4),I7

to be used on succeeding records.

As a further example, consider the statement

FORMAT (F7.2/(2(E15.5,E15.4),I7))

The first record has the format

F7.2

and successive records have the format

2(E15.5,E15.4),I7

5.1.1.12 Formats Stored as Data - The ASCII character string comprising a format specification may be stored as the values of an array. Input/output statements may refer to the format by giving the array name, rather than the statement number of a FORMAT statement. The stored format has the same form as a FORMAT statement excluding the word "FORMAT." The enclosing parentheses are included.

As an example, consider the sequence:

```
      DIMENSION SKELETON (2)
      READ 1, (SKELETON(I), I = 1,2)
    1 FORMAT (2A4)
      READ SKELETON,K,X
```

The first READ statement enters the ASCII string into the array SKELETON. In the second READ statement, SKELETON is referred to as the format governing conversion of K and X.

5.1.1.13 Carriage Control - The first character of each ASCII record controls the spacing of the line printer or Teletype. This character is usually set by beginning a FORMAT statement for an ASCII record with 1Ha, where a is the desired control character. The line spacing actions, listed below, occur before printing:

FORTRAN Character	Printer Character	Octal Value	Effect	Printer Channel
space	LF	012	Skip to next line with form feed after 60 lines	8
0 zero	LF,LF	012	Skip a line	8
1 one	FF	014	Form feed - go to top of next page	1
+ plus			Suppress skipping - overprint the line	
* asterisk	DC3	023	Skip to next line with no form feed	5
- minus	LF,LF,LF	012	Skip two lines	8
2 two	DLE	020	Space 1/2 of a page	2
3 three	VT	013	Space 1/3 of a page	7
/ slash	DC4	024	Space 1/6 of a page	6
. period	DC2	022	Triple space with a form feed after every 20 lines printed	4
, comma	DC1	021	Double space with a form feed after every 30 lines printed	3

NOTE: Printer control characters DLE, DC1, DC2, DC3, and DC4 affect only the line printer.

A $ (dollar sign) as a format field specification code suppresses the carriage return at the end of the Teletype or line printer line.

5.1.1.14 Spacing - Input and output can be made to begin at any position within a FORTRAN record by use of the format code

Tw

where T is the control character and w is an unsigned integer constant specifying the character position in a FORTRAN record where the transfer of data is to begin. When the output is printed, w corresponds to the (w-1)th print position. This is because the first character of the output buffer is a carriage control character and is not printed. It is recommended that the first field specification of the output format be 1x, except where a carriage control character is used.

For example,

```
2 FORMAT (T50, 'BLACK'T30, 'WHITE')
```

would cause the following line to be printed

```
Print Position 29        Print Position 49
        ↓                        ↓
      WHITE                    BLACK
```

For input, the statement

```
1 FORMAT(T35,'MONTH')
  READ (3,1)
```

cause the first 34 characters of the input data to be skipped, and the next 5 characters would replace the characters M, O, N, T, and H in storage. If an input record containing

```
ABCbbbXYZ
```

is read with the format specification

```
10 FORMAT (T7,A3,T1,A3)
```

then the characters XYZ and ABC are read, in that order.

5.1.1.15 Blank or Skip Fields - Blanks may be introduced into an output record or characters skipped on an input record by use of the specification nX. The control character is X; n is the number of blanks or characters skipped and must be greater than zero. For example, the statement

```
FORMAT (5H STEPI5, 10X2HY=F7.3)
```

may be used to output the line

```
bSTEPbbb28bbbbbbbbbbY=b-3.872
```

5.1.2 NAMELIST Statement

The NAMELIST statement, when used in conjunction with special forms of the READ and WRITE statements, provides a method for transmitting and converting data without using a FORMAT statement or an I/O list. The NAMELIST statement has the form

$$NAMELIST/X_1/A_1,A_2,\ldots,A_i/X_2/B_1,B_2,\ldots,B_i\ldots/X_m/C_1,C_2,\ldots C_n$$

where the X's are NAMELIST names, and the A's, B's, and C's are variable or array names.

Each list or variable mentioned in the NAMELIST statement is given the NAMELIST name immediately preceding the list. Thereafter, an I/O statement may refer to an entire list by mentioning its NAMELIST name. For example:

```
NAMELIST/FRED/A,B,C/MARTHA/D,E
```

states that A, B, and C belong to the NAMELIST name FRED, and D and E belong to MARTHA.

The use of NAMELIST statements must obey the following rules:

a. A NAMELIST name may not be longer than six characters; it must start with an alphabetic character; it must be enclosed in slashes; it must precede the list of entries to which it refers; and it must be unique within the program.

b. A NAMELIST name may be defined only once and must be defined by a NAMELIST statement. After a NAMELIST name has been defined, it may only appear in READ or WRITE statements. The NAMELIST name must be defined in advance of the READ or WRITE statement.

c. A variable used in a NAMELIST statement cannot be used as a dummy argument in a subroutine definition.

d. Any dimensioned variable contained in NAMELIST statement must have been defined in a DIMENSION statement preceding the NAMELIST statement.

5.1.2.1 Input Data For NAMELIST Statements - When a READ statement refers to a NAMELIST name, the first character of all input records is ignored. Records are searched until one is found with a $ or & as the second character immediately followed by the NAMELIST name specified. Data is then converted and placed in memory until the end of a data group is signaled by a $ or & either in the same record as the NAMELIST name, or in any succeeding record as long as the $ or & is the second character of the record. Data items must be separated by commas and be of the following form:

$$V=K_1,K_2,\ldots,K_n$$

where V may be a variable name or an array name, with or without subscripts. The K's are constants which may be integer, real, double precision, complex (written as (A, B) where A and B are real), or logical (written as T for true and F for false). A series of J identical constants may be represented by J*K where J is an unsigned integer and K is the repeated constant. Logical and complex constants must be equated to logical and complex variables, respectively. The other types of constants (real, double precision, and integers) may be equated to

any other type of variable (except logical or complex), and will be converted to the variable type. For example, assume A is a two-dimensional real array, B is a one-dimensional integer array, C is an integer variable, and that the input data is as follows:

```
$FRED A(7,2)=4, B=3,6*2.8, C=3.32$
↑
Column 2
```

A READ statement referring to the NAMELIST name FRED will result in the following: the integer 4 will be converted to floating point and placed in A(7,2). The integer 3 will be placed in B(1) and the floating point number 2.8 will be placed in B(2), B(3),..., B(7). The floating point number 3.32 will be converted to the integer 3 and placed in C.

5.1.2.2 Output Data For NAMELIST Statements - When a WRITE statement refers to a NAMELIST name, all variables and arrays and their values belonging to the NAMELIST name will be written out, each according to its type. The complete array is written out by columns. The output data will be written so that:

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

For example, if JOE is a 2x3 array, the statement

```
NAMELIST/NAM1/JOE,K1,ALPHA
WRITE (u,NAM1)
```

generate the following form of output.

```
Column 2
↓
$NAM1
JOE = -6.75,          .234E-04,          68.0,
      -17.8,             0.0,        -.197E+07,
  K1 = 73.1,          ALPHA=3,$
```

5.2 DATA TRANSMISSION STATEMENTS

The data transmission statements accomplish input/output transfer of data that may be listed in a NAMELIST statement or defined in a FORMAT statement. When a FORMAT statement is used to specify formats, the data transmission statement must contain a list of the quantities to be transmitted. The data appears on the external media in the form of records.

5.2.1 Input/Output Lists

The list of an input/output statement specifies the order of transmission of the variable values. During input, the new values of listed variables may be used in subscript or control expressions for variables appearing later in the list. For example:

```
READ 13,L,A(L),B(L+1)
```

reads a new value of L and uses this value in the subscripts of A and B.

The transmission of array variables may be controlled by indexing similar to that used in the DO statement. The list of controlled variables, followed by the index control, is enclosed in parentheses. For example,

```
READ 7, (X(K),K=1,4),A
```

is equivalent to:

```
READ 7, X(1),X(2),X(3),X(4),A
```

As in the DO statement, the initial, limit, and increment values may be given as integer expressions:

```
READ 5, N, (GAIN(K),K=1,M/2,N)
```

The indexing may be compounded as in the following:

```
READ 11, ((MASS(K,L),K=1,4),L=1,5)
```

The above statement reads in the elements of array MASS in the following order:

```
MASS(1,1), MASS(2,1),...,MASS(4,1),MASS(1,2),...,MASS(4,5)
```

If an entire array is to be transmitted, the indexing may be omitted and only the array identifier written. The array is transmitted in order of increasing subscripts with the first subscript varying most rapidly. Thus, the example above could have been written:

```
READ 11, MASS
```

Entire arrays may also be designated for transmission by referring to a NAMELIST name (see description of NAMELIST statement).

5.2.2 Input/Output Records

All information appearing on external media is grouped into records. The maximum amount of information in one record and the manner of separation between records depends upon the medium. For punched cards, each card constitutes one record; on a teletypewriter a record is one line, and so forth. The amount of information contained in each ASCII record is specified by the FORMAT reference and the I/O list. For magnetic tape binary records, the amount of information is specified by the I/O list.

Each execution of an input or output statement initiates the transmission of a new data record. Thus, the statement

```
READ 2, FIRST,SECOND,THIRD
```

is not necessarily equivalent to the statements

```
READ 2, FIRST
READ 2, SECOND
READ 2, THIRD
```

since, in the second case, at least three separate records are required, whereas, the single statement

```
READ 2, FIRST,SECOND,THIRD
```

may require one, two, three, or more records depending upon FORMAT statement 2.

If an input/output statement requests less than a full record of information, the unrequested part of the record is lost and cannot be recovered by another input/output statement without repositioning the record.

If an input/output list requires more than one ASCII record of information, successive records are read.

5.2.3 PRINT Statement

The PRINT statement assumes one of two forms

```
PRINT f, list
PRINT f
```

where f is a format reference.

The data is converted from internal to external form according to the designated format. If the data to be transmitted is contained in the specified FORMAT statement, the second form of the statement is used.

Examples:
```
PRINT 16,T,(B(K),K=1,M)
PRINT F106,SPEED,MISS
```

In the second example, the format is stored in array F106.

5.2.4 PUNCH Statement

The PUNCH statement assumes one of two forms

```
PUNCH f, list
PUNCH f
```

where f is a format reference.

Conversion from internal to external data forms is specified by the format reference. If the data to be transmitted is contained in the designated FORMAT statement, the second form of the statement is used.

Examples:
```
PUNCH 12,A,B(A),C(B(A))
PUNCH 7
```

5.2.5 TYPE Statement

The TYPE statement assumes one of two forms

```
TYPE f, list
TYPE f
```

where f is a format reference.

This statement causes the values of the variables in the list to be read from memory and listed on the user's teletypewriter. The data is converted from internal to external form according to the designated format. If the data to be transmitted is contained in the designated FORMAT statement, the second form of the statement is used.

Examples:
```
TYPE 14,K,(A(L),L=1,K)
TYPE FMT
```

5.2.6 WRITE Statement

The WRITE statement assumes one of the following forms

```
WRITE (u,f) list
WRITE(u,f)
WRITE(u,N)
WRITE(u) list
WRITE(u#R,f) list
```

where u is a unit designation, f is a format reference, N is a NAMELIST name, and R is a record number where I/O is to start.

The first form of the WRITE statement causes the values of the variables in the list to be read from memory and written on the unit designated in ASCII form. The data is converted to external form as specified by the designated FORMAT statement.

The second form of the WRITE statement causes information to be read directly from the specified format and written on the unit designated in ASCII form.

The third form of the WRITE statement causes the names and values of all variables and arrays belonging to the NAMELIST name, N, to be read from memory and written on the unit designated. The data is converted to external form according to the type of each variable and array.

The fourth form of the WRITE statement causes the values of the variables in the list to be read from memory and written on the unit designated in binary form.

The fifth form of the WRITE statement causes the variables in the list to be written in the specified record of the file on the disk unit designated. Either a pound sign (#) or a single quote (') can be used to separate the unit and the record. This allows a programmer to access fixed-length records directly, and eliminates the sequential writing of data to access one or more records within the file. The file must first be defined properly by a CALL to DEFINE FILE (see Section 12.4). Output begins when the random WRITE specifying the record to which the writing is desired is given in the correct format.

5.2.7 READ Statement

The READ statement assumes one of the following forms:

```
READ f, list
READ f
READ(u,f) list
READ(u,f)
READ(u,N)
READ(u)list
READ(u#R,f) list
READ(u,f,END=C, ERR=d) list
READ(u,f,END=C) list
READ(u,f, ERR=d) list
```

where f is a format reference, u is a unit designation, N is a NAMELIST name, R is a record number where I/O is to start, C is a statement number to which control is transferred upon encountering an end-of-file, and d is the statement number to which control is transferred upon encountering an error condition on the input data.

The first form of the READ statement causes information to be read from cards and put in memory as values of the variables in the list. The data is converted from external to internal form as specified by the referenced FORMAT statement.

Example: READ 28,Z1,Z2,Z3

The second form of the READ statement is used if the data read from cards is to be transmitted directly into the specified format.

Example: READ 10

The third form of the READ statement causes ASCII information to be read from the unit designated and stored in memory as values of the variables in the list. The data is converted to internal form as specified by the referenced FORMAT statement.

Example: READ(1,15)ETA,PI

The fourth form of the READ statement causes ASCII information to be read from the unit designated and transmitted directly into the specified format.

Example: READ(N,105)

The fifth form of the READ statement causes data of the form described in the discussion of input data for NAMELIST statements to be read from the unit designated and stored in memory as values of the variables or arrays specified.

Example: READ(2,FRED)

The sixth form of the READ statement causes binary information to be read from the unit designated and stored in memory as values of the variables in the list.

Example: READ(M)GAIN,Z,AI

The seventh form of the READ statement causes information to be read from the specified record in a disk file into the variables of the list. This allows random access of fixed-length records in a disk file. The file from which records are to be read is defined by the DEFINE FILE call (see Section 12.4).

Example:
```
DOUBLE PRECISION FIL
DIMENSION A(6)
DATA FIL/'FILE.ONE'/
CALL DEFINE FILE (4,30,NV,FIL,"11,"23)
READ (4#54,5)A
```

This example reads the 54th record from FILE.ONE on the disk area belonging to programmer [11,23] into the list variables A(1) through A(6).

The eighth form of the READ statement causes control to be transferred if an end-of-file or error condition is encountered on the input data. The arguments END=c and ERR=d are optional and if both are included, either may appear first. If an end-of-file is encountered, control transfers to the statement specified by END=c. If an END parameter is not specified, I/O on that device terminates and the program halts with an error message to the user's TTY. If an error on input is encountered, control transfers to the statement specified by ERR=d. If an ERR=d parameter is not specified, the program halts with an error message to the user's TTY.

Example:
```
      READ (7,7,END=888, ERR=999) A
      .
      .
      .
888   (control transfers here if an end-of-file is encountered)
      .
      .
      .
999   (control transfers here if an error on input is encountered)
```

5.2.8 REREAD Statement

The reread feature allows a FORTRAN program to reread information from the last used input file. The format used during the reread need not correspond to the original read format, and the information may be read as many times as desired.

a. To reread from an input device, the following coding would be used:

```
READ (16,100)A
      .
      .
      .
REREAD 105,A
```

The REREAD 105,A statement causes the last input device used to be reread according to format statement 105. The original read format and a subsequent reread format need not be the same.

b. The reread feature cannot be used until an input from a file has been accomplished. If the feature is used prematurely, an error message will be generated.

c. Information may be reread as many times as desired using either the same or a new format statement each time.

d. The reread feature must be used with some forethought and care since it rereads from the last input file used, i.e.:

The following example will reread from the file on Device No. 10, not Device No. 16:

```
READ (16,100)A
      .
      .
      .
READ (10,200)B
      .
      .
      .
REREAD 110,A
```

5.2.9 ACCEPT Statement

The ACCEPT statement assumes one of two forms

```
ACCEPT f, list
ACCEPT f
```

where f is a format reference.

This statement causes information to be input from the user's teletypewriter and put in memory as values of the variables in the list. The data is converted to internal form as specified by the format. If the transmission of data is directly into the designated format, the second form of the statement is used.

Examples:
```
ACCEPT 12,ALPHA,BETA
ACCEPT 27
```

5.3 DEVICE CONTROL STATEMENTS

Device control statements and their corresponding effects are listed in Table 5-3.

Table 5-3
Device Control Statements

Statement	Effect
BACKSPACE u	Backspaces designated tape one ASCII record or one logical binary record.
END FILE u	Writes an end-of-file.
REWIND u	Rewinds tape on designated unit.
SKIP RECORD u	Causes skipping of one ASCII record or one logical binary record.
UNLOAD u	Rewinds and unloads the designated tape.

5.4 ENCODE AND DECODE STATEMENTS

ENCODE and DECODE statements transfer data, according to format specifications, from one section of user's core to another. No peripheral equipment is involved. DECODE is used to change data in ASCII format to data in another format. ENCODE changes data of another format into data in ASCII format.

The two statements are of the form

```
ENCODE(c,f,v),L(1),...,L(N)
DECODE(c,f,v),L(1),...,L(N)
```

where

c = the number of ASCII characters
f = the format statement number
v = the starting address of the ASCII record referenced
L(1),...,L(N) = the list of variables.

A slash cannot appear in the FORMAT statement referenced by an ENCODE or DECODE statement.

Example: Assume the contents of the variables to be as follows:

A(1) contains the floating-point binary number 300.45

A(2) contains the floating-point binary number 3.0

J contains the binary integer value 1.

B is a four-word array of indeterminate contents

C contains the ASCII string 12345

```
      DO 2 J = 1,2
      ENCODE (16,10,B) J, A(J)
10    FORMAT (1X,2HA(,I1,4H) = ,F8.2)
      TYPE 11,B
11    FORMAT (4A5)
 2    CONTINUE
      DECODE (4, 12, C) B
12    FORMAT (3F1.0,1X,F1.0)
      TYPE 13,B
13    FORMAT (4F5.2)
      END
```

Array B can contain 20 ASCII characters. The result of the ENCODE statement after the first iteration of the DO loop is:

B(1)	A(1)
B(2)	=
B(3)	300.4
B(4)	5

Typed as

A(1) = 300.45

The result after the second iteration is:

B(1)	A(2)
B(2)	=
B(3)	3.0
B(4)	

Typed as

A(2) = 3.0

The result of the DECODE statement is to extract the digits 1, 2, and 3 from C and convert them to floating-point binary values and store them in B(1), B(2), and B(3). Then skip the next character (4) and extract the digit 5 from C, convert it to a floating-point binary value, and store it in B(4).

CHAPTER 6
SPECIFICATION STATEMENTS

Specification statements allocate storage and furnish information about variables and constants to the compiler. Specification statements may be divided into three categories, as follows:

a. Storage specification statements: DIMENSION, COMMON, and EQUIVALENCE.

b. Data specification statements: DATA and BLOCK DATA.

c. Type declaration statements: INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL, SUBSCRIPT INTEGER, and IMPLICIT.

By extending the USA Standard in regard to specification statements, PDP-10 FORTRAN IV allows the following statements to be used anywhere in the program, provided that the variables they specify appear in executable statements only after the particular specification statement. The specification statement must not appear in the range of a DO loop.

DIMENSION statement
EXTERNAL statement (described in Chapter 7)
COMMON statement
EQUIVALENCE statement
Type declaration statements
DATA statement

A sample program that incorporates these statements follows.

```
      DOUBLE PRECISION D
      DIMENSION Y(10), D(5)
      Y(1) = -1.0
      INTEGER XX(5)
      Y(2) = ABS(Y(1))
      DATA XX/1,2,3,4,5
      DO 10 I = 3,7
10    Y(I) = XX(I-2)
      COMMON Z
      Z=Y(1)*Y(2)/(Y(3) + Y(5))
      END
```

Only IMPLICIT statements and arithmetic function definition statements (described in Chapter 7) must appear in the program before any executable statement.

In addition, arrays must be dimensional before being referenced in a NAMELIST, EQUIVALENCE, or DATA statement. DOUBLE PRECISION and COMPLEX arrays must be declared before they are dimensioned.

6.1 STORAGE SPECIFICATION STATEMENTS

6.1.1 DIMENSION Statement

The DIMENSION statement is used to declare identifiers to be array identifiers and to specify the number and bounds of the array subscripts. The information supplied in a DIMENSION statement is required for the allocation of memory for arrays. Any number of arrays may be declared in a single DIMENSION statement. The DIMENSION statement has the form

DIMENSION $S_1, S_2, \ldots, S_n$

where S is an array specification.

Each array variable appearing in the program must represent an element of an array declared in a DIMENSION statement, unless the dimension information is given in a COMMON or TYPE statement. Dimension information may appear only once for a given variable.

Each array specification gives the array identifier and the minimum and maximum values which each of its subscripts may assume in the following form:

identifier(min/max,min/max,...,min/max)

The minima and maxima must be integers. The minimum must not exceed the maximum. For example, the statement

DIMENSION EDGE(-1/1,4/8)

specifies EDGE to be a two-dimensional array whose first subscript may vary from -1 to 1 inclusive, and the second from 4 to 8 inclusive.

Minimum values of 1 may be omitted. For example,

NET(5, 10)

is interpreted as:

NET(1/5,1/10)

Examples:
```
DIMENSION FORCE(-1/1,0/3,2,2,-7/3)
DIMENSION PLACE(3,3,3),JI(2,2/4),K(256)
```

Arrays may also be declared in the COMMON or type declaration statements in the same way:

```
COMMON X(10,4),Y,Z
INTEGER A(7,32),B
DOUBLE PRECISION K(-2/6,10)
```

6.1.1.1 Adjustable Dimensions - Within either a FUNCTION or SUBROUTINE subprogram, DIMENSION and TYPE statements may use integer variables in an array specification, provided that the array name and variable dimensions are dummy arguments of the subprogram. The actual array name and values for the dummy variables are given by the calling program when the subprogram is called. The variable dimensions may not be altered within the subprogram (i.e., typing the array DOUBLE PRECISION or COMPLEX after it has been dimensioned) and must be less than or equal to the explicit dimensions declared in the calling program.

Example:
```
      SUBROUTINE SBR(ARRAY,M1,M2,M3,M4)
      DIMENSION ARRAY (M1/M2,M3/M4)
                :
                :
      DO 27 L=M3,M4
      DO 27 K=M1,M2
                :
                :
27    ARRAY(K,L)=VALUE
                :
                :
      END
```

The calling program for SBR might be:

```
DIMENSION A1(10,20),A2(1000,4)
          :
          :
CALL SBR(A1,5,10,10,20)
          :
          :
CALL SBR(A2,100,250,2,4)
          :
          :
END
```

6.1.2 COMMON Statement

The COMMON statement causes specified variables or arrays to be stored in an area available to other programs. By means of COMMON statements, the data of a main program and/or the data of its subprograms may share a common storage area.

The common area may be divided into separate blocks which are identified by block names. A block is specified as follows:

/block identifier/identifier,identifier,...,identifier

The identifier enclosed in slashes is the block name. The identifiers which follow are the names of the variables or arrays assigned to the block and are placed in the block in the order in which they appear in the block specification. A common block may have the same name as a variable in the same program.

The COMMON statement has the general form

COMMON/BLOCK1/A,B,C/BLOCK2/D,E,F/...

where BLOCK1,BLOCK2,... are the block names, and A,B,C,... are the variables to be assigned to each block. For example, the statement

COMMON/R/X,Y,T/C/U,V,W,Z

indicates that the elements X,Y, and T are to be placed in block R in that order, and that U,V,W, and Z are to be placed in block C.

Block entries are linked sequentially throughout the program, beginning with the first COMMON statement. For example, the statements

```
COMMON/D/ALPHA/R/A,B/C/S
COMMON/C/X,Y/R/U,V,W
```

have the same effect as the statement

COMMON/D/ALPHA/R/A,B,U,V,W/C/S,X,Y

One block of common storage, referred to as blank common, may be left unlabeled. Blank common is indicated by two consecutive slashes. For example,

COMMON/R/X,Y//B,C,D

indicates that B, C, and D are placed in blank common. The slashes may be omitted when blank common is the first block of the statement.

```
COMMON B,C,D
```

Storage allocation for blocks of the same name begins at the same location for all programs executed together. For example, if a program contains

```
COMMON A,B/R/X,Y,Z
```

as its first COMMON statement, and a subprogram has

```
COMMON/R/U,V,W//D,E,F
```

as its first COMMON statement, the quantities represented by X and U are stored in the same location. A similar correspondence holds for A and D in blank common.

Common blocks may be any length provided that no program attempts to enlarge a given common block declared by a previously loaded program.

Array names appearing in COMMON statements may have dimension information appended if the arrays are not declared in DIMENSION or type declaration statements. For example,

```
COMMON ALPHA,T(15,10,5),GAMMA
```

specifies the dimensions of the array T while entering T in blank common. Variable dimension array identifiers may not appear in a COMMON statement, nor may other dummy identifiers. Each array name appearing in a COMMON statement must be dimensioned somewhere in the program containing the COMMON statement.

6.1.3 EQUIVALENCE Statement

The EQUIVALENCE statement causes more than one variable within a given program to share the same storage location. The EQUIVALENCE statement has the form

$$\text{EQUIVALENCE}(V_1, V_2, \ldots), (V_k, V_{k+1}, \ldots), \ldots$$

where the V's are variable names.

The inclusion of two or more references in a parenthetical list indicates that the quantities in the list are to share the same memory location. For example,

```
EQUIVALENCE(RED,BLUE)
```

specifies that the variables RED and BLUE are stored in the same location.

The relation of equivalence is transitive; e.g., the two statements,

```
EQUIVALENCE(A,B),(B,C)
EQUIVALENCE(A,B,C)
```

have the same effect.

The subscripts of array variables must be integer constants.

Example: EQUIVALENCE(X,A(3),Y(2,1,4)),(BETA(2,2),ALPHA)

6.1.4 EQUIVALENCE and COMMON

Identifiers may appear in both COMMON and EQUIVALENCE statements provided the following rules are observed.

a. No two quantities in common may be set equivalent to one another.

b. Quantities placed in a common block by means of EQUIVALENCE statements may cause the end of the common block to be extended. For example, the statements

```
COMMON/R/X,Y,Z
DIMENSION A(4)
EQUIVALENCE(A,Y)
```

causes the common block R to extend from X to A(4), arranged as follows:

```
X
Y A(1)      (same location)
Z A(2)      (same location)
  A(3)
  A(4)
```

c. EQUIVALENCE statements which cause extension of the start of a common block are not allowed. For example, the sequence

```
COMMON/R/X,Y,Z
DIMENSION A(4)
EQUIVALENCE(X,A(3))
```

is not permitted, since it would require A(1) and A(2) to extend the starting location of block R.

6.2 DATA SPECIFICATION STATEMENTS

The DATA statement is used to specify initial or constant values for variables. The specified values are compiled into the object program, and become the values assumed by the variables when program execution begins.

6.2.1 DATA Statement

The data to be compiled into the object program is specified in a DATA statement. The DATA statement has the form

DATA list/d_1,d_2,.../,list/d_k,d_{k+1},.../,...

where each list is in the same form as an input/output list, and the d's are data items for each list.

Indexing may be used in a list provided the initial, limit, and increment (if any) are given as constants. Expressions used as subscripts must have the form

$$c_1*i\pm c_2$$

where c_1 and c_2 are integer constants and i is the induction variable. If an entire array is to be defined, only the array identifier need be listed. Variables in COMMON may appear on the lists only if the DATA statement occurs in a BLOCK DATA subprogram. (See Chapter 7, Section 7.6)

The data items following each list correspond one-to-one with the variables of the list. Each item of the data specifies the value given to its corresponding variable with no implied type conversion. Thus, integer variables can only be defined numerically by integer constants, real variables by real constants, double precision variables by double precision constants, and so forth. Refer to Section 2.1 for definitions of the various constants. Data items may be numeric constants, alphanumeric strings, octal constants, or logical constants. For example,

```
DATA ALPHA, BETA/.5, 16.E-2/
```

specifies the value .5 for ALPHA and the value .16 for BETA.

Alphanumeric data is packed into words according to the data word size in the manner of A conversion; however, excess characters are not permitted. The specification is written as nH followed by n characters or is imbedded in single quotes. Double precision variables must have at least six characters assigned to them in DATA statements.

Octal data is specified by the letter O or the character ", followed by a signed or unsigned octal integer of one to twelve digits.

Logical constants are written as .TRUE.,.FALSE., T, or F.

Example:
```
DATA NOTE,K/4HFOOT, O-7712/
DATA QUOTE/'QUOTE'/
```

Any item of the data may be preceded by an integer followed by an asterisk. The integer indicates the number of times the item is to be repeated. For example,

```
DATA(A(K),K=1,20)/61E2, 19*32E1/
```

specifies 20 values for the array A; the value 6100 for A(1); the value 320 for A(2) through A(20). To cause an array or part of an array to be initialized to blanks, the blank areas must be specified explicitly in the DATA statement. For example,

```
DATA(A(I),I=1,10)/'12345',9*' '/
```

causes the first word of A to contain 12345 in ASCII and the next nine words of the array to be blank.

6.2.2 BLOCK DATA Statement

The BLOCK DATA statement has the form:

```
BLOCK DATA
```

This statement declares the program which follows to be a data specification subprogram. Data may be entered into labeled or blank common.

The first statement of the subprogram must be the BLOCK DATA statement. The subprogram may contain only the declarative statements associated with the data being defined.

Example:

```
BLOCK DATA
COMMON/R/S,Y/C/Z,W,V
DIMENSION Y(3)
COMPLEX Z
DATA Y/1E-1,2*3E2/,X,Z/11.877D0,(-1.41421,1.41421)/
END
```

Data may be entered into more than one block of common in one subprogram.

6.3 TYPE DECLARATION STATEMENTS

The type declaration statements INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL, IMPLICIT, and SUBSCRIPT INTEGER are used to specify the type of identifiers appearing in a program. An identifier may appear in only one type statement. Type statements may be used to give dimension specifications for arrays.

The explicit type declaration statements have the general form

```
type identifier,identifier,identifier...
```

where type is one of the following:

```
INTEGER,REAL,DOUBLE PRECISION,COMPLEX,LOGICAL,
SUBSCRIPT INTEGER
```

In addition, for the sake of compatibility the following types have been made equivalent:

```
SUBSCRIPT INTEGER is equivalent to INTEGER*2
INTEGER is equivalent to INTEGER*4
REAL is equivalent to REAL*4
DOUBLE PRECISION is equivalent to REAL*8
LOGICAL is equivalent to LOGICAL*1 and LOGICAL*4
COMPLEX is equivalent to COMPLEX*8
```

The listed identifiers are declared by the statement to be of the stated type. Fixed-point variables in a SUBSCRIPT INTEGER statement must fall between -2^{27} and 2^{27}.

6.3.1 IMPLICIT Statement

The IMPLICIT statement has the form

$$\text{IMPLICIT } type_1(a_1,a_2,\ldots),\ldots,type_2(a_3,a_4,\ldots)$$

where type represents INTEGER, REAL, LOGICAL, COMPLEX, DOUBLE PRECISION, or one of the equivalent types listed in Section 6.3, and $a_1a_2,\ldots$ represent single alphabetic characters, each separated by commas, or a range of characters (in alphabetic sequence) denoted by the first and last characters of the range separated by a minus sign (e.g., (A-D)).

This statement causes any program variable which is not mentioned in a type statement, and whose first character is one of those listed in the IMPLICIT statement, to be classified according to the type appearing before the list in which the character appears. As an example, the statement

```
IMPLICIT REAL(A-D,L,N-P)
```

causes all variables starting with the letters A through D, L, and N through P to be typed as real, unless they are explicitly declared otherwise.

The initial state of the compiler is set as if the statement

```
IMPLICIT REAL(A-H,O-Z), INTEGER(I-N)
```

were at the beginning of the program. This state is in effect unless an IMPLICIT statement changes the above interpretation; i.e., identifiers, whose types are not explicitly declared, are typed as follows.

a. Identifiers beginning with I, J, K, L, M, or N are assigned interger type.
b. Identifiers not assigned integer type are assigned real type.

If the program contains an IMPLICIT statement, this statement will override throughout the program the implicit state initially set by the compiler. No program may contain more than one IMPLICIT declaration for the same letter.

CHAPTER 7
SUBPROGRAM STATEMENTS

FORTRAN subprograms may be either internal or external. Internal subprograms are defined and may be used only within the program containing the definition. The arithmetic function definition statement is used to define internal functions.

External subprograms are defined separately from (i.e., external to) the programs that call them, and are complete programs which conform to all the rules of FORTRAN programs. They are compiled as closed subroutines; i.e., they appear only once in the object program regardless of the number of times they are used. External subprograms are defined by means of the statements FUNCTION and SUBROUTINE.

7.1 DUMMY IDENTIFIERS

Subprogram definition statements contain dummy identifiers, representing the arguments of the subprogram. They are used as ordinary identifiers within the subprogram definition and indicate the sort of arguments that may appear and how the arguments are used. The dummy identifiers are replaced by the actual arguments when the subprogram is executed.

7.2 LIBRARY SUBPROGRAMS

The standard FORTRAN IV library for the PDP-10 includes built-in functions, FUNCTION subprograms, and SUBROUTINE subprograms, listed and described in Chapter 8. Built-in functions are open subroutines; that is, they are incorporated into the object program each time they are referred to by the source program. FUNCTION and SUBROUTINE subprograms are closed subroutines; their names derive from the types of subprogram statements used to define them.

7.3 ARITHMETIC FUNCTION DEFINITION STATEMENT

The arithmetic function definition statement has the form:

```
identifier(identifier,identifier,...)=expression
```

This statement defines an internal subprogram. The entire definition is contained in the single statement. The first identifier is the name of the subprogram being defined.

Arithmetic function subprograms are single-valued functions with at least one argument. The type of the function is determined by the type of the function identifier.

The identifiers enclosed in parentheses represent the arguments of the function. These are dummy identifiers; they may appear only as scalar variables in the defining expression. Dummy identifiers have meaning and must be unique only within the defining statement. Dummy identifiers must agree in order, number, and type with the actual arguments given at execution time.

Identifiers, appearing in the defining expression, which do not represent arguments are treated as ordinary variables. The defining expression may include external functions or other previously defined arithmetic statement functions.

All arithmetic function definition statements must precede the first executable statement of the program.

Examples:
```
SSQR(K)=K*(K+1)*(2*K+1)/6
ACOSH(X)=(EXP(X/A)+EXP(-X/A))/2
```

In the last example above, X is a dummy identifier and A is an ordinary identifier. At execution time, the function is evaluated using the current value of the quantity represented by A.

7.4 FUNCTION SUBPROGRAMS

A FUNCTION subprogram is a single-valued function that may be called by using its name as a function name in an arithmetic expression, such as FUNC(N), where FUNC is the name of the subprogram that evaluates the corresponding function of the argument N. A FUNCTION subprogram begins with a FUNCTION statement and ends with an END statement. It returns control to the calling program by means of one or more RETURN statements.

7.4.1 FUNCTION Statement

The FUNCTION statement has the form:

```
FUNCTION identifier(argument,argument,...)
```

This statement declares the program which follows to be a FUNCTION subprogram. The identifier is the name of the function being defined. This identifier must not be used as a dummy argument or appear in any nonexecutable statement in the program other than as a scalar variable in a TYPE statement. It must appear as a scalar variable and be assigned a value during execution of the subprogram which is the function value.

Arguments appearing in the list enclosed in parentheses are dummy arguments representing the function argument. The arguments must agree in number, order, and type with the actual arguments used in the calling program. FUNCTION subprogram arguments may be expressions, alphanumeric strings, array names, statement labels preceded by an asterisk (*) or dollar sign ($), or subprogram names.

Dummy arguments may appear in the subprogram as scalar identifiers, array identifiers, subprogram identifiers, or an asterisk (*) or dollar sign ($), denoting statement labels in the calling program. A function must have at least one dummy argument. Dummy arguments representing array names must appear within the subprogram in a DIMENSION statement, or one of the type statements that provide dimension information. Dimensions given as constants must equal the dimensions of the corresponding arrays in the calling program. In a DIMENSION statement, dummy identifiers may be used to specify adjustable dimensions for array name arguments. For example, in the statement sequence:

```
FUNCTION TABLE(A,M,N,B,X,Y)
          :
          :
DIMENSION A(M,N),B(10),C(50)
```

The dimensions of array A are specified by the dummies M and N, while the dimension of array B is given as a constant. The various values given for M and N by the calling program must be those of the actual arrays which the dummy A represents. The arrays may each be of different size but must have two dimensions. The arrays are dimensioned in the programs that use the function.

Dummy dimensions may be given only for dummy arrays. In the example above the array C must be given absolute dimensions, since C is not a dummy identifier. A dummy identifier may not appear in an EQUIVALENCE statement in the FUNCTION subprogram.

Dummy arguments representing statement labels can be used only in connection with the RETURN statement. When the value of the function is not required, a FUNCTION subprogram can be used as a SUBROUTINE subprogram by utilizing the optional return. When the optional return appears in a FUNCTION subprogram, the value of the function is stored on return only if RETURN or RETURN i (where i = 0) is used.

Example: FUNCTION LIST (A,$,C)

A function must not modify any arguments which appear in the FORTRAN arithmetic expression calling the function. Modification of implicit arguments from the calling program, such as variables in COMMON and DO loop indexes, is not allowed. The only FORTRAN statements not allowed in a FUNCTION subprogram are SUBROUTINE, BLOCK DATA, and another FUNCTION statement.

7.4.1.1 Function Type - The type of the function is the type of identifier used to name the function. This identifier may be typed, implicitly or explicitly, in the same way as any other identifier. Alternatively, the function

may be explicitly typed in the FUNCTION statement itself by preceding the word FUNCTION with one of the types or equivalent types described in Section 6.3. For example:

```
INTEGER FUNCTION
REAL FUNCTION
COMPLEX FUNCTION
LOGICAL FUNCTION
DOUBLE PRECISION FUNCTION
REAL*8 FUNCTION
```

Thus, the statement

```
COMPLEX FUNCTION HPRIME(S,N)
```

is equivalent to the statements

```
FUNCTION HPRIME(S,N)
COMPLEX HPRIME
```

Examples:

```
FUNCTION MAY(RANGE,EP,YP,ZP)
COMPLEX FUNCTION COT(ARG)
DOUBLE PRECISION FUNCTION LIMIT(X,Y)
FUNCTION WORK (A,$,C)
```

7.5 SUBROUTINE SUBPROGRAMS

A SUBROUTINE subprogram may be multivalued and can be referred to only by a CALL statement. A SUBROUTINE subprogram begins with a SUBROUTINE statement and returns control to the calling program by means of one or more RETURN statements.

7.5.1 SUBROUTINE Statement

The SUBROUTINE statement has the form:

```
SUBROUTINE identifier(argument,argument,...)
```

This statement declares the program which follows to be a SUBROUTINE subprogram. The first identifier is the subroutine name. This identifier cannot be used as a dummy argument or appear in any nonexecutable statement in the program other than as a scalar variable in a TYPE statement. The subroutine name can, however, be used as a scalar variable in any executable statement in the program. The arguments in the list enclosed in parentheses are dummy arguments representing the arguments of the subprogram. The dummy arguments must agree in number, order, and type with the actual arguments used by the calling program.

SUBROUTINE subprograms may have expressions, alphanumeric strings, array names, statement labels, and subprogram names as arguments. The dummy arguments may appear as scalar, array, subprogram identifiers, or an

asterisk (*) or dollar sign ($) denoting a statement label in the calling program. Dummy arguments representing statement labels can be used only in connection with the RETURN statement.

Dummy identifiers which represent array names must be dimensioned within the subprogram by a DIMENSION or type declaration statement. As in the case of a FUNCTION subprogram, either constants or dummy identifiers

may be used to specify dimensions in a DIMENSION statement. The dummy arguments must not appear in an EQUIVALENCE or COMMON statement in the SUBROUTINE subprogram.

A SUBROUTINE subprogram may use one or more of its dummy identifiers to represent results. The subprogram name is not used for the return of results. A SUBROUTINE subprogram need not have any argument at all.

Examples:
```
SUBROUTINE FACTOR(COEFF,N,ROOTS)
SUBROUTINE RESIDU(NUM,N,DEN,M,RES)
SUBROUTINE SERIES
SUBROUTINE TYPE(A,$,B,*)
```

The only FORTRAN statements not allowed in a function subprogram are FUNCTION, BLOCK DATA, and another SUBROUTINE statement.

7.5.2 CALL Statement

The CALL statement assumes one of two forms:

```
CALL identifier
CALL identifier (argument,argument,...,argument)
```

The CALL statement is used to transfer control to SUBROUTINE subprogram. The identifier is the subprogram name.

The arguments may be expressions, array identifiers, alphanumeric strings, subprogram identifiers, or statement labels of the calling program preceded by an asterisk (*), dollar sign ($), or ampersand (&). Arguments may be of any type, but must agree in number, order, type, and array size (except for adjustable arrays, as discussed under the DIMENSION statement) with the corresponding arguments in the SUBROUTINE statement of the called subroutine. Unlike a function, a subroutine may produce more than one value and cannot be referred to as a basic element in an expression.

A subroutine may use one or more of its arguments to return results to the calling program. If no arguments at all are required, the first form is used.

Examples:
```
CALL EXIT
CALL SWITCH(SIN,2.LE.BETA,X**4,Y)
CALL TEST(VALUE,123,275)
CALL TYPE(A,$10,B,*20,&30)
```

The identifier used to name the subroutine is not assigned a type and has no relation to the types of the arguments. Arguments which are constants or formed as expressions must not be modified by the subroutine.

7.5.3 RETURN Statement

The RETURN statement has one of two forms:

RETURN
RETURN i

where i is an integer constant or an interger variable. The value of i must be positive, and specifies that the return is to the i-th argument of the referencing statement (where the i-th argument is a statement number preceded by a $ or *). If i=0, the return is the same as with the first form of the RETURN statement.

This statement returns control from a subprogram to the calling program. Normally, the last statement executed in a subprogram is a RETURN statement. Any number of RETURN statements may appear in a subprogram. For purposes of debugging functions and subroutines originally written as main programs, the RETURN statement has been made equivalent to the STOP statement in a main program.

7.6 BLOCK DATA SUBPROGRAMS

A BLOCK DATA subprogram is a data specification subprogram and is used to enter initial values into variables in COMMON for use by FORTRAN subprograms and MACRO-10 main programs (see Chapter 9). No executable statements may appear in a BLOCK DATA subprogram.

7.6.1 BLOCK DATA Statement

The BLOCK DATA statement has the form:

BLOCK DATA

This statement declares the program which follows to be a data specification subprogram and it must be the first statement of the subprogram (see Chapter 6, Section 6.2.2).

7.7 EXTERNAL STATEMENT

FUNCTION and SUBROUTINE subprogram names may be used as the actual arguments of subprograms. Such subprogram names must be distinguished from ordinary variables by their appearance in an EXTERNAL statement. The EXTERNAL statement has the form:

EXTERNAL identifier, identifier,...,identifier

This statement declares the listed identifiers to be subprogram names. Any subprogram name given as an argument to another subprogram must have previously appeared in an external declaration in the calling program (i.e., as an identifier in an EXTERNAL or CALL statement or as a function name in an expression).

Example:

```
EXTERNAL SIN,COS
          .
          .
          .
CALL TRIGF(SIN,1.5,ANSWER)
          .
          .
          .
CALL TRIGF(COS,.87,ANSWER)
          .
          .
          .
END
```

```
SUBROUTINE TRIGF(FUNC,ARG,ANSWER)
        :
ANSWER = FUNC(ARG)
        :
RETURN
END
```

To reference external variables from a MACRO-10 program by name, place the variables in named COMMON. Use the name of the variable as the name of the COMMON block:

```
COMMON /A/A/B/B(13)/C/C(6,7)
```

7.8 SUMMARY OF PDP-10 FORTRAN IV STATEMENTS

CONTROL STATEMENTS

General Form	Section References
ASSIGN i to m	4.1.3
CALL name (a_1,a_2,...)	7.5.2
CONTINUE	4.4
DO i m=m_1,m_2,m_3	4.3
GO TO i	4.1.1
GO TO m	4.1.3
GO TO m, (i_1,i_2,...)	4.1.3
GO TO (i_1,i_2,...),m	4.1.2
IF (e_1)i_1,i_2,i_3	4.2.1
IF (e_2)s	4.2.2
PAUSE	4.5
PAUSE j	4.5
PAUSE 'h'	4.5
RETURN	7.5.3
RETURN i	7.5.3
STOP	4.6
END	4.7

DATA TRANSMISSION STATEMENTS

General Form	Section References
ACCEPT f	5.2.9
ACCEPT f,list	5.2.9
BACKSPACE unit	5.3
DECODE (n,f,v)list	5.4
END FILE unit	5.3

General Form	Section References
ENCODE (n,f,v)list	5.4
FORMAT (g)	5.1.1
PRINT f	5.2.3
PRINT f, list	5.2.3
PUNCH f	5.2.4
READ f	5.2.7
READ f, list	5.2.7
READ (unit, f)	5.2.7
READ (unit,f)list	5.2.7
READ (unit)list	5.2.7
READ (unit,$name_1$)	5.2.7
READ (unit #R,f)list	5.2.7
READ (unit,f,END=c,ERR=d)list	5.2.7
READ (unit,f,END=c)list	5.2.7
READ (unit,f,ERR=d)list	5.2.7
REREAD f,list	5.2.8
REWIND unit	5.3
SKIP RECORD unit	5.3
TYPE f	5.2.5
TYPE f,list	5.2.5
WRITE (unit,f)	5.2.6
WRITE (unit,f)list	5.2.6
WRITE (unit)list	5.2.6
WRITE (unit,$name_1$)	5.2.6
WRITE (unit #R,f)list	5.2.6
UNLOAD unit	5.3

SPECIFICATION STATEMENTS

General Form	Section References
BLOCK DATA	6.2.2
COMMON a(n_1,n_2,...),b(n_3,n_4,...),...	6.1.2
COMMON /blk1/a,b/blk2/c,d/...	6.1.2
COMPLEX a(n_1,n_2,...),b(n_3,n_4,...),...	6.3
DATA t,u,.../k_1,k_2,k_3,.../ v,w,.../k_4,k_5,k_6,.../...	6.2.1

General Form	Section References
DIMENSION $a(n_1,n_2,\ldots),b(n_1,n_2,\ldots),\ldots$	6.1.1
DOUBLE PRECISION $a(n_1,n_2,\ldots),b(n_3,n_4,\ldots),\ldots$	6.3
EQUIVALENCE $(a(n_1,\ldots),b(n_2,\ldots),\ldots),\ldots$ $(c(n_3,\ldots),d(n_4,\ldots),\ldots),\ldots$	6.1.3
EXTERNAL y,z,...	7.7
IMPLICIT $type_1(l_1\text{-}l_2),type_2(l_3\text{-}l_4),\ldots$	6.3.1
INTEGER $a(n_1,n_2,\ldots),b(n_3,n_4,\ldots),\ldots$	6.3
LOGICAL $a(n_1,n_2,\ldots),b(n_3,n_4,\ldots),\ldots$	6.3
NAMELIST $/name_1/a,b,\ldots/name_2/c,d,\ldots$	5.1.2
REAL $a(n_1,n_2,\ldots)b(n_3,n_4,\ldots),\ldots$	6.3
SUBSCRIPT INTEGER $a(n_1,n_2,\ldots),b(n_3,\ldots),\ldots$	6.3

ARITHMETIC STATEMENT FUNCTION DEFINITION

General Form	Section Reference
name(a,b,...)=e	7.3

NOTE:

$a_1,a_2,\ldots$	are expressions
a,b,c,d	are variable names
blk1,blk2	are block names
c	is the statement number to which control is transferred upon encountering an end-of-file
d	is the statement number to which control is transferred upon encountering an error condition on the input data.
e	is an expression
e_1	is a noncomplex expression
e_2	is a logical expression
f	is a format number
g	is a format specification
'h'	is an alphanumeric
$i,i_1,i_2,\ldots$	are statement numbers
j	is an integer constant
$k_1,k_2,\ldots$	are constants of the general form j*k where k is any constant
$l_1,l_2,\ldots$	are letters

General Form	Section Reference
list	is an input/output list
m	is an integer variable name
m_1, m_2, m_3	are integer expressions
$n_1, n_2, \ldots$	are dimension specifications
n	are the number of ASCII characters
name	is a subroutine or function name
$name_1, name_2$	are NAMELIST names
#R	is a record number where I/O begins
s	is a statement (not DO or logical IF)
t,u,v,w	are variable names or input/output lists
$type_1, type_2, \ldots$	are type specifications
unit	is an integer variable or constant specifying a logical device number
v	is the starting address of the ASCII record referenced
y,z	are external subprogram names

SECTION II
THE RUN TIME SYSTEM

The five chapters of this section contain information on LIB40, SUBPROGRAM calling sequences, accumulator usage, compiler switches and diagnostic messages, and FORTRAN user programming.

CHAPTER 8
LIB40

LIB40 is a single file which contains all of the programs in the FORTRAN library. It is composed of three groups of programs:

(1) The FORTRAN Operating System.
(2) Science Library.
(3) FORTRAN Utility Subprograms.

There are two forms of LIB40, one for the KA-10 and the other for the KI-10. The KA-10 library will run on the KI-10, but will not take advantage of the speed of the KI-10. The KI-10 library will not run on the KA-10 because of the hardware differences. Also, the library used must match the compiler used, i.e., KA-10 compiled code must use the KA-10 LIB40 and the KI-10 compiled code must use the KI-10 LIB40.

8.1 THE FORTRAN OPERATING SYSTEM

The system programs in the FORTRAN Operating System act as the interface between the user's program and the PDP-10. All of these programs are invisible to the user's program. The FORTRAN Operating System is loaded automatically from LIB40 and resides in the user's core area along with the user's main programs and any library functions and subroutines that his programs reference.

8.1.1 FORSE.

FORSE. is the main program of the FORTRAN Operating System and is loaded whenever a FORTRAN main program is in core. The primary functions of FORSE. are

a. FORMAT statement processing,
b. Dispatching of all UUOs, and
c. Control of I/O devices at runtime.

8.1.1.1 FORMAT Processing - FORSE. assumes that all FORMAT statements are syntactically correct since the syntax of each statement is checked by the compiler. FORSE. scans the FORMAT statements and performs the indicated I/O operations. FORSE. invokes the required conversion routine to actually do data conversion. The conversion routine that is used is a function of the conversion indicated in the FORMAT statement and of the data type of the element in the I/O list.

8.1.1.2 UUO Dispatching - Some UUOs are handled minimally by FORSE. (NLIN, NLOUT, MTOP), but the others are handled almost entirely within FORSE.

8.1.1.3 I/O Device Control - FORSE. executes the required carriage control of output devices that are physical listing devices (LPT, TTY) and stores the carriage control character at the beginning of each line if the output is going to a retrievable medium for deferred listing. When listings are deferred, the appropriate switch in PIP can be used to list the file and execute the required carriage control.

8.1.1.4 Additional Functions of FORSE. - FORSE. is responsible for the following:

a. Control of REREAD and ENCODE/DECODE features.

b. Interaction with EOFTST and READ (unit,f,END=C)list to handle end-of-file testing.

c. Control of the assignment of devices to software channels.

d. Control of the handling of filenames for I/O associated with directory devices.

e. Control of the opening and closing of data files.

f. Control the handling of the functions associated with the MAGDEN, BUFFER, IBUFF, OBUFF, DEFINE FILE, TRAPS, and RELEASE subroutines.

8.1.2 I/O Conversion Routines

The I/O conversion routines convert data from internal PDP-10 format to external format or vice versa. The calls to these routines are implied by FORMAT and data transfer statements in the FORTRAN source program. The routines reside as relocatable binary files in LIB40. REL.

Table 8-1
I/O Conversion Routines

Routine	Description
ALPHI.	Alphanumeric ASCII input conversion
ALPHO.	Alphanumeric ASCII output conversion
FLIRT.*	Floating point and double precision input conversion
FLOUT.*	Floating point and double precision output conversion
INTI.	Integer input conversion
INTO.	Integer output conversion
LINT.	Logical input conversion
LOUT.	Logical output conversion

*FLIRT. contains two entry points, FLIRT and DIRT.
FLOUT. contains two entry points, FLOUT and DOUBT.

Table 8-1 (Cont)
I/O Conversion Routines

Routine	Description
BINWR.	Binary I/O
OCTI.	Octal input conversion
OCTO.	Octal output conversion
NMLST.	Namelist

8.1.3 FORTRAN UUOs

Operation codes 000 through 077 in the PDP-10 are programmed operators, sometimes referred to as UUO's (Unimplemented User Operators) since from a hardware point of view their function is not prespecified. Some of these op-codes trap to the Monitor and the rest trap to the user program. FORTRAN UUO's trap to the FORTRAN Operating System UUO Handler and are then processed.

Table 8-2
FORTRAN UUOs

UUO	Op Code	Meaning
RESET.	015	Resets all devices, clears tables and flags.
IN.	016	Initializes device for formatted input, does a LOOKUP.
OUT.	017	Initializes device for formatted output, does an ENTER.
DATA.	020	Converts one data element from external to internal format or vice versa depending upon whether input or output is being done. Actual data transfer takes place.
FIN.	021	Terminates data transfer statements.
RTB.	022	Initializes device for unformatted input, similar to IN.
WTB.	023	Initializes device for unformatted output, similar to OUT.
MTOP.	024	Performs Magtape operations, rewind, rewind and unload, backspace, end file, skip, write blank record.
SLIST.	025	Converts entire arrays from external to internal format or vice versa depending upon whether input or output is being done. Actual data transfer takes place.
INF.	026	IFILE. Sets up input filename, similar to IN. but with specified filename.
OUTF.	027	OFILE. Sets up output filename, similar to OUT. but with specified filename.
RERED.	030	REREAD. Reread last record.
NLI.	031	Namelist input.

Table 8-2 (Cont)
FORTRAN UUOs

UUO	Op Code	Meaning
NLO.	032	Namelist output.
DEC.	033	DECODE.
ENC.	034	ENCODE.

8.2 SCIENCE LIBRARY AND FORTRAN UTILITY SUBPROGRAMS

The Science Library and FORTRAN Utility Subprograms extend the capabilities of the FORTRAN language. These subprograms are called explicitly by the user. The subprograms include the built-in FORTRAN math functions and the user-called utility subroutines which provide optional I/O capabilities and control of and information about the program's environment. The optional I/O capabilities and environmental control are achieved by the subroutines from interactions with the FORTRAN Operating System.

8.2.1 FORTRAN IV Library Functions

This section contains descriptions of all standard function subprograms provided with the FORTRAN IV library for the PDP-10. These functions are called by using the function mnemonic as a function name in an arithmetic expression. The function mnemonics in Table 8-3 have the types specified unless their types are explicitly or implicitly changed. (Refer to Section 6.3, "Type Declaration Statements" and Section 6.3.1, "IMPLICIT Statement.")

Table 8-3
FORTRAN IV Library Functions

Function	Mnemonic	Definition	Number of Arguments	Type of Argument	Type of Function	External Calls
Absolute value:						
Real	ABS	|arg|	1	Real	Real	
Integer	IABS	|arg|	1	Integer	Integer	
Double precision	DABS	|arg|	1	Double	Double	
Complex to real	CABS	$c=(x^2+y^2)^{1/2}$	1	Complex	Real	SQRT
Conversion:						
Integer to real	FLOAT*		1	Integer	Real	
Real to integer	IFIX*	Result is largest integer $\leq a$	1	Real	Integer	
Double to real	SNGL		1	Double	Real	
Real to double	DBLE		1	Real	Double	
Integer to double	DFLOAT		1	Integer	Double	
Complex to real (obtain real part)	REAL		1	Complex	Real	
Complex to real (obtain imaginary part)	AIMAG		1	Complex	Real	
Real to complex	CMPLX	$c=Arg_1+i*Arg_2$	2	Real	Complex	
Truncation:						
Real to real	AINT	Sign of arg * largest integer $\leq$ |arg|	1	Real	Real	
Real to integer	INT*		1	Real	Integer	
Double to integer	IDINT		1	Double	Integer	
Remaindering:						
Real	AMOD	The remainder when Arg 1 is divided by Arg 2	2	Real	Real	ERROR., TRAPS
Integer	MOD		2	Integer	Integer	
Double precision	DMOD		2	Double	Double	
Maximum Value:						
	AMAX0	$Max(Arg_1, Arg_2, \ldots)$	≥ 2	Integer	Real	FLOAT
	AMAX1			Real	Real	
	MAX0			Integer	Integer	
	MAX1			Real	Integer	IFIX
	DMAX1			Double	Double	
Minimum Value:						
	AMIN0	$Min(Arg_1, Arg_2, \ldots)$	≥ 2	Integer	Real	FLOAT
	AMIN1			Real	Real	
	MIN0			Integer	Integer	
	MIN1			Real	Integer	IFIX
	DMIN1			Double	Double	

*These functions are not used on the KI-10 because they are unnecessary.

Table 8-3 (Cont)
FORTRAN IV Library Functions

Function	Mnemonic	Definition	Number of Arguments	Type of Argument	Type of Function	External Calls
Transfer of Sign:						
Real	SIGN	$\{ Sgn(Arg_2)*\|Arg_1\| \}$	2	Real	Real	
Integer	ISIGN		2	Integer	Integer	
Double precision	DSIGN		2	Double	Double	
Positive Difference:						
Real	DIM	$\{ Arg_1 - Min(Arg_1, Arg_2) \}$	2	Real	Real	
Integer	IDIM		2	Integer	Integer	
Exponential:						
Real	EXP	$\{ e^{Arg} \}$	1	Real	Real	ERROR.
Double	DEXP		1	Double	Double	
Complex	CEXP		1	Complex	Complex	EXP,SIN,COS, ALOG,ERROR.
Logarithm:						
Real	ALOG	$\log_e$ (Arg)	1	Real	Real	ERROR.
	ALOG10	$\log_{10}$ (Arg)	1	Real	Real	ERROR.
Double	DLOG	$\log_e$ (Arg)	1	Double	Double	
	DLOG10	$\log_{10}$ (Arg)	1	Double	Double	
Complex	CLOG	$\log_e$ (Arg)	1	Complex	Complex	ALOG,ATAN2, SQRT,ERROR.
Square Root:						
Real	SQRT	$(Arg)^{1/2}$	1	Real	Real	ERROR.
Double	DSQRT	$(Arg)^{1/2}$	1	Double	Double	
Complex	CSQRT	$c=(x+iy)^{1/2}$	1	Complex	Complex	SQRT
Sine:						
Real (radians)	SIN		1	Real	Real	
Real (degrees)	SIND	$\{ \sin(Arg) \}$	1	Real	Real	
Double (radians)	DSIN		1	Double	Double	
Complex	CSIN		1	Complex	Complex	SIN,SINH,COSH, ALOG,EXP
Cosine:						
Real (radians)	COS		1	Real	Real	
Real (degrees)	COSD	$\{ \cos(Arg) \}$	1	Real	Real	
Double (radians)	DCOS		1	Double	Double	
Complex	CCOS		1	Complex	Complex	SIN,SINH,COSH, ALOG,EXP

Table 8-3 (Cont)
FORTRAN IV Library Functions

Function	Mnemonic	Definition	Number of Arguments	Type of		External Calls
				Argument	Function	
Hyperbolic:						
Sine	SINH	sinh (Arg)	1	Real	Real	EXP, ERROR.
Cosine	COSH	cosh (Arg)	1	Real	Real	EXP, ERROR.
Tangent	TANH	tanh (Arg)	1	Real	Real	EXP
Arc - sine	ASIN	asin (Arg)	1	Real	Real	ATAN, SQRT, ERROR.
Arc - cosine	ACOS	acos (Arg)	1	Real	Real	ATAN, SQRT, ERROR.
Arc tangent						
Real	ATAN	atan (Arg)	1	Real	Real	
Double	DATAN	atan (Arg)	1	Double	Double	
quotient of two arguments	ATAN2	atan (Arg_1/Arg_2)	2	Real	Real	ATAN, ERROR., TRAPS
	DATAN2	atan (Arg_1/Arg_2)	2	Double	Double	DATAN, ERROR.
Complex Conjugate	CONJG	Arg=X+iY, C=X-iY	1	Complex	Complex	
Random Number	RAN	result is a random number in the range of 0 to 1.0.	1	Integer, Real, Double, or Complex	Real	

8.2.2 FORTRAN IV Library Subroutines

This section contains descriptions of all standard subroutine subprograms provided within the FORTRAN IV library for the PDP-10. These subprograms are closed subroutines and are called with a CALL statement.

Table 8-4
FORTRAN IV Library Subroutines

Subroutine Name	Effect
BUFFER	Allows the programmer to specify buffering for a device at one of fifteen levels. CALL BUFFER (unit*, in/out, number) where in/out is 1 for input buffering only, 2 for output buffering only, or 3 for both, and number is the level of buffering ($1 \leq$ number ≤ 15). If number is not specified, 2 is assumed. In calls to two entries in BUFFER, IBUFF and OBUFF, the programmer can specify a non-standard buffer size if the records in his data files exceed standard buffer sizes set by the Monitor. (See Table 12-1.) The programmer cannot change buffer sizes for the disk; IBUFF and OBUFF are designed primarily for Magtape. CALL IBUFF (d,n,s) where d is the device number, n is the number of buffers, and s is the size of buffer.
CHAIN	Reads a segment of coding (Chain file) into core and links it to a program already residing in core. CALL CHAIN (type,device,file) where type is 0 (the next Chain file is read into core immediately above the permanent resident area) or type is 1 (the next Chain file is read into core immediately above the FORTRAN IV program which marks the end of the removable resident). Device is 0,1,2,... FORTRAN IV logical device number (Chain files can be stored on DSK, MTA, or DTA only) corresponding to the device where the Chain file can be found. File is 0 for reading the next file from the selected magnetic tape or 1,2,... for the number of the magnetic tape unit where the Chain file is located.
DATE	Places today's date as left-justified ASCII characters into a dimensioned 2-word array. CALL DATE (array) where array is the 2-word array. The date is in the form dd-mmm-yy

*For explanation, see page 7-10.

Table 8-4 (Cont)
FORTRAN IV Library Subroutines

Subroutine Name	Effect
DATE (cont)	where dd is a 2-digit day (if the first digit is 0, it is converted to a blank), mmm is a 3-digit month (e.g., MAR), and yy is a 2-digit year. The date is stored in ASCII code, left-justified in the two words.
DUMP	Causes particular portions of core to be dumped and is referred to in the following form: CALL DUMP (L_1,U_1,F_1,...,L_n,U_n,F_n) where L_i and U_i are the variable names which give the limits of core memory to be dumped. Either L_i or U_i may be upper or lower limits. F_i is a number indicating the format in which the dump is to be performed: 0=octal, 1=real, 2=integer, and 3=ASCII. If F is not 0,1,2,3, the dump is in octal. If F_n is missing, the last section is dumped in octal. If U_n and F_n are missing, an octal dump is made from L to the end of the job area. If L_n, U_n, and F_n are missing, the entire job area is dumped in octal. The dump is terminated by a call to EXIT.
EOF1(unit*)	Skips one end-of-file terminator when found and returns the value TRUE if an end-of-file was found and FALSE if it was not found. Subsequent terminators produce an error message.
EOFC(unit*)	Skips more than one end-of-file terminators when found and returns the value TRUE if an end-of-file was found or FALSE if it was not found.
ERRSET	Allows the user to control the typeout of execution-time arithmetic error messages, ERRSET is called with one argument in integer mode. CALL ERRSET(N) Typeout of each type of error message is suppressed after N occurances of that error message. IF ERRSET is not called, the default value of N is 2.
EXIT	Returns control to the Monitor and, therefore, terminates the execution of the program.
IFILE	Performs LOOKUPs for files to be read from DECtape and disk. CALL IFILE(unit*,filnam) where filnam is a filename consisting of five or fewer ASCII characters enclosed in single quotes ('). e.g., CALL IFILE (12, 'FILE1')

*For explanation, see page 7-10.

Table 8-4 (Cont)
FORTRAN IV Library Subroutines

Subroutine Name	Effect
ILL	Sets the ILLEG flag. If the flag is set and an illegal character is encountered in floating-point/double-precision input, the corresponding word is set to zero. CALL ILL
LEGAL	Clears the ILLEG flag. If the flag is set and an illegal character is encountered in floating-point/double-precision input, the corresponding word is set to zero. CALL LEGAL
MAGDEN	Allows specification of magnetic tape density and parity. CALL MAGDEN(unit*,density,parity) where density is the tape density desired (200=200 bpi, 556=556 bpi, or 800=800 bpi) and parity is the tape parity desired (0=odd, 1=even). Even parity is intended for use with BCD-coded tapes only.
OFILE	Performs ENTERs for files to be written on DECtape and disk. CALL OFILE (unit*,filnam) where filnam is a filename consisting of five ASCII characters.
PDUMP	Is referred to in the following form: CALL PDUMP(L_1,U_1,F_1,...,L_n,U_n,F_n) where the arguments are the same as those for DUMP. PDUMP is the same as DUMP except that control returns to the calling program after the dump has been executed.
RELEAS	Closes out I/O on a device initialized by the FORTRAN Operating System and returns it to the uninitialized state. CALL RELEAS (unit*)
SAVRAN	SAVRAN is called with one argument in integer mode. SAVRAN sets its argument to the last random number (interpreted as an integer) that has been generated by the function RAN.
SETRAN	SETRAN has one argument which must be a non-negative integer $< 2^{31}$. The starting value of the function RAN is set to the value of this argument, unless the argument is zero. In this case, RAN uses its normal starting value.

*For explanation, see page 7-10.

Table 8-4 (Cont)
FORTRAN IV Library Subroutines

Subroutine Name	Effect
SLITE(i)	Turns sense lights on or off. i is an integer expression. For $1 \leq i \leq 36$ sense light i will be turned on. If i = 0, all sense lights will be turned off.
SLITE(i,j)	Checks the status of sense light i and sets the variable j accordingly and turns off sense light i. If i is on, j is set to 1; and if i is off, j is set to 2.
SSWTCH(i,j)	Checks the status of data switch i($0 \leq i \leq 35$) and sets the variable j accordingly. If i is set down, j is set to 1; and, if i is up, j is set to 2.
TIME	Returns the current time in its argument(s) in left-justified ASCII characters. If TIME is called with one argument, CALL TIME(X) the time is in the form hh : mm where hh is the hours (24-hour time) and mm is the minutes. If a second argument is requested, CALL TIME(X,Y) the first argument is returned as before and the second has the form ss.t where ss is the seconds and t is the tenths of a second.

CHAPTER 9
SUBPROGRAM CALLING SEQUENCES

This chapter describes the conventions used in writing MACRO subprograms which can be called by FORTRAN IV programs, and FORTRAN subprograms which can be linked to MACRO main programs. The reader is assumed to be familiar with the following texts:

MACRO-10 Assembler (DEC-10-AMZB-D)
Section 2.5.8 "Linking Subroutines"
Figure 7-1, "Sample Program, CLOG"

TOPS-10 Monitor Calls (DEC-10-MRRA-D)
Section 1.2.2 "Loading Relocatable Binary Files"

Science Library and FORTRAN Utility Subprograms (DEC-10-SFLE-D)
How to Use This Manual - FORTRAN calling sequences

9.1 MACRO SUBPROGRAMS CALLED BY FORTRAN MAIN PROGRAMS

9.1.1 Calling Sequences

The FORTRAN calling sequence, in the main program, for a subroutine is

FORTRAN Code	MACRO Code (Generated by Compiler)
CALL subprog (adr_1, adr_2,...)	JSA 16, subprog ARG $code_1$, adr_1 ARG $code_2$, adr_2 ⋮

where

subprog — is the name of the subprogram

adr_1, adr_2,... — are the addresses of the arguments

$code_1$, $code_2$ — are the accumulator fields of the ARG instructions which indicate the type of argument being passed to the subprogram. These codes are as follows:

0	Integer argument	4	Octal argument
1	Unused	5	Hollerith argument
2	Real argument	6	Double-precision argument
3	Logical argument	7	Complex argument

An example of a FORTRAN calling sequence for a subroutine and the MACRO-10 coding generated by the compiler is given below.

FORTRAN Code	MACRO Code
CALL PROG1 (REAL,INT)	JSA 16, PROG1
	ARG 02, REAL
	ARG 00, INT

The MACRO code generated by the compiler is the same for subroutines and functions; however, the FORTRAN code is different.

9.1.2 Returning of Answers

A subroutine returns to its answers in specified locations in the main program. These locations are often given as argument names or as variable names.

A function returns its answer in accumulator 0 (if a single word result) or in accumulators 0 and 1 (if a double-precision or complex result). A function may also return its answer in specified locations (given by argument names in the CALL) or variable names; in any event, however, it must return an answer in accumulator 0 (or accumulators 0 and 1).

A MACRO subprogram access COMMON by declaring as external common block names for labelled COMMON and by declaring .COMM. as external for blank common. A common block name always refers to the same core location as the first element following the block name in a COMMON statement. MACRO subprograms may refer to the remainder of the variables in the common block through additive globals.

9.1.3 Use of Accumulators

For accumulator usage, see Chapter 10, Accumulator Conventions for PDP-10 Main Programs and Subprograms.

9.1.4 Examples of Subprogram Linkage

Three examples of subprogram linkage, one of a subroutine, one of a function subprogram, and one of a FORTRAN main program and MACRO subprogram both referencing COMMON, are given below.

9.1.4.1 Example of a Subroutine Linkage - The coding of the subroutine in this example is followed by the calling sequence.

```
ENTRY     SUBA
SUBA:     0
          MOVE      1,@0(16)      ;GET FIRST ARGUMENT
          IMULI     1, 12         ;MULTIPLY BY 10
          MOVEM     1,@0(16)      ;RETURN RESULT IN ARGUMENT
          JRA       16, 1(16)     ;RETURN TO MAIN PROGRAM
```

FORTRAN Calling Sequence	MACRO Code (Generated by Compiler)
CALL SUBA(INT)	JSA 16, SUBA ARG 00, INT

9.1.4.2 Example of a Function Subprogram Linkage - The coding of the function subprogram in this example is followed by the calling sequence.

```
ENTRY     FNC
FNC:      0
          MOVE      00,@0(16)     ;PICK UP FIRST ARGUMENT
          MOVE      01,@1(16)     ;PICK UP SECOND ARGUMENT
          IMUL      00, 01        ;MULTIPLY BOTH ARGUMENTS
                                  ;RESULT IN AC0
          JRA       16, 2(16)     ;RETURN WITH ANSWER IN AC0
```

FORTRAN Calling Sequence	MACRO Code (Generated by Compiler)
X = FNC (I, 10)	JSA 16, FNC ARG 00, I ARG 00, CONST.

9.1.4.3 Example of a FORTRAN Main Program and a MACRO Subprogram Both Referencing COMMON.

```
T           F40         V013        28-NOV-69   12:24

1M          BLOCK       0

                                                        DIMENSION A(5),B(3,4),C(3)

                                                        COMMON C

                                                        COMMON/A/A/B/B/D/D

                                                        A(2)=B(2,3)+C(3)+D

            MOVE        02,D
            FADR        02,B+7
            FADR        02,C+2
            MOVEM       02,A+1

            JSA         16,SUB2                     CALL SUB2

            JSA         16,EXIT                     END
MAIN.%      RESET.      00,0
            JRST        1M

COMMON
C           /.COMM./                0
A           /A/         0
B           /B/         0
D           /D/         0

SUBPROGRAMS

FORSE.
JOBFF
SUB2
EXIT

SCALARS

D           0

ARRAYS
```

```
A               0
B               0
C               0

MAIN.           ERRORS DETECTED:  0

  2K CORE USED
```

```
.MAIN       MACRO.V36    12:23    28-NOV-69

                                                EXTERNAL .COMM.,A,B,D
                                                ENTRY       SUB2
            000000      000000      000000  SUB2:   0
            000001      200000      000002          MOVE        0,A+2       ;GET A(3)
            000002      202000      000003          MOVEM       0,B+3       ;STORE IN B(1,2)
            000003      200000      000000          MOVE        0,.COMM.    ;GET C
            000004      202000      000000          MOVEM       0,D         ;STORE IN D
            000005      267716      000000          JRA         16,(16)     ;RETURN TO FORTRAN PROGRAM
                                                    END                     ;END

NO ERRORS DETECTED

PROGRAM BREAK IS 000006

                SYMBOL TABLE

A                       000000  EXT         B               000000  EXT         D       000004' EXT
SUB2                    000000' INT         .COMM.          000003' EXT
```

003466 IS THE PROGRAM BREAK

STORAGE MAP

MAIN.	000140	000035
MAIN.	000146	
.COMM.	000150	
A	000153	
B	000160	
D	000174	
.MAIN	000175	000006
SUB2	000175	
JOBDAT	000203	000000
FORSE.	000203	002374
BUFCA.	001624	
BUFHD.	002337	
CHINN.	001121	
CLOS.	002002	
CLOSI.	002000	
CLROU.	001763	
CLRSY.	001770	
DADDR.	002276	
DATA.	000000	
DEPOT.	001004	
DEVIC.	002477	
DEVNO.	002172	
DYNDV.	002212	
DYNND.	002356	
ENDLN.	001047	
EOFFL.	002205	
EOFTS.	001214	
EOL.	002275	
FI.	001112	
FIN.	000000	
FMTBG.	002274	
FMTEN.	002273	
FNCTN.	001751	
IORTR.	000334	
LOOK.	002034	
MTOP.	000000	
MTPZ.	002030	
NLI.	000000	
NLO.	000000	
FORSE.	000203	
IIB.	001141	
IN.	000000	
INF.	000000	
INP.	002007	
INPDV.	002203	
NXTCR.	001162	
NXTLN.	001172	
ONLY1.	002204	
OUT.	000000	
OUTF.	000000	
OUTT.	002013	
OVFLS.	002202	
PAKFL.	002176	
RERDV.	002501	
RERED.	000000	
RESET.	000000	
RIN.	000245	
RTB.	000000	
SESTA.	002020	
SETOU.	001755	
SLIST.	000000	
STAT.	001774	
TCNT1.	002506	
TCNT2.	002507	
TEMP.	002232	
TNAM1.	002133	
TANM2.	002132	
TPNTR.	002505	
TYPE.	002504	
UUOH.	001234	
WAIT.	002024	
WTB.	000000	
XIO.	000424	
ERROR.	002577	000431

ALPHO.	003250	
DDIRT	003252	000002
DIRT.	003252	
DDOUBT	003254	000002
DOUBT.	003254	
DFLIRT	003256	000002
FLIRT.	003256	
DFLOUT	003260	000002
FLOUT.	003260	
DINTI	003262	000002
INTI.	003262	
DOCTI	003264	000002
OCTI.	003264	
DINTO	003266	000002
INTO.	003266	
DOCTO	003270	000002
OCTO.	003270	
DLINT	003272	000002
LINT.	003272	
DLOUT	003274	000002
LOUT.	003274	
DNMLST	003276	000003

BPHSE.	002777	
DEVER.	002667	
DPRER.	002767	
DUMER.	003041	
ENDTP.	002772	
ERROR.	002577	
ILLCH.	002634	
ILLMG.	003007	
ILRED.	003025	
ILUUO.	003051	
INIER.	002654	
LISTB.	002737	
LOGEN.	002627	
MSNG.	002707	
NMLER.	003020	
NOROM.	002720	
PARER.	003034	
QTY1	003170	
REDER.	002746	
TBLER.	002700	
UUOM	003067	
WLKER.	002731	
EXIT	003230	000002
EXIT	003230	
EXIT.	003231	
IOADR.	003232	000014
IOADR.	003232	
DALPHI	003246	000002
ALPHI.	003246	
DALPHO	003250	000002

DELIM.	003300	
NMLST.	003276	
DTFMT	003301	000002
TFMT.	003301	
DBINWR	003303	000002
BINDT.	003303	
BINEN.	003303	
BINWR.	003303	
INPT.	003303	
DTPFCN	003305	000002
TPFCN.	003305	
DEVTB.	003307	000123
DATTB.	003363	
DEVLS.	003344	
DEVND.	003352	
DEVTB.	003307	
DVTOT.	000035	
MBFBG.	003352	
MTABF.	003353	
MTACL.	003421	
NEG1.	000005	
NEG2.	000007	
NEG3.	000003	
NEG5.	000002	
TABP1.	003363	
TABPT.	003362	
PDLST.	003432	000025
PDLST.	003432	
ILL	003457	000007
ILL	003457	
ILLEG.	003465	
LEGAL	003462	

LOADER 3K CORE
3+3K MAX 1225 WORDS FREE

9.2 MACRO MAIN PROGRAMS WHICH REFERENCE FORTRAN SUBPROGRAMS

9.2.1 Calling Sequences

The MACRO code which calls the FORTRAN subprogram should be the same as that produced by the FORTRAN IV compiler when it calls a subroutine. That is:

MACRO Code

```
JSA 16, subprog
ARG code1, adr1
ARG code2, adr2
      .
      .
```

where

subprog	is the name of the subprogram
adr_1, adr_2,...	are the addresses of the arguments
$code_1$, $code_2$	are the accumulator fields of the ARG instruction which indicate the type of argument being passed to the subprogram. These codes are as follows:

0	Integer argument
1	Unused
2	Real argument
3	Logical argument
4	Octal argument
5	Hollerith argument
6	Double-precision argument
7	Complex argument

Both subroutines and functions are called in this manner.

9.2.2 Returning of Answers

A FORTRAN subroutine returns its answers in specified locations in the main program. These locations may be given as variable names in COMMON or as argument names.

A FORTRAN function returns its answer in accumulator 0, if a single word result, or in accumulators 0 and 1, if a double-precision or complex result. A function may also return its answer in specified locations given by argument names in the CALL, or variable names in COMMON; in any event, however, it must return an answer in accumulator 0 (or accumulators 0 and 1).

If it is desired to reference a common block of data in both the MACRO main program and the FORTRAN subprogram, it is necessary to set up the common area first by loading a FORTRAN BLOCK DATA program before the MACRO main program and the FORTRAN subprogram.

9.2.3 Example of Subprogram Linkage

The following is an example of a FORTRAN subroutine being called by a MACRO main program. Both programs reference common data. Read and write statements have been omitted for simplification. Because the FORTRAN operating system, FORSE., sets up I/O channels at run time, the MACRO programmer must be sure not to initialize a device on a channel that FORSE. will then try to use, unless he releases the device before FORSE. is called. FORSE. initializes the first device encountered in the user program on software channel 1, the second on channel 2, etc.

It is possible to release a device from its associated channel in a FORTRAN program by a call to the subroutine RELEAS. Channels one through seventeen are available for I/O. If a FORTRAN user wishes to write MACRO programs which do I/O, he may use either FORTRAN UUO's or the channel numbers less than or equal to seventeen but greater than the largest number used by FORSE.

The FORTRAN RESET. UUO should be the first instruction executed in any program which accesses FORTRAN subroutines. For this reason the FORTRAN operating system, which contains the FORTRAN UUO handler routine, must be declared external in the MACRO main program. This causes FORSE. to be loaded. In general, any program in the FORTRAN library referenced in a MACRO program must be declared external. This results in the searching of LIB40 by the Linking Loader and loading the referenced program.

```
BLKDTA.F4          F40       V016      22-JAN-70          15:46

1M          BLOCK    0                                   BLOCK DATA

                                                         COMMON/A/A/B/B/C/C

                                                         COMMON D

                                                         DIMENSION A(5),B(2,3)

                                                         END

DAT.        BLOCK    0

COMMON
A           /A/      0
B           /B/      0
C           /C/      0
D           /.COMM./          0

SUBPROGRAMS

JOBFF

SCALARS

C           0
D           0

ARRAYS

A           0
B           0

DAT.        ERRORS DETECTED:  0

2K CORE USED
```

```
.MAIN    MACRO.V40 16:05 22-JAN-70
START .MAC

                                        ENTRY     START
                                        EXTERNAL                .COMM.,A,B,C,ARGS,FORSE.,EXIT.
000000  015000  000000  START:  RESET.   00,0                    ;DO FORTRAN UUO RESET, FOUND IN FORSE
000001  200000  000000          MOVE      0,A                   ;GET A(1)
000002  202000  000000          MOVEM     0,B                   ;STORE IN B(1,1)
000003  200000  000000          MOVE      0,C                   ;GET C
000004  202000  000000          MOVEM     0,.COMM.              ;STORE IN D
000005  200040  000002          MOVE      1,A+2                 ;GET A(3)
000006  202040  000005          MOVEM     1,B+5                 ;STORE IN B(2,3)
000007  266700  000000          JSA       16,ARGS               ;GO TO FORTRAN SUBROUTINE ARGS
                                JSA       16,EXIT.              ;EXIT. FORTRAN EXIT ROUTINE WHICH PRINTS
000010  266700  000000                                          ;OUT SUMMARIES AND ALSO CALLS MONITOR
                                                              ;LEVEL EXIT UUO. USER HAS OPTION TO USE
                                                              ;EITHER
                                END       START               ;END

NO ERRORS DETECTED

PROGRAM BREAK IS 000011

START    .MAC    SYMBOL TABLE

A               000001' EXT     ARGS            000007' EXT     B       000002' EXT
C               000003' EXT     EXIT.           000010' EXT     FORSE.  000000  EXT
START           000000' ENT     .COMM.          000004' EXT
```

```
1M      BLOCK   0


        MOVE    02,C
        FADR    02,D
        FADR    02,B
        MOVEM   02,A

        JRST    2M

        JRST    2M
ARGS%   ARG     00,0
        MOVEM   15,TEMP.
        MOVEM   16,TEMP.+1
        JRST    1M
2M      MOVE    15,TEMP.
        MOVE    16,TEMP.+1
        JRA     16,0(16)

COMMON
A       /A/     0
B       /B/     0
C       /C/     0
D       /.COMM./        0

SCALARS

ARGS    17
C       0
D       0
```

```
SUBROUTINE ARGS
COMMON /A/A/B/B/C/C
COMMON D
DIMENSION A(5),B(2,3)
A(1)=B(1,1)+C+D


RETURN
END
```

ARRAYS

A 0
B 0

ARGS ERRORS DETECTED: 0
2K CORE USED

003471 IS THE LOW SEGMENT BREAK

.MAIN STORAGE MAP 16:06 22-JAN-70

STARTING ADDRESS 000155 PROG .MAIN FILE START

DAT. 000140 000015

	DAT. 000140	A 000140	B 000145	C 000153			
	.COMM. 000154						
.MAIN	000155	000011					
	START	000155					
ARGS	000166	000020					
	ARGS	000174					
JOBDAT	000206	000000					
FORSE.	000206	002374					
	BUFCA. 001627	BUFHD. 002342	CHINN. 001124	CLOS. 002005			
	CLOSI. 002003	CLROU. 001766	CLRSY. 001773	DADDR. 002301			
	DATA. 000000	DEPOT. 001007	DEVIC. 002502	DEVNO. 002175			
	DYNDV. 002215	DYNND. 002361	ENDLN. 001052	EOFFL. 002210			
	EOFTS. 001217	EOL. 002300	FI. 001115	FIN. 000000			
	FMTBG. 002277	FMTEN. 002276	FNCTN. 001754	FORSE. 000206			
	IIB. 001144	IN. 000000	INF. 000000	INP. 002012			
	INPDV. 002206	IORTR. 000337	LOOK. 002037	MTOP. 000000			
	MTPZ. 002033	NLI. 000000	NLO. 000000	NXTCR. 001165			
	NXTLN. 001175	ONLY1. 002207	OUT. 000000	OUTF. 000000			
	OUTT. 002016	OVFLS. 002205	PAKFL. 002201	RERDV. 002504			

	RERED.	000000	RESET.	000000	RIN.	000250	RTB.	000000
	SESTA.	002023	SETOU.	001760	SLIST.	000000	STAT.	001777
	TCNT1.	002511	TCNT2.	002512	TEMP.	002235	TNAM1.	002136
	TNAM2.	002135	TPNTR.	002510	TYPE.	002507	UUOH.	001237
	WAIT.	002027	WTB.	000000	XIO.	000427		
ERROR.	002602	000431						
	BPHSE.	003002	DEVER.	002672	DPRER.	002772	DUMER.	003044
	ENDTP.	002775	ERROR.	002602	ILLCH.	002637	ILLMG.	003012
	ILRED.	003030	ILUUO.	003054	INIER.	002657	LISTB.	002742
	LOGEN.	002532	MSNG.	002712	NMLER.	003023	NOROM.	002723
	PARER.	003037	QTY1	003173	REDER.	002751	TBLER.	002703
	UUOM	003072	WLKER.	002734				
EXIT	003233	000002						
	EXIT	003233	EXIT.	003234				
IOADR.	003235	000014						
	IOADR.	003235						
DALPHI	003251	000002						
	ALPHI.	003251						
DALPHO	003253	000002						
	ALPHO.	003253						
DDIRT	003255	000002						
	DIRT.	003255						
DDOUBT	003257	000002						
	DOUBT.	003257						
DFLIRT	003261	000002						
	FLIRT.	003261						
DFLOUT	003263	000002						
	FLOUT.	003263						
DINTI	003265	000002						
	INTI.	003265						

DOCTI	003267	000002						
	OCTI.	003267						
DINTO	003271	000002						
	INTO.	003271						
DOCTO	003273	000002						
	OCTO.	003273						
DLINT	003275	000002						
	LINT.	003275						
DLOUT	003277	000002						
	LOUT.	003277						
DNMLST	003301	000003						
	DELIM.	003303	NMLST.	003301				
DTFMT	003304	000002						
	TFMT.	003304						
DBINWR	003306	000002						
	BINDT.	003306	BINEN.	003306	BINWR.	003306	INPT.	003306
DTPFCN	003310	000002						
	TPFCN.	003310						
DEVTB.	003312	000123						
	DATTB.	003366	DEVLS.	003347	DEVND.	003355	DEVTB.	003312
	DVTOT.	000035	MBFBG.	003355	MTABF.	003356	MTACL.	003424
	NEG1.	000005	NEG2.	000007	NEG3.	000003	NEG5.	000002
	TABP1.	003366	TABPT.	003365				
PDLST.	003435	000025						
	PDLST.	003435						
ILL	003462	000007						
	ILL	003462	ILLEG.	003470	LEGAL	003465		

LOADER 3K CORE
3+3K MAX 1222 WORDS FREE

CHAPTER 10
ACCUMULATOR CONVENTIONS FOR MAIN PROGRAMS AND SUBPROGRAMS

10.1 LOCATIONS

Locations specified in the calling sequence for a FORTRAN subprogram may be either required locations or defined locations. A required location is a memory location whose address is specified in the calling sequence for a subprogram. For example, X is a required location in the calling sequence

```
JSA 16, SQRT
ARG     X
```

A defined location is a memory location whose address is specified in the definition of a calling sequence. The location does not appear in the calling sequence. For example in the calling sequence

```
MOVEI 16, MEMORY
PUSHJ 17, DFAS.0
```

MEMORY is required, and AC0, AC1, and AC2 are defined by DFAS.0.

10.2 ACCUMULATORS

10.2.1 Accumulators 0 and 1

When used for subprograms called by JSA, accumulators 0 and 1 may be used at any time without restoring their original contents. These accumulators cannot be required locations. A FORTRAN function returns its answer in accumulator 0 (if a single word result) or in accumulators 0 and 1 (if a double-precision or complex result). A function may also return its answer in specified locations (given by argument names in the CALL) or variable names; in any event, an answer must be returned either in accumulator 0 or in accumulators 0 and 1.

When used for subprograms called by PUSHJ 17, adr, accumulators 0 and 1 may have their contents destroyed. Some subprograms by their definition return an argument in accumulator 0 or 1.

10.2.2 Accumulators 2 Through 15

Accumulators 2 through 15 must not be destroyed by FORTRAN functions, but may be destroyed by FORTRAN subroutines. (Presently subroutines must preserve the contents of accumulator 15.) The contents of these accumulators must not be destroyed by subprograms called by PUSHJ unless the definition of the subroutines requires it.

10.2.3 Accumulators 16 and 17

Accumulator 16 should be used only for JSA-JRA subprogram calls unless the definition of the subprogram sequence requires otherwise. The contents of accumulator 16 may be destroyed by subprograms called by PUSHJ 17, adr.

Accumulator 17 must be used only for pushdown list operations.

10.3 UUOS

User UUO's are not considered subprograms and may not change any locations except those required for input and the contents of accumulators 0 or 1.

10.4 SUBPROGRAMS CALLED BY JSA 16, ADDRESS

The calling sequence is

```
JSA 16, address
ARG     adr1
ARG     adr2
        :
        :
ARG     adrN
```

where each ARG adrN corresponds to one argument of the subprogram.

There may or may not be arguments. If there are arguments, they must be in accumulators 2 through 15. Subroutines called with the FORTRAN CALL statement may, by definition, return an argument in accumulator 0 or 1. Subprograms that are FORTRAN functions (such as SIN or SQRT) may destroy the contents of accumulators 0 and 1. Results are returned in accumulator 0 for single word results and accumulators 0 and 1 for double word results.

10.5 SUBPROGRAMS CALLED BY PUSHJ 17, ADDRESS

See section 10.2. In addition, three consecutive accumulators are required for double-precision addition, subtraction, multiplication, and division operations. The contents of the third accumulator may be destroyed. The

"to memory" modes also leave the answer in the defined accumulators. The two arguments of the double-precision operation cannot be in the same accumulators. Complex addition, subtraction, multiplication, and division operations do not destroy locations except those required for the answer and accumulator 16. The two arguments of the complex operation must not be in the same accumulator.

10.6 SUBPROGRAMS CALLED BY UUOS

Subprograms called by UUO's may change the contents of accumulators 0 and 1 only.

Table 10-1
Accumulator Conventions for
PDP-10 FORTRAN IV Compiler and Subprograms

Subprogram Called By: Accumulators	JSA		PUSHJ	UUO
	Functions	Subroutines		
0, 1	1) May be destroyed. 2) May not be used to pass arguments. 3) A result must be returned in 0 or 0 and 1.	1) May be destroyed. 2) May not be used to pass arguments. 3) Results must not be returned.	1) May be destroyed. 2) May be used to pass arguments if the subprogram is defined with an argument in 0 or 0 and 1. 3) Results may be returned if the subprogram is so defined.	1) May be destroyed. 2) May be used to pass arguments except as defined. 3) Results must not be returned.
2-15	1) Must be preserved. 2) Arguments may be passed. 3) Results may be returned if required by calling sequence.	1) May be destroyed. 2) Arguments may be passed. 3) Results must not be returned.	1) Must be preserved unless the definition of subprogram forces results to be returned. 2) Arguments may be passed. 3) Results may be returned if the subprogram is so defined.	1) Must be preserved. 2) Arguments may be passed. 3) Results must not be returned.
16 Reserved for JSA-JRA Operations (except as noted for PUSHJ)	1) Must be preserved. 2) May not be used to pass arguments. 3) Results must not be returned.	1) Must be preserved. 2) May not be used to pass arguments. 3) Results must not be returned.	1) Is destroyed. 2) Used for argument address. 3) Results must not be returned.	1) Must be preserved. 2) May not be used to pass arguments. 3) Results must not be returned.
17 Reserved for Pushdown List Operations	1) Must be preserved. 2) May not be used to pass arguments. 3) Results must not be returned.	1) Must be preserved. 2) May not be used to pass arguments. 3) Results must not be returned.	1) Must be preserved. 2) May not be used to pass arguments. 3) Results must not be returned.	1) Must be preserved. 2) May not be used to pass arguments. 3) Results must not be returned.

CHAPTER 11
SWITCHES AND DIAGNOSTICS

11.1 FORTRAN SWITCHES AND DIAGNOSTICS

Table 11-1
FORTRAN Compiler Switch Options

Switch	Meaning
A†	Advance magnetic tape reel by one file.
B†	Backspace magnetic tape reel by one file.
C†	Generate a CREF-type cross-reference listing. (DSK:CREF.TMP assumed if no list-dev specified)
	Complement: Do not produce cross-reference information (standard procedure).
E	Print an octal listing of the binary program produced by the compiler in addition to the symbolic listing output.
	Complement: Do not produce octal listing (standard procedure).
I	Translate the letter D in column 1 as a space and treat the line as a normal FORTRAN statement.
	Complement: Translate the letter D in column 1 as a comment character and treat the line as a comment (standard procedure).
M	Include MACRO coding in the output listing.
	Complement: Eliminate the MACRO coding from the output listing (standard procedure).
N	Suppress output of error messages on the Teletype.
	Complement: Output error messages on TTY (standard procedure).
S	If the compiler is running on the KA-10, produce code for execution on the KI-10 and vice-versa.
T†	Skip to the logical end of the magnetic tape reel.
W†	Rewind the magnetic tape reel.
Z†	Zero the DECtape directory.

†Switches A through C and T, W, and Z must immediately follow the device name or filename.ext to which the individual switch applies.

Table 11-2
FORTRAN Compiler Diagnostics
(Command Errors)

Message	Meaning
?BINARY OUTPUT ERROR dev:filename.ext	An output error has occurred on the device specified for the binary program output.
?CANNOT FIND dev:filename.ext	Filename.ext cannot be found on this device.
?DEVICE INPUT ERROR for command string	Device error occurred while attempting to read Monitor command file.
IMPROPER IO FOR DEVICE dev:	An input device is specified for output (or vice versa) or an illegal data mode was specified (e.g., binary output to TTY).
ILLEGAL MEMORY REFERENCE AT loc COMPILATION TERMINATED	An illegal memory reference has occurred and compilation has stopped. The current output files will be closed and the next source files read.
?INPUT DATA ERROR dev:filename.ext	A read error has occurred on the source device.
?x IS A BAD SWITCH	This specified switch is not recognizable.
?x IS AN ILLEGAL CHARACTER	A character in a command string typein is not recognizable (e.g., FORM-FEED).
?dev: IS NOT AVAILABLE	Either the device does not exist or it has been assigned to another job.
LINKAGE ERROR	Input device error while doing Dump Mode I/O, or not enough core was available to execute the newly loaded program.
?LINKAGE ERROR FOR dev:filename	Specified dev:filename appears in a ! Monitor command string, but cannot be run for some reason.
?LISTING OUTPUT ERROR	An output error has occurred on the device specified for the listing output.
?NO ROOM FOR filename.ext	The directory on dev: DTAn is full and cannot accept filename.ext as a new file, or a protection failure occurred for a DSK output file.
?NO FILE NAMED filename.ext	An illegal filename has been used.
?NOT ENOUGH CORE FOR LINKAGE	Not enough core available to load (with dump mode I/O) the program specified in a ! Monitor command string.
?SYNTAX ERROR IN COMMAND STRING	A syntax error has been detected in a command string typein (e.g., the ← has been omitted).
?X SWITCH ILLEGAL AFTER LEFT ARROW	Cannot change machine type with a file or clear source directory.
?X SWITCH ILLEGAL AFTER FIRST STANDARD FILE	Cannot clear directory after start of compilation (Batch Mode).
?X SWITCH, NO LISTING FILE	A CREF listing requires a listing file.
?INSUFFICIENT CORE - COMPILATION TERMINATED	The compiler has insufficient table space to compile the program.

Table 11-2 (Cont)
FORTRAN Compiler Diagnostics
(Command Errors)

Message	Meaning
WORK STACK OVERFLOW AT loc COMPILATION TERMINATED	The pushdown list used by the compiler for machine language subroutine calls has overflowed. Compilation has stopped. The current output files will be closed and the next source file read.

Table 11-3
FORTRAN Compiler Diagnostics
(Compilation Errors)

	Message	Meaning
I-1	DUPLICATED DUMMY VARIABLE IN ARGUMENT STRING	A dummy variable (identifier) may appear only once in any one argument set representing the arguments of a subprogram. (See Section 7.3)
I-2	ARRAY NAME ALREADY IN USE	Any attempt to re-dimension a variable or redefine a scalar as an array is illegal. (See Section 6.1.1)
I-3	ATTEMPT TO REDEFINE VARIABLE TYPE	Once a variable has been defined as either complex, double precision, integer, logical, or real it may not be defined again. (See Section 2.2, 6.3)
I-4	NOT A VARIABLE FORMAT ARRAY	The variable which contains the FORMAT specification read-in at object time must be a dimensioned variable, i.e., an array (see Section 5.1.1) or a subprogram argument was used as a NAMELIST name with the subprogram (see Section 5.1.2).
I-5	NAME ALREADY USED AS NAMELIST NAME	After a NAMELIST name has been defined, it may appear only in READ or WRITE statements and may not be defined again. (See Section 5.1.2)
I-6	DUPLICATED NAMELIST NAME	A NAMELIST name has already been used as a scalar array or global dummy argument. (See Section 5.1.2)
I-7	A NAME APPEARS TWICE IN AN EXTERNAL STATEMENT	A subprogram name has been declared EXTERNAL more than once. (See Section 7.7)
I-8	ARGUMENT TYPE DOESN'T AGREE WITH FUNCTION SPEC	The actual arguments for a function do not agree in type with the dummy arguments in the specification of the function.
I-9	THIS FUNCTION REQUIRES MORE ARGUMENTS	Not enough arguments were supplied for a function.
I-10	SUBPROGRAM NAME ALREADY IN USE	A subprogram name has appeared in another statement as a scalar or array variable, arithmetic function statement name, or COMMON block name. (See Section 7.5)
I-11	DUMMY ARGUMENT IN DATA STATEMENT	Dummy arguments may not appear in DATA statements. (See Section 6.2.1)

Table 11-3 (Cont)
FORTRAN Compiler Diagnostics
(Compilation Errors)

Message		Meaning
I-12	NOT A SCALAR OR ARRAY	The variable defining the starting address for an ENCODE/DECODE statement must be a scalar or an array. (See Section 5.4) The I/O unit name of a READ/WRITE statement is not a scalar or array. (See Sections 5.2.6, 5.2.7) An attempt to ASSIGN a label number to a variable that is not a scalar or array. (See Section 2.2) An attempt to GO TO through a variable that is not a scalar or array. (See Section 4.1)
I-13	ILLEGAL USE OF DUMMY ARGUMENT	Dummy arguments may be used with functions or subprograms only. (See Sections 7.4.1, 7.5.1)
I-14	ILLEGAL DO LOOP PARAMETER	The DO index must be a non-subscripted integer variable while the initial, limit, and increment values of the index must be an integer expression - the index may not be zero. (See Section 4.3)
I-15	I/O VARIABLES MUST BE SCALARS OR ARRAYS	Referencing data in an I/O statement other than scalars or arrays is illegal. (See Section 5.2)
I-16	A CONFLICT EXISTS WITH A COMMON DECLARATION	The function name used was previously declared a scalar variable in a COMMON statement.
S-1	ILLEGAL NAME OR DELIMITER OR KEY CHARACTER	A variable name doesn't start with an alphabetic character, or a delimiter such as the left parenthesis that begins a format is missing, or a key character such as the letter D in BLOCK DATA is missing.
S-2	STATEMENT KEYWORD NOT RECOGNIZED	A statement keyword such as ERASE was not recognized, possibly due to misspelling (e.g., ERASC 16).
S-3	ILLEGAL FIELD SPECIFICATION	The field width or decimal specification in a FORMAT statement must be integer. The number of Hollerith characters in an H specification must be equal to the number specified. (See Sections 5.1.1.1, 5.1.1.6)
S-4	SCALAR VARIABLE - MAY NOT BE SUBSCRIPTED	An undimensioned variable (a scalar variable) is being illegally subscripted (see Section 2.2.1) or a scalar variable is subscripted in an ENCODE/DECODE statement (see Section 5.4).
S-5	ILLEGAL TYPE SPECIFICATION	The type of constant specified is illegal or misspelled. (See Section 2.1)
S-6	ARGUMENT IS NOT SINGLE LETTER	Arguments in parentheses must be single letters in IMPLICIT statement. (See Section 6.3.1)
S-7	'NAMELIST' NOT FOLLOWED BY "/"	The first character following NAMELIST must be /. (See Section 5.1.2)
S-8	ILLEGAL CHARACTER IN LABEL	A non-numeric character was detected in the label field of the statement, possibly because tabs or spaces are missing.

Table 11-3 (Cont)
FORTRAN Compiler Diagnostics
(Compilation Errors)

Message	Meaning
S-9 MISSING COMMA OR SLASH IN SPECIFICATION STATEMENT	A specification statement (see Section 7.8) requires a comma or slash and it is missing.
S-10 ILLEGAL ARITHMETIC "IF" - TOO MANY LABELS	An arithmetic "IF" statement must have no more or less than three statement labels to transfer to. Special optimization will occur if two of the labels are the same, or one or more labels refer to the next statement.
S-11 A NUMBER WAS EXPECTED	Only arrays which are subprogram arguments can have adjustable dimensions. (See Section 6.1.1.1)
S-12 IMPLICIT TYPE RANGE OVERLAPS PREVIOUS SPECIFICATION	An implicit type range encompasses a character that has already been given an implicit type.
S-13 ATTEMPT TO USE AN ARRAY OR FUNCTION NAME AS A SCALAR	Variables may be either scalar or array but not both. Variables appearing in a DIMENSION statement must be subscripted when used. (See Section 2.2) Function names must be followed by at least one argument enclosed in parentheses (See Section 7.4).
S-14 ARRAY NOT SUBSCRIPTED	See S-13
S-15 ILLEGAL USE OF AN ARITHMETIC FUNCTION NAME	Arithmetic function definition statement name is being used without arguments (i.e., as a scalar) in an arithmetic expression. (See Section 7.3)
S-16 MULTIPLE RETURN ILLEGAL WITHOUT STATEMENT LABEL ARG	A dollar sign ($) or an asterisk (*) must have appeared in the argument list of this subprogram to represent the position of a statement label argument in the call.
S-17 INCORRECT PAREN COUNT OR MISSING IMPLIED DO INDEX	The number of left and right parentheses does not match, or an undefined index variable was used in defining a DO loop (see Section 5.2.1), or the number of implied DO loops and the number of matching parentheses differ in a DATA statement. (See Section 6.2.1)
S-18 INVALID INDEX IN DO-LOOP OR IMPLIED DO-LOOP	The index of a DO statement must be a non-subscripted integer variable and must not be zero. (See Section 4.3) The index is not used as a subscript in a DATA list. (See Section 6.2.1)
S-19 EQUIVALENCE REQUIRES TWO OR MORE ELEMENTS	The EQUIVALENCE statement must have more than one argument because it causes variables to share the same location. (See Section 6.1.3)
S-20 ILLEGAL DEFINITION OF AN ARITHMETIC STATEMENT FUNCTION	The statement function continues past its recognized end point.
S-21 MISSING COMMA IN INPUT/OUTPUT LIST	An input/output list continues past its recognized end point.
S-22 STATEMENT CONTINUES PAST RECOGNIZED END POINT	A statement other than those mentioned above continued past its recognized end point.
S-23 ILLEGAL COMPLEX CONSTANT	The parentheses of the complex constant enclose a logical, Hollerith, or complex constant.

Table 11-3 (Cont)
FORTRAN Compiler Diagnostics
(Compilation Errors)

Message		Meaning
O-1	BLOCK DATA NOT SEPARATE PROGRAM	Block Data must exist as a separate program. (See Sections 6.2.2, 7.6)
O-2	SUBROUTINE IS NOT A SEPARATE PROGRAM	A subroutine following a main program or another subroutine subprogram may have no statement between it and the preceding programs END statement and must begin with a SUBROUTINE statement. The previous program must have been terminated properly. (See Section 7.5)
O-3	STATEMENT OUT OF PLACE	The IMPLICIT specification statement and any arithmetic function definition statement must appear before any executable statement. (See Chapter 6)
O-4	EXECUTABLE STATEMENTS ILLEGAL IN BLOCK DATA	Block DATA statements cannot contain executable statements.
A-1	MINIMUM VALUE EXCEEDS MAXIMUM VALUE	Minimum value of an array exceeds the maximum value specified. (See Section 6.1.1)
A-2	ATTEMPT TO ENTER A VARIABLE INTO COMMON TWICE	A variable name may appear in COMMON statement only once. (See Section 6.1.2)
A-3	ATTEMPT TO EQUIVALENCE A SUBPROGRAM NAME OR DUMMY ARGUMENT	An identifier defined as a subprogram name cannot appear in EQUIVALENCE statements in the defining program. Dummy argument identifiers of a subprogram may not appear in EQUIVALENCE statements in that subprogram. (See Sections 6.1.3, 7.1)
A-4	NOT A CONSTANT OR DUMMY ARGUMENT	Only constant and dummy arguments may be used as arguments in dimension statements. (See Section 7.4.1)
A-5	CAUTION ** COMMON VARIABLE PASSED AS ARGUMENT	The variable may be multiply defined in the called subprogram. (See Sections 7.4.1, 7.5.1)
M-1	TOO MANY SUBSCRIPTS	An array variable appears with more subscripts than specified. (See Sections 2.2.2, 6.1.1)
M-2	WRONG NUMBER OF SUBSCRIPTS	An array variable appears with too few subscripts. (See Sections 2.2.2, 6.1.1)
M-3	CONSTANT OVERFLOW	Too many significant digits in the formation of a constant or the exponent is too large. (See Section 2.1)
M-4	ILLEGAL 'IF' ARGUMENT	Logical IF or DO statement adjacent to a logical IF statement, or illegal expression within a logical IF statement. (See Sections 4.2.2, 4.3)
M-5	ILLEGAL CONVERSION IMPLIED	Attempt to mix double precision and complex data in the same expression. (See Section 2.3.1)
M-6	LABEL OUT OF RANGE OR ARRAY TOO LARGE	Illegal statement label (See Section 1.1.1) or array size is greater than $2^{18}-1$.
M-7	UNTERMINATED HOLLERITH STRING	A missing single quote or fewer than n characters following an "nH" specification. (See Section 5.1.1.6)

Table 11-3 (Cont)
FORTRAN Compiler Diagnostics
(Compilation Errors)

Message	Meaning
M-8 SYSTEM ERROR - NO MORE SPACE FOR RECURSIVE STORAGE	The compiler's work roll is too small to hold the parts of all the subexpressions this statement implies. Break this statement or reassemble the compiler with a larger work-roll parameter (WORLEN=150_8 at present).
M-9 TOO MUCH DATA - WRONG ARRAY SIZE OR LITERAL TOO LONG	The list of DATA constants defines more words than the list of DATA variables specifies. This may be due to an array of the wrong size in the list of DATA variables, or definition of an integer, real, or logical DATA variable with a Hollerith constant of more than five characters.
M-10 ILLEGAL DO LOOP CLOSE	Illegal statement terminating a DO loop. (See Section 4.3)
M-11 MORE DATA NEEDED - LITERAL TOO SHORT OR TYPE CONVERSION EXPECTED	The list of DATA constants defines fewer words than the list of DATA variables specifies. This may be due to a double precision or complex DATA variable defined with a Hollerith constant of less than six characters, or a double precision DATA variable defined with a real constant.
M-12 NON-INTEGER PARAMETER IN 'DO' STATEMENT	DO statement parameters must be integers. (See Section 4.3)
M-13 NON-INTEGER SUBSCRIPT	Array subscripts must be integer constants, variables, or expressions. (See Section 4.3)
M-14 ILLEGAL COMPARISON OF COMPLEX VARIABLES	The only comparison allowed of complex variables is .NE. or .EQ. (See Sections 2.2, 2.3)
M-15 TOO MANY CONTINUATION CARDS	More than 19 continuation cards. (See Section 1.1.2)
M-16 NON-INTEGER I/O UNIT OR CHARACTER COUNT	The I/O unit variable of a READ/WRITE statement, or the character count variable of an ENCODE/DECODE statement, is not an integer variable. (See Sections 5.2.6, 5.2.7, 5.4)
M-17 SYSTEM ERROR-ROLL OUT OF RANGE	Compiler error. Report this message and its circumstances via a Software Trouble Report.
M-18 SYSTEM ERROR - NO MORE SPACE FOR RECURSIVE CALLS	The compiler's exit roll is too small to hold the return addresses for all the recursive subroutine calls this statement requires to be compiled. Break up the statement or reassemble the compiler with a larger exit roll parameter (EXLEN1=201_8 at present).
M-19 ILLEGAL USE OF STATEMENT LABEL	A GO TO or IF statement transfers to itself.
M-20 ILLEGAL RECURSIVE CALL	The statement function called itself. Recursive calls are illegal in the FORTRAN language.
EXCESSIVE COUNT	The number specified is greater than the maximum possible number of characters in a statement.
OPEN DO LOOPS	The list of statements are specified in DO statements but not defined.
UNDEFINED LABELS	The list of labels that do not appear in the label field.

Table 11-3 (Cont)
FORTRAN Compiler Diagnostics
(Compilation Errors)

Message	Meaning
MULTIPLY DEFINED LABELS	The list of labels that appeared more than once in the label field.
ALLOCATION ERRORS	The list of EQUIVALENCEd COMMON variables which have attempted to extend the beginning of a COMMON block.

Table 11-4
FORTRAN Operating System Diagnostics
(Execution Errors)

Message	Meaning
?BLOCK TOO LARGE OR QUOTA EXCEEDED ON dev	The user's program attempted to add blocks to a random access file, which caused the block to be too large or caused him to exceed his disk quota.
?CANNOT ACCESS FORTR.SHR- GETSEG ERROR CODE xx	An error occurred when a GETSEG UUO was issued to access FORTR.SHR. The codes are listed in Appendix E of the TOPS-10 Monitor Calls manual.
?dev: NOT AVAILABLE	FORSE. tried to initialize a device which either does not exist or has been assigned to another job.
?DEVICE NUMBER n IS ILLEGAL	A nonexistent device number was selected.
?DEVICE NUMBER n MUST BE DSK FOR RANDOM ACCESS	The device for random access operations must be disk.
?DIRECT ACCESS DEVICE NUMBER n IS ILLEGAL	Only devices 1 through 17 can be used for random access.
ENCODE - DECODE ERROR	The character count in an ENCODE or DECODE statement was incorrect.
END OF FILE ON dev:	A premature end-of-file has occurred on an input device.
?END OF TAPE ON dev:	The end of tape marker has been sensed during input or output
?FILE NAME filename.ext NOT ON DEVICE dev:	Filename.ext cannot be found in the directory of the specified device.
?ILLEGAL CHARACTER, x, IN FORMAT	The illegal character x is not valid for a FORMAT statement.
?ILLEGAL CHARACTER, x, IN INPUT STRING	The illegal character x is not valid for this type of input.
?ILLEGAL MAGNETIC TAPE OPERATION, TAPE dev:	An attempt was made to skip a record after performing output on a magnetic tape.
?ILLEGAL PHYSICAL RECORD COUNT, TAPE dev:	FORSE. has encountered an inconsistency in the physical record count on a magnetic tape.
?ILLEGAL USER UUO uuu AT USER loc	An illegal user UUO to FORSE. was encountered at location loc.
?INPUT DEVICE ERROR ON dev:	A data transmission error has been detected in the input from a device.

Table 11-4 (Cont)
FORTRAN Operating System Diagnostics
(Execution Errors)

Message	Meaning
?LIBRARY (FORTR.SHR) AND USER PROGRAM VERSION NUMBERS ARE DIFFERENT	The user's executable program is using an obsolete version of the library. The program should be recompiled so that the correct version of the library is used.
?MORE THAN 15 DEVICES REQUESTED	Too many devices have been requested.
?NAMELIST SYNTAX ERROR	Improper mode of I/O (octal or Hollerith), incorrect variable name.
?NO ROOM FOR FILE filename.ext ON DEVICE dev:	There is no room for the file in the directory of the named device.
?NOT ENOUGH CORE FOR BUFFERS	Either a call to BUFFER or a random access operation tried to set up a buffer ring when not enough core was available.
program name NOT LOADED	A dummy routine was loaded instead of the real one. Generally, this error occurs when a loaded program is patched to include a call to a library program which was not called by the original program at load time.
?OUTPUT DEVICE ERROR ON dev:	A data transmission error has been detected during output to a device.
?OUTPUT FIELD WIDTH OVERFLOW	A field overflowed on output and was filled with asterisks.
?PARITY ERROR ON dev:	A parity error has been detected.
?REREAD EXECUTED BEFORE FIRST READ	A reread was attempted before initializing the first input device.
?TAPE RECORD TOO SHORT ON UNIT n	The data list is too long on a binary tape READ operation.
?dev: WRITE PROTECTED	The device is WRITE locked.
WARNING! / IS ILLEGAL IN ENCODE-DECODE, END OF FORMAT ASSUMED	A slash was used in the FORMAT statement referenced by an ENCODE or DECODE statement. Since slashes are illegal in these statements, the operating system assumes that the slash is the end of the format.

The following messages are typed twice, when the error occurs, and in a final summary. When the error occurs, the PC value is appended to the message. When the message appears in the final summary, the number of times that the error occurred in the program is appended to the message.

?ACOS OF ARG > 1.0 IN MAGNITUDE
?ASIN OF ARG > 1.0 IN MAGNITUDE
?ATTEMPT TO TAKE LOG OF NEGATIVE ARG
?ATTEMPT TO TAKE SQRT OF NEGATIVE ARG
?FLOATING DIVIDE CHECK
?FLOATING OVERFLOW
?FLOATING UNDERFLOW
?INTEGER DIVIDE CHECK
?INTEGER OVERFLOW

Table 11-4 (Cont.)
FORTRAN Operating System Diagnostics
(Execution Errors)

Message	Meaning
The following messages are issued when a LOOKUP, ENTER, or RENAME UUO error occurs. The number in parentheses indicates the error code. Refer to Appendix E in the TOPS-10 Monitor Calls manual.	
?(0) ILLEGAL FILENAME WAS NOT FOUND FILE xx ON DEVICE yy	
?(1) NO DIRECTORY FOR PROJECT -PROGRAMMER NUMBER FILE xx ON DEVICE yy	
?(2) PROTECTION FAILURE FILE xx ON DEVICE yy	
?(3) FILE WAS BEING MODIFIED FILE xx ON DEVICE yy	
?(4) RENAME FILE NAME ALREADY EXISTS FILE xx ON DEVICE yy	
?(5) ILLEGAL SEQUENCE OF UUOS FILE xx ON DEVICE yy	
?(6) BAD UFD OR BAD RIB FILE xx ON DEVICE yy	
?(7) NOT A SAV FILE FILE xx ON DEVICE yy	
?(10) NOT ENOUGH CORE FILE xx ON DEVICE yy	
?(11) DEVICE NOT AVAILABLE FILE xx ON DEVICE yy	
?(12) NO SUCH DEVICE FILE xx ON DEVICE yy	
?(13) NOT TWO RELOC REG. CAPABILITY FILE xx ON DEVICE yy	
?(14) NO ROOM OR QUOTA EXCEEDED FILE xx ON DEVICE yy	
?(15) WRITE LOCK ERROR FILE xx ON DEVICE yy	
?(16) NOT ENOUGH MONITOR SPACE FILE xx ON DEVICE yy	
?(17) PARTIAL ALLOCATION ONLY FILE xx ON DEVICE yy	
?(20) BLOCK NOT FREE ON ALLOCATION FILE xx ON DEVICE yy	
NOTE With the exception of the messages ILLEGAL USER UUO uuu AT USER loc and ENCODE/DECODE ERROR, all messages are followed by a second message LAST FORTRAN I/O AT USER LOC adr	

Several arithmetic error conditions can occur during execution time.

a. Overflow - An attempt was made to create either a positive number greater than the largest representable positive number or a negative number greater in magnitude than the most negative representable number (in the appropriate mode).

Example: For I an integer,

$377777777777 < I < 400000000000$ (octal)

b. Underflow - An attempt was made to create either a positive non-zero number smaller than the smallest representable positive non-zero number or a negative number smaller in magnitude than the negative number whose magnitude is the smallest representable.

Example: For X a real non-zero number,

777400000000 < X < 000400000000

c. Divide Check - An attempt was made to divide by zero.

d. Improper Arguments for LIB40 math routines - For example, an attempt was made to find the arc sine of an argument greater than 1.0.

When overflow, underflow, or divide check errors occur in the user's FORTRAN program, the Monitor calls the LIB40 routine OVTRAP. This routine replaces the resulting numbers, if the numbers are floating point, with either zero in the case of underflow or ± the largest representable number in the cases of overflow and divide check. OVTRAP does not affect numbers in integer mode.

Overflow, underflow, and divide check errors occurring in LIB40 math routines are handled differently from when they occur in the user's program: only if the final answer from a routine is in error is an error condition considered to exist. If the answer is floating point, it is set to the appropriate value as for user program errors. Integer answers are handled in various ways. (See the Science Library and FORTRAN Utility Subprograms, DEC-10-SFLE-D.)

When an error condition occurs in a user program or in a final answer from a LIB40 math routine, an error message is typed. Presently there are eight distinct error messages.

Error Message No.	Error Message
1	INTEGER OVERFLOW PC=nnnnnn
2	INTEGER DIVIDE CHECK PC=nnnnnn
3	FLOATING OVERFLOW PC=nnnnnn
4	FLOATING UNDERFLOW PC=nnnnnn
5	FLOATING DIVIDE CHECK PC=nnnnnn
6	ATTEMPT TO TAKE SQRT OF NEGATIVE ARG
7	ACOS OF ARG > 1.0 IN MAGNITUDE
8	ASIN OF ARG > 1.0 IN MAGNITUDE
final	1 ? FATAL I/O ERROR

NOTE

nnnnnn = location at which the error occurred.

After two typeouts of a particular error message, further typeout of that error message is suppressed. At the end of execution, a summary listing the actual number of times each error message occurred is typed out. If the user wishes to permit more than two typeouts for each error message, he may do so by calling the routine ERRSET at the beginning of the executable part of his main program. ERRSET accepts one argument in integer mode. This argument is the number of typeouts that are permitted for each error message before suppression occurs. This routine is used to obtain the PC information which would otherwise be lost. Alternatively, because of the slow-

ness of the Teletype output, the user may wish to suppress typeout of the messages entirely. This can be done by calling ERRSET with an argument of zero. Suppression of typeout can also be accomplished during execution by typing ↑O on the Teletype.

Error messages and the summary are output to the Teletype (or the output device when running BATCH), regardless of the device assignments that have been made.

The treatment of overflow, underflow, and divide check errors in MACRO programs (those that are loaded with OVTRAP) can, to a certain extent, be manipulated by the user. (See OVTRAP in the Science Library and FORTRAN Utility Subprogram manual.)

CHAPTER 12

FORTRAN USER PROGRAMMING

12.1 ASCII CHARACTER SET

Table 12-1
ASCII Character Set

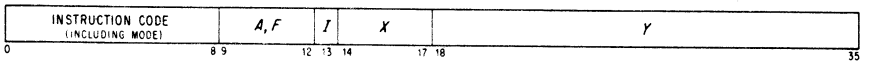
SIXBIT	Character	ASCII 7-Bit†	SIXBIT	Character	ASCII 7-Bit†	Character	ASCII 7-Bit†
00	Space	040	40	@	100	`	140
01	!	041	41	A	101	a	141
02	"	042	42	B	102	b	142
03	#	043	43	C	103	c	143
04	$	044	44	D	104	d	144
05	%	045	45	E	105	e	145
06	&	046	46	F	106	f	146
07	'	047	47	G	107	g	147
10	(	050	50	H	110	h	150
11	)	051	51	I	111	i	151
12	*	052	52	J	112	j	152
13	+	053	53	K	113	k	153
14	,	054	54	L	114	l	154
15	-	055	55	M	115	m	155
16	.	056	56	N	116	n	156
17	/	057	57	O	117	o	157
20	0	060	60	P	120	p	160
21	1	061	61	Q	121	q	161
22	2	062	62	R	122	r	162
23	3	063	63	S	123	s	163
24	4	064	64	T	124	t	164
25	5	065	65	U	125	u	165
26	6	066	66	V	126	v	166
27	7	067	67	W	127	w	167
30	8	070	70	X	130	x	170
31	9	071	71	Y	131	y	171
32	:	072	72	Z	132	z	172
33	;	073	73	[	133	{	173
34	<	074	74	\	134	|	174
35	=	075	75	]	135	}	175
36	>	076	76	↑	136	~	176
37	?	077	77	←	137	Delete	177

†FORTRAN IV also accepts the following control codes in 7-bit ASCII:

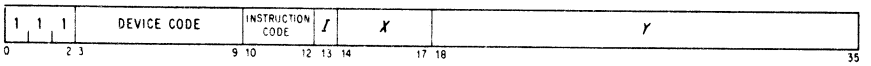
Horizontal Tab	011	Carriage Return	015
Line Feed	012	Form Feed	014

12.2 PDP-10 WORD FORMATS

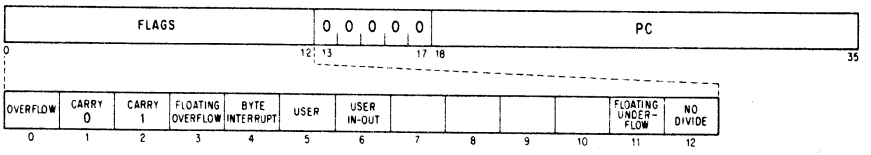
BASIC INSTRUCTIONS



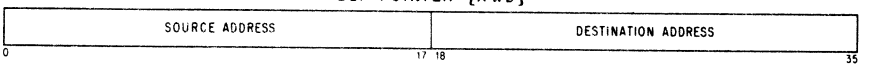
IN-OUT INSTRUCTIONS



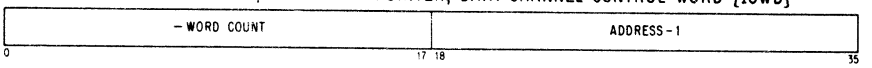
PC WORD



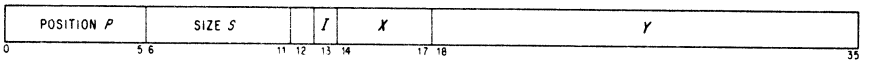
BLT POINTER {XWD}



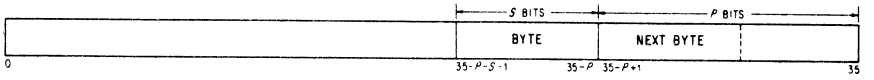
BLKI / BLKO POINTER, PUSHDOWN POINTER, DATA CHANNEL CONTROL WORD {IOWD}



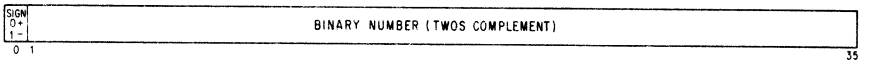
BYTE POINTER



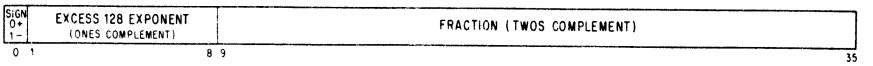
BYTE STORAGE



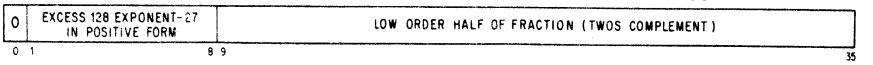
FIXED POINT OPERANDS



FLOATING POINT OPERANDS



LOW ORDER WORD IN DOUBLE LENGTH FLOATING POINT OPERANDS



12.3 FORTRAN INPUT/OUTPUT

In addition to the arithmetic functions, the PDP-10 FORTRAN IV library (LIB40) contains several subprograms which control FORTRAN IV I/O operations at runtime. The I/O subprograms are compatible with the PDP-10 Monitors.

In general FORTRAN IV I/O is done with double buffering unless the user has either specified otherwise through calls to IBUFF and OBUFF or is doing random access I/O to the disk. In these cases, single buffers are used. The standard buffer sizes for the devices normally available to the user are given in Table 12-2. Note that the devices and buffer sizes are determined by the Monitor and may be changed by a particular installation. Also a user may specify buffer sizes for magtape operations through the use of IBUFF and OBUFF.

The logically first device in a FORTRAN program is initialized on software I/O channel one, the second on software I/O channel two, and so forth. Software I/O channel 0 is reserved for error message and summary output. The SIXBIT name of the device that is initialized on channel N can be found in a dynamic device table at location DYNDV. + N. A device may be initialized for input and output on the same I/O channel. Devices are initialized only once and are released through either the CALL [SIXBIT/EXIT/] executed at the end of every FORTRAN program or the LIB40 subroutine RELEAS.

Table 12-2
PDP-10 FORTRAN IV Standard Peripheral Devices

Name	Mnemonic	Input/Output		Buffer Size In Words	Operation
		Formatted	Unformatted		
Card Punch	CDP	Yes	Yes	26	WRITE
Card Reader	CDR	Yes	Yes	28	READ
Disk (includes disk packs and drums)	DSK	Yes	Yes	128	READ/WRITE
DECtapes	DTA	Yes	Yes	127	READ/WRITE
Line Printer	LPT	Yes	No	26	WRITE
Magtape	MTA	Yes	Yes	128	READ/WRITE
Plotter	PLT	Yes	Yes	36	WRITE
Paper Tape Punch	PTP	Yes	Yes	33	WRITE
Paper Tape Reader	PTR	Yes	Yes	33	READ
Pseudo Teletype	PTY	Yes	No	17	READ/WRITE
Teletype - User	TTY	Yes	No	17	READ/WRITE
Teletype - Console	CTY	Yes	No	17	READ/WRITE

12.3.1 Logical and Physical Peripheral Device Assignments

Logical and physical device assignments are controlled by either the user at runtime or a table called DEVTB. The first entry in DEVTB. is the length of the table. Each entry after the first is a sixbit ASCII device name. The position in the table of the device name corresponds to the FORTRAN logical number for that device. For example, in Table 12-3, magnetic tape 0 is the 16th entry in DEVTB. Therefore, the statement

```
WRITE (16, 13)A
```

refers to magnetic tape 0. The last five entries in DEVTB. correspond to the special FORTRAN statements READ, ACCEPT, PRINT, PUNCH, and TYPE. Any device assignments may be changed by reassembling DEVTB.

If the user gives the Monitor command

```
ASSIGN DSK 16
```

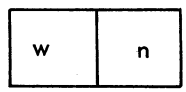
prior to the running of his program, a file named FOR16.DAT would be written on the disk. Similarly, the Monitor command

```
ASSIGN LPT 16
```

causes output to go to the line printer.

12.3.2 DECtape and Disk Usage

12.3.2.1 Binary Mode - In binary mode, each block contains 127 data words, the first of which is a record control word of the form:



where w is the word count specifying the number of FORTRAN data words in the block (126 for a full block) and n is 0 in all but the last block of a logical record, in which case n is the number of blocks in the logical record. A logical record contains all the data corresponding to one READ or WRITE statement, that is, the maximum number of logical records per disk/DECtape block is one.

12.3.2.2 ASCII Mode - In ASCII mode, blocks are packed with as many full lines (a line is a unit record as specified by a format statement) as possible. Lines always begin with a new word. If a line terminates in the middle of a word, the word is filled out with null characters and the next line begins with the next word. Lines are not split across blocks. Such a file is created by FORTRAN during output or by PIP with the A switch. FORTRAN input files must be in this format.

Table 12-3
Device Table for FORTRAN IV

```
TITLE    DEVTB  V.017
SUBTTL   1-APR-69

ENTRY    DEVTB.,DEVND.,DEVLS.,DVTOT.
ENTRY    MTABF.,MBFBG.,TABPT.,TABP1.
ENTRY    MTACL.,DATTB.,NEG1.,NEG2.,NEG3.,NEG5.
P=17

DEVTB.:  EXP      DEVND.-.        ;NO. OF ENTRIES
                                  ;LOGICAL#/FILENAME/DEVICE
         SIXBIT   .DSK.           ; 1 FOR01.DAT      DISC
CDRPOS:  SIXBIT   .CDR.           ; 2 FOR02.DAT      CARD READER
LPTPOS:  SIXBIT   .LPT.           ; 3 FOR03.DAT      LINE PRINTER
         SIXBIT   .CTY.           ; 4 FOR04.DAT      CONSOLE TELETYPE
TTYPOS:  SIXBIT   .TTY.           ; 5 FOR05.DAT      USER TELETYPE
         SIXBIT   .PTR.           ; 6 FOR06.DAT      PAPER TAPE READER
PTPPOS:  SIXBIT   .PTP.           ; 7 FOR07.DAT      PAPER TAPE PUNCH
         SIXBIT   .DIS.           ; 8 FOR08.DAT      DISPLAY
         SIXBIT   .DTA1.          ; 9 FOR09.DAT      DECTAPE
         SIXBIT   .DTA2.          ; 10 FOR10.DAT
         SIXBIT   .DTA3.          ; 11 FOR11.DAT
         SIXBIT   .DTA4.          ; 12 FOR12.DAT
         SIXBIT   .DTA5.          ; 13 FOR13.DAT
         SIXBIT   .DTA6.          ; 14 FOR14.DAT
         SIXBIT   .DTA7.          ; 15 FOR15.DAT
         SIXBIT   .MTA0.          ; 16 FOR16.DAT     MAGNETIC TAPE
         SIXBIT   .MTA1.          ; 17 FOR17.DAT
         SIXBIT   .MTA2.          ; 18 FOR18.DAT
         SIXBIT   .FORTR.         ; 19 FORTR.DAT     ASSIGNABLE DEVICE, FORTR

         SIXBIT   .DSK0.          ; 20 FOR20.DAT     DISK
         SIXBIT   .DSK1.          ; 21 FOR21.DAT
         SIXBIT   .DSK2.          ; 22 FOR22.DAT
         SIXBIT   .DSK3.          ; 23 FOR23.DAT
         SIXBIT   .DSK4.          ; 24 FOR24.DAT
         SIXBIT   .DEV1.          ; 25 FOR25.DAT     ASSIGNABLE DEVICES
         SIXBIT   .DEV2.          ; 26 FOR26.DAT
         SIXBIT   .DEV3.          ; 27 FOR27.DAT
         SIXBIT   .DEV4.          ; 28 FOR28.DAT
DEVLS.:  SIXBIT   .DEV5.          ; 29 FOR29.DAT       V.006
         SIXBIT   .REREAD.        ;       -6         REREAD
         SIXBIT   .CDR.           ;       -5         READ
         SIXBIT   .TTY.           ;       -4         ACCEPT
         SIXBIT   .LPT.           ;       -3         PRINT
         SIXBIT   .PTP.           ;       -2         PUNCH
DEVND.:  SIXBIT   .TTY.           ;       -1         TYPE
```

12.3.2.3 File Names - File names may be declared for DECtapes or the disk through the use of the library subprograms IFILE and OFILE. In order to make an entry of the file name FILE1 on unit u, the following statement could be used:

```
CALL OFILE (u,'FILE1')
```

Similarly, the following statements might be used to open the file, RALPH, for reading:

```
RALPH=5HRALPH
CALL IFILE(u,RALPH)
```

After writing a file, the END FILE u statement must be given in order to close the current file and allow for reading or writing another file or for reading or rewriting the same file. If no call to IFILE or OFILE has been given before the execution of a READ or WRITE referencing DECtape or the disk the file name FORnn.DAT is assumed where nn is the FORTRAN logical number used in the I/O statement that references device nn.

The FORTRAN programmer can make logical assignments such that each device has its own unique file as intended, but each can be on the DSK. In order to use the devices available, the programmer can make assignments at run time and assign the DSK to those not available.

For example, the FORTRAN logical device numbers, e.g., 1 = DSK, 2 = CDR, 3 = LPT, are used in the file name. The written file names are FOR01.DAT, FOR02.DAT, etc. The same is true for READ. For example, a WRITE (3, 1) A, B, C, in the FORTRAN program generates the file name FOR03.DAT on the DSK if the DSK has been assigned LPT or 3 prior to running the program. (Note: REREAD rereads from the file belonging to the device last referenced in a READ statement, not FOR-6.DAT, as usual.) The programmer must, of course, realize his own mistake in assigning the DSK as the TTY in the case that FORSE tries to type out error messages or PAUSE messages.

More than one DSK File may be accessed, without making logical assignments at runtime, by using logical device numbers 1, and 20 through 24 in the FORTRAN program. Logical device number 19 refers to logical device FORTR which must be assigned at runtime and accesses file name FORTR.DAT to maintain compatibility with the past system of default file name FORTR.DAT. In all cases when the operating system fails to find a file specified, an attempt will be made to read from file FORTR.DAT as before.

The magnetic tape operation REWIND is simulated on DECtape or the disk; a REWIND closes the file and clears the filename. A call to IFILE or OFILE should be made after a REWIND to open the file and preserve the filename, if desired. A program which uses READ, WRITE, END FILE, and REWIND for magnetic tape need only have the logical device number changed or assigned to a DECtape or disk at runtime in order to perform the proper input/output sequences on DECtape or the disk.

12.3.3 Magnetic Tape Usage

Magnetic tape and disk/DECtape I/O are different in the following ways. When a READ is issued, a record is read in for both magnetic tape and disk. If a WRITE is then issued, the next sequential record is written on

magnetic tape but not on disk. When one or more READs have been executed on a disk file and a WRITE is issued, the next record is written. Unless records are written past the existing end-of-file, that end-of-file is not changed, i.e., the file is not truncated.

12.3.3.1 Binary Mode - The format of binary data on magnetic tape is similar to that for DECtape except that the physical record size depends on the magnetic tape buffer size assigned in the Time-Sharing Monitor or by IBUFF/OBUFF (see Section 8.2.2). Normally, the buffer size is set at either 129 or 257 words so that either 128 or 256 word records are written (containing a control word and 127 or 255 FORTRAN data words).

The first word, control word, of each block in a binary record contains information used by the operating system. The left half of the first word contains the word count for that block. The right half of the first word contains a null character except for the last block in a logical record. In this case, the right half of the first word contains the number of blocks in the logical record.

12.3.3.2 ASCII Mode - The format for ASCII data is the same as that used on DECtape.

12.3.3.3 Backspacing and Skipping Records - Both the BACKSPACE u and SKIP RECORD u statements are executed on a logical basis for binary records and on a line basis for ASCII records.

a. Binary Mode - Both BACKSPACE and SKIP RECORD space magnetic tape physically over one (1) logical record; i.e., the result of one WRITE (u) statement.

b. ASCII Mode - ASCII records are packed, that is WRITE (u, f) statements do not cause physical writing on the tape until the output buffers are full or a BACKSPACE, END FILE, or REWIND command is executed by the program. BACKSPACE and SKIP RECORD on ASCII record space over one (1) line.

c. BACKSPACE and SKIP RECORD following WRITE ASCII commands.

(1) BACKSPACE closes the tape, writes 2 EOF's (tapemark) and backspaces over the last line.

(2) SKIP RECORD cannot be used during a WRITE operation. This is an input function only.

12.4 RANDOM ACCESS PROGRAMMING

In random access programming, data is obtained from (or placed into) storage, where the time required for this access is independent of the location of the data most recently obtained from (or placed into) storage. Random access programming allows a programmer to access any record within a file with a single READ or WRITE statement independent of the location of the previously accessed record within that file. For example, a programmer may read or write only the 10th record in a file if he wishes. Random I/O is desirable when only a few records in a large file are to be accessed, or when a file is to be read or wirtten in a non-sequential manner, as in a sort.

Random access applies only to data files on the disk with fixed-length record sizes. Any fixed-length record file (formatted or unformatted) which has been written on the disk with FORTRAN or with PIP using the A switch may be read or rewritten non-sequentially.

12.4.1 How to Use Random Access

A programmer may directly access fixed-length records in a disk file by defining the structure of the file with a CALL DEFINE FILE and then specifying the record he wishes to access with a READ or WRITE statement. The file from which records are to be accessed is defined as follows:

```
CALL DEFINE FILE (U,S,V,F,PJ,PG)
```

where

U = the unit number expressed as an integer. The number must refer to the disk. The numbers from 1 to 10 are available unless a particular installation decides to change this range.

S = the size of the records within the file expressed as an integer. The size is specified by the number of characters per record for formatted records, and the number of words per record for unformatted records. The size of the records must be constant within the file and may be from 1 to 132 characters in formatted records, or one word to any size limited by core in unformatted records.

V = the associated integer variable. Contains any integer value. The record number which would be accessed next if I/O were to continue sequentially is returned as an integer in the associated variable after each random read or write. The associated variable may be used in the I/O statements as part of the integer expression which defines the record number.

F = the filename and extension. This may be zero, in which case standard default names are used.

PJ = the project number in octal of the disk area being accessed.

PG = the programmer number in octal of the disk area being accessed. The project-programmer numbers may be zero, in which case the user's disk area is accessed. Note that the writing on another user's disk area is restricted by the monitor.

I/O begins when the random WRITE or READ is specified in the correct format. (See Sections 5.2.6 and 5.2.7.)

12.4.2 Restrictions

A number of restrictions are imposed in random access programming:

a. A logical unit may not be used for sequential and then random I/O in the same program unless an intervening CALL to RELEASE is issued. For example, if sequential I/O is done to unit 3 then random I/O to unit 3 is illegal and will fail.

b. If the name of a file to be accessed randomly is specified in a DATA statement or is read in at run-time, the user must use a full 6-character filename and a 3-character extension.

c. Mixed formatted and unformatted files are not accessible randomly.

d. Before random I/O is performed through a READ or WRITE statement, the file must be properly defined through a CALL to DEFINE FILE.

e. All FORTRAN data files must be created by FORTRAN or PIP with the A switch.

f. The records within the file must be of a fixed length.

g. Random access is used for disk files only.

h. Access to files is controlled by the file protection scheme in effect at each installation. (Refer to the Timesharing Monitors Manual for a discussion of file access privileges.)

i. CALLs to IFILE or OFILE open files for sequential I/O. They must not be issued for units to be used for random I/O. If it is desired to open a file for sequential I/O on a unit that has been used for random I/O, a CALL to RELEASE must be issued before a CALL to IFILE or OFILE.

12.4.3 Examples

Example 1:

Assume a standard FORTRAN program, the purpose of which is to read the Kth record in a file and ignore all other records. A section of the program might be as follows:

```
                .
                .
                DO 10 I=1, K
10              READ (1,1) A,B,C
1               FORMAT (3A5)
```

If K is a large number, time is wasted in obtaining the Kth record using sequential I/O. Now consider a program written to perform the same function using random access:

```
                CALL DEFINE FILE (1,15,N,0,0,0)
                  .
                  .
                  .
                READ (1#K,1) A,B,C

1               FORMAT (3A5)
```

Note that the default filename FOR01.DAT and the user's project-programmer number are used in both examples.

Example 2:

Consider a program the purpose of which is to change the contents of the Kth record within the file FOR01.DAT on the user's disk area. Using sequential I/O, the code might be as follows:

```
                      .
                      .
                      .
                DO 10 I=1, K-1
                READ (1,1) A,B,C
10              WRITE (2,1) A,B,C
                READ (1,1) A,B,C
                WRITE (2,1) D,E,F
                DO 20 I=K+1,NEND
                READ (1,1) A,B,C
20              WRITE (2,1) A,B,C
1               FORMAT (3A5)
```

There would be two files on the disk, FOR01.DAT and FOR02.DAT, which are identical except for the Kth record. The code that accomplishes the same result using random access is:

```
      CALL DEFINE FILE (1,15,N,0,0,0)

      WRITE (1#K,1) D,E,F
1     FORMAT (3A5)
        .
        .
        .
```

A new file is not created; the old file remains with the Kth record changed.

Example 3:

The following code creates a new file for random output by first writing K blank records and then updating the file in non-sequential output:

```
C     40 SPACES PER RECORD USERS
C     NEED NO WORRY ABOUT CARRIAGE
C     RETURNS AND LINE FEEDS.

      DIMENSION A(8), B(8)
      DO 10 I=1,K
10    WRITE (1,1) A
        .
        .
        .
      CALL DEFINE FILE (2,40,N,'FOR01.DAT',0,0)
      N=3
      DO 20 I=1,5
20    WRITE (2#N*8,2) B
1     FORMAT (8A5)
2     FORMAT (4I5,2A5,F10,3)
        .
        .
        .
```

Example 4:

Read a 1000 record file, the records of which are 27 characters long, backwards. The file is named FOR01.DAT and resides on the user's disk area. The following program creates a disk file and then reads it backwards. (Note that the same unit number may not be used for both sequential and random I/O in the same program):

```
        DIMENSION A(6)
        CALL DEFINE FILE (2,27,NV,'FOR01.DAT',0,0)
        DO 10 I=1, 1000
10      WRITE (1,1) I
        REWIND (1)
1       FORMAT ('THIS IS RECORD NUMBER', I5)
        NV=1002
        DO 20 I=1,1000
20      READ (2#NV-2,2) A
2       FORMAT(5A5,A2)
        END
```

Example 5:

Use random WRITES to change every 7th record, beginning with record 10, in the file named DATA on the user's disk area. The file contains 100 records, each of which is 35 characters long.

```
        DIMENSION LIST(7)
        CALL DEFINE FILE(5,35,NV,'DATA',0,0)
        DO 10 I=10,100,7
10      WRITE (5#I,5) LIST
5       FORMAT (2A5,5I5)
        END
```

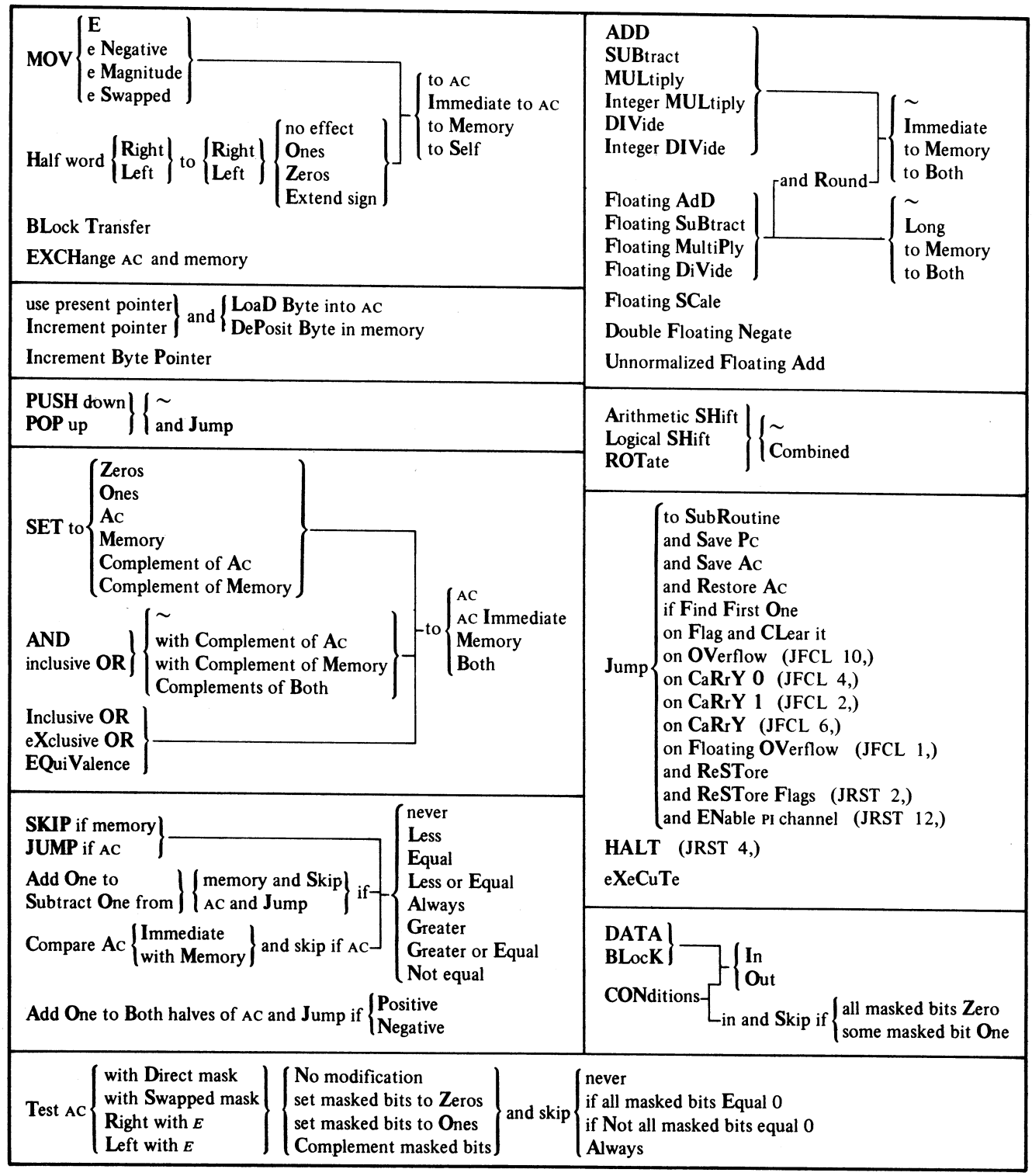
Example 6:

Read one-word binary records, starting with record 26 and ending with record 7, from file FOR07.DAT. The following program creates a 50-record file of the numbers from 1 to 50, reads the file backwards, and types the contents of the record it read, NP, along with the contents of the associated variable, NV. Note that FORTRAN binary output creates files with a maximum of one record per disk block.

```
.TY BINTST
C       BINARY RANDOM ACCESS TEST
C
        DOUBLE PRECISION FIL
        DATA FIL /'FOR07.DAT'/
        CALL DEFINE FILE (2,1,NV,FIL,0,0)
        DO 7 I=1,50
        WRITE(7)I
7       CONTINUE
        END FILE (7)
        NV=28
        DO 2 I=1,20
        READ(2#NV-2)NP
        WRITE(5,5)NP,NV
2       CONTINUE
5       FORMAT(' NP= ',I3,' NV= ',I3)
        END
```

```
RUN DSK BINTST

NP=  26 NV=  27
NP=  25 NV=  26
NP=  24 NV=  25
NP=  23 NV=  24
NP=  22 NV=  23
NP=  21 NV=  22
NP=  20 NV=  21
NP=  19 NV=  20
NP=  18 NV=  19
NP=  17 NV=  18
NP=  16 NV=  17
NP=  15 NV=  16
NP=  14 NV=  15
NP=  13 NV=  14
NP=  12 NV=  13
NP=  11 NV=  12
NP=  10 NV=  11
NP=   9 NV=  10
NP=   8 NV=   9
NP=   7 NV=   8
```

12.5 PDP-10 INSTRUCTION SET



APPENDIX A
THE SMALL FORTRAN IV COMPILER

This compiler runs in 5.5K of core, and to the user, is identical to the large compiler, with the exception of the following language differences. Operating procedures are given in the Systems User's Guide (DEC-10-NGCC-D).

Language Differences

The IMPLICIT, DATA, and NAMELIST statements are not recognized; constant strings are not collapsed (for example, A=5*3 will not be treated as A=15).

DEC-10-KJZE-D

decsystem10 BASIC
CONVERSATIONAL LANGUAGE MANUAL

This manual reflects the software as of version 17A.

DIGITAL EQUIPMENT CORPORATION • MAYNARD, MASSACHUSETTS

1st Printing December 1968
2nd Printing (Rev) May 1969
3rd Printing September 1969
4th Printing (Rev) January 1970
5th Printing (Rev) September 1970
6th Printing (Rev) August 1971
7th Printing (Rev) February 1972
Update Pages May 1972

CONTENTS

CONTENTS (Cont)

CONTENTS (Cont)

CONTENTS (Cont)

ILLUSTRATION

TABLES

PREFACE

WHY BASIC? BASIC is a problem-solving language that is easy to learn and conversational, and has wide application in the scientific, business, and educational communities. It can be used to solve both simple and complex mathematical problems from the user's Teletype® and is particularly suited for time-sharing.

In writing a computer program, it is necessary to use a language or vocabulary that the computer recognizes. Many computer languages are currently in use, but BASIC is one of the simplest of these because of the small number of clearly understandable and readily learned commands that are required, its easy application in solving problems, and its practicality in an evolving educational environment.

BASIC is similar to other programming languages in many respects; and is aimed at facilitating communication between the user and the computer in a time-sharing system. As with most programming languages, BASIC is divided into two sections:

a. Elementary statements that the user must know to write simple programs, and
b. Advanced techniques needed to efficiently organize complicated problems.

As a BASIC user, you type in a computational procedure as a series of numbered statements by using common English syntax and familiar mathematical notation. You can solve almost any problem by spending an hour or so learning the necessary elementary commands. After becoming more experienced, you can add the advanced techniques needed to perform more intricate manipulations and to express your problem more efficiently and concisely. Once you have entered your statements via the Teletype, simply type in RUN or RUNNH. These commands initiate the execution of your program and return your results almost instantaneously.

SPECIAL FEATURES OF BASIC - BASIC incorporates the following special features:

a. Matrix Computations - A special set of 13 commands designed exclusively for performing matrix computations.

®Teletype is the registered trademark of Teletype Corporation.

b. Alphanumeric Information Handling - Single alphabetic or alphanumeric strings or vectors can be read, printed, and defined in LET and IF...THEN statements. Individual characters within these strings can be easily accessed by the user. Conversion can be performed between characters and their ASCII equivalents. Tests can be made for alphabetic order.

c. Program Control and Storage Facilities - Programs or data files can be stored on or retrieved from various devices (disk, DECtape, card reader, card punch, high-speed paper-tape reader, high-speed paper-tape punch and line printer). The user can also input programs or data files from the low-speed Teletype paper-tape reader, and output them to the low-speed Teletype paper-tape punch.

d. Program Editing Facilities - An existing program or data file can be edited by adding or deleting lines, by renaming it, or by resequencing the line numbers. The user can combine two programs or data files into one and request either a listing of all or part of it on the Teletype or a listing of all of it on the high-speed line printer.

e. Formatting of Output - Controlled formatting of Teletype output includes tabbing, spacing, and printing columnar headings.

f. Documentation and Debugging Aids - Documenting programs by the insertion of remarks within procedures enables recall of needed information at some later date and is invaluable in situations in which the program is shared by other users. Debugging of programs is aided by the typeout of meaningful diagnostic messages which pinpoint syntactical and logical errors detected during execution.

CHAPTER 1
INTRODUCTION

This chapter introduces the user to PDP-10 BASIC and to its restrictions and characteristics. The best introduction lies in beginning with a BASIC program and discussing each step completely.

1.1 EXAMPLE OF A BASIC PROGRAM

The following example is a complete BASIC program, named LINEAR, that can be used to solve a system of two simultaneous linear equations in two variables

$$ax + by = c$$
$$dx + ey = f$$

and then used to solve two different systems, each differing from the above system only in the constants c and f. If ae - bd is not equal to 0, this system can be solved to find that

$$x = \frac{ce - bf}{ae - bd} \qquad \text{and} \qquad y = \frac{af - cd}{ae - bd}$$

If ae - bd = 0, there is either no solution or there are many, but there is no unique solution. Study this example carefully and then read the commentary and explanation. (In most cases the purpose of each line in the program is self-evident.)

```
10      READ A,B,D,F
15      LET G=A*E-B*D
20      IF G=0 THEN 65
30      READ C,F
37      LET X=(C*E-B*F)/G
42      LET Y=(A*F-C*D)/G
55      PRINT X,Y
60      GO TO 30
65      PRINT "NO UNIQUE SOLUTION"
70      DATA 1,2,4
80      DATA 2,-7,5
85      DATA 1,3,4,-7
90      END
```

NOTE

All statements are terminated by pressing the RETURN key (represented in this text by the symbol ↵). The RETURN key echoes as a carriage return, line feed.

1.2 DISCUSSION OF THE PROGRAM

Each line of the program begins with a line number of 1 to 5 digits that serves to identify the line as a statement. A program is made up of statements. Line numbers serve to specify the order in which these statements are to be performed. Before the program is run, BASIC sorts out and edits the program, putting the statements into the orders specified by their line numbers; thus, the program statements can be typed in any order, as long as each statement is prefixed with a line number indicating its proper sequence in the order of execution. Each statement starts after its line number with an English word which denotes the type of statement. Unlike statements, commands are not preceded by line numbers and are executed immediately after they are typed in. (Refer to Chapter 9 for a further description of commands.) Spaces and tabs have no significance in BASIC programs or commands, except in messages which are printed out, as in line number 65 above. Thus, spaces or tabs may, but need not be, used to modify a program and make it more readable.

With this preface, the above example can be followed through step-by-step.

```
10      READ A,B,D,E
```

The first statement, 10, is a READ statement and must be accompanied by one or more DATA statements. When the computer encounters a READ statement while executing a program, it causes the variables listed after the READ to be given values according to the next available numbers in the DATA statements. In this example, we read A in statement 10 and assign the value 1 to it from statement 70 and, similarly, with B and 2, and with D and 4. At this point, the available data in statement 70 has been exhausted, but there is more in statement 80, and we pick up from it the value 2 to be assigned to E.

```
15      LET G=A*E-B*D
```

Next, in statement 15, which is a LET statement, a formula is to be evaluated. [The asterisk (*) is used to denote multiplication.] In this statement, we compute the value of AE - BD, and call the result G. In general, a LET statement directs the computer to set a variable equal to the formula on the right side of the equal sign.

```
20      IF G=0 THEN 65
```

If G is equal to zero, the system has no unique solution. Therefore, we next ask, in line 20, if G is equal to zero.

```
65      PRINT "NO UNIQUE SOLUTION"
70      DATA 1,2,4
80      DATA 2,-7,5
85      DATA 1,3,4,-7
90      END
```

If the computer discovers a "yes" answer to the question, it is directed to go to line 65, where it prints NO UNIQUE SOLUTION. Since DATA statements are not executable statements, the computer then goes to line 90 which tells it to END the program.

```
30      READ C,F
```

If the answer is "no" to the question "Is G equal to zero?", the computer goes to line 30. The computer is now directed to read the next two entries, -7 and 5, from the DATA statements (both are in statement 80) and to assign them to C and F, respectively. The computer is now ready to solve the system

$$x + 2y = -7$$
$$4x + 2y = 5$$

```
37      LET X=(C*E-B*F)/G
42      LET Y=(A*F-C*D)/G
```

In statements 37 and 42, we instruct the computer to compute the value of X and Y according to the formulas provided, using parentheses to indicate that C*E - B*F is calculated before the result is divided by G.

```
55      PRINT X,Y
60      GO TO 30
```

The computer prints the two values X and Y in line 55. Having done this, it moves on to line 60 where it is reverted to line 30. With additional numbers in the DATA statements, the computer is told in line 30 to take the next one and assign it to C, and the one after that to F. Thus,

$$x + 2y = 1$$
$$4x + 2y = 3$$

As before, it finds the solutions in 37 and 42, prints them out in 55, and then is directed in 60 to revert to 30.

In line 30, the computer reads two more values, 4 and -7, which it finds in line 85. It then proceeds to solve the system

$$x + 2y = 4$$
$$4x + 2y = -7$$

and print out the solutions. Since there are no more pairs of numbers in the DATA statement available for C and F, the computer prints OUT OF DATA IN 30 and stops.

If line number 55 (PRINT X, Y) had been omitted, the computer would have solved the three systems and then told us when it was out of data. If we had omitted line 20, and G were equal to zero, the computer would print DIVISION BY ZERO IN 37 and DIVISION BY ZERO IN 42. Had we omitted statement 60 (GO TO 30), the computer would have solved the first system, printed out the values of X and Y, and then gone to line 65, where it would be directed to print NO UNIQUE SOLUTION.

The particular choice of line numbers is arbitrary as long as the statements are numbered in the order the machine is to follow. We would normally number the statements 10, 20, 30, ..., 130, so that later we can insert additional statements. Thus, if we find that we have omitted two statements between those numbered 40 and 50, we can give them any two numbers between 40 and 50 -- say 44 and 46. Regarding DATA statements, we need only put the numbers in the order that we want them read (the first for A, the second for B, the third for D, the fourth for E, the fifth for C, the sixth for F, the seventh for the next C, etc.). In place of the three statements numbered 70, 80, and 85, we could have written the statement:

```
75      DATA 1,2,4,2,-7,5,1,3,4,-7
```

or, more naturally,

```
70      DATA 1,2,4,2
75      DATA -7,5
80      DATA 1,3
85      DATA 4,-7
```

to indicate that the coefficients appear in the first data statement and the various pairs of right-hand constants appear in the subsequent statements.

The program and the resulting run is shown below as it appears on the Teletype.

```
10      READ A,B,D,E
15      LET G=A*E-B*D
20      IF G=0 THEN 65
30      READ C,F
37      LET X=(C*E-B*F)/G
42      LET Y=(A*F-C*D)/G
55      PRINT X,Y
60      GO TO 30
65      PRINT "NO UNIQUE SOLUTION"
70      DATA 1,2,4
80      DATA 2,-7,5
85      DATA 1,3,4,-7
90      END
RUN
```

(continued on next page)

```
LINEAR          11:03          19-OCT-69

 4              -5.50000
 0.666667        0.166667
-3.66667         3.83333
OUT OF DATA IN 30
TIME:  0.10 SECS.
```

NOTE

Remember to terminate all statements by pressing the RETURN key.

After typing the program, we type the command RUN and press the RETURN key to direct the computer to execute the program. Note that the computer, before printing out the answers, printed the name LINEAR which we gave to the problem (refer to Paragraph 4.1) and the time and date of the computation. The message OUT OF DATA IN 30, may be ignored here. However, in some instances, it indicates an error in the program. The TIME message, printed out at the end of execution, indicates the compile and execute time used by the program; this time is slightly dependent upon other jobs being processed by the computer and consequently will not be exactly the same each time the same program is run.

1.3 FUNDAMENTAL CONCEPTS OF BASIC

BASIC can perform many operations such as adding, subtracting, multiplying, dividing, extracting square roots, raising a number to a power, and finding the sine of an angle measured in radians.

1.3.1 Arithmetic Operations

The computer performs its primary function (that of computation) by evaluating formulas similar to those used in standard mathematical calculation, with the exception that all BASIC formulas must be written on a single line. The following operators can be used to write a formula.

Operator	Example	Meaning
+	A + B	add B to A
+	+A	A itself
-	A - B	subtract B from A
-	-A	make A negative
*	A * B	multiply B by A
/	A / B	divide A by B
↑	X ↑ 2	find X^2 (the symbols ↑ and ** have
**	X**2	find X^2 the same meaning)

If we type A + B * C ↑ D, the computer first raises C to the power D, multiplies this result by B, and then adds the resulting product to A. We must use parentheses to indicate any other order. For

example, if it is the product of B and C that we want raised to the power D, we must write A + (B * C) ↑ D; or if we want to multiply A + B by C to the power D, we write (A + B) * C ↑ D. We could add A to B, multiply their sum by C, and raise the product to the power D by writing ((A + B) * C) ↑ D. The order of precedence is summarized in the following rules.

a. The formula inside parentheses is evaluated before the parenthesized quantity is used in computations.

b. Normally two operators cannot be contiguous. However the operators + and - can follow the operators *, /, **, or ↑ (e.g., *-). In such a case, the + or - takes precedence over its leading *, /, **, or ↑. Otherwise:

c. In the absence of parentheses in a formula, ** and ↑ take precedence over * and /, which take precedence over + and -.

d. In the absence of parentheses in a formula whose only operators are * and /, BASIC performs the operations from left to right, in the order that they are read.

e. In the absence of parentheses in a formula whose only operators are + and -, BASIC performs the operations from left to right, in the order that they are read.

The rules tell us that the computer, faced with A - B - C, (as usual) subtracts B from A, and then C from their difference; faced with A/B/C, it divides A by B, and that quotient by C. Given A ↑ B ↑ C, the computer raises the number A to the power B and takes the resulting number and raises it to the power C. If there is any question about the precedence, put in more parentheses to eliminate possible ambiguities.

1.3.2 Mathematical Functions

In addition to these five arithmetic operations, BASIC can evaluate certain mathematical functions. These functions are given special three-letter English names.

Function		Interpretation	
SIN	(X)	Find the sine of X	X interpreted as an angle measured in radians
COS	(X)	Find the cosine of X	
TAN	(X)	Find the tangent of X	
COT	(X)	Find the cotangent of X	
ATN	(X)	Find the arctangent of X	
EXP	(X)	Find e raised to the X power (e^X)	
LOG	(X), or LN(X), or LOGE(X)	Find the natural logarithm of X (log to the base e)	X interpreted as a number
ABS	(X)	Find the absolute value of X (| X |)	
SQR	(X) or SQRT(X)	Find the square root of X ($\sqrt{X}$)	
CLOG	(X) or LOG10(X)	Find the common logarithm of X (log to the base 10)	

Five other functions are also available in BASIC: INT, RND, SGN, NUM, and DET; these are reserved for explanation in Chapters 5 and 7. In place of X, we may substitute any formula or any number in parentheses following any of these functions. For example, we may ask the computer to find

$\sqrt{4 + X^3}$ by writing SQR (4 + X ↑ 3), or the arctangent of $3X - 2e^X + 8$ by writing ATN (3 * X - 2 * EXP (X) + 8). If the above value of $(\frac{5}{6})^{17}$ is needed, the two-line program can be written:

```
10      PRINT(5/6)↑17
20      END
```

and the computer finds the decimal form of this number and prints it out.

1.3.3 Numbers

A number may be positive or negative and it may contain up to eight digits, but it must be expressed in decimal form (i.e., 2, -3.675, 12345678, -.98765432, and 483.4156). The following are not numbers in BASIC: 14/3 and SQR(7). The computer can find the decimal expansion of 14/3 or SQR(7), but we may not include either in a list of DATA. We gain further flexibility by using the letter E, which stands for: times ten to the power. Thus, we may write .0012345678 as .12345678E-2 or 12345678E-10 or 1234.5678E-6. We do not write E7 as a number, but write 1E7 to indicate that it is 1 that is multiplied by 10^7.

1.3.4 Variables

A simple (i.e., unsubscripted) numeric variable in BASIC is denoted by any letter or by any letter followed by a single digit. (Refer to Chapter 3 for a discussion of subscripted numeric variables and to Chapter 8 for a discussion of subscripted and unsubscripted string variables.) Thus, the computer interprets E7 as a variable, along with A, X, N5, I0, and O1. A numeric variable in BASIC stands for a number, usually one that is not known to the programmer at the time the program is written. Variables are given or assigned values by LET and READ statements. The value so assigned does not change until the next time a LET or READ statement is encountered with a value for that variable. However, all numeric variables are set equal to 0 before a RUN. Consequently, it is only necessary to assign a value to a numeric variable when a value other than 0 is required.

Although the computer does little in the way of correcting during computation, it sometimes helps if an absolute value hasn't been indicated. For example, if you ask for the square root of -7 or the logarithm of -5, the computer gives the square root of 7 along with an error message stating that you have asked for the square root of a negative number, or it gives the logarithm of 5 along with the error message that you have asked for the logarithm of a negative number.

1.3.5 Relational Symbols

Six other mathematical symbols of relation are used in IF-THEN statements where it is necessary to compare values. An example of the use of these relation symbols was given in the sample program LINEAR.

Any of the following six standard relations may be used:

Symbol	Example	Meaning
=	A = B	A is equal to B
<	A <B	A is less than B
<=	A <=B	A is less than or equal to B
>	A >B	A is greater than B
>=	A >=B	A is greater than or equal to B
<>	A <>B	A is not equal to B

Note that while BASIC outputs its answers with only six places of accuracy, variables and formulas may have values accurate to more than six places. If it is desired that result X be checked to only N places, the function

```
INT       (X*10↑N+.5)/10↑N
```

should be used.

1.4 SUMMARY

Several elementary BASIC statements have been introduced in our discussions. In describing each of these statements, a line number is assumed, and brackets are used to denote a general type. For example, [variable] refers to any variable.

1.4.1 LET Statement

The LET statement is used when computations must be performed. This command is not of algebraic equality, but a command to the computer to perform the indicated computations and assign the answer to a certain variable. Each LET statement is of the form:

```
   LET [variable] = [formula]
or
   [variable] = [formula]
```

Generally, several variables may be assigned the same value by a single LET statement. Examples of assigning a value to a single variable are given in the following two statements:

```
100      LET X=X+1
259      W7=(W-X↑3)*(Z-A/(A-B)-17)
```

Examples of assigning a value to more than one variable are given in the following statements:

```
50      X=Y3=A(3,1)=1
```
The variables X, Y3, and A(3,1) are assigned the value 1.

```
90      LET W=Z=3*X-4*X↑2
```
The variables W and Z are assigned the value $3X-4X^2$

1.4.2 READ and DATA Statements

READ and DATA statements are used to enter information into the computer. We use a READ statement to assign to the listed variables those values which are obtained from a DATA statement. Neither statement is used without the other. A READ statement causes the variables listed in it to be given in order, the next available numbers in the collection of DATA statements. Before the program is run, the computer takes all of the DATA statements in the order they appear and creates a large data block. Each time a READ statement is encountered anywhere in the program, the data block supplies the next available number or numbers. If the data block runs out of data, the program is assumed to be finished and we get an OUT OF DATA message.

Since we have to read in data before we can work with it, READ statements normally occur near the beginning of a program. The location of DATA statements is arbitrary, as long as they occur in the correct order. A common practice is to collect all DATA statements and place them just before the END statement.

Each READ statement is of the form:

READ [sequence of variables]

Each DATA statement is of the form:

DATA [sequence of numbers]

```
150     READ X,Y,Z,X1,Y2,G9
330     DATA 4,2,1.7
340     DATA 6.734E-3,-174.321,3.1415927

234     READ B(K)
263     DATA 2,3,5,7,9,11,10,8,6,4

10      READ R(I,J)
440     DATA -3,5,-9,2.37,2.9876,-437.234E-5
450     DATA 2.765, 5.5576, 2.3789E2
```

Remember that numbers, not formulas, are put in a DATA statement, and that 15/7 and SQR(3) are formulas. Refer to Chapter 3 for a discussion of the subscripted variables.

1.4.3 PRINT Statement

The common uses of the PRINT statement are:

a. to print out the results of some computations

b. to print out verbatim a message included in the program

c. a combination of the two

d. to skip a line.

The following are examples of a type a.:

```
100       PRINT X,SQR(X)
135       PRINT X,Y,Z, B*B-4*A*C, EXP(A-B)
```

The first example prints X, and a few spaces to the right, the square root of X. The second prints five different numbers:

$$X, Y, Z, B^2, -4AC, \text{ and } e^{A-B}$$

The computer computes the two formulas and prints up to five numbers per line in this format.

The following are examples of type b.:

```
100       PRINT "NO UNIQUE SOLUTION"
430       PRINT "X VALUE", "SINE", "RESOLUTION"
500       PRINT X,M,D
```

Line 100 prints the sample statement, and line 430 prints the three labels with spaces between them. The labels in 430 automatically line up with the three numbers called for in PRINT statement 500.

The following is an example of type c.:

```
150       PRINT "THE VALUE OF X IS" X
30        PRINT "THE SQUARE ROOT OF" X "IS" SQR(X)
```

If the first has computed the value of X to be 3, it prints out: THE VALUE OF X IS 3. If the second has computed the value of X to be 625, it prints out: THE SQUARE ROOT OF 625 IS 25.

The following is an example of type d.:

```
250       PRINT
```

The computer advances the paper one line when it encounters this command.

1.4.4 GO TO Statement

The GO TO statement is used when we want the computer to unconditionally transfer to some statement other than the next sequential statement. In the LINEAR problem, we direct the computer to go through the same process for different values of C and F with a GO TO statement. This statement is in the form of GO TO [line number].

```
150      GO TO 75
```

1.4.5 IF - THEN Statement

The IF - THEN statement is used to transfer conditionally from the sequential order of statements according to the truth of some relation. It is sometimes called a conditional GO TO statement. Each IF - THEN statement is of the form:

IF [formula] [relation] [formula], THEN [line number]

The comma preceding THEN is optional and can be omitted.

The following are two examples of this statement:

```
40       IF SIN(X)<=M THEN 80
20       IF G=0, THEN 65
```

The first asks if the sine of X is less than or equal to M, and skips to line 80 if so. The second asks if G is equal to 0, and skips to line 65 if so. In each case, if the answer to the question is no, the computer goes to the next line.

1.4.6 ON - GO TO Statement

The IF - THEN statement allows a two-way fork in a program; the ON - GO TO statement allows a many-way switch. The ON - GO TO statement has the form:

ON [formula], GO TO [line number], [line number], ... [line number]

The comma preceding the GO TO can be omitted. For example:

```
80       ON X GO TO 100, 200, 150
```

This condition causes the following to occur:

If X = 1, the program goes to line 100,
If X = 2, the program goes to line 200,
If X = 3, the program goes to line 150

In other words, any formula may occur in place of X, and the instruction may contain any number of line numbers, as long as it fits on a single line. The value of the formula is computed and its integer part is taken. If this is 1, the program transfers to the line whose number is first on the list; if its integer part is 2, the program transfers to the line whose number is the second one, etc. If the integer part of the formula is below 1, or larger than the number of line numbers listed, an error message is printed. To increase the similarity between the ON - GO TO and IF - THEN instructions, the instruction

```
75      IF X>5 THEN 200
```

may also be written as:

```
75      IF X>5 GO TO 200
```

Conversely, THEN may be used in an ON - GO TO statement.

1.4.7 END Statement

Every program must have an END statement, and it must be the statement with the highest line number in the program.

```
999     END
```

CHAPTER 2
LOOPS

We are frequently interested in writing a program in which one or more portions are executed a number of times, usually with slight variations each time. To write a program in such a way that the portions of the program to be repeated are written just once, we use loops. A loop is a block of instructions that the computer executes repeatedly until a specified terminal condition is met.

The use of loops is illustrated and explained by using two versions of a program that performs the simple task of printing out the positive integers 1 through 100 together with the square root of each. The first version, which does not use a loop, is 101 statements long and reads

```
10      PRINT 1,SQR(1)
20      PRINT 2,SQR(2)
30      PRINT 3,SQR(3)
        . . . . . .
990     PRINT 99,SQR(99)
1000    PRINT 100,SQR(100)
1010    END
```

The second version, which uses one type of loop, obtains the same results with far fewer instructions (5 instead of 101):

```
10      LET X=1
20      PRINT X,SQR(X)
30      LET X=X+1
40      IF X<=100 THEN 20
50      END
```

Statement 10 gives the value of 1 to X and initializes the loop. In line 20, both 1 and its square root are printed. Then, in line 30, X is increased by 1, to a value of 2. Line 40 asks whether X is less than or equal to 100; an affirmative answer directs the computer back to line 20, where it prints 2 and $\sqrt{2}$ and goes to 30. Again, X is increased by 1, this time to 3, and at 40 it goes back to 20. This process is repeated -- line 20 (print 3 and $\sqrt{3}$), line 30 (X = 4), line 40 (since 4 < 100, go back to line 20), etc. -- until the loop has been traversed 100 times. Then, after it has printed 100 and its square root, X becomes 101. The computer now receives a negative answer to the question in line 40 (X is greater than 100, not less than or equal to it), does not return to 20 but moves on to line 50, and ends the program. All loops contain four characteristics:

a. initialization (line 10)

b. the body (line 20)

c. modification (line 30)

d. an exit test (line 40)

2.1 FOR AND NEXT STATEMENTS

BASIC provides two statements to specify a loop: the FOR statement and the NEXT statement.

```
10      FOR X=1 TO 100
20      PRINT X,SQR(X)
30      NEXT X
50      END
```

In line 10, X is set equal to 1, and a test is executed, like that of line 40 above. Line 30 carries out two tasks: X is increased by 1, and control transfers back to the test in line 10. There the test is carried out to determine whether to execute the body of the loop again or to go on to the statement following line 30. Thus, lines 10 and 30 take the place of lines 10, 30, and 40 in the previous program.

Note that the value of X is increased by 1 each time BASIC goes through the loop. If we want a different increase, e.g., 5, we could specify it by writing the following:

```
10      FOR X=1 TO 100 STEP 5
```

and then the value of X on the first time through the loop would be 1, on the second time 6, on the third 11, and on the last time 96. The step of 5 which would take X beyond 100 to 101 causes control to transfer to line 50, which ends the program. The STEP may be positive, negative, or zero. We could have caused the original results to be printed in reverse order by writing line 10 as follows:

```
10      FOR X=100 TO 1 STEP-1
```

In the absence of a STEP instruction, a step-size of +1 is assumed.

The word BY may be substituted for the word STEP; FOR TO BY and FOR TO STEP statements are completely equivalent.

More complicated FOR statements are allowed. The initial value, the final value, and the step-size may all be formulas of any complexity. For example, we could write the following:

```
FOR X=N+7*Z TO (Z-N)/3 BY(N-4*Z)/10
```

For a positive or zero step-size, the loop continues as long as the control variable is less than or equal to the final value. For a negative step-size, the loop continues as long as the control variable is greater than or equal to the final value.

If the initial value is greater than the final value (less than the final value for negative step-size), the body of the loop is not performed at all, but the computer immediately passes to the statement following the NEXT. The following program for adding up the first n integers gives the correct result 0 when n is 0.

```
10      READ N
20      LET S=0
30      FOR K=1 TO N
40      LET S=S+K
50      NEXT K
60      PRINT S
70      GO TO 10
90      DATA 3,10,0
99      END
```

In the following description of the instructions used to specify a loop, a line number is assumed and brackets are used to denote a general type.

A FOR statement has one of two forms:

FOR [numeric variable] = [formula] TO [formula] STEP [formula]

or

FOR [numeric variable] = [formula] TO [formula] BY [formula]

Most commonly, the expressions are integers and the STEP or BY is omitted. In the latter case a step-size of +1 is assumed. The accompanying NEXT statement is simple in form, but the variable must be precisely the same one as that following FOR in the FOR statement. Its form is as follows:

NEXT [variable]

Note that for each FOR statement there is one and only one NEXT statement, and vice versa. Some examples of FOR and NEXT statements are:

```
30      FOR X=0 TO 3 STEP D
80      NEXT X

120     FOR X4=(17+COS(Z))/3 TO 3* SQR(10) BY 1/4
235     NEXT X4

240     FOR X=8 TO 3 STEP -1
456     FOR J=-3 TO 12 BY 2
500     NEXT J
505     NEXT X
```

Line 120 specifies that the successive values of X4 are .25 apart, in increasing order. Line 240 specifies that the successive values of X will be 8, 7, 6, 5, 4, 3. Line 456 specifies that J will take on values -3, -1, 1, 3, 5, 7, 9, and 11. If the initial, final, or step-size index values are given as formulas, these formulas are evaluated only upon entering the FOR statement; therefore, if after this evaluation we change the value of a variable in one of these formulas, we do not affect the index value.

The control variable can be changed in the body of the loop; it should be noted that the exit test always uses the latest value of this variable.

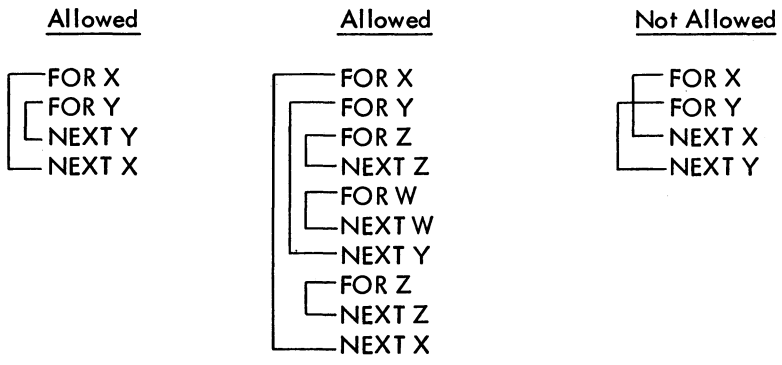
The following difficulty can occur with loops, both FOR-NEXT loops and loops explicitly written with LET and IF statements (as in the example on page 2-1). The calculation of the index values (initial, final, and step-size) is subject to precision limitations inherent in the computer. These values are represented in the computer as binary numbers. When the values are integer, they can be represented exactly in binary; however, it is not always possible to represent decimal values exactly in binary when they contain a fractional part. For example, a loop of the form:

```
40 FOR X=0 TO 10 STEP 0.1
95 NEXT X
```

executes 100 times instead of 101 times because the internal value for 0.1 is not exactly 0.1 After the hundredth execution of the loop, X is not exactly equal to 10, it is slightly larger than 10, so the loop stops. Whenever possible, it is advisable to use indices that have integer values because then the loop will always execute the correct number of times.

2.2 NESTED LOOPS

Nested loops (loops within loops) can be expressed with FOR and NEXT statements. They must be nested and not crossed as the following skeleton examples illustrate:



CHAPTER 3
LISTS AND TABLES

In addition to the ordinary variables used by BASIC, variables can be used to designate the elements of a list or a table. Many occasions arise where a list or a table of numbers is used over and over, and, since it is inconvenient to use a separate variable for each number, BASIC allows the programmer to designate the name of a list or table by a single letter.

Lists are used when we might ordinarily use a single subscript, as in writing the coefficients of a polynomial ($a_0, a_1, a_2, \ldots, a_n$). Tables are used when a double subscript is to be used, as in writing the elements of a matrix ($b_{i,j}$). The variables used in BASIC consist of a single letter, which is the name of the list or table, followed by the subscript in parentheses. Thus,

A(0), A(1), A(2), ..., A(N)

represents the coefficients of a polynomial, and

B(1,1), B(1,2), ..., B(N,N)

represents the elements of a matrix. (Refer to Chapter 8 for a discussion of string variables.)

The single letter denoting a list or a table name may also be used without confusion to denote a simple variable. However, the same letter may not be used to denote both a list and a table in the same program because BASIC recognizes a list as a special kind of table having only one column. The form of the subscript is flexible: A list item B(I + K) may be used, or a table item Q(A(3,7), B-C) may be used. The value of the subscript must not be less than zero.

We can enter the list A(0), A(1), ..., A(10) into a program by the following lines:

```
10      FOR I=0 TO 10
20      READ A(I)
30      NEXT I
40      DATA 0,2,3,-5,2.2,4,-9,123,4,-4,3
```

3.1 THE DIMENSION STATEMENT (DIM)

BASIC automatically reserves room for any list or table with subscripts of 10 or fewer. However, if we want larger subscripts, we must use a DIM statement. This statement indicates to the computer that the specified space is to be allowed for the list or table. For example, the instruction

```
10      DIM A(15)
```

reserves 16 spaces for list A (A(0), A(1), A(2), ..., A(15)). The instruction

```
20      DIM Y(10,15)
```

reserves 176 spaces for matrix Y (10 + 1 rows * 15 + 1 columns). Space may be reserved for more than one list and/or table with a single DIM statement by separating the entries with commas, as shown in the following example:

```
30      DIM A(100),B(20,30),C(25)
```

A DIM statement is not executed; therefore, it may appear on any line before the END statement. However, the best place to put it is at the beginning so that it is not forgotten. If we enter a table with a subscript greater than 10, without a DIM statement, BASIC gives an error message, telling us that we have a subscript error. This condition can be rectified by entering a DIM statement with a line number less than the line number of the END statement.

A DIM statement is normally used to reserve additional space, but in a long program that requires many small tables, it may be used to reserve less space for tables in order to have more space for the program. When in doubt, declare a larger dimension than you expect to use, but not one so large that there is no room for the program. For example, if we want a list of 15 numbers entered, we may write the following:

```
10      DIM A(25)
20      READ N
30      FOR I=1 TO N
40      READ A(I)
50      NEXT I
60      DATA 15
70      DATA 2,3,5,7,11,13,17,19,23,29,31,37,41,43,47
```

Statements 20 and 60 could have been eliminated by writing 30 as FOR I = 1 TO 15 but the program as typed allows for the lengthening of the list by changing only statement 60, as long as the list does not exceed 25 and there is sufficient data.

We could enter a 3-by-5 table into a program by writing the following:

```
10      FOR I=1 TO 3
20      FOR J=1 TO 5
30      READ B(I,J)
40      NEXT J
50      NEXT I
60      DATA 2,3,-5,-9,2
70      DATA 4,-7,3,4,-2
80      DATA 3,-3,5,7,8
```

Again, we may enter a table with no DIM statement: BASIC then handles all the entries from B(0,0) to B(10,10).

3.2 EXAMPLE

Below are the statements and run of a problem which uses both a list and a table. The program computes the total sales of five salesmen, all of whom sell the same three products. The list, P, gives the price per item of the three products and the table, S, tells how many items of each product each man sold. Product 1 sells for $1.25 per item, product 2, for $4.30 per item, and product 3, for $2.50 per item; also, salesman 1 sold 40 items of the first product, 10 of the second, 35 of the third, and so on. The program reads in the price list in lines 40 through 80, using data in lines 910 through 930. The same program could be used again, modifying only line 900 if the prices change, and only lines 910 through 930 to enter the sales in another month. This sample program does not need a DIM statement, because the computer automatically reserves enough space to allow all subscripts to run from 0 to 10.

NOTE

Since spaces are ignored, statements may be indented for visual identity of the various loops within the program.

```
10      FOR I=1 TO 3
20        READ P(I)
30      NEXT I
40      FOR I=1 TO 3
50        FOR J=1 TO 5
60      READ S(I,J)
70          NEXT J
80       NEXT I
90      FOR J=1 TO 5
100       LET S=0
110       FOR I=1 TO 3
120        LET S=S+P(I)*S(I,J)
130       NEXT I
140       PRINT "TOTAL SALES FOR SALESMAN"J,"$"S
150      NEXT J
900      DATA 1.25,4.30,2.50
910      DATA 40,20,37,29,42
920      DATA 10,16,3,21,8
930      DATA 35,47,29,16,33
999      END
```

(continued on next page)

```
RUN
SALES1          11:06          20-OCT-69
TOTAL SALES FOR SALESMAN 1 $ 180.500
TOTAL SALES FOR SALESMAN 2 $ 211.300
TOTAL SALES FOR SALESMAN 3 $ 131.650
TOTAL SALES FOR SALESMAN 4 $ 166.500
TOTAL SALES FOR SALESMAN 5 $ 169.400
TIME: 0.16 SECS.
```

3.3 SUMMARY

Because the number of simple variable names is limited, BASIC allows a programmer to use lists and tables to increase the number of problems that can be programmed easily and concisely. A single letter is used for the name of the list or table, and the subscript that follows is enclosed in parentheses. The subscripts may be explicitly stated or may be any legal expression.

Lists and tables are called subscripted variables, and simple variables are called unsubscripted variables. Usually, you can use a subscripted variable anywhere that you use an unsubscripted variable. However, the variable mentioned immediately after FOR in the FOR statement and after NEXT in the NEXT statement must be an unsubscripted variable. The initial, terminal, and step values may be any legal expression.

3.3.1 The DIM Statement

To enter a list or a table with a subscript greater than 10, a DIM statement is used to retain sufficient space, as in the following examples:

```
20      DIM H(35)
35      DIM Q(5, 25)
```

The first example enables us to enter list H with 36 items (H(0), H(1), ..., H(35)). The second reserves space for a table of 156 items (5 + 1 rows * 25 + 1 columns).

CHAPTER 4
HOW TO RUN BASIC

After learning how to write a BASIC program, we must learn how to gain access to BASIC via the Teletype so that we can type in a program and have the computer solve it. Steps required to communicate with the monitor must first be performed. These steps are fully explained in the PDP-10 Reference Handbook and the TOPS-10 Operating System Commands manual.

4.1 GAINING ACCESS TO BASIC

Once the monitor has responded with a period to indicate that it is ready to receive a monitor command, type in the following command:

```
.R BASIC
```

This command establishes contact with the BASIC CUSP (Commonly Used System Program). BASIC responds with the following:

```
NEW OR OLD--
```

Type in:

```
NEW
```

if you are going to create a new program. BASIC responds with the following:

```
NEW FILE NAME--
```

Type in the name of your new program. If you want to work with a previously created program that you saved on a storage device, type in the following:

```
OLD
```

BASIC then asks for the name of the old program, as follows:

```
OLD FILE NAME--
```

Respond by typing in the name of your old file. If your old file is stored on a device other than the disk, you must type in the device name as in the following example:

```
OLD FILE NAME--DTA6:SAMPLE
```

BASIC retrieves the file named SAMPLE from DECtape 6 and replaces the current contents of user core with the file SAMPLE. The disk may be specified as the device on which the old program is stored, but this is not necessary because the disk is the device used when no device is specified. For example, the following statements are equivalent:

```
OLD FILE NAME--DSK:TEST1
OLD FILE NAME--TEST 1
```

Device names are as follows:

DSK	the disk
DTA0 through DTA7	DECtapes number 0 through 7
TTY	your Teletype
TTY0 through TTY177	Teletypes number 0 through 177
LPT	the line printer
MTA0 through MTA7	magnetic tapes number 0 through 7
PTP	the high-speed paper-tape punch
PTR	the high-speed paper-tape reader
CDP	the card punch
CDR	the card reader
SYS	the system device where system programs are stored

Not all installations have all of these devices; if you specify a device that does not exist or that is not available for your use, BASIC returns an error message. Also, while it is possible to store a file on the card punch, for example, the file cannot be retrieved from this device but must be retrieved from the card reader. If you specify for OLD a device that can only do output, an error message will be returned.

Program names can be any combination of letters and digits up to and including six characters in length. In addition to specifying a program name, you may also specify an extension. The extension follows the name and is separated from it by a period. An extension is any combination of letters and digits up to and including three characters in length. In previous chapters we have used program names such as LINEAR and SALES1. If you recall an old program from storage, you must use exactly the same name and extension you assigned to it when it was saved.

You can also type the name of your file (and the device on which it is located) on the same line as the NEW or OLD command. In this case, BASIC will not ask for the name of the file. For example:

```
NEW TEST

OLD DTA6:SAMPLE
```

The NEW OR OLD -- request can be answered not only by NEW or OLD, but also by any other command (refer to Chapter 9 for a description of the commands) or statement. If NEW OR OLD -- is answered by a NEW, OLD, or RENAME command, the current device, filename, and extension are established by the arguments specified with the command; if a device is not specified explicitly, the disk is assumed; if a filename is specified without an extension, the extension BAS is assumed; it is illegal to specify an extension without specifying a filename.

If NEW OR OLD -- is answered by anything other than a NEW, OLD, or RENAME command, the current device, filename, and extension are established as DSK, NONAME, and BAS, respectively. For example, the following sequence creates a disk file called NONAME.BAS.

```
.R BASIC
NEW OR OLD -- 5 PRINT "TESTING"
10 END
SAVE
```

A new current device, filename, and extension are established whenever a NEW, OLD, or RENAME command is given.

4.2 ENTERING THE PROGRAM

After you type in your filename (whether it is old or new), BASIC responds with the following:

```
READY
```

You can begin to type in your program. Make sure that each line begins with a line number containing no more than five digits and containing no spaces or nondigit characters. Also, be sure to start at the beginning of the Teletype line for each new line. Press the RETURN key upon completion of each line.

If, in the process of typing a statement, you make a typing error and notice it before you terminate the line, you can correct it by pressing the RUBOUT key once for each character to be erased, going backward until the character in error is reached. Then continue typing, beginning with the character in error. The following is an example of this correcting process:

```
10      PRNIT\TIN\INT 2,3
```

NOTE

The RUBOUT key echoes as a backslash (\), followed by the deleted characters and a second backslash.

4.3 EXECUTING THE PROGRAM

After typing the complete program (do not forget to end with an END statement), type RUN or RUNNH, followed by the RETURN key. BASIC types the name of the program, the time of day, the current date (unless RUNNH is specified), and then it analyzes the program. If the program can be run, BASIC executes it and, via PRINT statements, types out any results that were requested. The typeout of results does not guarantee that the program is correct (the results could be wrong), but it does indicate that no grammatical errors exist (e.g., missing line numbers, misspelled words, or illegal syntax). If errors of this type do exist, BASIC types a message (or several messages) to you. A list of these diagnostic messages, with their meanings, is given in Appendix B.

4.4 CORRECTING THE PROGRAM

If you receive an error message typeout informing you, for example, that line 60 is in error, the line can be corrected by typing in a new line 60 to replace the erroneous one. If the statement on line 110 is to be eliminated from your program, it is accomplished by typing the following:

```
110
```

If you wish to insert a statement between lines 60 and 70, type a line number between 60 and 70 (e.g., 65), followed by the statement.

4.5 INTERRUPTING THE EXECUTION OF THE PROGRAM

If the results being typed out seem to be incorrect and you want to stop the execution of your program, type ↑O (hold down CTRL key and at the same time type O) to suppress the typeout, or type ↑C twice, as indicated in the following example:

```
↑C
↑C      } Stops execution of your program, and Returns control to Monitor
```

If you typed ↑C, the monitor responds with a period and waits for you to type a monitor command. If you wish to reinitialize, type either of the following:

```
.START    or    .REENTER
```

BASIC responds with the following:

```
READY
```

whereupon you can modify or add statements and/or type RUN. If you wish to continue at the point where you interrupted the execution, type the following:

```
.CONT
```

4.6 LEAVING THE COMPUTER

When you wish to leave the computer, type the BYE or GOODBYE command.

If the system monitor is a 4-Series monitor, it responds with the message:

```
CONFIRM:
```

Then, if you simply want to get off the machine, type the following:

```
K)
```

Neither data files created by WRITE # or PRINT # statements nor files that were SAVEd, REPLACEd, or COPIed on the disk are deleted by this procedure. Other options available following the typeout of CONFIRM: are listed for you if you respond to the CONFIRM: message with a carriage return (RETURN key) only. The monitor then lists all options available, along with the response required to request each option.

If the system monitor is a 5-Series monitor, it responds to the BYE or GOODBYE command by logging you off the system completely, unless your files stored on the disk take up so much room that you are over the logged-out quota set by the system administrator. In that case, the following message is typed out (n and m are the appropriate integers):

```
DSK LOGGED OUT QUOTA n EXCEEDED BY m BLOCKS
CONFIRM:
```

If you then type

```
H)
```

instructions for deleting files at logout time are typed on your Teletype.

4.7 EXAMPLE OF BASIC RUN

The following is a simple example of the use of BASIC under a timesharing monitor:

.↑C	GO TO MONITOR LEVEL
.LOGIN	REQUEST LOGIN
JOB 7 5S0318A TTY34	MONITOR TYPES OUT YOUR ASSIGNED JOB NUMBER, THE CURRENT VERSION NUMBER OF THE MONITOR, AND YOUR TELETYPE NUMBER
.	MONITOR REQUESTS YOUR PROJECT-PROGRAMMER NUMBER; TYPE IT IN
#27,20	

(continued on next page)

```
PASSWORD:                       MONITOR REQUESTS YOUR PASSWORD;
                                TYPE IT IN; IT WILL NOT ECHO BACK

0927 29-OCT-69 WED              MONITOR TYPES OUT THE TIME OF
                                DAY, THE CURRENT DATE, THE DAY OF
                                THE WEEK, AND A PERIOD

.R BASIC                        INSTRUCT MONITOR TO BRING BASIC
                                INTO CORE AND START ITS EXECUTION

NEW OR OLD--NEW                 BASIC ASKS WHETHER NEW OR OLD
                                PROGRAM IS TO BE RUN

NEW FILE NAME--SAMPLE           BASIC ASKS FOR NEW FILENAME

READY                           BASIC IS NOW READY TO RECEIVE
                                STATEMENTS

10      FOR N=1 TO 7            TYPE IN STATEMENTS

20      PRINT N, SQR(N)

30      NEXT N

40      PRINT "DONE"

50      END

RUN                             RUN PROGRAM

SAMPLE          11:14             29-OCT-69

 1               1

 2               1.41421

 3               1.73205

 4               2

 5               2.23607

 6               2.44949

 7               2.64575

DONE

TIME: 0.20 SECS.

READY

^C

.KJOB /F

JOB 7, USER [27, 20] LOGGED OFF TTY34 0930 29-OCT-69

SAVED ALL 1 FILE (5 DISK BLOCKS)

RUNTIME 0 MIN, 01 SEC
```

4.8 ERRORS AND DEBUGGING

Occasionally, the first run of a new problem is free of errors and gives the correct answers, but, more commonly, errors are present and have to be corrected. Errors are of two types: errors of form (grammatical errors) which prevent the running of the program, and logical errors in the program which cause the computer to produce wrong answers or no answers at all.

Errors of form cause error messages to be printed, and the various types of error messages are listed and explained in Appendix B. Logical errors are more difficult to uncover, particularly when the program gives answers which seem to be nearly correct. In either case, after the errors are discovered, they can be corrected by changing lines, by inserting new lines, or by deleting lines from the program. As indicated previously, a line is changed by typing it correctly with the same line number; a line is inserted by typing it with a line number between those of two existing lines; and a line is deleted by typing its line number and pressing the RETURN key. Note that you can insert a line only if the original line numbers are not consecutive integers. For this reason, most programmers begin by using arbitrary line numbers that are multiples of five or ten.

These corrections can be made either before or after a run. Since BASIC sorts out lines and arranges them in order, a line may be retyped out of sequence. Simply retype the offending line with its original line number.

4.8.1 Example of Finding and Correcting Errors

We can best illustrate the process of finding the errors (bugs) in a program and correcting (debugging) them by an example. Consider the problem of finding that value of X between 0 and 3 for which the sine of X is a maximum, and ask the machine to print out this value of X and the value of its sine. Although we know that $\pi/2$ is the correct value, we use the computer to test successive values of X from 0 to 3, first using intervals of .1, then of .01, and finally of .001. Thus, we ask the computer to find the sine of 0, of .1, of .2, of .3..., of 2.8, of 2.9, and of 3, and to determine which of these 31 values is the largest. It does so by testing SIN(0) and SIN(.1) to see which is larger, and calling the larger of these two numbers M. It then picks the larger of M and SIN (.2) and calls it M. This number is checked against SIN (.3). Each time a larger value of M is found, the value of X is "remembered" in X0. When it finishes, M will have been assigned to the largest value. It then repeats the search, this time checking the 301 numbers 0, .01, .02, .03, ..., 2.98, 2.99, and 3, finding the sine of each, and checking to see which has the largest sine. At the end of each of these three searches, we want the computer to print three numbers: the value X0 which has the largest sine, the sine of that number, and the interval of search.

Before going to the Teletype, we write a program such as the following:

```
10      READ D
20      LET X0=0
30      FOR X=0 TO 3 STEP D
40      IF SIN(X)<=M THEN 100
50      LET X0=X
60      LET M=SIN(X0)
70      PRINT X0,X,D
80      NEXT X0
90      GO TO 20
100     DATA .1,.01,.001
110     END
```

The following is a list of the entire sequence on the Teletype with explanatory comments on the right side:

```
NEW OR OLD--NEW
NEW FILE NAME--MAXSIN
READY
10      READ D
20      LWR X0=0
30      FOR X=0 TO STEP 3 D
40      IF SINE\E\(X)<=M THEN 100
50      LET X0=X
60      LET M=SIN(X)
70      PRINT XO,X,D
80      NEXT T\T\X0
90      GO TO 20
20      LET X0=0
100     DATA .1,.01,.001
110     END
RUN

MAXSIN          11:35           20-OCT-69

ILLEGAL VARIABLE IN 70
NEXT WITHOUT FOR IN 80
FOR WITHOUT NEXT IN 30
TIME: 0.05 SECS.
READY

70      PRINT X0,X,D
40      IF SIN (X) <=M THEN 80
80      NEXT X
RUN

MAXSIN          11:36           20-OCT-69

 0.1             0.1             0.1
 0.2             0.2             0.1
 0.3             ↑C↑C
.REEN
READY

20
RUN

MAXSIN          11:37           20-OCT-69
```

Note the use of the RUBOUT key (echoes as a \) to erase a character in line 40 (which should have started IF SIN (X), etc.) and in line 80.

We discover that LET was mistyped in line 20, and we correct it after 90.

After receiving the first error message, we inspect line 70 and find that we used XO for a variable instead of X0. The next two error messages relate to lines 30 and 80 having mixed variables. These are corrected by changing line 80.

Both of these changes are made by retyping lines 70 and 80. In looking over the program, we also discover that the IF - THEN statement in 40 directed the computer to a DATA statement and not to line 80 where it should go. This is obviously incorrect. We are having every value of X printed, so we direct the machine to cease operations by typing ↑C twice even while it is running. We notice that SIN(0) is compared with M on the first time

(continued on next page)

```
UNDEFINED LINE NUMBER 20 IN 90
TIME: 0.03 SECS.

90      GO TO 10
RUN

MAXSIN          11:43           20-OCT-69

 0.1             0.1             0.1
 0.2             0.2             0.1
 0.3             ↑C↑C
.REEN
READY
```

through the loop, but we had assigned a value to X0 but not to M. However, we recall that all variables are set equal to zero before a RUN; therefore, line 20 is unnecessary.

Line 90, of course, sent us back to line 20 to repeat the operation and not back to line 10 to pick up a new value for D. We retype line 90 and then type RUN again.

We are about to print out the same table as before. Each time that it goes through the loop, it is printing out X0, the current value of X, and the interval size.

```
70
85      PRINT X0,M,D
5       PRINT "X VALUE","SIN",RESOLUTION"
RUN

MAXSIN          11:44           20-OCT-69

ILLEGAL VARIABLE IN 5
TIME: 0.08 SECS.
READY
5       PRINT "X VALUE","SIN","RESOLUTION"
RUN


MAXSIN          11:47           20-OCT-69
X VALUE         SINE            RESOLUTION
 1.60            0.999574        0.1
 1.57            1.              0.01
 1.57099         1.              0.001
OUT OF DATA IN 10
TIME: 0.96 SECS.
READY
```

We rectify this condition by moving the PRINT statement outside the loop. Typing 70 deletes that line, and line 85 is outside of the loop. We also realize that we want M printed, not X. We also decide to put in headings for the columns by a PRINT statement.

There is an error in our PRINT statement: no left quotation mark for the third item.

Retype line 5, with all of the required quotation marks.

These are the desired results. Of the 31 numbers (0, .1, .2, .3,..., 2.8, 2.9, 3), it is 1.6 which has the largest sine, namely .999574; this is true for finer subdivisions.

```
LIST

MAXSIN 11:48 20-OCT-69

5       PRINT "X VALUE","SINE","RESOLUTION"
10      READ D
30      FOR X=0 TO 3 STEP D
40      IF SIN(X)<=M THEN 80
50      LET X0=X
60      LET M=SIN(X)
80      NEXT X
85      PRINT X0,M,D
90      GO TO 10
100     DATA .1, .01,.001
110     END

READY
SAVE
READY
```

Having changed so many parts of the program, we ask for a list of the corrected program.

The program is saved for later use.

A PRINT statement could have been inserted to check on the machine computations. For example, if M were checked, we could have inserted 65 PRINT M, and seen the values.

CHAPTER 5
FUNCTIONS AND SUBROUTINES

5.1 FUNCTIONS

Occasionally, you may want to calculate a function, for example, the square of a number. Instead of writing a small program to calculate this function, BASIC provides 14 functions as part of the language, 9 of which are described in Chapter 1. Three of the remaining functions are described here, and the last two are described in Chapter 7.

The desired function is called by a three-letter name. The value to be used is expressed explicitly or implicitly in parentheses and follows the function name. The expression enclosed in parentheses is the argument of the function, and it is evaluated and used as indicated by the function name. For example:

```
15      LET B=SQR(4+X↑3)
```

indicates that the expression (4 + X ↑3) is to be evaluated and then the square root taken.

5.1.1 The Integer Function (INT)

The INT function appears in algebraic notation as [X] and returns the greatest integer of X that is less than or equal to X. For example:

```
INT (2.35) = 2
INT (-2.35) = -3
INT (12) = 12
```

One use of this function is to round numbers to the nearest integer by asking for INT (X + .5). For example:

```
INT (2.9 + .5) = INT (3.4) = 3
```

rounds 2.9 to 3. Another use is to round to any specific number of decimal places. For example:

```
INT (X * 10 ↑2 + .5) / 10 ↑2
```

rounds X correct to two decimal places and

```
INT (X * 10 ↑ D + .5) /10 ↑ D
```

rounds X correct to D decimal places.

5.1.2 The Random Number Generating Function (RND)

The RND function produces random numbers between 0 and 1. This function is used to simulate events that happen in a somewhat random way. RND does not need an argument.

If we want the first 20 random numbers, we can write the program shown below and get 20 six-digit decimals.

```
10      FOR L=1 TO 20
20      PRINT RND,
30      NEXT L
40      END
RUN

RANDOM          13:24          20-OCT-69

 0.406533        0.88445        0.681969        0.939462        0.253358
 0.863799        0.880238       0.638311        0.602898        0.990032
 0.863799        0.897931       0.628126        0.613262        0.303217
 5.00548E-2      0.393226       0.680219        0.632246        0.668218
```

NOTE

This is a sample run of random numbers. The format of the PRINT statement is discussed in Chapter 6.

```
RUN

RANDOM          13:25          20-OCT-69

 0.406533        0.88445        0.681969        0.939462        0.253358
 0.863799
```

A second RUN gives exactly the same random numbers as the first RUN; this is done to facilitate the debugging of programs. If we want 20 random one-digit integers, we could change line 20 to read as follows:

```
20      PRINT INT (10*RND);
RUN
```

We would obtain the following:

```
RANDOM          13:26          20-OCT-69

 4              8              6              9              2
 8              8              6              6              9
 5              8              6              6              3
 Ø              3              6              6              6
```

To vary the type of random numbers (20 random numbers ranging from 1 to 9, inclusive), change line 20 as follows:

```
20      PRINT INT(9*RND +1);
RUN

RANDOM          13:28          20-OCT-69

 4  8  7  9  3  8  8  6  6  9  6  6  3  1  4  7  6  7
```

To obtain random numbers which are integers from 5 to 24, inclusive, change line 20 to the following:

```
20      PRINT INT(20*RND +5);
RUN

RANDOM          13:30          20-OCT-69

 13  22  18  23  10  22  22  17  17  24  16  22  17  17  11  6
 12  18  17  18
```

If random numbers are to be chosen from the A integers of which B is the smallest, call for INT (A*RND+B).

5.1.3 The RANDOMIZE Statement

As noted when we ran the first program of this chapter twice, we got the same numbers in the same oraer each time. However, we get a different set with the RANDOMIZE statement, as in the following program:

```
5       RANDOMIZE
10      FOR L=1 TO 20
20      PRINT INT(10*RND);
30      NEXT L
40      END
RUN

RNDNOS          13:32          20-OCT-69

 1  9  4  2  1  1  6  6  3  8  4  9  8  6  5  8  6  2  6  Ø

RUN

RNDNOS          13:33          20-OCT-69

 1  1  4  6  6  6  Ø  5  3  8  4  Ø  8  1  Ø  5  1  8  Ø  1
```

RANDOMIZE (RANDOM) resets the numbers in a random way. For example, if this is the first instruction in a program using random numbers, then repeated RUNs of the program produce different results. If the instruction is absent, then the official list of random numbers is obtained in the usual order. It is suggested that a simulated model should be debugged without this instruction so that one always obtains the same random numbers in test runs. After the program is debugged, and before starting production runs, you insert the following:

```
1       RANDOM
```

5.1.4 The Sign Function (SGN)

The SGN function is one which assigns the value 1 to any positive number, 0 to zero, and -1 to any negative number. Thus, SGN (7.23) = 1, SGN (0) = 0, and SGN (-.2387) = -1. For example, the following statement:

```
50      ON SGN(X)+2 GO TO 100,200,300
```

transfers to 100 if X < 0, to 200 if X = 0, and to 300 if X > 0.

5.1.5 The Define User Function (DEF) and Function End Statement (FNEND)

In addition to the 14 functions BASIC provides, you may define up to 26 functions of your own with the DEF statement. The name of the defined function must be three letters, the first two of which are FN, e.g., FNA, FNB, ..., FNZ. Each DEF statement introduces a single function. For example, if you repeatedly use the function $e^{-X^2} + 5$, introduce the function by the following:

```
30      DEF FNE(X)=EXP(-X↑2)+5
```

and call for various values of the function by FNE (.1), FNE (3.45), FNE (A+2), etc. This statement saves a great deal of time when you need values of the function for a number of different values of the variable.

The DEF statement may occur anywhere in the program, and the expression to the right of the equal sign may be any formula that fits on one line. It may include any combination of other functions, such as those defined by different DEF statements; it also can involve other variables besides those denoting the argument of the function.

As in the following example each defined function may have zero, one, two, or more numeric variables; string variables (refer to Chapter 8) are not allowed:

```
10      DEF FNB(X,Y)=3*X*Y-Y↑3
105     DEF FNC(X,Y,Z,W)=FNB(X,Y)/FNB(Z,W)
530     DEF FNA=3.1416*R↑2
```

In the definition of FNA, the current value of R is used when FNA occurs. Similarly, if FNR is defined by the following:

```
70      DEF FNR(X)=SOR(2+LOG(X)-EXP(Y*Z):(X+SIN(2*Z)))
```

you can ask for FNR(2.7), and give new values to Y and Z before the next use of FNR.

The method of having multiple line DEFs is illustrated by the "max" function shown below. Using this method, the possibility of using IF...THEN as part of the definition is a great help as shown in the following example:

```
10      DEF FNM(X,Y)
20      LET FNM=X
30      IF Y<=X THEN 50
40      LET FNM=Y
50      FNEND
```

The absence of the equals sign (=) in line 10 indicates that this is a multiple line DEF. In line 50, FNEND terminates the definition. The expression FNM, without an argument, serves as a temporary variable for the computation of the function value. The following example defines N-factorial:

```
10      DEF FNF(N)
20      LET FNF=1
30      FOR K=1 TO N
40      LET FNF=K*FNF
50      NEXT K
60      FNEND
```

Any variable which is not an argument of FN_ in a DEF loop has its current value in the program. Multiple line DEFs may not be nested and there must not be a transfer from inside the DEF to outside its range, or vice versa. GOSUB and RETURN statements (refer to Section 5.2) are not allowed in multiple line DEFs.

5.2 SUBROUTINES

When you have a procedure that is to be followed in several places in your program, the procedure may be written as a subroutine. A subroutine is a self-contained program which is incorporated into the main program at specified points. A subroutine differs from other control techniques in that the computer remembers where it was before it entered the subroutine, and it returns to the appropriate place in the main program after executing the subroutine.

5.2.1 GOSUB and RETURN Statements

Two new statements, GOSUB and RETURN, are required with subroutines. The subroutine is entered with a GOSUB statement which can appear at any place in the main program except within a multiple

line DEF. The GOSUB statement is similar to a GO TO statement; however, with a GOSUB statement, the computer remembers where it was prior to the transfer. Following is an example of the GOSUB statement:

```
90      GOSUB 210
```

where 210 is the line number of the first statement in the subroutine. The last line in the subroutine is a RETURN statement which directs the computer to the statement following the GOSUB from which it transferred. For example:

```
350     RETURN
```

returns to the next highest line number greater than the GOSUB call.

Subroutines may appear anywhere in the main program except within the range of a multiple line DEF. Care should be taken to make certain that the computer enters a subroutine only through a GOSUB statement and exits via a RETURN statement.

5.2.2 Example

A program for determining the greatest common divisor (GCD) of three integers, using the Euclidean Algorithm, illustrates the use of a subroutine. The first two numbers are selected in lines 30 and 40, and their GCD is determined in the subroutine, lines 200 through 310. The GCD just found is called X in line 60; the third number is called Y, in line 70; and the subroutine is entered from line 80 to find the GCD of these two numbers. This number is, of course, the greatest common divisor of the three given numbers and is printed out with them in line 90.

A GOSUB inside a subroutine to perform another subroutine is called a nested GOSUB. It is necessary to exit from a subroutine only with a RETURN statement. You may have several RETURNs in the subroutine, as long as exactly one of them will be used.

```
10      PRINT "A", "B", "C", "GCD"
20      READ A, B, C
30      LET X=A
40      LET Y=B
50      GOSUB 200
60      LET X=G
70      LET Y=C
80      GOSUB 200
90      PRINT A,B,C,G
100     GO TO 20
110     DATA 60,90,120
120     DATA 38456,64872,98765
130     DATA 32,384,72
200         LET Q=INT(X/Y)
```

(continued on next page)

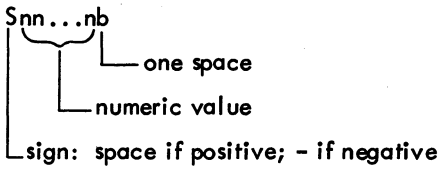
```
210           LET R=X-Q*Y
220           IF R=0 THEN 300
230           LET X=Y
240           LET Y=R
250           GO TO 200
300           LET G=Y
310           RETURN
320       END
RUN

GCD3NO          13:38          20-OCT-69

A               B              C               GCD

 60              90             120             30
 38456           64872          98765           1
 32              384            72              8

OUT OF DATA IN 20
```

CHAPTER 6
MORE SOPHISTICATED TECHNIQUES

The preceding chapters have covered the essential elements of BASIC. At this point, you are in a position to write BASIC programs and to input these programs to the computer via your Teletype. The commands and techniques discussed so far are sufficient for most programs. This chapter and remaining ones are for a programmer who wishes to perform more intricate manipulations and to express programs in a more sophisticated manner.

6.1 MORE ABOUT THE PRINT STATEMENT

The PRINT statement permits a greater flexibility for the more advanced programmer who wishes to have a different format for his output. BASIC normally outputs items from PRINT statements in the forms described in this chapter*. Numeric items are printed in the format:



String items (refer to Chapter 8) are printed exactly as they appear but without the enclosing quotes.

The Teletype line is divided into zones of 14 spaces each. A comma in a PRINT statement is a signal to the Teletype to move to the next print zone on the current line or, if necessary, to the beginning of the first print zone of the next line. A semicolon in a PRINT statement causes no motion of the Teletype. <PA> (page) in a PRINT statement moves the Teletype to the beginning of the first print zone of the first line on the next page of output. Commas, semicolons, and <PA> delimiters can appear in PRINT statements without intervening data items. Each delimiter causes Teletype movement as previously described. For example, PRINT A,,B causes the value of A to be printed in the first zone,

*This chapter describes the noquote mode of output. The user can explicitly change the mode to quote mode by using a QUOTE statement. Refer to Chapter 10 for the description of quote and noquote modes and their associated statements.

the Teletype to be moved to the third zone, and the value of B to be printed in the third zone. If two items in a PRINT statement are clearly distinct, the separating commas, semicolons, or <PA> delimiters can be omitted and the items are treated as though they were separated by one semicolon.

When you type in the following program:

```
10      FOR I=1 TO 15
20      PRINT I
30      NEXT I
40      END
```

the Teletype prints 1 at the beginning of a line, 2 at the beginning of the next line, and, finally, 15 on the fifteenth line. But, by changing line 20 to read as follows:

```
20      PRINT I,
```

the numbers are printed in the zones, reading as follows:

```
1               2               3               4               5
6               7               8               9               10
11              12              13              14              15
```

If you want the numbers printed in this fashion, but compressed, change line 20 by replacing the comma with a semicolon as in the following example:

```
20      PRINT I;
```

The following results are printed:

```
1  2  3  4  5  6  7  8  9  10  11  12  13  14  15
```

The end of a PRINT statement signals a new line, unless a comma or semicolon is the last symbol. Thus, the following instruction:

```
50      PRINT X, Y
```

prints two numbers and then returns to the next line, while the instruction:

```
50      PRINT X, Y,
```

prints these two values and does not return. The next number to be printed appears in the third zone, after the values of X and Y in the first two zones.

Since the end of a PRINT statement signals a new line,

```
250     PRINT
```

causes the Teletype to advance the paper one line, to put a blank line for vertical spacing of your results, or to complete a partially filled line.

```
50      FOR M=1 TO N
110     FOR J=0 TO M
120     PRINT B(M,J);
130     NEXT J
140     PRINT
150     NEXT M
```

This program prints B(1,0) and next to it B(1,1). Without line 140, the Teletype would go on printing B(2,0), B(2,1), and B(2,2) on the same line, and then B(3,0), B(3,1), etc. After the Teletype prints the B(1,1) value corresponding to M = 1, line 140 directs it to start a new line; after printing the value of B(2,2) corresponding to M = 2, line 140 directs it to start another new line, etc.

The following instructions:

```
50      PRINT "TIME-"; "SHAR"; "ING";
51      PRINT " ON"; " THE "; "PDP-10"
```

cause the printing of the following:

```
TIME-SHARING ON THE PDP-10
```

(The items enclosed in quotes in statements 50 and 51 are strings.)

The following instructions:

```
10      N=5
20      PRINT "END OF PAGE" N <PA>
30      PRINT "ITEM",,"NO. ORDERED",,"TOTAL PRICE"
```

cause the printing of

```
END OF PAGE 5
```

followed by a form-feed to position the Teletype paper at the top of a new page, where the following is printed:

```
ITEM                        NO. ORDERED                    TOTAL PRICE
```

Formatting of output can be controlled even further by means of the TAB function, in the form TAB(n), where n is the desired print position. TAB can contain any numeric formula as its argument. The value of the numeric formula is computed and then truncated to an integer. This integer is treated modulo the current output right margin. Setting the output right margin is described in Section 6.7. For example, if the output right margin is 72, which is the default margin, a value in the range 0 through 71 is obtained. The first print position on the line is column 0. Thus, TAB(17) causes the Teletype to

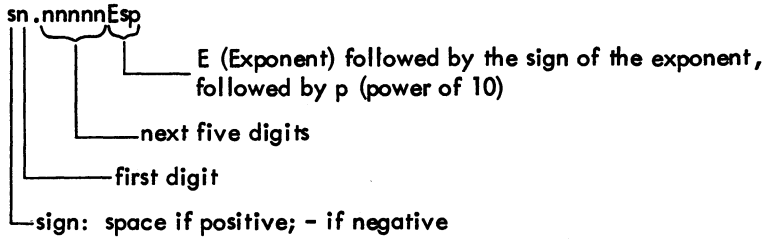
move to column 17 (unless it has already passed this position, in which case the TAB is ignored). For example, inserting the following line in a loop

```
55      PRINT X; TAB(12); Y; TAB(27);Z
```

causes the X values to start in column 0, the Y values in column 12, and the Z values in column 27.

The following rules are used to interpret the printed results:

a. If a number is an integer, the decimal point is not printed. If the integer contains more than eight digits, it is printed in the format as follows.



For example, 32,437,580,259 is written as 3.24376E+10.

b. For any decimal number, no more than six significant digits are printed.

c. For a number less than 0.1, the E notation is used, unless the entire significant part of the number can be printed as a 6-digit decimal number. Thus, 0.03456 indicates that the number is exactly .0345600000, while 3.45600E-2 indicates that the number has been rounded to .0345600.

d. Trailing zeros after the decimal point are not printed.

The following program, in which powers of 2 are printed out, demonstrates how numbers are printed.

```
10      FOR N=-5 TO 30
20      PRINT 2↑N;
30      NEXT N
40      END

POWERS          11:54          20-OCT-69

 0.03125  0.0625  0.125  0.25  0.5  1  2  4  8  16  32  64  128  256
 512  1024  2048  4096  8192  16384  32768  65536  131072  262144
 524288  1048576  2097152  4194304  8388608  16777216  33554432
 67108864  1.324218E+8  2.68435E+8  5.36871E+8  1.07374E+9
```

6.2 INPUT STATEMENT

At times, during the running of a program, it is desirable to have data entered. This is particularly true when one person writes the program and saves it on the storage device as a library program (refer to SAVE command, Chapter 9), and other persons use the program and supply their own data. Data may be entered by an INPUT statement, which acts as a READ but accepts numbers of alphanumeric data from the Teletype keyboard. For example, to supply values for X and Y into a program, type the following:

```
40      INPUT X,Y
```

prior to the first statement which uses either of these numbers. When BASIC encounters this statement, it types a question mark. The user types two numbers, separated by a comma, and presses the RETURN key, and BASIC continues the program. No number can be longer than 8 digits.

Frequently, an INPUT statement is combined with a PRINT statement to make sure that the user knows what the question mark is asking for. You might type in the following statement:

```
20      PRINT "YOUR VALUES OF X,Y, AND Z ARE";
30      INPUT X,Y,Z
```

and BASIC types out the following:

```
YOUR VALUE OF X,Y, AND Z ARE?
```

Without the semicolon at the end of line 20, the question mark would have been printed on the next line. Data entered via an INPUT statement is not saved with the program. Therefore, INPUT should be used only when small amounts of data are to be entered, or when necessary during the running of the program.

6.3 STOP STATEMENT

STOP is equivalent to GO TO xxxxx, where xxxxx is the line number of the END statement in the program. For example, the following two program portions are exactly equivalent:

```
250     GO TO 999           250     STOP
        .........                   ....
340     GO TO 999           340     STOP
        .........                   ....
999     END                 999     END
```

6.4 REMARKS STATEMENT (REM)

REM provides a means for inserting explanatory remarks in the program. BASIC completely ignores the remainder of that line, allowing you to follow the REM with directions for using the program, with identifications of the parts of a long program, or with any other information. Although what follows REM is ignored, its line number may be used in a GO TO or IF-THEN statement as in the following:

```
100     REM INSERT IN LINES 900-998.  THE FIRST
110     REM NUMBER IS N, THE NUMBER OF POINTS.  THEN
120     REM THE DATA POINTS THEMSELVES ARE ENTERED, BY
200     REM THIS IS A SUBROUTINE FOR SOLVING EQUATIONS

        ......
300     RETURN
        .......
520     GOSUB 200
```

A second method for adding comments to a program consists of placing an apostrophe (') at the end of the line, and following it by a remark. Everything following the apostrophe is ignored. This method cannot be used in an image statement. Image statements are described in Chapter 11. Apostrophes within string constants are not treated as remark characters.

6.5 RESTORE STATEMENT

The RESTORE statement permits READing the data in the DATA statements of a program more than once. Whenever RESTORE is encountered in a program, BASIC restores the data block pointer to the first number. A subsequent READ statement then starts reading the data all over again. However, if the desired data is preceded by code numbers or parameters, superfluous READ statements should be used to pass over these numbers. As an example, the following program portion reads the data, restores the data block to its original state, and reads the data again. Note the use of line 570 (READ X) to pass over the value of N, which is already known.

```
100       READ N
110       FOR I=1 TO N
120       READ X
          ......

200       NEXT I
          ......
560       RESTORE
570       READ X
580       FOR I=1 TO N
590       READ X
          ......
700       DATA .....
710       DATA .....
```

6.6 CHAIN STATEMENT

The CHAIN statement provides a means for one program to call another program so that programs can be written separately and executed together in a chain. The CHAIN statement has one of the forms:

```
     CHAIN [alphabetic string]
or   CHAIN [alphabetic string], [numeric formula]
```

The alphabetic string is either: a) the name of the program being chained to, in the form device:filename.ext (optionally enclosed in quotes), or b) a string variable* that has as its value the name of the program being chained to, in the form device:filename.ext. The device and the extension

*A string variable is a variable that is used to store an alphabetic string. A string variable is composed of a letter and a dollar sign ($) or a letter, a number, and a dollar sign ($), e.g., A$ or B2$. String variables are described in Chapter 8.

can be omitted, but the filename must be present. If the device is omitted, DSK: is assumed; if the extension is omitted, .BAS is assumed.

The numeric formula specifies a line number in the program being chained to; its value is truncated to an integer.

A few examples of the CHAIN statement are:

```
CHAIN A$
CHAIN B2$, N*EXP(W)
CHAIN PTR:MAIN, 50
```

When BASIC encounters a CHAIN statement in a program, it stops execution of that program, retrieves the program named in the CHAIN statement from the specified device, compiles the chained program, and begins execution either at the line number specified in the CHAIN statement or at the beginning of the program if no line number was specified. Only the heading of the first program in the chain is printed, and the TIME: message is printed only after the last program in the chain has been executed. Error messages for the programs in the chain, excluding the first program, have the name of the program appended. For example:

```
OVERFLOW IN 1100 IN TEST4.BAK
```

indicates that an overflow error occurred in line 1100 in the chained program TEST4.BAK. Programs that run individually, or the first program in a chain will not have the program name appended.

The following is an example of program chaining.

```
LIST

PROG3      12:05      25-JAN-71

10      PRINT 10
11      STOP
20      PRINT 20
21      END

READY
SAVE
NEW
NEW FILENAME -- PROG2
READY
10      INPUT N
20      CHAIN PROG3, N
30      END
RUNNH
?10
 10
TIME: 0.02 SECS
```

6.7 MARGIN STATEMENT

Normally, the right margin for output to the Teletype is 72 characters. The MARGIN statement allows the user to specify a right margin of 1 to 132 characters. This margin becomes effective on the first new line of output after the MARGIN statement, and remains in effect until the next time the margin is set by a MARGIN statement or until the end of the program's execution, whichever is sooner. At the end of program execution, the output margin is reset to 72 characters.

The form of the margin statement is:

MARGIN [numeric formula]

The numeric formula is a numeric constant, variable, or expression that specifies the right margin; it is truncated to an integer before the margin is set. Some examples of the MARGIN statement are:

```
MARGIN   75
MARGIN   132*N
```

The right margin for input from the Teletype is not affected by MARGIN statements; it is always 142 characters. Lines of input that are longer than 142 characters will result in error messages.

The monitor, as well as BASIC, considers the normal Teletype output margin to be 72 characters. Therefore, when a margin greater than 72 characters is needed, the monitor command SET TTY WIDTH must be used in addition to the BASIC MARGIN statement. Otherwise, the monitor will output a leading carriage return-line feed if an attempt is made to output a seventy-third character on a line. Before the program is run, the user must twice press the CTRL and C keys simultaneously (two CONTROL-C's) and then type:

```
SET TTY WIDTH 132
REENTER
```

to reenter BASIC. The monitor will not output its carriage return-line feed until after the 132nd character on a line; consequently, BASIC can control the margin as the MARGIN statements specify without interference from the monitor. The SET TTY WIDTH monitor command is implemented in 5.02 and later monitors.

6.8 PAGE STATEMENT

Normally, output to the Teletype is not divided into pages. The PAGE statement allows the user to set a page size of any positive number of lines. This page size remains in effect until the page size is set again by a PAGE statement, or until the Teletype is set back into nopage mode by a NOPAGE statement (described in Section 6.9), or until the end of the program's execution. At the end of program execution, the Teletype is reset to nopage mode.

The form of the PAGE statement is:

PAGE [numeric formula]

The numeric formula specifies the page size; it is truncated to an integer before the page size is set.

When a PAGE statement is executed, BASIC ends the current output line (if necessary), outputs a form-feed to position the Teletype paper at the top of the next page, and starts counting lines beginning with the next line of output. As soon as a new page is necessary, a form-feed is output. Whenever a PRINT statement containing <PA> is executed, the line count for the Teletype page is set back to zero.

6.9 NOPAGE STATEMENT

The NOPAGE statement sets the Teletype back to nopage mode (i.e., the output to the Teletype is no longer automatically divided into pages). The NOPAGE statement need only be used to change the mode back from page mode (set by a PAGE statement) because the default is nopage mode for all Teletype output. The form of the statement is:

NOPAGE

The NOPAGE statement has no effect on the execution of <PA> delimiters in PRINT statements; they are executed as usual.

CHAPTER 7
VECTORS AND MATRICES

Operations on lists and tables occur frequently; therefore, a special set of 13 instructions for matrix computations, all of which are identified by the starting word MAT, is used. These instructions are not necessary and can be replaced by combinations of other BASIC instructions, but use of the MAT instructions results in shorter programs that run much faster.

The MAT instructions are as follows:

MAT READ a, b, c	Read the three matrices, their dimensions having been previously specified.
MAT c = ZER	Fill out c with zeros.
MAT c = CON	Fill out c with ones.
MAT c = IDN	Set up c as an identity matrix.
MAT PRINT a, b, c	Print the three matrices. (Semicolons can be used immediately following any matrix which you wish to have printed in a closely packed format.)
MAT INPUT v	Call for the input of a vector.
MAT b = a	Set the matrix b equal to the matrix a.
MAT c = a + b	Add the two matrices a and b.
MAT c = a - b	Subtract the matrix b from the matrix a.
MAT c = a * b	Multiply the matrix a by the matrix b.
MAT c = TRN(a)	Transpose the matrix a.
MAT c = (k) * a	Multiply the matrix a by the number k. The number, which must be in a parentheses, may also be given by a formula.
MAT c = INV (a)	Invert the matrix a.

7.1 MAT INSTRUCTION CONVENTIONS

The following convention has been adopted for MAT instructions: while every vector has a component 0, and every matrix has a row 0 and a column 0, the MAT instructions ignore these. Thus, if we have a matrix of dimension M-by-N in a MAT instruction, the rows are numbered 1, 2, ..., M, and the columns 1, 2, ..., N.

If a numeric array is referenced in a MAT statement other than MAT INPUT BASIC sets up the array as a matrix with two dimensions unless the user has specifically declared in a DIM statement that the array is a vector.

The DIM statement may simply indicate what the maximum dimension is to be. Thus, if we write the following:

```
DIM M(20,35)
```

M may have up to 20 rows and up to 35 columns. This statement is written to reserve enough space for the matrix; consequently, the only concern at this point is that the dimensions declared are large enough to accommodate the matrix. However, in the absence of DIM statements, all vectors may have up to 10 components and matrices up to 10 rows and 10 columns. This is to say that in the absence of DIM statements, this much space is automatically reserved for vectors and matrices on their appearance in the program. The actual dimension of a matrix may be determined either when it is first set up (by a DIM statement) or when it is computed. Thus the following

```
10      DIM M(20,7)
        -----
50      MAT READ M
```

reads a 20-by-7 matrix for M, while the following:

```
50      MAT READ M(17,30)
```

reads a 17-by-30 matrix for M, provided sufficient space has been saved for it by writing

```
10      DIM M(20,35)
```

7.2 MAT C = ZER, MAT C = CON, MAT C = IDN

The following three instructions:

```
MAT M = ZER     (sets up matrix M with all components equal to zero)
MAT M = CON     (sets up matrix M with all components equal to one)
MAT M = IDN     (sets up matrix M as an identity matrix)
```

act like MAT READ as far as the dimension of the resulting matrix is concerned. For example,

```
MAT M = CON (7,3)
```

sets up a 7-by-3 matrix with 1 in every component, while in the following:

```
MAT M = CON
```

sets up a matrix, with ones in every component, and a 10-by-10 dimension (unless previously given other dimensions). It should be noted, however, that these instructions have no effect on row and column zero. Thus, the following instructions:

```
10      DIM M(20,7)
20      MAT READ M(7,3)
........
35      MAT M=CON
70      MAT M=ZER(15,7)
........
90      MAT M=ZER(16,10)
```

first read in a 7-by-3 matrix for M. Then they set up a 7-by-3 matrix of all 1s for M (the actual dimension having been set up as 7-by-3 in line 20). Next they set up M as a 15-by-7 all-zero matrix. (Note that although this is larger than the previous M, it is within the limits set in 10.) An error message results because of line 90. The limit set in line 10 is (20 + 1) x (7 + 1) = 168 components, and in 90 we are calling for (16 + 1) x (10 + 1) = 187 components. Thus, although the zero rows and columns are ignored in MAT instructions, they play a role in determining dimension limits. For example,

```
90      MAT M=ZER(25,5)
```

would not yield an error message.

Perhaps it should be noted that an instruction such as MAT READ M(2,2) which sets up a matrix and which, as previously mentioned, ignores the zero row and column, does, however, affect the zero row and column. The redimensioning which may be implicit in an instruction causes the relocation of some numbers; therefore, they may not appear subsequently in the same place. Thus, even if we have first LET M(1,0) = M(2,0) = 1, and then MAT READ M(2,2), the values of M(1,0) and M(2,0) now are 0. Thus when using MAT instructions, it is best not to use row and column zero.

7.3 MAT PRINT A, B, C

The following instruction:

```
MAT PRINT A, B; C
```

causes the three matrices to be printed with A and C in the normal format (i.e., with five components to a line and each new row starting on a new line) and B closely packed.

Vectors may be used in place of matrices, as long as the above rules are observed. Since a vector like V(I) is treated as a column vector by BASIC, a row vector has to be introduced as a matrix that has only one row, namely row 1. Thus,

```
DIM X(7), Y(0,5)
```

introduces a 7-component column vector and a 5-component row vector.

If V is a vector, then

```
MAT PRINT V
```

prints the vector V as a column vector.

```
MAT PRINT V,
```

prints V as a row vector, five numbers to the line, while

```
MAT PRINT V;
```

prints V as a row vector, closely packed.

7.4 MAT INPUT V AND THE NUM FUNCTION

The following instruction:

```
MAT INPUT V
```

calls for the input of a vector. The number of components in the vector need not be specified. Normally, the input is limited by its having to be typed on one line. However, by ending the line of input with an ampersand (&) before the carriage return, the machine asks for more input on the next line. There must be at least one data item preceding the ampersand on the line or an error message will be issued. Note that, although the number of components need not be specified, if we wish to input more than 10 numbers, we must save sufficient space with a DIM statement. After the input, the function NUM equals the number of components, and V(1), V(2), ..., V(NUM) become the numbers that are input, allowing variable length input. For example,

```
5       LET S=0
10      MAT INPUT V
20      LET N=NUM
30      IF N=0 THEN 99
40      FOR I=1 TO N
45      LET S=S+V(I)
50      NEXT I
60      PRINT S/N
70      GO TO 5
99      END
```

allows the user to type in sets of numbers, which are averaged. The program takes advantage of the fact that zero numbers may be input, and it uses this as a signal to stop. Thus, the user can stop by simply pushing RETURN on an input request. If an ampersand is used, it need only be preceded by a comma when the item immediately preceding it is an unquoted string.

7.5 MAT B = A

This instruction sets up B to be the same as A and, in doing so, dimensions B to be the same as A, provided that sufficient space has been saved for B.

7.6 MAT C = A + B AND MAT C = A - B

For these instructions to be legal, A and B must have the same dimensions, and enough space must be saved for C. These statements cause C to assume the same dimensions as A and B. Instructions such as MAT A = A ± B are legal; the indicated operation is performed and the answer stored in A. Only a single arithmetic operation is allowed; therefore, MAT D = A + B - C is illegal but may be achieved with two MAT instructions.

7.7 MAT C = A * B

For this instruction to be legal, it is necessary that the number of columns in A be equal to the number of rows in B. For example, if matrix A has dimension L-by-M and matrix B has dimension M-by-N, then C = A * B has dimension L-by-N. It should be noted that while MAT A = A + B may be legal, MAT A = A * B is self-destructive because, in multiplying two matrices, we destroy components which would be needed to complete the computation. MAT B = A * A is, of course, legal provided that A is a "square" matrix.

7.8 MAT C = TRN(A)

This instruction lets C be the transpose of the matrix A. Thus, if matrix A is an M-by-N matrix, C is an N-by-M matrix. The instruction MAT C = TRN (C) is legal.

7.9 MAT C = (K) * A

This instruction allows C to be the matrix A multiplied by the number K (i.e., each component of A is multiplied by K to form the components of C). The number K, which must be in parentheses, may be replaced by a formula. MAT A = (K) * A is legal.

7.10 MAT C = INV(A) AND THE DET FUNCTION

This instruction allows C to be the inverse of A. (A must be a "square" matrix.) The function DET is available after the execution of the inversion, and it will equal the determinant of A. Consequently, the user can obtain the determinant of a matrix by inverting the matrix and then noting what value DET has. If the determinant of a matrix is zero, the matrix is singular and its inverse is meaningless. When an attempt is made to invert a matrix whose determinant is nearly zero, the warning message

```
NEARLY SINGULAR MATRIX INVERTED IN nn
```

is printed, DET is set equal to zero, and the program execution continues.

7.11 EXAMPLES OF MATRIX PROGRAMS

The first example reads in A and B in line 30 and, in so doing, sets up the correct dimensions. Then, in line 40, A + A is computed and the answer is called C. This automatically dimensions C to be the same as A. Note that the data in line 90 results in A being 2-by-3 and in B being 3-by-3. Both MAT PRINT formats are illustrated, and one method of labeling a matrix print is shown.

```
10      DIM A(20,20), B(20,20), C(20,20)
20      READ M,N
30      MAT READ A(M,N),B(N,N)
40      MAT C=A+A
50      MAT PRINT C;
60      MAT C=A*B
70      PRINT
75      PRINT "A*B=",
80      MAT PRINT C
90      DATA 2,3
91      DATA 1,2,3
92      DATA 4,5,6
93      DATA 1,0,-1
94      DATA 0,-1,-1
95      DATA -1,0,0
99      END
RUN

MATRIX          08:31          09-MAR-71


 2   4   6

 8   10  12

A*B=

-2              -2              -3

-2              -5              -9


TIME:  0.13 SECS.
```

The second example inverts an n-by-n Hilbert matrix:

1	1/2	1/3 . . .	1/n
1/2	1/3	1/4 . . .	1/n + 1
1/3	1/4	1/5 . . .	1/n + 2
.	.	. . . .	
.	.	. . . .	
.	.	. . . .	
1/n	1/n + 1	1/n + 2	1/2n-1

Ordinary BASIC instructions are used to set up the matrix in lines 50 to 90. Note that this occurs after correct dimensions have been declared. A single instruction then results in the computation of the inverse, and one more instruction prints it. Because the function DET is available after an inversion, it is taken advantage of in line 130, and is used to print the value of the determinant of A. In this example, we have supplied 4 for N in the DATA statement and have made a run for this case:

```
5       REM THIS PROGRAM INVERTS AN N-BY-N HILBERT MATRIX
10      DIM A(20,20), B(20,20)
20      READ N
30      MAT A=CON(N,N)
50      FOR I=1 TO N
60      FOR J=1 TO N
70      LET A(I,J)=1/(I+J-1)
80      NEXT J
90      NEXT I
100     MAT B=INV(A)
115     PRINT "INV(A)="
120     MAT PRINT B
125     PRINT
130     PRINT "DETERMINANT OF A=" DET
190     DATA 4
199     END
RUN

HILMAT          13:52           20-OCT-69

INV(A)=

 16.0001       -120.001         240.003        -140.002
-120.001        1200.01        -2700.03         1680.02
 240.003       -2700.03         6480.08        -4200.05
-140.002        1680.02        -4200.05         2800.03

DETERMINANT OF A=1.65342E-7
```

A 20-by-20 matrix is inverted in about 0.5 seconds. However, the reader is warned that beyond n = 7, the Hilbert matrix cannot be inverted because of severe round-off errors.

7.12 SIMULATION OF N-DIMENSIONAL ARRAYS

Although it is not possible to create n-dimensional arrays in BASIC, the method outlined below does simulate them. The example is of a three-dimensional array, but it has been written in such a way

that it could be easily changed to four dimensions or higher. We use the fact that functions can have any number of variables, and we set up a 1-to-1 correspondence between the components of the array and the components of a vector which equals the product of the dimensions of the array. For example, if the array has dimensions 2, 3, 5, then the vector has 30 components. A multiple line DEF could be used in place of the simple DEF in line 30 if the user wished to include error messages. The printout is in the form of two 3-by-5 matrices.

```
10      DIM V(1000)
20      MAT READ D(3)
30      DEF FNA(I,J,K)=(I-1)*D(2)+(J-1))*D(3)+K
50      FOR I=1 TO D(1)
55      FOR J=1 TO D(2)
60      FOR K=1 TO D(3)
80      LET V(FNA(I,J,K))=I+2*J+K↑2
90      PRINT V(FNA(I,J,K)),
100     NEXT K
110     NEXT J
112     PRINT
115     PRINT
120     NEXT I
900     DATA 2,3,5
999     END
RUN

3ARRAY          08:07           27-OCT-69
 4               7               12              19              28
 6               9               14              21              30
 8               11              16              23              32

 5               8               13              20              29
 7               10              15              22              31
 9               12              17              24              33
```

CHAPTER 8
ALPHANUMERIC INFORMATION (STRINGS)

In previous chapters, we have dealt only with numerical information. However, BASIC also processes alphanumeric information in the form of strings. A string is a sequence of characters, each of which is a letter, a digit, a space, or some other character. A string, however, cannot contain a character that is a line terminator (i.e., a line feed, form feed, or vertical tab), or a carriage return.

String constants are normally enclosed in quotes (e.g., "TOTAL VALUE"). In some cases in some statements, the quotes can be omitted. Where this is allowed, it is explicitly stated in the description of the particular type of statement found elsewhere in this manual.

Variables may be introduced for simple strings and string vectors, but not for string matrices. Any simple variable, followed by a dollar sign ($), stands for a string; e.g., A$ and C7$. A vector variable, followed by $, denotes a list of strings; e.g., V$(n) or A2$(n), where n is the nth string in the list. For example, V$(7) is the seventh string in the list V.

8.1 READING AND PRINTING STRINGS

Strings may be read and printed. For example:

```
10      READ A$, B$, C$
20      PRINT C$; B$; A$
30      DATA ING,SHAR,TIME-
40      END
```

causes TIME-SHARING to be printed. The effect of the semicolon in the PRINT statement is consistent with that discussed in Chapter 6; i.e., it causes output of the alphanumeric items in a close-packed form. Commas, <PA> delimiters, and TABs may be used as in any other PRINT statement. The loop:

```
70      FOR I=1 TO 12
80      READ M$(I)
90      NEXT I
```

reads a list of 12 strings.

In place of the READ and PRINT, corresponding MAT instructions may be used for lists. For example, MAT PRINT M$; causes the members of the list to be printed without spaces between them. We may also use INPUT or MAT INPUT. After a MAT INPUT, the function NUM equals the number of strings inputted. When using the MAT INPUT statement, you can continue inputting strings on the next line by typing an ampersand (&) on the current line immediately before pressing the RETURN key. A comma must precede the ampersand if the string immediately before the ampersand is unquoted. If the string is unquoted and a comma does not separate the string from the ampersand, the ampersand will be treated as part of the string. Thus, either MARY,& or "MARY"& is legal input.

As usual, lists are assumed to have no more than 10 elements; otherwise, a DIM statement is required. The following statement:

```
10      DIM M$(20)
```

saves space for 20 strings in the M$ list.

In the DATA statements, numbers and strings may be intermixed. Numbers are assigned only to numerical variables, and strings only to string variables. Strings in DATA statements are recognized by the fact that they start with a letter. If they do not, they must be enclosed in quotes. The same requirement holds for a string containing a comma. For example:

```
90      DATA 10,ABC,5,"4FG",SEPT. 22," 1968",2
```

The only convention on INPUT and MAT INPUT is that a string containing a comma must be enclosed in quotes. The following example shows the correct format for a response to a MAT INPUT:

```
MR. JONES, "146 MAIN ST., MAYNARD, MASS."
```

8.2 STRING CONVENTIONS

In every method of inputting string information into a program (DATA, INPUT, MAT INPUT, etc.), leading blanks are ignored unless the string, including the blanks, is enclosed in quotes. String constants (which must be enclosed in quotes) or string variables may occur in LET and IF-THEN statements. The following two examples are self-explanatory:

```
10      LET Y$="YES"
20      IF A7$="YES" THEN 200
```

The relation "<" is interpreted as "earlier in alphabetic order." The other relational symbols work in a similar manner. In any comparison, trailing blanks in a string are ignored, as in the following:

```
"YES" = "YES  "
```

We illustrate these possibilities by the following program, which reads a list of strings and alphabetizes them:

```
10      DIM L$(50)
20      READ N
30      MAT READ L$(N)
40      FOR I=1 TO N
50      FOR J=1 TO N-I
60      IF L$(J) < L$(J+1) THEN 100
70      LET A$=L$(J)
80      LET L$(J)=L$(J+1)
90      LET L$(J+1)=A$
100     NEXT J
110     NEXT I
120     MAT PRINT L$
900     DATA 5,ONE,TWO,THREE,FOUR,FIVE
999     END
```

Omitting the $ signs in this program serves to read a list of numbers and to print them in increasing order.

A rather common use is illustrated by the following:

```
330     PRINT "DO YOU WISH TO CONTINUE";
340     INPUT A$
350     IF A$="YES" THEN 10
360     STOP
```

8.3 NUMERIC AND STRING DATA BLOCKS

Numeric and string data are kept in two separate blocks, and these act independently of each other. The RESTORE statement resets both the data pointers for the numerical data and string data back to the beginning of their blocks. RESTORE* resets the pointer only for the numerical data and RESTORE $ only for the string data.

8.4 THE CHANGE STATEMENT

In BASIC, it is very easy to obtain the individual digits in a number by using the function INT. One way to obtain the individual characters in a string is with the instruction CHANGE. The use of CHANGE is best illustrated with the following examples.

```
5     DIM A(65)
10    READ A$
15    CHANGE A$ TO A
20    FOR I=0 TO A(0)
25    PRINT A(I);
35    NEXT I
40    DATA ABCDEFGHIJKLMNOPQRSTUVWXYZ
45    END
RUN

CHANGE          13:55          20-OCT-69

 26  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79
 80  81  82  83  84  85  86  87  88  89  90
```

In line 15, the instruction CHANGE A$ TO A has caused the vector A to have as its zero component the number of characters in the string A$ and, also, to have certain numbers in the other components. These numbers are the American Standard Code for Information Interchange (ASCII) numbers for the characters appearing in the string (e.g., A(1) is 65 - the ASCII number for A).

Table 8-1 lists the ASCII numbers for printing and nonprinting characters. Note that the nonprinting characters are shown in the table as codes containing two or three letters. These codes are not output; the actual meaning of the ASCII number is output (e.g., 7 causes the bell to ring, it does not print BEL).

Table 8-1
ASCII Numbers and Equivalent Characters

ASCII Decimal Number	Character	Meaning	ASCII Decimal Number	Character	Meaning
0	NUL	Null	14	SO	Shift out
1	SOH	Start of heading	15	SI	Shift in
2	STX	Start	16	DLE	Data link escape
3	ETX	End of text	17	DC1	Device control 1
4	EOT	End of transmission	18	DC2	Device control 2
5	ENQ	Enquiry	19	DC3	Device control 3
6	ACK	Acknowledge	20	DC4	Device control 4
7	BEL	Bell	21	NAK	Negative acknowledgement
8	BS	Backspace	22	SYN	Synchronous idle
9	HT	Horizontal tab	23	ETB	End of transmission block
10	LF	Line feed	24	CAN	Cancel
11	VT	Vertical tab	25	EM	End of medium
12	FF	Form feed	26	SUB	Substitute
13	CR	Carriage return	27	ESC	Escape

Note: Recall that line feed (LF), form feed (FF), vertical tab (VT), and carriage return (CR) are illegal in strings.

(continued on next page)

Table 8-1 (Cont)
ASCII Numbers and Equivalent Characters

ASCII Decimal Number	Character	Meaning	ASCII Decimal Number	Character	Meaning
28	FS	File separator	72	H	Upper case H
29	GS	Group separator	73	I	Upper case I
30	RS	Record separator	74	J	Upper case J
31	US	Unit separator	75	K	Upper case K
32	SP	Space or blank	76	L	Upper case L
33	!	Exclamation mark	77	M	Upper case M
34	"	Quotation mark	78	N	Upper case N
35	#	Number sign	79	O	Upper case O
36	$	Dollar sign	80	P	Upper case P
37	%	Percent sign	81	Q	Upper case Q
38	&	Ampersand	82	R	Upper case R
39	'	Apostrophe	83	S	Upper case S
40	(	Left parenthesis	84	T	Upper case T
41	)	Right parenthesis	85	U	Upper case U
42	*	Asterisk	86	V	Upper case V
43	+	Plus sign	87	W	Upper case W
44	,	Comma	88	X	Upper case X
45	-	Minus sign or hyphen	89	Y	Upper case Y
46	.	Period or decimal point	90	Z	Upper case Z
47	/	Slash	91	[	Left square bracket
48	0	Zero	92	\	Back slash
49	1	One	93	]	Right square bracket
50	2	Two	94	^ or ↑	Circumflex or up arrow
51	3	Three	95	← or _	Back arrow or underscore
52	4	Four	96	`	Grave accent
53	5	Five	97	a	Lower case a
54	6	Six	98	b	Lower case b
55	7	Seven	99	c	Lower case c
56	8	Eight	100	d	Lower case d
57	9	Nine	101	e	Lower case e
58	:	Colon	102	f	Lower case f
59	;	Semicolon	103	g	Lower case g
60	<	Left angle bracket	104	h	Lower case h
61	=	Equal sign	105	i	Lower case i
62	>	Right angle bracket	106	j	Lower case j
63	?	Question mark	107	k	Lower case k
64	@	At sign	108	l	Lower case l
65	A	Upper case A	109	m	Lower case m
66	B	Upper case B	110	n	Lower case n
67	C	Upper case C	111	o	Lower case o
68	D	Upper case D	112	p	Lower case p
69	E	Upper case E	113	q	Lower case q
70	F	Upper case F	114	r	Lower case r
71	G	Upper case G	115	s	Lower case s

(continued on next page)

Table 8-1 (Cont)
ASCII Numbers and Equivalent Characters

ASCII Decimal Number	Character	Meaning	ASCII Decimal Number	Character	Meaning
116	t	Lower case t	122	z	Lower case z
117	u	Lower case u	123	{	Left brace
118	v	Lower case v	124	|	Vertical line
119	w	Lower case w	125	}	Right brace
120	x	Lower case x	126	~	Tilde
121	y	Lower case y	127	DEL	Delete

The other use of CHANGE is illustrated by the following:

```
10      FOR I=0 TO 5
15      READ A(I)
20      NEXT I
25      DATA 5,65,66,67,68,69
30      CHANGE A TO A$
35      PRINT A$
40      END
```

This program prints ABCDE because the numbers 65 through 69 are the code numbers for A through E. Before CHANGE is used in the vector-to-string direction, we must give the number of characters which are to be in the string as the zero component of the vector. In line 15, A(0) is read as 5. The following is a final example:

```
5       DIM V(128)
10      PRINT "WHAT DO YOU WANT THE VECTOR V TO BE";
20      MAT INPUT V
30      LET V(0)=NUM
35      IF NUM=0 THEN 70
40      CHANGE V TO A$
50      PRINT A$
60      GO TO 10
70      END
RUN

EXAMPLE         13:59           20-OCT-69

WHAT DO YOU WANT THE V TO BE? 40,45,60,45,89,90
(-<-YZ
WHAT DO YOU WANT THE VECTOR V TO BE? 33,34,35,36,37,38,39,40,41,42 &
? 43,44,45,46,47,48,49,50
!"#$%&'()*+,-./012
WHAT DO YOU WANT THE VECTOR V TO BE?
TIME: 0.10 SECS.
```

Note that in this example we have used the availability of the function NUM after a MAT INPUT to find the number of characters in the string which is to result from line 40.

8.5 STRING CONCATENATION

Strings can be concatenated by means of the plus sign operator (+). The plus sign can be used to concatenate string formulas wherever a string formula is legal, with the exception that information cannot be stored by means of LET or CHANGE statements in concatenated string variables. That is, concatenated string variables cannot appear to the left of the equal sign in a LET statement or as the right-hand argument in a CHANGE statement. For example, LET A$=B$+C$ is legal, but LET A$+B$=C$ is not; and similarly, CHANGE A$+B$ TO X is legal, but CHANGE X TO A$+B$ is not. An example of string concatenation is:

```
10      INPUT A$
20      CHAIN A$+"MAIN.PRG"
30      END
RUNNH

?DTA4:
```

The program causes chaining to DTA4:MAIN.PRG, which is the program MAIN.PRG on DECtape drive 4.

8.6 STRING MANIPULATION FUNCTIONS

A number of functions have been implemented that perform manipulations on strings. These functions are LEN, ASC, CHR$, VAL, STR$, LEFT$, RIGHT$, MID$, SPACE$, and INSTR. Functions that return strings have names that end in a dollar sign ($); those functions that return numbers have names that do not end in a dollar sign.

8.6.1 The LEN Function

The LEN function returns the number of characters in a string. It has the form:

LEN (string formula)

Examples:

```
10      READ A$, B$
20      PRINT LEN(A$+B$+"AROUND")
30      DATA "UP, ", "DOWN, AND "
40      END
RUNNH
20

10      IF LEN (A$)<>0 THEN 30
20      PRINT "A$ IS A NULL STRING"
30      END
```

8.6.2 The ASC and CHR$ Functions

The ASC and CHR$ functions perform conversion of ASCII numbers in the same manner as the CHANGE statement. The ASC function converts one character to its ASCII decimal equivalent, and the CHR$ function converts an ASCII decimal number to its equivalent character.

The ASC function has the form:

ASC (argument)

The argument can be either one character or the two- or three-letter code that represents a nonprinting character (refer to Table 8-1 for these codes). ASC returns the equivalent ASCII decimal number for the character.

The CHR$ function has the form:

CHR$ (numeric formula)

The value of the numeric formula is truncated to an integer that must be in the range 0 through 127 and cannot be the numbers 10 through 13. If the integer is less than 0 or greater than 127 or one of the numbers 10 through 13, an error message is issued. This integer is then interpreted as an ASCII decimal number that is converted to its equivalent character (refer to Table 8-1 for the ASCII numbers and the equivalent characters).

An example of the ASC and CHR$ functions follows.

```
5     FOR T=ASC(A) TO ASC(A)+3
10    PRINT "THIS IS TEST " + CHR$(T)
```

This is the beginning of a FOR loop that successively prints:

```
THIS IS TEST A
      .
      .
      .
THIS IS TEST B
      .
      .
      .
THIS IS TEST C
      .
      .
      .
THIS IS TEST D
```

8.6.3 The VAL and STR$ Functions

The VAL and STR$ functions perform conversions from numbers to strings and strings to numbers. The form of the VAL function is:

VAL (string formula)

The string formula must look like a number; if it does not, an error message is issued. VAL returns the actual number that the string represents. The VAL function does not return the ASCII value of the number that the string represents, it returns the number. For example, VAL ("25") returns the number 25. The 25 that is the argument to VAL is a string, the 25 that VAL returns is a number.

If the string argument represents a number that is greater than about 1.7E38 in magnitude or non-zero, but less than about 1.4E-39 in magnitude, the appropriate overflow or underflow message is issued and the value returned is about 1.7E38, about -1.7E38, or zero, whichever is appropriate.

Example:

```
10      INPUT A$
20      PRINT VAL (A$)*2
          .
          .
          .
100     END
RUNNH

?2.4611121
 4.92222
```

The STR$ function returns the string representation (as a number) of its argument. The form of STR$ is:

STR$ (numeric formula)

The string that is returned is in the form in which numbers are output in BASIC (see Section 6.1). For example, PRINT STR$ (1.76111124) prints the string 1.76111.

Examples:

```
10      A=2561
20      B$=STR$(A)
30      PRINT B$
40      END
RUNNH

2561


10      A=25
20      B$=STR$(A)
30      CHANGE B$ TO X
40      PRINT X(0); X(1); X(2)
50      END
RUNNH

 2   50   53
```

8.6.4 The LEFT$, RIGHT$, and MID$ Functions

The LEFT$, RIGHT$, and MID$ functions return substrings of their string arguments.

The LEFT$ function returns a substring of a specified number of characters starting with the leftmost character of its string argument. The LEFT$ function has the form:

LEFT$ (string formula, numeric formula)

The value of the numeric formula is truncated to an integer that specifies the number of characters in the substring. If the specified number of characters is greater than the length of the string argument, the entire string is returned. If the specified number of characters is less than or equal to zero, an error message is issued. For example,

```
10      PRINT LEFT$("THIS IS A TEST",7)
```

prints the substring

```
THIS IS
```

The RIGHT$ function returns a substring of specified length ending at the rightmost character of its string argument. The form of the RIGHT$ function is:

RIGHT$ (string formula, numeric formula)

The value of the numeric formula is truncated to an integer that specifies the number of characters in the substring to be returned. If the number of characters is greater than the length of the string argument, the entire string is returned. If the specified number of characters is less than or equal to zero, an error message is issued. For example,

```
5       A$="HERE AND THERE"
10      PRINT RIGHT$(A$,5)
```

prints the substring

```
THERE
```

The MID$ function returns a substring of its string argument starting a specified number of characters from the leftmost character of the string argument. The number of characters in the substring can also be specified. The form of the MID$ function is:

MID$ (string formula, numeric formula-1, numeric formula-2)

The second numeric formula, which is truncated to an integer that specifies the number of characters in the substring, is optional and can be omitted. If this argument is omitted, the substring includes all the remaining characters in the string argument. The first numeric formula is truncated to an integer that specifies the leftmost character at which the substring is to start. MID$ returns a null string if the first numeric formula when truncated to an integer is greater than the number of characters in the string argument; if it is less than or equal to zero, an error message is issued. If the number of characters in the substring is specified (by the second numeric formula) and is greater than the number of characters in the string argument beginning at the specified character, MID$ returns the string argument starting at the specified character. If the number of characters is less than or equal to zero, an error message is issued.

Examples:

```
10      PRINT MID$ ("TOTAL OUTPUT IN MARCH",17)
                .
                .
                .

RUNNH

MARCH


10      PRINT MID$ ("ABCDEF",3,1)
                .
                .
                .

RUNNH

C
```

8.6.5 The SPACE$ Function

The SPACE$ function returns a string of spaces. The form of the SPACE$ function is:

SPACE$ (numeric formula)

The value of the numeric formula is truncated to an integer that specifies the number of spaces in the string to be returned. If the integer is less than or equal to zero or greater than 132, an error message is issued.

Example:

```
10      A$=B$="HERE"
20      FOR T=1 TO 3
30      PRINT A$; SPACE$(T); B$
                .
                .
                .

RUNNH


HERE HERE
HERE  HERE
HERE   HERE
```

8.6.6 The INSTR Function

The INSTR function searches for a specified substring within a string and returns the position of the first character of that substring within the string. The positions are numbered from the leftmost character in the string. The user can optionally specify that the search for the substring begin at a character position other than the first. The form of the INSTR function is:

INSTR (numeric formula, string formula-1, string formula-2)

The numeric formula, which is truncated to an integer that specifies the starting character position, is optional and can be omitted. If the numeric argument is omitted, the search begins at the first character position. The first string argument is the string searched; the second string argument is the substring searched for. If the value of the numeric formula (if specified) is greater than the number of characters in the string or if the substring cannot be found in the string, INSTR returns a value of zero. If the value of the numeric formula is less than or equal to zero, an error message is issued. If the second string argument is a null string, INSTR returns the character position at which the search started, unless that position is past the last character in the string. In that case, INSTR returns a value of zero.

Examples:

```
10      PRINT INSTR ("ABCDCEF", "C")
                    •
                    •
                    •

RUNNH

3

10      PRINT INSTR (4,"ABCDCEF", "C")
                    •
                    •
                    •

RUNNH

5
```

Note that if the second string argument occurs more than once within that part of the first string argument that is searched, the first occurrence found is used.

CHAPTER 9
EDIT AND CONTROL

Several commands for editing BASIC programs and for controlling their execution enable the user to perform such operations as:

a. deleting lines
b. listing the program
c. changing or resequencing line numbers with set increments
d. saving programs on various storage devices (disk, DECtape, card punch, etc.)
e. calling in programs from storage devices
f. deleting programs on disk or DECtape

These commands are summarized in Table 9-1:

Table 9-1
Commands for Editing BASIC Programs

Command	Action
BYE	Exits from BASIC and partiall logs out. Refer to Section 4.6.
CATALOG device:	Lists on the Teletype the names and extensions of the user's files on either the disk or DECtape or the system programs on the system device SYS. If device is omitted, disk is assumed. The colon following the device is optional.
COPY device:filename.ext >device:filename.ext	The file specified by the first argument is copied to the device specified by the second argument and given the name specified in the second argument. If device is omitted, DSK: is assumed. If the device is not disk or DECtape, the filename and extension can be omitted. If the filename is omitted, the extension must also be omitted. If the device is either disk or DECtape, the filename must be specified; however, the extension can be omitted and the extension .BAS will be assumed. The file need not have line numbers to be acceptable to COPY. The program currently in core is not disturbed by the COPY command.

(continued on next page)

Table 9-1 (Cont)
Commands for Editing BASIC Programs

Command	Action
DELETE n	Deletes line number n and the contents of the line from user core.
DELETE n-m†	Deletes lines numbered n through m from user core.
GOODBYE	Equivalent to BYE.
KEY	Sets BASIC to accept Keyboard mode input from the user's Teletype (refer to Appendix C). If neither KEY nor TAPE was specified, KEY is assumed.
LENGTH	Prints approximate length of source program (expressed as the number of characters).
LIST†	Lists program with heading.
LIST n	Lists line number n of the program, with heading.
LIST n-m	Lists program with heading, from line number n through m.
LISTNH† LISTNH n LISTNH n-m	Same as LIST, but with heading suppressed.
LISTREVERSE	Lists program in reverse order, with heading.
LISTNHREVERSE	Same as LISTREVERSE, but with heading suppressed.
NEW filename.ext	The user must specify the filename explicitly; if this is not done on the same line as the NEW command, BASIC outputs a prompting message in which it asks for the name. The file currently in user core is then deleted and the new program name is established as the current name.
OLD dev:filename.ext	The user must specify the filename explicitly; if this is not done on the same line as the OLD command, BASIC outputs a prompting message in which it asks for the name. BASIC then replaces the file currently in user core with the existing program of that name from the storage device. That program name is established as the current name. The file must have line numbers.
QUEUE filename.ext /UNSAVE/nCOPIES /LIMITm	Causes the specified file to be output from the disk to the line printer when the line printer is available. The line printer does not have to be available when the QUEUE command is given. The file currently in core is not affected by the QUEUE command. If the extension is omitted, .BAS is assumed. The three optional switches, /UNSAVE, /nCOPIES, and /LIMITm can be in any order. UNSAVE and LIMIT can be abbreviated to as little as one letter and the word COPY can be omitted entirely. The /UNSAVE switch causes the file to be deleted from the disk after it is printed;

†LIST, LISTNH and DELETE commands can be given more than one argument; arguments are separated by commas. An example is as follows:

LIST n,m-l,k

Table 9-1 (Cont)
Commands for Editing BASIC Programs

Command	Action
QUEUE filename.ext /UNSAVE/nCOPIES /LIMITm (cont)	normally, the file is saved. The /nCOPIES switch causes n copies of the file to be printed to a maximum of 63 copies; normally, one copy is printed. The /LIMITm switch specifies the number of line-printer pages to be printed; normally, 200 is the maximum number of pages that can be printed. The values specified by n and m must be integers. More than one file and its associated switches can be specified in the QUEUE command. Arguments must be separated by commas.
RENAME filename	Changes name of program currently in user core.
REPLACE dev:filename.ext	Replaces an existing file of the specified name on the specified device with the file currently in user core. If the device is DSK or DECtape, the old file must be present on the device or an error message will be issued. The default assumptions are the same as those described for SAVE.
RESEQUENCE n	Changes line numbers to n, n + 10, ...
RESEQUENCE n,,k	Changes line numbers to n, n+k, ... Commas are necessary as argument delimiters.
RESEQUENCE n,f,k	Changes line numbers from line f upward to n, n+k, ... f must not be greater than n.
RUN	Compiles and executes the entire program currently in core.
RUN n	Compiles the entire program currently in core and begins execution at line number n.
RUNNH	Same as RUN, but with heading suppressed.
RUNNH n	Same as RUN n, but with heading suppressed.
SAVE device:filename.ext	Saves on the specified device the file currently in user core. If device: is omitted, DSK: is assumed. If .ext is omitted, .BAS is assumed. The extension cannot be specified if the filename is omitted. If filename.ext is omitted, the current filename and extension are used. Note that the SAVE command does not overwrite an existing file of the same name (use REPLACE instead).
SCRATCH	Deletes all program statements from user core.
SYSTEM	Exits to Monitor.
TAPE	Sets BASIC to accept input from the paper-tape reader attached to the user's Teletype (refer to Appendix C).
UNSAVE dev:filename.ext, dev:filename.ext, ...	Deletes from each device specified each file indicated by filename.ext. The default assumptions are the same as those described for SAVE. When more than one argument is specified, they must be separated by commas.

(continued on next page)

Table 9-1 (Cont)
Commands for Editing BASIC Programs

Command	Action
WEAVE dev:filename.ext	Reads program statements from the file indicated by filename.ext on the specified device. The file must have line numbers, and existing statements in user core are replaced by new statements having same line numbers.
↑C	To stop a running program and enter Monitor level, type ↑C twice.
↑O	To suppress output (typeout), type ↑O.
Any command root can be abbreviated to its first three letters. For example, LISNHREV is the same as LISTNHREVERSE.	

CHAPTER 10
DATA FILE CAPABILITY

The data file capability allows a program to write information into and read information from data files that are on the disk.

Nine input/output channels are reserved for handling data files from a program. A data file must be assigned to a channel before it can be referenced in the program. At any given time, a program can have one and only one file on each channel and one and only one channel assigned to each file. Consequently, a maximum of nine files can be open simultaneously. However, because it is possible for a program to change or establish file/channel assignments while it is running, there is no limit to the number of data files that can be referenced in one program.

10.1 TYPES OF DATA FILES

There are two types of data files acceptable to BASIC: sequential access files and random access files.

10.1.1 Sequential Access Files

Sequential access files are those files that contain information that must be read or written sequentially, one item after another, from the beginning of the file. A sequential access file is either in write mode or read mode, but cannot be in both modes at the same time. When read mode is established, reading starts at the beginning of the file. When write mode is established, the file is erased and writing starts at the beginning of the file.

An important distinction to note about sequential access files is that they can be listed in readable form on the user's Teletype or the line printer. Sequential access files consist of lines that contain data items. A sequential access file is either a line-numbered file or a nonline-numbered file, depending upon whether or not its lines begin with line numbers. Line-numbered files are like BASIC programs in that they can be manipulated by any of the commands described in Chapter 9 (e.g., OLD, LIST, DELETE) except the RUN(NH) and CHAIN commands. Nonline-numbered files cannot be handled by any of the commands that expect a file to have line numbers; they can only be manipulated by the

COPY, QUEUE, and UNSAVE commands. They can be listed on the user's Teletype by means of the COPY command; for example:

```
COPY TEST4 > TTY:
```

Sequential access files do not necessarily have to be created by a program; they can be created at the editing level in BASIC. Line-numbered files can be created or modified just as a BASIC program is created or modified. Nonline-numbered files can be created at the Teletype and then transferred to a storage device such as the disk by means of the COPY command. The following conventions must be observed when dealing with a sequential access file at the editing level:

a. In line-numbered files, each line number must be followed immediately by at least one space, a tab, or the letter D.

b. A line can contain any number of data items separated from one another by at least one space, a comma, or a tab. However, the line must not be longer than 142 characters (counting the line number and its following delimiter, but not the carriage return and line feed that terminate the line). It is not necessary to have a space, comma, or tab after the last data item on the line. Note that blanks and tabs are not ignored in a data file as they are in a program.

c. A data item is any numeric constant (refer to Section 1.3.3) or string constant (refer to Chapter 8). Numeric constants must not contain blanks or tabs. If a string is to contain a blank, comma, or tab, the user must enclose the string in quotes; otherwise it will be read as more than one data item by the statements that read data.

Section 10.4 contains an example of the use of a line-numbered data file created at the editing level. Section 10.5.1 contains an example of a program that creates both a line-numbered data file and a nonline-numbered data file and shows what these files look like when they are copied to the Teletype.

Because it requires execution time for a program to read and write line numbers in a data file, a nonline-numbered data file should be used in preference to a line-numbered data file unless the user specifically wishes to edit the data file with commands such as DELETE.

Another distinction between sequential access files is whether the file is a pure data file or a text file. A pure data file is used primarily for the storage of data. A text file contains data that is probably destined for output to the line printer, because it is a report, a financial statement, or the like. The user must follow slightly different procedures in his program depending on the type of file he wishes to handle. For example, a string that contains a blank must be enclosed in quotes when it is written into a pure data file, otherwise it will be seen as more than one string when data is read from the file. However, such a string should not be enclosed in quotes when it is written into a text file because text files are not normally read back into a program, and the superfluous quotes would spoil the appearance of the file when it is printed. The procedures to follow when handling each type of file are explained in Sections 10.5.1 and 10.7.

10.1.2 Random Access Files

Random access files are data files that are not necessarily read or written sequentially. The user can read items from or write items into a random access file without having the items follow one after the other. The items in a random access file are not recorded in a form suitable for listing, and therefore cannot be output to the user's Teletype or the line printer. Random access files cannot be handled by any of the BASIC commands other than COPY, QUEUE, and UNSAVE. A random access file can be copied to the disk, DECtape, or magnetic tape, but not to any other device (Teletype, paper-tape punch, card punch, or line printer). Copying a random access file to a device other than disk, DECtape, or magnetic tape will cause errors to be introduced into the file. If the system program PIP is used to transfer a random access file to disk, DECtape, or magnetic tape, the file must be transferred in binary mode. Refer to the PIP manual for more information.

Random access files, unlike sequential access files, do not distinguish between read mode and write mode. The user can read and write any item in a random access file at any time by first setting a pointer to that item. A random access file contains either string data or numeric data, but not both. Each data item in a random access file takes up the same amount of storage space, called a record, on the disk. BASIC must know the record size for the random access file in order to correctly move the pointer for that file from one data item to another. The record size for a random access numeric file is set by BASIC because the storage space required for a number in such a file is always the same. The storage space required for a string, however, is dependent upon the number of characters in the string. Thus, for a random access string file the user must specify the number of characters in the longest string in the file so that BASIC can set the record size accordingly. This specification takes place when the file is assigned to a channel. Refer to the description of the FILES and FILE statements in Section 10.2. When creating a new random access string file, if the user specifies too few characters an error message is issued when a string too long to fit into a record is written. If too many characters are specified for a record, the strings will always fit, but space will be wasted on the disk. When he is dealing with an existing file, the user does not have to specify a record size. If he does specify a record size for an existing file, the record size must match that with which the file was written.

BASIC processes random access files more quickly than it processes sequential access files. Consequently, if the user wishes to read or write large amounts of data in sequential order, but does not require that the data be in listable form, he should consider using a random access file to take advantage of its speed. A random access file can easily be read or written in sequential order.

10.2 THE FILE AND FILES STATEMENTS

The FILE and FILES statements perform identical functions. They both assign a file to a channel and establish the type of the file (sequential, random access string, or random access numeric). The difference between FILE and FILES is that FILE is an executable statement while FILES is not. Before execution of the program begins, BASIC collects all of the FILES statements in the program, makes the channel assignments, and sets the file types as they were declared in the FILES statements. The FILES statements are not used again during that execution of the program. GO TO and GOSUB statements to FILES statements work just as they do to REM statements; i.e., execution will transfer to the first executable statement following the FILES statement. The FILE statement, on the other hand, assigns channels and establishes file types during program execution, thereby allowing the user to change file/channel assignments during the running of his program.

The FILE and FILES statements accept filename arguments of the form:

filenm.ext type

where filenm and .ext are the filename and extension of the file in the form described in Chapter 4. The filename must be specified, but the extension can be omitted. If the extension is omitted, .BAS is assumed. Type can be a percent sign (%); a dollar sign ($) optionally followed by one, two, or three digits; or omitted. If type is omitted, the file is assumed to be a sequential access file. If a percent sign is specified, the file is assumed to be a random access numeric file. A dollar sign optionally followed by a one- to three-digit number indicates a random access string file. The number following the dollar sign specifies the number of characters in the longest string that the file will contain. A maximum of 132 characters and a minimum of one character can be specified. If the number is omitted from the dollar sign type and the file does not presently exist, a default length of 34 characters is established. If the number is omitted from the dollar sign type and the file does exist, the length with which the file was previously written is established.

The FILES statement has the form:

FILES filenm.ext type, filenm.ext type,...filenm.ext type

where the arguments can be separated by a comma or a semicolon. Channels are assigned consecutively to the arguments of all the FILES statements in the program. If an argument is omitted, the channel for the missing argument is skipped. For example, if a program contains only these FILES statements:

```
10      FILES ,, A;,B
20      FILES C,D,
30      FILES E
```

file A will be assigned to channel 3, file B to channel 5, file C to channel 6, file D to channel 7, and file E to channel 9.

The FILE statement has the form:

FILE arg1, arg2, ... argn

where the arguments can be separated by a comma or a semicolon. At least one argument must be present in a FILE statement. Each argument that assigns a sequential access file to a channel is of the form:

#N, string formula
or #N: string formula

Each argument that assigns a random access file to a channel is of the form:

:N, string formula
or :N: string formula

N is a digit from 1 to 9 specifying the channel, and the string formula is of the form:

filenm. ext type

Note that the channel specifier for a random access file is preceded by a colon (:) while the channel specifier for a sequential access file is preceded by a number sign (#). This is true of all data file statements and functions that include channel specifiers. Some data file statements and functions do not require the number sign or colon to be specified explicitly, but default to one or the other. See the description of the various statements and functions in the following sections for details. An attempt to reference a file with a channel specifier of the wrong type causes an error message.

The FILE statement does not permit the enclosing quotes to be omitted when its string formula argument is a constant. This is because a statement of the form FILE :1, B$ would cause an ambiguity. The B$ could be taken as a variable (B$) or as a random access string file named B.

Before the FILE statement assigns a file to a channel, it checks to see if a file already exists on that channel; if so, the old file is closed and removed from the channel before the new file is assigned. The type of the old file is immaterial; it is permissible, for example, to close an old sequential access file on a channel and then open a random access file on that channel. Any file open on a channel at the end of program execution or whenever BASIC is reentered is automatically closed and removed from that channel.

Examples of FILES and FILE statements are:

```
10      FILE #1, "ONEDAT": #4,"OUTDAT"
20      FILE #9: "CHKDAT.4", :4, B$+"%"
30      FILES FOUR.OUT$, MAIN.B;;; PROG$16
```

10.3 THE SCRATCH AND RESTORE STATEMENTS

The SCRATCH statement has the form:

SCRATCH arg1, arg2, ... argn

The RESTORE statement has the form:

RESTORE arg1, arg2, ... argn

where the arguments can be separated by a comma or a semicolon. An argument is of the form:

For sequential access files:

#N

For random access files:

:N

where N is a digit from 1 through 9 specifying the channel. If neither a number sign nor a colon is present in front of the N, the number sign is assumed. At least one argument must be present in a SCRATCH or RESTORE statement.

Scratching a sequential access file erases it and sets it in write mode. Writing will start at the beginning of the file. Referencing a sequential access file with a statement that does input (READ, INPUT, or IF END, described in Sections 10.4 and 10.10) while it is in write mode results in a fatal error.

Scratching a random access file simply erases it and sets the pointer for the file to the first record in the file.

Restoring a sequential access file sets the file in read mode. Reading will start at the beginning of the file. Referencing a sequential access file with a statement that does output (WRITE or PRINT, described in Section 10.5) while it is in read mode results in a fatal error. When a sequential access file is opened by a FILES or FILE statement and the file exists at that time, it is automatically set in read mode; it is not necessary to restore it. It is only necessary to restore a sequential access file if it has been set in write mode and the user wishes to set it to read mode in the same program.

Restoring a random access file simply sets the pointer for the file to the first record in the file. When a random access file is opened on a channel by a FILE or FILES statement, its pointer is automatically set to point to the first record of the file.

Examples of the SCRATCH and RESTORE statements are:

```
10      SCRATCH #4, :2, #3, 1
20      SCRATCH #1, 2, 3, 4
80      RESTORE :2 $9, 1
90      RESTORE 1, 2, 3, 7
```

10.4 THE READ AND INPUT STATEMENTS

The READ and INPUT statements read data items from files. The READ statement has the following forms:

For sequential access files:
READ #N, variable, variable, ... variable

For random access files:
READ :N, variable, variable, ...variable

The INPUT statement has the following forms:

For sequential access files:
INPUT #N, variable, variable, ... variable

For random access files:
INPUT :N, variable, variable, ...variable

N is a digit from 1 through 9 specifying the channel. At least one variable must be present in each READ or INPUT statement. The delimiter following N can be a comma or a colon. The variables are separated from one another by a comma or semicolon.

The variables in a READ or INPUT statement for a sequential access file can be string or numeric or a mixture of both. The variables in a READ or INPUT statement for a random access file can be string or numeric, but not both, because a given random access file cannot contain both string and numeric data items.

READ and INPUT statements for sequential access files differ from one another in the following way. The READ statement expects each line of data in the file to begin with a line number, which it then skips. That is, the line number is not treated as data. If a line number is not present, an error message is issued. The INPUT statement, on the other hand, does not expect a line number on each line of data. If one is present, it is read as data. It is illegal to use both INPUT and READ statements to read from the same sequential access file unless the file has been restored between the two types of statements. An attempt to mix READ and INPUT statements for sequential access files results in a fatal error message.

Examples of the READ and INPUT statements for sequential access files are:

```
10      READ #2, A(I), L, B$
30      READ #6, Z$
105     INPUT #4, B(K)
120     INPUT #7, W$, M
```

READ and INPUT statements for random access files are completely equivalent. They both begin reading at the item that the pointer for the file specifies, and continue reading sequentially until all of

the variables have been filled. It is legal to use both READ and INPUT statements to input from the same random access file.

If the user attempts to read beyond the last item in either a sequential access or a random access file, a fatal error message is issued. In a random access file, it is possible to have items that have not been written but that are within the file (because some subsequent item has been written). If such an item is in a numeric file and is read, a value of zero is input. If such an item is in a string file, a string containing no characters is returned.

Examples of READ and INPUT statements for random access files are:

```
20      READ :2, A, B(I), C; F2
50      READ :4, F$, G$(8)
210     INPUT :1, Q(2)
240     INPUT :5: N1; N2; N3
```

The following example shows a sequential access file being created at the editing level and then read by a program.

```
NEW
NEW FILE NAME--TEST2

READY
10  "LANTHANIDE SERIES"
20  LA,CE,PR,ND,PM,SM,EU,GD,TB,DY,HO,ER
25  TM,YB,LU,57,71
SAVE

READY
OLD
OLD FILE NAME--TABLE

READY
LISNH
1   DIM A$(15)
5   FILES TEST2
12  READ #1,B$
15  FOR X=1 TO 15
20  READ #1,A$(X)
25  NEXT X
30  READ #1,N1,N2
35  PRINT "THIS IS THE ";B$
40  PRINT
42  PRINT "ELEMENT", "ATOMIC NUMBER"
44  PRINT
45  FOR Y=1 TO 15
50  PRINT A$(Y),N1-1+ Y
55  NEXT Y
100 END
```

The user types in and then SAVEs the data file "TEST2".

The old file "TABLE" is retrieved and listed.

(continued on next page)

```
READY
RUN

TABLE           13:31           15-JUL-70

THIS IS THE LANTHANIDE SERIES

ELEMENT         ATOMIC NUMBER

LA              57
CE              58
PR              59
ND              60
PM              61
SM              62
EU              63
GD              64
TB              65
DY              66
HO              67
ER              68
TM              69
YB              70
LU              71

TIME:  0.18 SECS.
READY
```

An example of reading from a random access file is given in Section 10.6.

10.5 THE WRITE AND PRINT STATEMENTS

The WRITE and PRINT statements write data items into files.

10.5.1 WRITE and PRINT Statements for Sequential Access Files

The WRITE and PRINT statements for sequential access files have the following forms:

```
WRITE #N, list of formulas and delimiters
PRINT #N, list of formulas and delimiters
```

where N is the channel specifier. The delimiter following N can be a comma or a colon; it can be omitted if the list is omitted. The formulas in the list can be string or numeric or both. The TAB function can be used. The delimiters can be commas, semicolons, or <PA> delimiters; they have the same meanings that they have in the PRINT statement for the Teletype (refer to Chapter 6).

WRITE and PRINT statements for sequential access files differ from one another in the following way. The WRITE statement begins each line of output with a line number followed by a tab. The first line in the file is numbered 1000 and subsequent line numbers are incremented by 10. The PRINT statement, on the other hand, does not begin lines with line numbers. It is illegal to use both WRITE and PRINT

statements to write to the same sequential access file unless the file has been erased (by means of the SCRATCH command) between the two types of statement. An attempt to mix WRITE and PRINT statements results in a fatal error message.

Files created by WRITE statements are normally read by READ statements. Files created by PRINT statements are normally read by INPUT statements.

Examples of the WRITE and PRINT statements for sequential access files are:

```
50      WRITE #2, SQR(A)+EXP(G); Q(I)
75      PRINT #7, <PA> B(I),,C(I),,D(I)
110     WRITE #3
```

The normal mode of output for WRITE and PRINT statements for sequential access data files is noquote mode. In noquote mode, strings are not enclosed in quotes even if they contain characters that the READ and INPUT statements see as delimiters. Also, strings are concatenated if they are output with a semicolon separating them. Noquote mode is the mode used when writing a text file (refer to Section 10.1.1 for a description of text files and pure data files). Noquote is the default mode; a sequential access file is automatically set in noquote mode when it is assigned to a channel by a FILE or FILES statement. However, noquote mode is not suitable when writing pure data files because the integrity of the data is not maintained. In order to write a pure data file, the file must be set in quote mode. This can be done by the QUOTE or QUOTE ALL statement, both of which are described in Section 10.7. When a file is in quote mode, BASIC accepts WRITE and PRINT statements that are in the usual form, but it makes whatever small changes that are necessary to the formatting in order to preserve the integrity of the data items. Refer to Section 10.7 for details about the changes that are made.

An example of the actions performed by the WRITE and PRINT statements follows.

```
10      FILES A, B
20      SCRATCH #1,2
30      WRITE #1, 1; 2, TAB(70), 3
40      PRINT #2, "A"; 4
50      END

RUNNH

TIME:  0.02 SECS.

READY

COPY A > TTY:

01000 1 2
01010 3
```

(continued on next page)

```
READY
COPY B > TTY:

A 4

READY
```

10.5.2 WRITE and PRINT Statements for Random Access Files

The WRITE and PRINT statements for random access files have the forms:

```
WRITE :N, formula, formula, ... formula
PRINT :N, formula, formula, ... formula
```

where N is the channel specifier. The delimiter following the channel specifier can be a comma or a colon. At least one formula must be present in each statement. The formulas are separated from one another by a comma or semicolon. In a given statement, all of the formulas must be string or all of them must be numeric because a random access file is either string or numeric but not both.

WRITE and PRINT statements for random access files are exactly equivalent; they both begin writing into the record that the pointer for the file specifies, and continue writing sequentially until all of their arguments have been written. It is legal to use both WRITE and PRINT statements to write to the same random access file.

Examples of WRITE and PRINT statements for random access files are:

```
25      WRITE :2, N, L; M
35      PRINT :4: A$, B$+Q$(I)
```

An example of writing to a random access file is shown below in Section 10.6.

10.6 THE SET STATEMENT AND THE LOC AND LOF FUNCTIONS

The SET statement has the form:

```
SET arg1, arg2, ... argn
```

where the arguments can be separated by commas or semicolons. Each argument has the form:

```
    :N, numeric formula
or  :N: numeric formula
```

where N is the channel specifier. The colon preceding the channel specifier can be omitted because SET is only used for random access files; the colon is therefore redundant. Each SET statement must have at least one argument. When a SET statement is executed, the pointer for the file on the specified channel is moved so that it points to the item in the file that is specified by the numeric formula, which has been truncated to an integer. If the numeric formula after truncation is less than or equal to

zero, an error message is issued. The items in the file are numbered sequentially; the first item in the file is 1, the second 2, and so forth. The next statement in the program that reads from or writes to the random access file will read or write the item to which the pointer was set, provided that the pointer has not been moved again by a subsequent SET statement or another statement.

Examples of SET statements are:

```
55      SET  :3, 100, :4, 150
85      SET  :1,1; :4,215
```

An example of a program using the SET statement follows.

```
10      FILES TEST4%
20      FOR T=1 TO 10
30      WRITE :1, T
40      NEXT T
50      FOR T=1 TO 10 BY 2
60      SET :1, T
70      READ :1, X
80      PRINT X
90      NEXT T
100     END

RUNNH

1
3
5
7
9

TIME:  0.01 SECS.
```

Two functions, LOC and LOF, return information about random access files. LOC returns the number of the record to which the pointer for the file currently points, and LOF returns the number of the last record in the file.

The forms of LOC are:

```
LOC(N)
LOC(:N)
```

The forms of LOF are:

```
LOF(N)
LOF(:N)
```

where N is the channel specifier. An error message is issued if a random access file is not assigned to the specified channel when the function is executed.

An example of these functions is:

```
10      IF LOC(2)<=LOF(2) THEN 30
20      PRINT "FINISHED FILE ON CHANNEL 2"
```

10.7 THE QUOTE, QUOTE ALL, NOQUOTE, AND NOQUOTE ALL STATEMENTS

As was discussed in Section 10.5.1, the default mode for output to sequential access data files or to the TELETYPE is noquote mode. The QUOTE and QUOTE ALL statements allow the user to change the mode of the Teletype and sequential access files to quote mode. Quote mode changes the way that the data items are written into the files or onto the Teletype. In quote mode, strings are enclosed in double quotes by BASIC if they contain blanks, tabs, or commas; a leading blank is output immediately before strings and negative numbers; and a double quote character cannot be output by the user. If such an attempt is made to output a double quote character, an error message is issued. Also a data item cannot be longer than the maximum amount of space available on a new line. If an attempt is made to output a data item longer than this, a fatal error message results. In noquote mode, the data item would be split across two or more lines. These modifications to the normal formatting are sufficient to insure that the integrity of the data is maintained, as was discussed in Section 10.5.1.

The opposite of quote mode is noquote mode, which can be set by the NOQUOTE and NOQUOTE ALL statements. Noquote mode is the default mode for the Teletype and sequential access files. Whenever a sequential access file is assigned to a channel by a FILES or a FILE statement, it is automatically set in noquote mode. NOQUOTE and NOQUOTE ALL statements are only necessary if the user wishes to change a file from quote to noquote mode.

When creating a pure data file, in addition to setting the file in quote mode, it is good practice to separate the formulas in the WRITE or PRINT statements with semicolons to pack the data items close together. Although separating the formulas with commas is permissible, it will waste space on the disk.

The form of the QUOTE statement is:

QUOTE arg1, arg2, ... argn

where each argument has the form:

#N
or N

where N is the channel specifier. If an argument is omitted, the Teletype is specified; for example,

```
30      QUOTE , 1, 4
```

refers to the Teletype and the files on channels 1 and 4.

Since QUOTE is assumed to have at least one argument, the statement

```
50      QUOTE
```

specifies the Teletype.

The form of the QUOTE ALL statement is:

QUOTE ALL

QUOTE ALL refers to channels 1 through 9, but not to the Teletype.

When a channel is referenced in a QUOTE or QUOTE ALL statement and that channel has a sequential access file currently assigned to it, output to the file is done in quote mode. If a sequential access file is not presently assigned to the channel, nothing is done and no error message is returned.

The form of the NOQUOTE statement is the same as that of the QUOTE statement, except that the word NOQUOTE is substituted for the word QUOTE. Examples of NOQUOTE statements are:

```
10      NOQUOTE #7,,2
20      NOQUOTE
```

The first example specifies the files on channels 7 and 2 and the Teletype. The second example specifies the Teletype.

The form of the NOQUOTE ALL statement is:

NOQUOTE ALL

When a channel is referenced by a NOQUOTE or NOQUOTE ALL statement and that channel has a sequential access file currently assigned to it, output to the file will be written in noquote mode. If a sequential access file is not presently assigned to the channel, nothing is done and no error message is returned.

The use of the QUOTE ALL or NOQUOTE ALL statement is a convenient way to set all sequential access files currently assigned to channels into the appropriate mode, since the statements will not return error messages about or affect unassigned channels or the Teletype, and will not damage any of the random access files currently assigned to channels.

Quote or noquote mode can be set even if the file is in read mode because these modes have no effect on input. They will affect the output if the file is subsequently put into write mode.

If the mode is changed from quote to noquote or vice versa, the change takes effect immediately.

10.8 THE MARGIN AND MARGIN ALL STATEMENTS

Normally, the right output margin for the Teletype and sequential access files is 72 characters. Whenever a sequential access file is assigned to a channel by a FILES or a FILE statement, the file's output margin is automatically set to 72 characters. At the beginning of and also at the end of program execution, the Teletype output margin is set to 72 characters. There is no margin in a random access file.

The MARGIN and MARGIN ALL statements allow the user to set the right output margin for the Teletype or any sequential access file from 1 to 132 characters.[1] The form of the MARGIN statement is:

MARGIN arg1, arg2, ... argn

where each argument has the form:

#N, numeric formula

The arguments can be separated by commas or semicolons. N is the channel specifier. The numeric formula specifies the margin size; it is truncated to an integer. Either a comma or a colon can be used to separate the channel number from the margin size.

If only the margin size is present in the argument, that argument refers to the Teletype. For example:

```
35      MARGIN 75, #8:132
```

sets a margin of 75 characters for the Teletype and a margin of 132 characters for the file on channel 8.

The form of the MARGIN ALL statement is:

MARGIN ALL numeric formula

This statement sets the sequential access files on channels 1 through 9 to the margin specified by the numeric formula, the value of which is truncated to an integer before the margin is set. The Teletype is not affected by the MARGIN ALL statement. Examples of the MARGIN ALL statement are:

```
60      MARGIN ALL 132
65      MARGIN ALL N*ABS(K(I))
```

Neither the MARGIN nor MARGIN ALL statement has any effect on random access files or on channels that have no files assigned to them. Consequently, the MARGIN ALL statement is a convenient way to set a margin for all sequential access files currently assigned to channels.

---

[1] The monitor command SET TTY WIDTH must be used in addition to the BASIC MARGIN statement if the user wishes to set the output margin for the Teletype to any size greater than 72 characters. Refer to Section 6.7 for details.

The margins set by the MARGIN and MARGIN ALL statements apply only to output. The margin for input lines for both the Teletype and sequential access files is not affected by these statements; it is always 142 characters. An attempt to input a line longer than 142 characters results in an error message.

A margin set by a MARGIN or MARGIN ALL statement takes effect as soon as a new line of output is begun for the Teletype or the sequential access file.

Although the right margin can be set to any number between 1 and 132 characters, the margin for lines output by WRITE statements must be at least 7 characters to allow for the line number and its following tab. If the margin is less than 7 characters for a line-numbered file, an error message is issued by the first WRITE statement referencing the file.

10.9 THE PAGE, PAGE ALL, NOPAGE, AND NOPAGE ALL STATEMENTS

Normally, output to the Teletype or to sequential access files is not divided into pages; that is, it is in nopage mode. Whenever a sequential access file is assigned to a channel by a FILES or a FILE statement, it is automatically set in nopage mode. At the beginning and also at the end of program execution, the Teletype is set to nopage mode. The PAGE and PAGE ALL statements allow the user to set a page size of any positive number of lines for the Teletype and sequential access files. The NOPAGE and NOPAGE ALL statements allow the user to set the Teletype and sequential access files to nopage mode. Nopage and page modes are meaningless for random access files.

The form of the PAGE statement is:

```
PAGE arg1, arg2, ... argn
```

where each argument has the form:

```
#N, numeric formula
```

The arguments can be separated by commas or semicolons. N is the channel specifier. The numeric formula is truncated to an integer and used to specify the page size. Either a comma or a colon can be used to separate the channel number from the page size.

If only a page size is present in an argument, that argument refers to the Teletype; for example:

```
40      PAGE #1, 66; 50, #7:62
```

sets the files on channels 1 and 7 to page sizes of 66 and 62 lines respectively, and the Teletype to a page size of 50 lines.

The form of the PAGE ALL statement is:

PAGE ALL numeric formula

This statement sets the sequential access files on channels 1 through 9 to a page size specified by the numeric formula; however, the Teletype is not affected. The value of the numeric formula is truncated to an integer before the page size is set. An example of the PAGE ALL statement is:

```
90      PAGE ALL Q(2)*B
```

Neither the PAGE nor PAGE ALL statement has any effect on random access files or on channels that have no files assigned to them. Consequently, the PAGE ALL statement is a convenient way to set a page size for all of the sequential access files currently assigned to channels. If a PAGE or PAGE ALL statement specifies a page size of zero or less than zero, an error message is issued.

When a PAGE or PAGE ALL statement is executed for a sequential access file that is in write mode or for the Teletype, BASIC ends the current line of output (if necessary), outputs a leading form feed, and starts counting lines beginning with the next line output. Subsequently, whenever a new page becomes necessary, a form feed is output and the line count is set back to zero. Execution of a <PA> delimiter sets the line count to zero. PAGE and PAGE ALL statements can be executed for sequential access files in read mode; in this case, the leading form feed is not output. A page size remains in effect until another PAGE or PAGE ALL statement changes it, until a NOPAGE or NOPAGE ALL statement is executed for that file or the Teletype, or until the end of program execution. Setting the page size for the Teletype is further described in Chapter 6.

The form of the NOPAGE statement is:

NOPAGE arg1, arg2, ... argn

where each argument has the form:

```
   #N
or N
```

where N is the channel specifier. If an argument is omitted, the Teletype is specified; for example:

```
10      NOPAGE #3,, 2
```

refers to the Teletype and the files on channels 2 and 3.

Since the NOPAGE statement is assumed to have at least one argument, the statement

```
70      NOPAGE
```

refers to the Teletype.

The form of the NOPAGE ALL statement is:

NOPAGE ALL

The NOPAGE ALL statement sets all of the sequential access files on channels 1 through 9 in nopage mode, but does not affect the Teletype.

Like the PAGE and PAGE ALL statements, NOPAGE and NOPAGE ALL statements have no effect on channels that have random access files or no files assigned to them. Consequently, the NOPAGE ALL statement is a convenient way to set all of the sequential access files currently assigned to channels into nopage mode.

10.10 THE IF END STATEMENT

The IF END statement allows the user to determine whether or not there is any data left in a file between the current position in the file and the end of the file.

The statement forms are:

For sequential access files:

IF END #N, {GO TO / THEN} line number

For random access files:

IF END :N, {GO TO / THEN} line number

where N is the channel specifier. The line number must refer to a line in the program and must follow the rules for line numbers discussed in Chapter 1. Either THEN or GO TO must be used in the statement. The comma preceding THEN or GO TO is optional.

The IF END statement will execute for a sequential access file only if the file is in read mode; an error message will be issued if the file is in write mode or if it does not exist. The IF END statement will always execute for a random access file that exists because such a file does not distinguish between read and write modes. For the purposes of the IF END statement, the end of a random access file is considered to be just beyond the final record in the file. The LOC and LOF functions described in Section 10.6 can also be used to determine whether or not there is any data between the current pointer position in a random access file and the end of the file.

If an IF END statement is executed for a sequential access file that is in read mode but that has not yet been referenced by a READ or INPUT statement, the IF END statement will assume that the file does not have line numbers. Thus, if an IF END statement is executed for a line-numbered file that has not been referenced by a READ statement, the IF END statement will treat line numbers as data items and

will erroneously report that there is data in the file if only line numbers remain in the file. As soon as a READ or INPUT statement is executed for a file, the IF END statement correctly interprets the kind of file (line-numbered or nonline-numbered) and can distinguish between line numbers and data.

The following example shows how the IF END statement works for sequential access files.

```
10      FILES TEST
20      SCRATCH #1
30      FOR X=1 TO 5
40      READ A
50      WRITE #1, A
60      NEXT X
70      RESTORE #1
80      FOR I=1 TO 1 TO 10
90      PRINT "I =2; I,
100     IF END #1 THEN 170
110     READ #1, B(I)
120     PRINT B(I)
130     NEXT I
140     PRINT "FAILED"
150     STOP
160     DATA -1,-2,-3,-4,-5,-6,-7,-8,-9,-10
170     END

RUNNH

I = 1   -1
I = 2   -2
I = 3   -3
I = 4   -4
I = 5   -5
I = 6

TIME:   0.10 SECS.
```

If the final record written into a random access file is record number 1804, for example, the IF END statement will cause a transfer when it is executed only if the pointer for that file has a value of 1805 or greater at that time.

CHAPTER 11
FORMATTED OUTPUT

The user who wishes to control the format of his output more than is permitted by the PRINT, PRINT#, and WRITE# statements described in Chapters 6 and 10 can use the statements described in this chapter. These statements are PRINT USING, PRINT USING#, and WRITE USING#. They all use a special formatting string, called an image, to format their output.

11.1 THE USING STATEMENTS

The PRINT USING statement allows formatting of string and numeric output to the Teletype. The forms of the PRINT USING statement are:

```
PRINT USING line number, list
PRINT USING string formula, list
```

The PRINT USING# and WRITE USING# statements allow formatting of output to data files. PRINT USING# formats output to data files without line numbers; WRITE USING# formats output to line-numbered data files. The forms of the PRINT USING# statement are:

```
PRINT USING #N, line number, list
PRINT USING #N, string formula, list
PRINT #N, USING line number, list
PRINT #N, USING string formula, list
```

The forms of the WRITE USING# statement are:

```
WRITE USING #N, line number, list
WRITE USING #N, string formula, list
WRITE #N, USING line number, list
WRITE #N, USING string formula, list
```

N is a digit from 1 through 9 that specifies the channel that the file is on. The comma following N can be omitted in the forms in which N precedes the word USING. The list has the form:

```
formula delimiter formula delimiter...formula
```

The formulas are either string or numeric and the delimiters are commas or semicolons. At least one formula must be present in the list.

The USING statements output each formula in their lists under the control of an image that specifies the format. The image is a string of characters that describe the form of the output (integer, decimal, string, etc.) and the placement of the output on the output line. If the USING statement contains a line number as its argument, the image is on the line specified by that line number. Such a line is called an image statement and has the form:

line number : string formula

The string formula in an image statement is not enclosed in quotes. For example:

```
10      PRINT USING 20, A
20      : THE ANSWER IS ####
```

Image statements cannot be terminated by the apostrophe remarks indicator because an apostrophe can be used as a format control character in an image.

If the USING statement contains a string formula as its argument, the image is the value of the string formula. If the string formula is a string constant, it must be enclosed in quotes. An example of the image in the USING statement is:

```
10      PRINT USING "THE ANSWER IS ####", A
```

When a USING statement is executed, BASIC begins a new line of output, and the first argument in the USING statement is output into the first specification in the image. If there are more arguments in the USING statement than specifications in the image, a new output line is begun and the specifications in the image are used again. USING statements always write complete lines. The current margin set for the Teletype or the data file referenced does not affect USING statements; however, an attempt to create a line longer than 132 characters results in an error message. Quote and noquote modes do not affect USING statements; USING statements ignore both modes.

The WRITE USING# statement performs the same functions as the PRINT USING# statement except that WRITE USING# places a line number and a tab at the beginning of each line. Neither the line number nor the tab are specified in the image. WRITE USING# statements must be used for files that have line numbers, and PRINT USING# statements must be used for files that do not have line numbers. If an attempt is made to use a WRITE# or WRITE USING# statement for a file that was previously written by PRINT# or PRINT USING# statements, an error message will be issued unless an intervening SCRATCH# statement erased the file. Similarly, an attempt to use PRINT# or PRINT USING# statements for a file that was previously referenced by WRITE# or WRITE USING# statements results in an error message unless an intervening SCRATCH# statement erased the file.

An example of PRINT USING# and WRITE USING# is shown below.

```
10      FILES TEST1, TEST2
20      SCRATCH #1, #2
30      A$ = "THE INDEX IS ##"
40      FOR T = 1 TO 3
50      PRINT USING #1, A$, T
60      WRITE USING #2, A$, T
70      NEXT T
80      END
RUNNH

TIME:  0.01 SECS.

READY
COPY TEST1 > TTY:
THE INDEX IS 1
THE INDEX IS 2
THE INDEX IS 3

READY
COPY TEST2 > TTY:
1000 THE INDEX IS 1
1010 THE INDEX IS 2
1020 THE INDEX IS 3

READY
```

11.2 IMAGE SPECIFICATIONS

An image is a string that contains format characters and printing characters. The format characters form specifications that describe how the values of the arguments of the USING statement will be arranged on an output line. More than one specification can be present in an image, but to avoid ambiguities when outputting numbers, the user should separate numeric specifications by string specifications, printing characters, or spaces. Note that spaces are printing characters and, therefore, as many spaces as are inserted between specifications will be inserted between the output items. That is, if two spaces separate a numeric specification from the preceding specification, two spaces will separate the numbers that are output according to these specifications. If numeric specifications are not separated from one another, ambiguities will generally exist and BASIC will make arbitrary decisions about the specifications. In general, it will accept as much of the specifications as it can, stopping when a character is seen that clearly delimits a specification because it considers that it has reached the end of the specification. String specifications need not be separated from one another because they are not ambiguous. Printing characters are output exactly as they appear in the image.

Image specifications can be divided into three major kinds:

a. Numeric image specifications
b. Edited numeric image specifications
c. String image specifications

11.2.1 Numeric Image Specifications

Numeric image specifications are used to describe the formats of integer and decimal numbers. Format characters within the image specification indicate the digits, sign, decimal point, and exponent of the number. Numbers in BASIC normally contain eight significant digits, and never contain more than nine significant digits. If a numeric image specification would cause a number to be output with more than nine significant digits, zeroes are substituted for all digits after the ninth. The format characters in all numeric image specifications must be contiguous.

The format characters used in numeric image specifications are:

(number sign)
. (decimal point)
↑↑↑↑[1] (four up-arrows)

Number signs in the specification indicate the digits in the number and a minus sign if the number is negative. At least two number signs must be present at the beginning of the image specification; an isolated number sign is treated as a printing character. A number sign is written in the image specification for each digit in the number to be output plus one additional number sign to indicate a minus sign if the number to be output is negative. For example, to output a negative four-digit integer, at least five number signs should be written in the image specification; a non-negative number containing four digits requires only four number signs.

11.2.1.1 Integer Image Specifications - Numbers can be output as integers by means of an image specification containing only number signs. As stated above, an additional number sign must be included in the image specification for a minus sign if the number is negative. If the number is positive or zero, no sign is output; if the number is negative, a minus sign is output. If insufficient characters are present in the image specification, an ampersand (&) is placed in the first position of the output field and the field is widened to the right to accommodate the number. If the image specification width is larger than necessary to accommodate the number, the number is right-justified in the output field. The number to be output is truncated to an integer if it is not an integer. An example showing integer image specifications follows.

```
10      READ A,B,C,D,E
20      DATA 25.6, -14.7, 4, -9.1, 41876.3
30      PRINT USING "#### ###", A, B, C
40      A$ ="#####"
50      PRINT USING A$, D, E
60      END

RUNNH

  25   -14
   4
  -9
& -41876
```

[1] On some Teletypes, the circumflex (^) is used instead of the up-arrow (↑).

11.2.1.2 Decimal Image Specifications - Decimal image specifications must contain number signs, as in integer image specifications, and a single decimal point. Optionally, the user can include four up-arrows (↑↑↑↑) at the end of a decimal specification to indicate that the number is to be output with an explicit exponent. A number output under control of a decimal image specification always contains an explicit decimal point.

When four up-arrows are present in the image specification, an explicit exponent is output in the form E±nn. The sign of the exponent is always output, a plus sign for positive or zero exponents, a minus sign for negative exponents (e.g., E+01).

The decimal point in the image specification causes the decimal point to be fixed in the output field. Thus number signs that precede the decimal point in the image specification reserve space in the output field for the digits before the decimal point and a minus sign if the number is negative. At least one digit is always output before the decimal point, even if the digit is zero. The number signs that follow the decimal point in the image specification reserve space for the digits after the decimal point in the output field.

If the number is to be output with an explicit exponent, a position must be reserved for the sign of the number even if the number is positive (a space is output for the sign of a positive number). When the number with the exponent is output all of the positions before the decimal point in the output field are used and the exponent is adjusted accordingly. If the number is not to be output with an explicit exponent, and more spaces are reserved before the decimal point than are necessary, the number is right-justified in the output field and leading spaces are appended. If insufficient spaces are reserved before the decimal point, an ampersand (&) is placed in the first position of the output field and the field is widened to the right to accommodate the number.

Whether or not the number is output with an explicit exponent, as many digits are output following the decimal point as there are number signs following the decimal point in the image specification. The number is rounded or trailing zeros are added if necessary.

An example of the use of decimal image specifications follows.

```
10      READ A,B,C,D,E,F
20      :####.## ##. ##.####
30      PRINT USING 20, A,B,C,D,E,F
40      DATA 100.256, 3.6, 19.11112
50      DATA -4.6, 3, 0.01256
60      PRINT
70      PRINT USING 80, 100.2, 14
80      :###.#↑↑↑↑  ###.↑↑↑↑
90      END

RUNNH

 100.26    4.  % 19.1111
  -4.60    3.    0.0126

 10.0E+01    14.E+00
```

11.2.2 Edited Numeric Image Specification

For those users who wish to output numbers in a form suitable for accounting reports, payrolls, and the like, additional format characters can be included in numeric image specifications to cause the numbers to be edited. The format characters used for edited numeric specifications are:

, (comma)
- (minus sign)
* (asterisk)
$ (dollar sign)

Comma

One or more commas in the integer part of a numeric image specification causes the digits in the output number to be grouped into hundreds, thousands, etc., and separated by commas (e.g., 1,000,000). The commas, however, cannot be in the first two places in the specification. Only one comma need be present in the image specification for the number to be output with commas in the required places, but a pound sign or a comma must be present in the image specification to reserve space for each comma to be output. For example, to print the number 1,365,072, the image specification must contain one comma and at least eight pound signs and/or commas. It is useful, however, to position commas in the specification where they will appear when they are output, e.g., ##,###,###. A comma that is not part of a numeric image specification is treated as a printing character.

Example:

```
10      PRINT USING " ####,###",1E4,1E5,1E6
20      PRINT
30      PRINT USING 40, -141516.8
40      :###,#####.#
50      END

RUNNH

   10,000
  100,000
 &1,000,000

 -141,516.8
```

Trailing Minus Sign

A trailing minus sign in a numeric image specification causes the number to have its sign printed at its end, rather than at its beginning (e.g., 27-). A trailing minus sign in a number is often used in a report to indicate a debit. When a trailing minus sign is present in the image specification, a position need not be saved at the beginning of the image specification for the sign of the number, since the minus sign reserves a place for the sign. When the trailing minus sign is present in the image specification and the output number is positive or zero, the sign field on output is blank. A minus sign that does not end a numeric image specification is treated as a printing character.

Example:

```
10      PRINT USING "###-",10,-14,137.8
20      PRINT
30      PRINT USING 40, -141516.8, -14
40      :##,#####.#- ##.↑↑↑↑-
50      END

RUNNH

 10
 14-
137

141,516.8- 14.E+00-
```

Leading Asterisk

If a numeric image specification begins with two or more asterisks instead of number signs, the number is output with leading asterisks filling any unused positions in the output field. Leading asterisks are often used when printing checks or in any application that requires that the numbers be protected (i.e., so that no additional digits can be added).

Within an image specification, an asterisk can replace one or all of the number signs. In image specifications with leading asterisks, negative numbers can be output only if there is a trailing minus sign in the image specification. If a trailing minus sign is not present in the image specification and an attempt is made to output a negative number, an error message will be issued. Thus, an additional position need not be saved for a leading sign. Four up-arrows cannot be present in an image specification that contains leading asterisks. Thus, numbers with explicit exponents cannot be output with leading asterisks. An isolated asterisk in an image is treated as a printing character.

An example showing image specifications with leading asterisks follows.

```
10      A$="***.**"
20      READ X,Y,Z,W,U
25      DATA 13.56, 4.577, 3.1, 19.612, 100.50
30      PRINT USING A$, X,Y,Z,W,U
35      PRINT
40      PRINT USING "*****,*** ****-", 1E6,-1E3
50      END

RUNNH

*13.56
**4.58
**3.10
*19.61
100.50

1,000,000 1000-
```

Floating Dollar Sign

If a numeric image specification begins with two or more dollar signs instead of number signs, the number is output with a floating dollar sign. That is, a dollar sign is always output in the position immediately preceding the first digit of the number, even if there are fewer digits in the number than there positions specified in the image specification. This capability is often used to protect checks so that there are never any spaces left between the dollar sign and the number.

The dollar sign can replace any or all of the number signs in the image specification (i.e., $$$$.$$ is exactly the same as $$##.##). An additional position at the beginning of the image specification must be indicated to save a place for the dollar sign in the output field.

When the floating dollar sign is used in the image specification, negative numbers can be output only if there is a trailing minus sign. If a trailing minus sign is not present in the image specification and an attempt is made to output a negative number, an error message is issued. Thus, a space need not be reserved for a leading sign in the output field. Four up-arrows cannot be present in a numeric image specification that contains dollar signs. Thus, numbers with explicit exponents cannot be output with floating dollar signs. An isolated dollar sign in an image is treated as a printing character.

An example showing floating dollar sign specifications follows. Note that the image in line 10 contains a decimal numeric image specification that is preceded by a dollar sign. This single dollar sign is treated as a printing character and, as shown in the example, is fixed in the output field.

```
10      :$$$$.$$  $####.##
20      READ A,B,C
25      DATA 100.43, 19.678, 0.97
30      PRINT USING 10, A,A,B,B,C,C
35      PRINT
40      PRINT USING "$$,,,,",1000
50      END

RUNNH

$100.43   $ 100.43
 $19.68   $  19.68
  $0.97   $   0.97

$1,000
```

11.2.3 String Image Specifications

The string image specifications allow the user to right-justify, left-justify, or center strings in the output field. In addition, the user can specify an image that causes the width of the output field to be extended if the string is larger than the image specifies. The format characters used for string output are:

' (apostrophe)
C (center)
L (left-justify)
R (right-justify)
E (extend)

A string image specification contains one apostrophe (') and as many of the format characters C, L, R, or E as are necessary to output the string. The apostrophe is counted with the format characters when BASIC determines the length of the output field. The format characters cannot be mixed within an image specification. If the image specification contains only the apostrophe, only the first character in the string is output. The characters in a string image specification must be contiguous.

C Format Character

C format characters following the apostrophe in a string image specification cause the string to be centered in the output field. If a string cannot be exactly centered (e.g., a two-character string in a three-character field), it will be off-center one character position to the left. If the string to be output is longer than the image specification, the string is left-justified in the output field and the rightmost characters that overflow are truncated.

L Format Character

L format characters following the apostrophe in a string image specification cause the string to be left-justified in the output field. If the string to be output is longer than the image specification, the string is left-justified in the field and the rightmost characters that overflow are truncated.

R Format Character

R format characters following the apostrophe in a string image specification cause the string to be right-justified in the output field. If the string to be output is longer than the image specification, the string is left-justified in the field and the rightmost characters that overflow are truncated.

E Format Character

E format characters following the apostrophe in a string image specification cause the string to be left-justified in the output field. If the string to be output is longer than the image specification, the output field is widened (extended) to the right to accommodate all the characters in the string.

The following example shows the use of string image specifications.

```
100      : 'CCCC  'EEEE  'LLLL  'KKKK  '
110      INPUT A$
120      IF A$="STOP" GO TO 150
130      PRINT USING 100, A$,A$,A$,A$,A$
140      GO TO 110
150      END

RUNNH

 ?ABCD
 ABCD   ABCD   ABCD    ABCD  A
 ?ABCDEF
 ABCDE  ABCDEF  ABCDE  ABCDE  A
 ?A
   A    A       A         A  A
 ?STOP
```

Note that the last three fields in the second line printed are displaced one position because of the field extension necessary in the second field of the line.

11.2.4 Printing Characters

All characters in an image that are not format control characters are printing characters. Printing characters are output exactly as they appear in the image. Format control characters only appear as part of image specifications; if a character used as a format control character (e.g., $, E, *) does not appear as part of an image specification, it is treated as a printing character. If the USING statement does not use all of the specifications in an image, the printing characters following the unused specifications are not printed. Similarly, if the USING statement uses the specifications in an image more than once, the printing characters in the image will be output as many times as the image is used. An example showing the use of printing characters in images follows.

```
10       :A=### AND THE SQUARE ROOT OF A=###
20       A=25
25       PRINT USING 10, A, SQR(A)
30       END

RUNNH

A= 25 AND THE SQUARE ROOT OF A= 5
```

APPENDIX A
SUMMARY OF BASIC STATEMENTS

A.1 ELEMENTARY BASIC STATEMENTS

The following subset of the BASIC command repertoire includes the most commonly used commands and is sufficient for solving most problems.

DATA [data list] READ [sequence of variables]	DATA statements are used to supply one or more numbers or alphanumeric strings to be accessed by READ statements. READ statements, in turn, assign the next available data, numeric or string as appropriate, in the DATA statement to the variables listed.
PRINT [sequence of formulas and format control characters]	Types the values of the specified formulas, which may be separated by format control characters. If two formulas are not separated by one or more format control characters, they are treated as though they were separated by a semicolon.
LET [variable] = [formula] or [variable] = [formula]	Assigns the value of the formula to the specified variable.
GO TO [line number]	Transfers control to the line number specified and continues execution from that point.
IF [formula] [relation] [formula], {THEN / GO TO} [line number]	If the stated relationship is true, then transfers control to the line number specified; if not, continues in sequence. The comma preceding THEN and GO TO is optional.
FOR [numeric variable] = [formula$_1$] TO [formula$_2$] STEP [formula$_3$] NEXT [numeric variable] or FOR [numeric variable] = [formula$_1$] TO [formula$_2$] BY [formula$_3$] NEXT [numeric variable]	Used for looping repetitively through a series of steps. The FOR statement initializes the variable to the value of formula$_1$ and then performs the following steps until the NEXT statement is encountered. The NEXT statement increments the variable by the value of formula$_3$. (If omitted, the increment value is assumed to be +1.) The resultant value is then compared to the value of formula$_2$. If variable $<$ formula$_2$, control is sent back to the step following the FOR statement and the sequence of steps is repeated; eventually, when variable $>$ formula$_2$, control continues in sequence at the step following NEXT.

ON [x] , {GO TO / THEN} [line number$_1$,] [line number$_2$,] .... [line number$_n$]	If the integer portion of x = 1, transfers control to line number$_1$, if x = 2, to line number$_2$, etc. [x] may be a formula. The comma preceding GO TO and THEN is optional.
DIM [variable] (subscript)	Enables the user to enter a table or array with a subscript greater than 10 (i.e., more than 10 items).
END	Last statement to be executed in the program, and must be present.

A.2 ADVANCED BASIC STATEMENTS

GOSUB [line number] Subroutine { [line number] . . . . . RETURN	Simplifies the execution of a subroutine at several different points in the program by providing an automatic RETURN from the subroutine to the next sequential statement following the appropriate GOSUB (the GOSUB which sent control to the subroutine).
INPUT [variable(s)]	Causes typeout of a ? to the user and waits for user to respond by typing the value(s) of the variable(s).
STOP	Equivalent to GO TO [line number of END statement].
REM	Permits typing of remarks within the program. The insertion of short comments following any BASIC statement (except an image statement) is accomplished by preceding such comments with an apostrophe (').
RESTORE	Sets pointer back to beginning of string of DATA values.
CHANGE [string formula / or / numeric vector] TO [numeric vector / or / string variable]	Changes a string formula to a numeric vector, or changes a numeric vector to a string variable.
CHAIN [string formula] or CHAIN [string formula] , [numeric formula]	Stops execution of the current program and begins execution of the new program at the beginning or at the specified line.
MARGIN [numeric formula]	Sets the Teletype to the specified output margin.
PAGE [numeric formula]	Output to the Teletype is divided into pages of the specified length.
NOPAGE	Output to the Teletype is not divided into pages.
QUOTE	The Teletype is set to quote mode (see Chapter 10).
NOQUOTE	The Teletype is set to noquote mode (see Chapter 10).

PRINT USING [line number / or / string formula] , [sequence of formulas]	Types the values of the formulas in the format determined by the image specified by the line number or string formula.

A.3 MATRIX INSTRUCTIONS

NOTE

The word "vector" may be substituted for the word "matrix" in the following explanations.

MAT READ a, b, c	Read the three matrices, their dimensions having been previously specified.
MAT c = ZER	Fill out c with zeros.
MAT c = CON	Fill out c with ones.
MAT c = IDN	Set up c as an identity matrix.
MAT PRINT a, b, c	Print the three matrices.
MAT INPUT v	Input a vector.
MAT b = a	Set matrix b = matrix a.
MAT c = a + b	Add the two matrices, a and b.
MAT c = a - b	Subtract matrix b from matrix a.
MAT c = a * b	Multiply matrix a by matrix b.
MAT c = TRN(a)	Transpose matrix a.
MAT c = (k) * a	Multiply matrix a by the number k. (k, which must be in parentheses, may also be given by a numeric formula.)
MAT c = INV(a)	Invert matrix a.

(Refer to Section A.5 for the special matrix functions NUM and DET.)

A.4 DATA FILE STATEMENTS

FILE [sequence of [channel specifier] [filename arguments]]	Assigns files to channels during program execution.
FILES [sequence of filename arguments]	Assigns files to channels before program execution begins.
SCRATCH [sequence of channel specifiers]	Erases a sequential access file and puts it in write mode; or erases a random access file and sets the record pointer to the beginning of the file.
RESTORE [sequence of channel specifiers]	Puts a sequential access file in read mode or sets the record pointer for a random access file to the beginning of the file.

WRITE [channel specifier] [sequence of formulas]	Causes data to be output to a file on the specified channel. Used for sequential access files with line numbers, or for random access files.
READ [channel specifier] [sequence of variables]	Causes data to be input from a file on the specified channel. Used for sequential access files with line numbers or for random access files.
PRINT [channel specifier] [sequence of formulas]	Causes data to be output to a file on the specified channel. Used for sequential access files without line numbers or for random access files.
INPUT [channel specifier] [sequence of variables]	Causes data to be input from a file on the specified channel. Used for sequential access files without line numbers or for random access files.
IF END [channel specifier], {THEN / GO TO} [line number]	Determines whether or not there is data in a file between the current position and the end of the file. The comma preceding THEN or GO TO is optional.
MARGIN [sequence of [channel specifier] [numeric formula]]	Sets the specified output margins for the sequential access files on the specified channels.
MARGIN ALL [numeric formula]	Sets the specified output margin for the sequential access files on channels 1 through 9.
PAGE [sequence of [channel specifier] [numeric formula]]	Sets the specified output page sizes for the sequential access files on the specified channels.
PAGE ALL [numeric formula]	Sets the specified output page size for the sequential access files on channels 1 through 9.
NOPAGE [sequence of channel specifiers]	Output to the sequential access files on the specified channels is not divided into pages.
NOPAGE ALL	Output to the sequential access files on channels 1 through 9 is not divided into pages.
QUOTE [sequence of channel specifiers]	Puts the sequential access files on the specified channels into quote mode (see Chapter 10).
QUOTE ALL	Puts the sequential access files on channels 1 through 9 into quote mode (see Chapter 10).
NOQUOTE [sequence of channel specifiers]	Puts the sequential access files on the specified channels into noquote mode (see Chapter 10).
NOQUOTE ALL	Puts the sequential access files on channels 1 through 9 into noquote mode (see Chapter 10).
SET [sequence of [channel specifier] [numeric formula]]	Moves the record pointers for random access files.
PRINT [channel specifier], USING [line number / string formula], [sequence of formulas] or PRINT USING [channel specifier], [line number / string formula], [sequence of formulas]	Causes data to be output to a sequential access file without line numbers on the specified channel. The data is output in the format determined by the image specified by the line number or string formula. In the first form, the comma following the channel specifier can be omitted.

WRITE [channel specifier], USING [line number or string formula], [sequence of formulas]
or
WRITE USING [channel specifier], [line number or string formula], [sequence of formulas]

Causes data to be output to a line-numbered sequential access file on the specified channel. The data is output in the format determined by the image specified by the line number or string formula. In the first form, the comma following the channel specifier can be omitted.

A.5 FUNCTIONS

In addition to the common arithmetic operators of addition (+), subtraction (-), multiplication (*), division (/), and exponentiation (↑ or **), BASIC includes the following elementary numeric functions:

SIN	(numeric formula)	COT	(numeric formula)	LOG, or LN, or LOGE	(numeric formula)
COS	(numeric formula)	ATN	(numeric formula)	ABS	(numeric formula)
TAN	(numeric formula)	EXP	(numeric formula)	SQR or SQRT	(numeric formula)
				CLOG or LOG10	(numeric formula)

Some advanced numeric functions include the following:

INT (numeric formula)	Finds the greatest integer not greater than its argument.
RND	Generates random numbers between 0 and 1. The same set of random numbers can be generated repeatedly for purposes of program testing and debugging. The statement RANDOMIZE can be used to cause the generation of new sets of random numbers.
SGN (numeric formula)	Assigns a value of 1 if its argument is positive, 0 if its argument is 0, or -1 if its argument is negative.

Two functions used with matrix computations are as follows:

NUM	Equals number of components following an INPUT.
DET	Equals the determinant of a matrix after inversion.

Two functions for use with random access files are:

LOC (channel specifier)	Returns the number of the current record in the file on the specified channel.
LOF (channel specifier)	Returns the number of the last record in the file on the specified channel.

Functions for manipulating strings are:

ASC (one character or a 2- or 3-letter code)	Returns the decimal ASCII code for its argument. The two- or three-letter codes are listed in Table 8-1.
CHR$ (numeric formula)	The opposite function to ASC. The argument is truncated to an integer that is interpreted as an ASCII decimal number; a one-character string is returned.
INSTR (numeric formula, string formula, string formula) or INSTR (string formula, string formula)	Searches for the second string within the first string argument. In the first form, the search starts at the character position specified by the numeric formula, truncated to an integer. In the second form, the search starts at the beginning of the string. Returns zero if the substring not found; returns the position of the first character in the substring if it is found.
LEFT$ (string formula, numeric formula)	Returns a substring of the string formula, starting from the left. The substring contains the number of characters specified by the numeric formula truncated to an integer.
LEN (string formula)	Returns the number of characters in its argument.
MID$ (string formula, numeric formula, numeric formula) or MID$ (string formula, numeric formula)	Returns a substring of the string formula, starting at the character position specified by the first numeric formula truncated to an integer. In the first form, the substring contains the number of characters specified by the second numeric formula truncated to an integer. In the second form, the substring continues to the end of the string.
RIGHT$ (string formula, numeric formula)	Returns a substring of the string formula, starting from the right, containing the number of characters specified by the numeric formula truncated to an integer.
SPACE$ (numeric formula)	Returns a string of the number of spaces specified by the numeric formula truncated to an integer.
STR$ (numeric formula)	Returns a string representation of its argument.
VAL (string formula)	The opposite function to STR$. Returns the number that the string argument represents.

The user can also define his own functions by use of the DEFine statement. For example,

```
[line number]   DEF   FNC(x) = SIN (x) + TAN(x) - 10
```

where x is a dummy variable. (Define the user function FNC as the formula SIN(x) + TAN(x) - 10.)

NOTE

DEFine statements may be extended onto more than one line; all other statements are restricted to a single line (refer to Section 5.1.5).

APPENDIX B
BASIC DIAGNOSTIC MESSAGES

Most messages typed out by BASIC are self-explanatory. BASIC diagnostic messages are divided into three categories and listed in the three tables below:

a. Command errors in Table B-1
b. Compilation errors in Table B-2
c. Execution errors in Table B-3

Table B-1
Command Error Messages

Message	Explanation
?CANNOT OUTPUT filenm.ext	A COPY, SAVE, or REPLACE command could not enter a file to output it. The actual name of the file is typed, not filenm.ext.
?CATALOG DEVICE MUST BE DISK OR DECtape	A device other than disk or DECtape was specified in a CATALOG command.
?COMMAND ERROR (YOU MAY NOT OVER-WRITE LINES OR CHANGE THE ORDER)	The given RESEQUENCE command would have changed the order of lines in the file. The command is ignored.
?COMMAND ERROR (LINE NUMBERS MAY NOT EXCEED 99999)	The given RESEQUENCE command is not executed for that reason.
?DELETE COMMAND MUST SPECIFY WHICH LINES TO DELETE	A DELETE command has no arguments.
?DUPLICATE FILENAME, REPLACE OR RENAME	User tried to SAVE a file that already exists.
?FILE NOT FOUND	A file that was requested did not exist.
?LINE TOO LONG	A line of input is greater than 142 characters, not counting the terminating carriage return, line feed.
?MISSING LINE NUMBER FOLLOWING LINE nn*	During a WEAVE or OLD command, a line without a line number was found in the file. The line is thrown away.
?NO SUCH DEVICE	The device is not available.

*If the current program was called by a CHAIN statement, the name of the current program is appended to the error message.

Table B-1 (Cont)
Command Error Messages

Message	Explanation
?QUOTA EXCEEDED ON OUTPUT DEVICE	All of the space allowed on the output device has been used; no more can be output to this device unless some of the user's files are deleted from it.
?UNSAVE DEVICE MUST BE DISK OR DECtape	A device other than disk or DECtape was specified in an UNSAVE command.
?WHAT?	Catchall command error.
?mm IN LINE nn	During a RESEQUENCE command, line nn was found to contain undefined line number mm.

Table B-2
Compilation Error Messages

Message	Explanation
?BAD DATA INTO LINE n	Input data is not in correct form.
?DATA IS NOT IN CORRECT FORM	Incorrect number or string data in: DATA statement, TTY INPUT, or TTY MAT INPUT. Following an INPUT error, the user is invited to retype the line.
?END IS NOT LAST IN nn	
?EOF IN LINE nn	An attempt was made to read data from a file after all data had been read.
?FAILURE ON ENTRY IN LINE nn	Channel is not available for SCRATCH command.
?FILE filenm.ext ON MORE THAN ONE CHANNEL IN nn	The user tried to establish the same file on more than one channel. The actual filename and extension are typed, not filenm.ext.
?FILE NEVER ESTABLISHED-REFERENCED IN LINE nn	File was not referenced in a FILES command.
?FILE NOT FOUND BY RESTORE COMMAND IN LINE nn	File was never written.
?FILE RECORD LENGTH OR TYPE DOES NOT MATCH IN nn	An existing random access file does not match the type or record length specified for it in a FILES statement.
?FNEND BEFORE DEF IN nn	FNEND occurs, but not in a function DEF.
?FNEND BEFORE NEXT IN nn	A FOR occurred in a DEF, but its NEXT did not.
?FOR WITHOUT NEXT IN nn	
?GOSUB WITHIN DEF IN nn	A GOSUB statement is within a multiple line DEF.
?FUNCTION DEFINED TWICE IN nn	
?ILLEGAL CHARACTER IN nn	A meaningless character; e.g., DIM# (1).
NOTE: If the current program was called by a CHAIN statement, the name of the current program is appended to all compilation error messages. For example, NO DATA IN TEST.BAK.	

Table B-2
Compilation Error Messages

Message	Explanation
?BAD DATA INTO LINE n	Input data is not in correct form.
?CHANNEL SPECIFIER NOT IN CORRECT FORM IN LINE nn	The channel specifier in a file-handling statement is not in the correct form.
?DATA IS NOT IN CORRECT FORM	Incorrect number or string data in a DATA statement.
?END IS NOT LAST IN nn	
?EOF IN LINE nn	An attempt was made to read data from a file after all data had been read.
?FAILURE ON ENTRY IN LINE nn	Channel is not available for SCRATCH command.
?FILE filenm.ext ON MORE THAN ONE CHANNEL IN LINE nn	The user tried to establish the same file on more than one channel. The actual filename and extension are typed, not filenm.ext.
?FNEND BEFORE DEF IN LINE nn	FNEND occurs, but not in a function DEF.
?FNEND BEFORE NEXT IN LINE nn	A FOR occurred in a DEF, but its NEXT did not.
?FOR WITHOUT NEXT IN LINE nn	
?GOSUB WITHIN DEF IN LINE nn	A GOSUB statement is within a multiple line DEF.
?FUNCTION DEFINED TWICE IN LINE nn	
?ILLEGAL ARGUMENT FOR ASC FUNCTION IN LINE nn	
?ILLEGAL CHARACTER IN LINE nn	A meaningless character; e.g., DIM# (1).
?ILLEGAL CONSTANT IN LINE nn	
?ILLEGAL FORMAT IN LINE nn	Catchall for other syntax errors.
?ILLEGAL FORMULA IN LINE nn	Syntax error in arithmetic formula.
?ILLEGAL INSTRUCTION IN LINE nn	The first three non-blank, non-tab characters of the statement do not match the first three characters of any legal statement.
?ILLEGAL LINE REFERENCE IN LINE nn	BASIC syntax required an integer, but user typed something else; e.g., GO TO A.
?ILLEGAL LINE REFERENCE mm IN LINE nn	In line nn, line mm was referred to illegally because: a. Line mm is a REM b. The first character in line mm is an apostrophe ('). c. One of the lines nn or mm is inside a function; the other is not inside that function.
?ILLEGAL RELATION IN LINE nn	Incorrect IF relation.
?ILLEGAL VARIABLE IN LINE nn	

NOTE: If the current program was called by a CHAIN statement, the name of the current program is appended to all compilation error messages. For example, NO DATA IN TEST.BAK.

Table B-2 (Cont)
Compilation Error Messages

Message	Explanation
?OUT OF ROOM	Cannot get more core to make room for: a. More compilation space. b. Maximum space for all the vectors and arrays. c. Space to store another string during execution.
?RETURN WITHIN DEF IN nn	A RETURN statement is within a multiple line DEF.
?STRING RECORD LENGTH > 132 OR < 1 IN nn	The length of a record in a string random access file was specified as greater than 132 or less than 1.
?STRING VECTOR IS 2-DIM ARRAY	The user managed to do this error despite many other checks.
?SYSTEM ERROR	An I/O error, or the UUO mechanism drops a bit, or something similar to those errors.
?TOO MANY FILES	A maximum of nine files can be open at one time in a program.
?UNDEFINED FUNCTION--FNx	The actual function name, not FNx, is typed.
?UNDEFINED LINE NUMBER mm IN nn	In line nn, mm is used as a line number. Line number mm does not exist.
?USE VECTOR, NOT ARRAY IN nn	A variable previously defined as a two-dimensional array is now used in MAT input or CHANGE.
?VARIABLE DIMENSIONED TWICE IN nn	
?VECTOR CANNOT BE ARRAY IN nn	A variable previously used in a MAT INPUT or CHANGE statement is now defined as a 2-dimensional array in a DIM statement.
NOTE: If the current program was called by a CHAIN statement, the name of the current program is appended to all compilation error messages. For example, NO DATA IN TEST.BAK.	

Table B-3
Execution Error Messages

Message	Explanation
ABSOLUTE VALUE RAISED TO POWER IN nn	
?ATTEMPT TO OUTPUT A NEGATIVE NUMBER TO A $ OR * FIELD IN nn	A USING statement attempted to output a negative number to a floating dollar sign or leading asterisk field that did not end in a minus sign.
?CHARACTER POSITION < = 0 IN nn	A character position less than or equal to zero was specified in an INSTR or MIDS function.
NOTE: If the current program was called by a CHAIN statement, the name of the current program is appended to all execution error messages. For example, LOG OF ZERO IN 20 IN TEST.BAK.	

Table B-2 (Cont)
Compilation Error Messages

Message	Explanation
?UNDEFINED LINE NUMBER mm IN LINE nn	In line nn, mm is used as a line number. Line number mm does not exist.
?USE VECTOR, NOT ARRAY IN LINE nn	A variable previously defined as a two-dimensional array is now used in MAT input or CHANGE.
?VARIABLE DIMENSIONED TWICE IN LINE nn	
?VECTOR CANNOT BE ARRAY IN LINE nn	A variable previously used in a MAT INPUT or CHANGE statement is now defined as a 2-dimensional array in a DIM statement.
NOTE: If the current program was called by a CHAIN statement, the name of the current program is appended to all compilation error messages. For example, NO DATA IN TEST.BAK.	

Table B-3
Execution Error Messages

Message	Explanation
%ABSOLUTE VALUE RAISED TO POWER IN LINE nn	
?ATTEMPT TO OUTPUT A NEGATIVE NUMBER TO A $ OR * FIELD IN LINE nn	A USING statement attempted to output a negative number to a floating dollar sign or leading asterisk field that did not end in a minus sign.
?ATTEMPT TO OUTPUT A NUMBER TO A STRING FIELD OR A STRING TO A NUMERIC FIELD IN LINE nn	A USING statement attempted to output a number to a string field or a string to a numeric field.
?ATTEMPT TO READ# OR INPUT# FROM A FILE WHICH DOES NOT EXIST IN LINE nn	An attempt was made to read from a file that does not exist on the user's disk area.
?ATTEMPT TO READ# OR INPUT# FROM A FILE WHICH IS IN WRITE# OR PRINT# MODE IN LINE nn	An attempt was made to read from a sequential access file that is not in read mode.
?ATTEMPT TO WRITE A LINE NUMBER >99,999 IN LINE nn	
?ATTEMPT TO WRITE# OR PRINT# TO A FILE WHICH HAS NOT BEEN SCRATCH#ED IN LINE nn	An attempt was made to write to a sequential access file that is not in write mode.
?ATTEMPT TO WRITE# OR PRINT# TO A FILE WHICH IS IN READ# OR INPUT# MODE IN LINE nn	An attempt was made to write to a sequential access file that is not in write mode.
?CHR$ ARGUMENT OUT OF BOUNDS IN LINE nn	The argument to the CHR$ function was less than zero or greater than 127.
?DATA FILE LINE TOO LONG IN LINE nn	An attempt has been made to read from a data file a line which is greater than 142 characters long.
?DIMENSION ERROR IN LINE nn	
NOTE: If the current program was called by a CHAIN statement, the name of the current program is appended to all execution error messages. For example, LOG OF ZERO IN 20 IN TEST.BAK.	

Table B-3 (Cont)
Execution Error Messages

Message	Explanation
%DIVISION BY ZERO IN LINE nn	†
?EXPONENT REQUESTED FOR * OR $ FIELD IN LINE nn	
?FILE IS NOT RANDOM ACCESS IN LINE nn	
?FILE NEVER ESTABLISHED -- REFERENCED IN LINE nn	
?FILE NOT FOUND BY RESTORE COMMAND IN LINE nn	
?FILE filenm.ext ON MORE THAN ONE CHANNEL IN LINE nn	The user tried to establish the same file on more than one channel. The actual filename and extension are typed, not filenm.ext.
?FILE NOT IN CORRECT FORM IN LINE nn	A data error has been detected in a string random access file.
?FILE RECORD LENGTH OR TYPE DOES NOT MATCH IN LINE nn	An existing random access file does match the type or record length specified for it in a FILE statement.
?IF END ASKED FOR UNREADABLE FILE IN LINE nn	
?ILLEGAL CHARACTER IN STRING IN LINE nn	An attempt has been made to write onto a data file a string containing an embedded line terminator or quote.
?ILLEGAL CHARACTER SEEN IN LINE nn	An attempt has been made to create an illegal character in a CHANGE statement.
?ILLEGAL FILENAME IN LINE nn	The string argument is not in the correct form. If the argument is variable, check that it has been defined.
?ILLEGAL LINE REFERENCE IN RUN (NH) OR CHAIN	The line at which execution is to begin is inside a multiline DEF.
?IMPOSSIBLE VECTOR LENGTH IN LINE nn	In a CHANGE (to string) statement, the zero element of the number vector was negative or exceeded its maximum dimension.
?INPUT DATA NOT IN CORRECT FORM -- RETYPE LINE	
?INSTR ARGUMENT OUT OF BOUNDS IN LINE nn	

†An OVERFLOW error message means that an attempt has been made to create a number larger in magnitude than the largest number representable in the computer (approximately 1.7E+38); when this occurs, the largest representable number is returned (with the correct sign) and execution continues. An UNDERFLOW error message means that an attempt has been made to create a nonzero number smaller in magnitude than the smallest representable positive number (approximately 1.4E-39); in this case, zero is returned and execution continues. Division by zero is considered overflow; the largest representable positive number is returned.

NOTE: If the current program was called by a CHAIN statement, the name of the current program is appended to all execution error messages. For example, LOG OF ZERO IN 20 IN TEST.BAK.

Table B-3 (Cont)
Execution Error Messages

Message	Explanation
?LEFT$ ARGUMENT OUT OF BOUNDS IN LINE nn	
?LINE NUMBER OUT OF BOUNDS IN LINE nn	The line number argument is less than zero or greater than 99,999. The RUN (NH) commands return a ?WHAT? message in this situation.
%LOG OF NEGATIVE NUMBER IN LINE nn	
%LOG OF ZERO IN LINE nn	
?MARGIN OUT OF BOUNDS IN LINE nn	A MARGIN or MARGIN ALL statement specified a margin greater than 132 characters or less than 1 character.
?MARGIN TOO SMALL IN LINE nn	A WRITE# statement referenced a file that has a margin of fewer than seven characters.
?MID$ ARGUMENT OUT OF BOUNDS IN LINE nn	
?MIXED RANDOM & SEQUENTIAL ACCESS IN LINE nn	A random access statement or function referenced a sequential access file, or vice versa.
?MIXED READ#/INPUT# IN LINE nn	An attempt was made to reference a file with both a READ# and an INPUT# statement without an intervening RESTORE# statement.
?MIXED WRITE#/PRINT# IN LINE nn	An attempt was made to reference a file with both a WRITE# and a PRINT# statement without an intervening SCRATCH# statement.
%SINGULAR MATRIX INVERTED IN LINE nn	
?NEGATIVE STRING LENGTH IN LINE nn	In a MID$, LEFT$, or RIGHT$ function, a negative number of characters was specified for a substring.
?NO FIELDS IN IMAGE IN LINE nn	An image contains neither string nor numeric fields.
?NO ROOM FOR STRING IN LINE nn	In a CHANGE A$ TO A, the number of characters in A$ exceeds the maximum size of A. A DIM statement appropriately increasing the size of A will cover this.
?NO SUCH LINE IN RUN (NH) OR CHAIN	The specified line does not exist in the program.
?NOT ENOUGH INPUT -- ADD MORE	
?ON EVALUATED OUT OF RANGE IN LINE nn	The value of the ON index was < 1 or >the number of branches.
?OUT OF DATA IN LINE nn	
?OUTPUT ITEM TOO LONG FOR LINE IN LINE nn	In quote mode, an attempt was made to write a string or number that is too long to fit on one line.

NOTE: If the current program was called by a CHAIN statement, the name of the current program is appended to all execution error messages. For example, LOG OF ZERO IN LINE 20 IN TEST.BAK.

Table B-3 (Cont)
Execution Error Messages

Message	Explanation
?OUTPUT LINE >132 CHARACTERS IN LINE nn	A line of output created by a USING statement is greater than 132 characters.
?OUTPUT STRING LENGTH >RECORD LENGTH IN LINE nn	An attempt has been made to output to a random access file a string which is too long to fit in one record.
%OVERFLOW IN LINE nn	†
%OVERFLOW IN EXP IN LINE nn	An exponent greater than 88.028 has been specified for the EXP function. An answer of the largest representable number is returned and execution continues.†
?PAGE LENGTH OUT OF BOUNDS IN LINE nn	A PAGE or PAGE ALL statement specified a page length of less than one line.
?QUOTA EXCEEDED OR BLOCK NO. TOO LARGE ON OUTPUT DEVICE	Normally, this indicates that all of the space allowed on the output device has been used; no more can be output to this device unless some of the user's files are deleted from it. It may also mean that the block number is too large for the output device.
?RETURN BEFORE GOSUB IN LINE nn	
?RIGHT$ ARGUMENT OUT OF BOUNDS IN LINE nn	
?SET ARGUMENT OUT OF BOUNDS IN LINE nn	The user attempted to set the value of the pointer to zero or to a negative number.
?SPACE$ ARGUMENT OUT OF BOUNDS IN LINE nn	The SPACE$ function was requested to return a string that was less than or equal to zero or greater than 132 characters.
%SQRT OF NEGATIVE NUMBER IN LINE nn	
?STRING FORMULA >132 CHARACTERS IN LINE nn	A string formula contains more than 132 characters.
?STRING RECORD LENGTH >132 OR <1 IN LINE nn	The record length for a string random access file was specified as less than one or greater than 132 characters.

†An OVERFLOW error message means that an attempt has been made to create a number larger in magnitude than the largest number representable in the computer (approximately 1.7E +38); when this occurs, the largest representable number is returned (with the correct sign) and execution continues. An UNDERFLOW error message means that an attempt has been made to create a nonzero number smaller in magnitude than the smallest representable positive number (approximately 1.4E -39); in this case, zero is returned and execution continues. Division by zero is considered overflow; the largest representable positive number is returned.

NOTE: If the current program was called by a CHAIN statement, the name of the current program is appended to all execution error messages. For example, LOG OF ZERO IN 20 IN TEST.BAK.

Table B-3 (Cont)
Execution Error Messages

Message	Explanation
?SUBROUTINE OR FUNCTION CALLS ITSELF IN LINE nn	FNA is defined in terms of FNB which is defined in terms of FNA, or a similar situation with FUNCTIONS or GOSUBS.
%TAN OF P1/2 OR COTAN OF ZERO IN LINE nn	
?TOO MANY ELEMENTS -- RETYPE LINE	
%UNDERFLOW IN EXP IN LINE nn	An exponent less than -88.028 has been specified for the EXP function. An answer of zero is returned and the execution continues. †
%UNDERFLOW IN LINE nn	†
?VAL ARGUMENT NOT IN CORRECT FORM IN LINE nn	The string argument to the VAL function does not represent a legal number.
%ZERO TO A NEGATIVE POWER IN LINE nn	

†An OVERFLOW error message means that an attempt has been made to create a number larger in magnitude than the largest number representable in the computer (approximately 1.7E + 38); when this occurs, the largest representable number is returned (with the correct sign) and execution continues. An UNDERFLOW error message means that an attempt has been made to create a nonzero number smaller in magnitude than the smallest representable positive number (approximately 1.4E - 39); in this case, zero is returned and execution continues. Division by zero is considered overflow; the largest representable positive number is returned.

NOTE: If the current program was called by a CHAIN statement, the name of the current program is appended to all execution error messages. For example, LOG OF ZERO IN 20 IN TEST.BAK.

APPENDIX C
TAPE AND KEY COMMANDS

The TAPE and KEY commands are designed for user Teletypes with attached paper-tape readers; for example, the LT33B shown in Figure C-1.

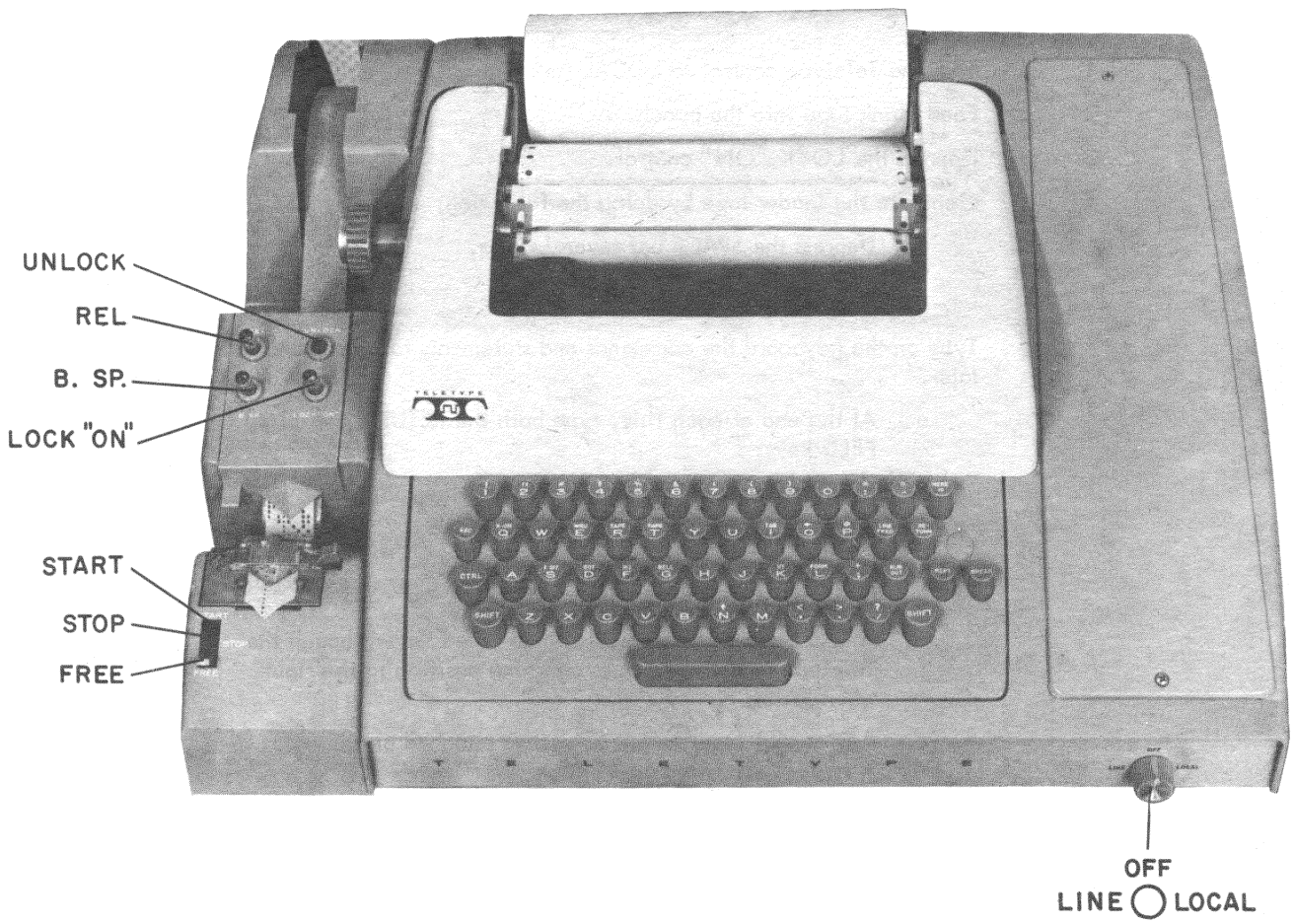


Figure C-1 LT33B Teletype

C.1 KEY AND TAPE MODES

KEY mode is produced by typing the KEY command to BASIC. In this mode, the user types input to BASIC on the keyboard in the normal manner. KEY mode is also the default mode.

TAPE mode is produced by typing the TAPE command to BASIC. The user initiates this mode whenever he wants to input from the paper tape reader while the Teletype is in LINE mode.

C.2 PREPARING AN INPUT TAPE IN LOCAL MODE

The following procedure should be followed for preparing an input tape.

Step	Procedure
1	Turn the Teletype control to LOCAL (see Figure C-1).
2	Feed blank tape into the punch.
3	Depress the LOCK "ON" control.
4	Generate the leader tape by doing the following: a. Depress the SPACE bar several times. b. Depress the RETURN key once. c. Depress the LINE FEED key once.
5	Type on the keyboard the commands and statements to be punched on the tape. a. At the end of each line, type both the RETURN and LINE FEED keys. b. If an incorrect character is typed, do the following: (1) Depress the BACKSPACE control (2) Depress the RUBOUT key. (3) Type the correct character. c. A TAB is received correctly when typed, even though the Teletype typewheel moves only one position to the right when TAB is typed. d. Any normal input to BASIC can be punched on the tape. A typical example is as follows: NEW TEST4 5 PRINT "THIS IS A TEST" 10 END LIS RUNNH
6	Generate a trailer tape by doing the following: a. Depress the SPACE bar several times. b. Depress the RETURN key once. c. Depress the LINE FEED key once.

Step	Procedure
7	Depress the UNLOCK control.
8	Remove the tape from the punch.
9	Depress the CTRL and T keys simultaneously.

C.3 SAVING AN EXISTING PROGRAM ON TAPE

The following procedure should be performed to save an existing program on tape.

Step	Procedure
1	Turn the Teletype control to LINE.
2	Depress the LOCK "ON" control.
3	Generate a leader tape by doing the following: a. Depress the SPACE bar several times. b. Depress the RETURN key once.
4	Turn the Teletype control to LOCAL.
5	Depress the UNLOCK control.
6	Depress the CTRL and T keys simultaneously.
7	Turn the Teletype control to LINE.
8	Type the LISTNH command, but do not depress the RETURN key.
9	Depress the LOCK "ON" control.
10	Depress the RETURN key.
11	Wait until the program has been listed and the READY message has been typed. NOTE The tape will contain not only your program but also an extra line at the end with the READY message on it. This is not important. Since READY is not a legal command, it will simply produce a WHAT? error message when the tape is input to BASIC, and then it will be ignored.
12	Generate a trailer tape by doing the following: a. Depress the SPACE bar several times. b. Depress the RETURN key once.
13	Turn the Teletype control to LOCAL.
14	Depress the UNLOCK control.
15	Remove the tape from the punch.
16	Depress the CTRL and T keys simultaneously.
17	Turn the Teletype control to LINE; now you are back in BASIC.

Step	Procedure
18	To stop the tape output while the program is being listed, do the following: a. Depress the UNLOCK control. b. Depress the CTRL and T keys simultaneously. c. Twice depress the CTRL and C keys simultaneously. d. Type REEN. e. Depress the RETURN key.

C.4 INPUTTING TO BASIC FROM THE READER

The following procedure should be followed for inputting to BASIC from the reader.

Step	Procedure
1	Turn the Teletype control to LINE.
2	With the reader control on STOP (see Figure C-1), position the tape on the sprocket wheel and close the tape retainer cover.
3	Type the command TAPE to BASIC.
4	Depress the RETURN key.
5	When BASIC answers READY, set the reader control to START.
6	When the tape has been read in, set the reader control to STOP.
7	Type KEY.
8	Depress the RETURN key.
9	Depress the LINE FEED key. (The Key mode is now restored.)
10	To stop the tape input while it is in progress, do the following: a. Switch the reader control to STOP. b. Twice depress the CTRL and C keys simultaneously. c. Type REEN. d. Depress RETURN. e. Depress LINE FEED. f. Type KEY. g. Depress RETURN. h. Depress LINE FEED. NOTE Do not type on the keyboard without first stopping the tape.

C.5 LISTING AN INPUT TAPE

An input tape is listed in the following manner:

Step	Procedure
1	Turn the Teletype control to LOCAL. (In LOCAL mode the tape is not inputted to the computer.)

Step	Procedure
2	Set the reader to STOP.
3	Put the tape in the reader.
4	Set the reader to START. (The contents of the tape is then printed on the console.)

decsystem10

ALGOL PROGRAMMER'S REFERENCE MANUAL

This manual reflects version 2A of the ALGOL System.

DIGITAL EQUIPMENT CORPORATION • MAYNARD, MASSACHUSETTS

1st Edition September 1971
2nd Edition December 1971
Update Pages May 1972

CONTENTS

CONTENTS (Cont)

CONTENTS (Cont)

CONTENTS (Cont)

CONTENTS (Cont)

TABLES

CHAPTER 1
INTRODUCTION

1.1 GENERAL

DECsystem-10 ALGOL is an implementation of ALGOL-60; ALGOL is an abbreviation of ALGOrithmic Language, and 1960 is the year it was defined. The authoritative definition of ALGOL-60 is contained in the "Revised Report on the Algorithmic Language ALGOL-60",[1] hereafter referred to as the "Revised Report". This report leaves a number of ALGOL-60 features undefined, notably input/output, and permits the implementer of the language some latitude in interpreting other features. Many of these features have been discussed extensively since the publication of the Revised Report; some have been given rigorous interpretations in various versions of ALGOL and the proposed ALGOL-68 Language.[2]

Where there is need for interpretation in the Revised Report, such interpretations as seem reasonable have been made in light of current ALGOL opinion. Where no guidelines exist, ALGOL-68 is used as a basis. These points are discussed in Chapter 19.

1.2 DECsystem-10 ALGOL

The purpose of this manual is to teach the use of DECsystem-10 ALGOL. The manual is written both for the user who is familiar with ALGOL implementations and for the user who has no knowledge of ALGOL but is reasonably fluent in a high-level scientific programming language such as FORTRAN IV. This manual is <u>not</u> a primer in high-level languages.[3]

---

[1] "Revised Report on the Algorithmic Language ALGOL-60", Backus et al., <u>Communications of the ACM</u>, 1963, vol. 6, no. 1, pp. 1-17.

[2] "Report on the Algorithmic Language ALGOL-68", A. Van Wijngaarden (Editor), B. M. Mailloux, J. E. L. Peck, and C. H. A. Koster, Mathematisch Centrum, Amsterdam, MR101, October 1969.

[3] <u>A Primer of ALGOL-60 Programming</u>, E. W. Dijkstra, Academic Press, London, 1962.

Readers not thoroughly familiar with ALGOL should read the entire manual. Readers already familiar with ALGOL-60 should read all chapters except Chapters 5, 6, 7, 8, 9, 10, 11, 12, and 14, which need be referred to only briefly.

1.3 THE ALGOL COMPILER

The DECsystem-10 ALGOL Compiler is that part of the DECsystem-10 ALGOL System that reads programs written in DECsystem-10 ALGOL and converts them into a form (relocatable binary) that is acceptable to the DECsystem-10 Linking Loader. The compiler is also responsible for finding errors in the user's source program and reporting them to the user.

Slight constraints are imposed on the way the user writes his program. These constraints, made to gain the most desirable feature of a single-pass compiler, concern the order in which the user declares the identifiers in the program and the use of forward declarations under certain special circumstances.

Such a compiler can process ALGOL programs rapidly and does not require the use of any backing store. The minor restrictions imposed will not normally affect the user.

1.3.1 Compiler Extensions

The following ALGOL-60 extensions are allowed by the compiler:

a. A LONG REAL type, equivalent to FORTRAN's double precision, is added that gives the user power to handle double-precision real numbers.

b. An EXTERNAL procedure facility allows the user to compile procedures separately from the main program.

c. A WHILE statement, and an abbreviated form of the FOR statement, allow the user greater flexibility of iteration.

d. A new type STRING allows the user to manipulate strings of various size bytes. In addition, the user can individually manipulate the bytes within a string by means of a byte subscripting facility.

e. An integer remainder function is provided.

f. Assignments are permitted within expressions.

g. Delimiter words may be represented in either reserved word format (upper case) or as non-reserved words enclosed in single quotes (primes).

The compiler, as supplied, uses reserved word delimiters. A version using non-reserved delimiter words enclosed in primes can be produced by means of the /Q compiler switch. Refer to Chapter 18.

1.3.2 Compiler Restrictions

If the user is unfamiliar with any of the following terminology, he should refer to the Revised Report and to Paragraph 1.5.

The compiler imposes the following restrictions on ALGOL-60:

a. Numeric labels are not permitted.

b. All formal parameters must be specified.

c. Identifiers are restricted to 64 characters in length.

d. Some minor restrictions are made in the ordering of declarations.

e. Forward references for procedures and labels must be given under certain circumstances.

1.4 THE ALGOL OPERATING ENVIRONMENT

Programs compiled by the ALGOL compiler are run in a special operating environment that provides special services, including input/output facilities for the object program.

The ALGOL operating environment consists of:

a. The ALGOL Library, known as ALGLIB - a set of routines, some of which are incorporated into the user's program by the linking loader.

b. The ALGOL Object Time System, known as ALGOTS - responsible for organizing the smooth running of the program and providing services such as core management, peripheral device allocation, and fault monitoring in case the program encounters an error condition at run time.

Refer to Chapters 17 and 18 for a description of ALGLIB and ALGOTS.

1.5 TERMINOLOGY

Some of the following words, used in this manual, may be new to the reader. Many have a FORTRAN equivalent; where such an equivalent exists, it is enclosed in parentheses.

Delimiter Word - a single, English language word that is an inherent part of the structure of the ALGOL language. Such words cannot normally be used for other purposes. Example: BEGIN IF ARRAY.

Identifier - a name, established by user declaration, that represents some quantity within a program.

Label (Statement Number) - an identifier used to mark a certain statement in a program. Control of program execution can be transferred to the statement following the label. A numeric label (not available in DECsystem-10 ALGOL) is similar to a FORTRAN statement number.

(continued on next page)

Procedure (Subroutine, Function) - part of a program, which may be invoked by "calling". In general, parameters are supplied as arguments and a result may be returned.

Parameter (Formal Parameter - Dummy Variable, Actual Parameter - Argument) See Procedure. - A Formal Parameter is an identifier used within the procedure that represents the argument supplied when the procedure is called.

CHAPTER 2
PROGRAM STRUCTURE

2.1 BASIC SYMBOLS

DECsystem-10 ALGOL programs consist of a sequence of symbols from the DECsystem-10 ASCII character set. The meaning of individual characters, given in Table 2-1, is much the same as in other high-level languages.

Table 2-1
DECsystem-10 ALGOL Symbols

Symbol	Meaning or Use
A - Z	Used to construct identifiers and delimiter words.
a - z	Lower case letters; are treated as upper case letters except when they appear in string constants and ASCII constants.
0 - 9	Decimal digits; used to construct numeric constants and identifiers.
+	Arithmetic addition operator.
-	Arithmetic subtraction operator.
*	Arithmetic multiplication operator.
/	Arithmetic division operator.
↑	Arithmetic exponentiation operator.
( )	Parentheses; used in arithmetic expressions and to enclose parameters in procedure specifications and calls.
[ ]	Square brackets; used to enclose subscript bounds in array declarations, and array subscript lists.
,	Comma; general separator, placed between array subscripts, procedure parameters, items in switch lists, etc.
.	Decimal point; used in numeric constants and byte subscripting. Also, used as a readability symbol in identifiers.
;	Semicolon; used to terminate statements.

(continued on next page)

Table 2-1 (Cont)
DECsystem-10 ALGOL Symbols

Symbol	Meaning or Use
:	Colon; used to indicate labels, and separate lower and upper bounds in array declarations.
=	Equality; used in arithmetic and string comparisons.
#	Nonequality.
< >	Less than, greater than.
& @	Introduces exponent in floating-point numbers.
'	Prime, or single quote; used to enclose delimiter words when the non-reserved word implementation is used.
"	Opening and closing string quotes.
!	Comment.
%	Introduces an octal constant.
$	Introduces an ASCII constant.
←	Alternative to := (refer to Table 2-2).

2.2 COMPOUND SYMBOLS

Compound symbols consist of two adjacent basic symbols. Any intervening spaces or tabs do not affect their use. The compound symbols are shown in Table 2-2.

Table 2-2
Compound Symbols

Symbol	Usage
:=	Assignment
< =	Less than or equal to
>=	Greater than or equal to

2.3 DELIMITER WORDS

Certain upper-case letter combinations are reserved as part of the structure of the language and may not be used as identifiers unless the compiler used is a version accepting delimiter words in single quotes. Such an option is selected by using a special switch option (refer to Chapter 18). It is assumed

throughout this manual that the standard method of delimiter word representation is used, that is, reserved words.

For example, the delimiter word

```
BEGIN
```

will always appear in the text of this manual as shown above and cannot be used as an identifier in a program. If the alternative method of representation is used, it would appear as

```
'BEGIN'
```

and

```
BEGIN
```

could be used as an identifier. Table 2-3 contains a list of all the delimiter words used in the language.

Table 2-3
Delimiter Words Used in DECsystem-10 ALGOL

Reserved Word	Chapter Reference
AND	5.2.1
ARRAY	9
BEGIN	10
BOOLEAN	5.2
CHECKOFF	18
CHECKON	18
COMMENT	2.4
DIV	5.1
DO	8
ELSE	7.3
END	10
EQV	5.2.1
EXTERNAL	11.9
FALSE	4.2
FOR	8
FORWARD	11.8
GO	7.2
GOTO	7.2
IF	7.3
IMP	5.2.1
INTEGER	3.2
LABEL	11
LINE	18
LISTOFF	18
LISTON	18
LONG	3.2

(continued on next page)

Table 2-3 (Cont)
Delimiter Words Used in DECsystem-10 ALGOL

Reserved Word	Chapter Reference
NOT	5.2.1
OR	5.2.1
OWN	15
PROCEDURE	11
REAL	3.2
REM	5.1
STEP	8
STRING	13
SWITCH	12
THEN	7.3
TRUE	4
UNTIL	8
VALUE	11
WHILE	8

2.4 USE OF SPACING AND COMMENTARY

The readability of ALGOL programs can be enhanced greatly by the judicious use of spacing, tab formatting, and commentary. Spaces, tabs, and form feeds (page throws) may be used freely in a source program subject to the following constraints:

a. Spaces, tabs, line feed, or form feed characters may not appear within delimiter words.

b. Where two delimiter words are adjacent, or where an identifier follows a delimiter word, they must be separated by one or more spaces and/or tabs.

c. Spaces, tabs etc., are significant within string constants.

d. Where the carriage return, line feed at the end of a line of source is to be ignored, a control-back arrow character should be inserted immediately before the carriage return.

Comments are introduced by either the word COMMENT or the symbol ! (available in DECsystem-10 ALGOL, but not necessarily in other implementations of ALGOL). Such a comment may appear anywhere in a program; the comment text is terminated by a semicolon. Refer to Section 11.10 for additional means to add comments to a program.

CHAPTER 3
IDENTIFIERS AND DECLARATIONS

3.1 IDENTIFIERS

An identifier must begin with an upper-case letter and optionally be followed by one or more upper-case letters and/or decimal digits. An identifier may not contain more than 64 characters.

NOTES

1. Unlike FORTRAN, there is no implied type attached to an identifier.
2. All identifiers in a program (except labels) have to be "declared", that is, the use to which they are to be put must be specified, usually before they are used.

Examples:

The following are identifiers:

```
I

ALPHA

P43

J4K5

HOUSEHOLDERTRIDIAGONALIZATION
```

The following are not identifiers:

4P	does not begin with letter
BOOLEAN	unless the non-reserved word delimiter representation is used
ONCE AGAIN	space not allowed

DECsystem-10 ALGOL also permits the use of a decimal point as a "readability symbol" in the alphabetic portion of identifiers. These readability symbols can appear between two alphabetic characters of an identifier and are ignored by the compiler. Thus:

```
ONCE.AGAIN
```

and

```
PI.BY.TWO
```

have exactly the same effect as

```
ONCEAGAIN
```

and

```
PIBYTWO
```

respectively.

Note that

```
ALPHA3.5
```

and

```
BETA.22
```

are not identifiers, since the decimal point does not appear between two alphabetic characters.

3.2 SCALAR DECLARATIONS

A declaration reserves an identifier to represent a particular quantity used in a program. Such declarations are mandatory in ALGOL. At any particular point during program execution, the form of the variable or quantity associated with the identifier depends on the type of variable. The type of variable is controlled by the type of identifier which represents it.

There are five scalar variables, that is, variables which contain a single value:

a. Integer
b. Real
c. Long Real
d. Boolean
e. String

Integer, real, and long real variables are capable of holding numerical values of the appropriate type (and only of that type). The range of values is as follows: integer: -34,359,738,368 through 34,359,738,367; real and long real: approximately -1.7&38 through 1.7&38; values less than approximately 1.4&-39 in magnitude are represented by zero.

Boolean variables (similar to FORTRAN's Logical variables) can hold a Boolean quantity, which is usually one of the states TRUE or FALSE but, in general, can be any pattern of 36 bits.

String variables are somewhat more complicated. A full discussion of their properties is presented in Chapter 13. At this point, it is sufficient to say that string variables are really pointers to byte strings.

All of the above variables can be declared for use by preceding a list of the identifiers to be used by the appropriate delimiter word for their type. Throughout this manual, a "list of items" consists of those items arranged sequentially and separated by commas.

Examples:

```
INTEGER I,J,K;
LONG REAL DOUBLE,P,Q,ELEPHANT;
BOOLEAN ISITREALLYTRUE;
STRING S,T;
```

CHAPTER 4
CONSTANTS

4.1 NUMERIC CONSTANTS

There are three forms of numeric constants:

a. Integer constants
b. Real constants
c. Long Real constants

4.1.1 Integer Constants

Integer constants consist of a number of adjacent decimal digits, subject to the constraint that the number represented must be in the range 0 through 34,359,738,367.

NOTE

Any preceding sign that appears in the program is not considered part of the constant.

Examples:

```
3

24

9276541
```

4.1.2 Real Constants

Real constants consist of a decimal number (containing either an integral part or a fractional part, or both) followed by an optional exponent. If the decimal number is unity, it may be omitted. The exponent consists of either the & or @ symbol followed by an optionally signed integer. This has the effect of multiplying the decimal number by the power of ten specified in the exponent. If no decimal number appears, a value of unity is assumed.

The range of real constants is approximately 1.4&-39 to 1.7&38; numbers less than 1.4&-39 are represented by zero. Real numbers are stored to a significance of approximately eight and one-half decimal digits.

Examples:

Representation	Value
3.141592653589793	3.14159265
.0001	0.0001
4.37&5	437000.0
5&-3	0.005
&-6	0.000001

4.1.3 Long Real Constants

Long real constants are used to represent numeric quantities to approximately twice the precision available with real numbers: about seventeen decimal digits. Long real constants are formed by writing a real constant in floating-point form, but replacing the & or @ by && or @@.

The range of long real constants is the same as that of real constants, except numbers below approximately 3.0&&-30 can only be represented to single precision due to hardware considerations.

Examples:

Representation	Value
3.14159265358979323846&&0	3.1415926535897932
12&&-3	0.012

4.2 OCTAL AND BOOLEAN CONSTANTS

Octal constants consist of the symbol % followed by a number of octal digits. Up to twelve significant digits may appear (leading zeros are ignored); these digits are right justified.

Examples:

```
%777777777774

%0470
```

Octal constants may only be used in Boolean expressions.

Boolean constants consist of the words TRUE and FALSE. They are equivalent to the octal constants %777777777777 and %000000000000, respectively.

4.3 ASCII CONSTANTS

Up to five ASCII symbols can be packed right justified to give an integer-type constant. The format is a dollar sign ($), followed by up to five ASCII symbols enclosed within a delimiting symbol pair. The leading delimiter symbol immediately follows the $, and may be a readable character or an invisible one such as a space. Thus, the user can generate a single ASCII character constant by placing one space on each side of it, and preceding the triplet by a dollar sign.

Examples:

Text	Octal Value
$ A	000000 000101
$/01234/	160713 516674

4.4 STRING CONSTANTS

String constants allow the user to store any reasonable length string of ASCII characters within a program. The length of such a constant is restricted only by the amount of core storage available to the user for the execution of the program. String constants may be used, typically, to output a message during the execution of the program.

The string of symbols is enclosed within quotes ("). There are restrictions on the symbols that may appear within the string.

a. [ ] ; and " may not appear alone.

b. [and] may appear if they are properly paired.

c. Single occurrences of [ ] ; and " are represented by [[ ]] ;; and "", respectively.

Note that [[and]] are stored as such in the byte string generated by the compiler. ;; and "" are stored as a single ; or ", respectively.

Square brackets are used to enclose symbols that have a specific effect when the string is output. These are discussed in Paragraph 16.6.2.

Examples:

```
"ABCDEFGHIJKLMNOPQRSTUVWXYZ"

"REMEMBER THAT SPACES ETC. ARE SIGNIFICANT"

"[PSC]INPUT DATA:[SC]"

""""A[[I]] := 0.1;;""""
```

CHAPTER 5
EXPRESSIONS

5.1 ARITHMETIC EXPRESSIONS

DECsystem-10 ALGOL arithmetic expressions are written in a form similar to that used in FORTRAN and many other high-level scientific computer languages. The usual algebraic rules concerning precedence of operators and brackets are followed (see Table 5-1).

Table 5-1
Operator Precedence

Operator	Priority (decreasing)
parentheses	1
exponentiation	2
multiplication and division	3
addition and subtraction	4

There are two additional operators, DIV and REM, that indicate integer division and remainder, respectively. They have the same precedence as ordinary division. Within the precedence scheme, the order of evaluation is always from left to right. For example:

```
X↑Y↑Z means (X↑Y)↑Z
```

and

```
I DIV J REM K means (I DIV J) REM K
```

Unlike FORTRAN, when ordinary division of one integer by another is performed, the real result is not rounded to an integer value.

NOTE

The operator REM is normally not available in other ALGOL implementations.

The difference between the various types of division is clarified by the following examples:

7/4 yields a result of 1.75, whereas
7 DIV 4 yields a result of 1, and
7 REM 4 yields a result of 3

The interpretation of integer division for negative integers follows:

Let M, N > 0, then

```
-M DIV N = M DIV (-N) = -(M DIV N)
-M DIV (-N) = M DIV N
```

The integer remainder operator, REM, is defined so that for all integral M,N:

```
M REM N = M - N*(M DIV N)
```

5.1.1 Identifiers and Constants

Arithmetic expressions consist of operands, that is, identifiers and constants, of the three types, integer, real and long real, together with the arithmetic operands + - * / DIV REM and ↑ and parentheses where necessary.

Identifiers are used to represent variables whose values are used when they appear in some calculation.

Since automatic conversion takes place as necessary when an expression is evaluated, the user may freely mix the three different types of identifiers and constants.

Integer quantities may have more precision than can be represented in a real variable. The user must beware of possible loss of significance in integral quantities used in mixed type expressions.

5.1.2 Special Functions

Three special functions are provided for use in arithmetic expressions. The first is the transfer function, ENTIER, which converts a real or long real quantity into an integer quantity defined as the largest integer value not exceeding the argument.

Thus

```
ENTIER(3.5) = 3
```

and

```
ENTIER(-3.5) = -4
```

The special function ABS yields the absolute value (also known as the modulus) of its argument. The argument may be any integer, real, or long real quantity; the result is always of the same type as the argument.

Thus

```
ABS(-3.5) = 3.5
```

and

```
ABS(-3) = 3
```

The special function SIGN is the signum function whose argument can be integer, real, or long real. The result is always integral, being minus one or zero or plus one, depending on whether the argument is negative, zero, or greater than zero, respectively.

Thus

```
SIGN(-3.5) = -1
SIGN(Ø) = Ø
SIGN(3.5) = 1
```

NOTE

ENTIER, ABS, and SIGN are not reserved words. They may be used for other purposes in a program.

Examples of simple arithmetic expressions follow:

```
X
I + 3
X*Y/Z
P+Q/R
X2 + Y
XJ-4
J + ENTIER(K-2)
SIGN(ENTIER(J/K) + 1)
(X + Y) ↑ (-I)
```

5.2 BOOLEAN EXPRESSIONS

Boolean expressions involve Boolean identifiers, Boolean and octal constants, arithmetic conditions, and Boolean operators interspersed in an order similar to that of arithmetic expressions.

5.2.1 Boolean Operators

There are five Boolean operators arranged in decreasing order of precedence.

a. NOT (unary operator)
b. AND
c. OR
d. IMP (implication)
e. EQV (equivalence)

NOT is a unary operator that complements a Boolean quantity in the same way that a unary minus sign negates an arithmetic quantity in an arithmetic expression. In this case, it changes FALSE to TRUE, and vice versa.

Table 5-2 gives the result of A OP B where OP stands for one of the Boolean operators AND, OR, IMP, or EQV, for all values of A and B.

Table 5-2
Function of Boolean Operators

A	FALSE		TRUE	
B	FALSE	TRUE	FALSE	TRUE
A AND B	FALSE	FALSE	FALSE	TRUE
A OR B	FALSE	TRUE	TRUE	TRUE
A IMP B	TRUE	TRUE	FALSE	TRUE
A EQV B	TRUE	FALSE	FALSE	TRUE

In addition, the following theorems hold true:

A IMP B is equivalent to NOT A OR B,
A EQV B is equivalent to A AND B OR NOT A AND NOT B.

Actually, Boolean variables may have a value consisting of any pattern of bits, rather than be confined to the values TRUE and FALSE. The logical operations operate on a bit-by-bit basis according to the preceding rules.

The actual test employed to determine the truth of a Boolean expression such as

B AND C

is to evaluate it and regard it as true if its value is nonzero, i.e., at least one bit is set, otherwise it is false.

This is particularly important when octal constants are used in Boolean expressions. For example, if the user wishes to test a particular bit in a Boolean variable, an appropriate octal constant can be used, for example:

B AND %1

is a Boolean expression that is true if and only if the bottom (least significant) bit of B is a one.

5.2.2 Arithmetic Conditions

Arithmetic conditions are used as operands in Boolean expressions. They consist of two arithmetic expressions coupled with a comparator. The comparator, which decides the particular type of test to be performed on the two expressions, is one of the following:

<	less than
<=	less than or equal to
=	equals
>	greater than
>=	greater than or equal to
#	not equal to

Such an arithmetic condition can be regarded as true or false according to whether the condition specified by the comparator is met when the arithmetic expressions on each side of it are evaluated. The resulting condition may form part of a Boolean expression.

The following examples of Boolean expressions, shown in Table 5-3, also involve arithmetic conditions.

5.3 INTEGER AND BOOLEAN CONVERSIONS

An integer quantity can be converted to a Boolean quantity by means of the dummy function BOOL. Similarly, the dummy function INT converts a Boolean quantity to an integer quantity.

Table 5-3
Boolean Expressions

Expression	Meaning
NOT B	NOT B
B AND NOT C	B AND (NOT C)
A OR B AND C	A OR (B AND C)
B EQV X<Y	B EQV (X<Y)
X+Y<Z AND B OR P=Q	(((X+Y)<Z) AND B) OR (P=Q)

The value passed by these functions is unchanged: the functions are included for semantic correctness. Thus:

BOOL(I)

may be regarded as a Boolean operand, and

INT(B)

INT(%400000000000)

as integer operands.

NOTE

This feature is not generally available in other ALGOL-60 implementations.

BOOL and INT are not reserved words. They can be used for other purposes by declaring them as required. However, this practice should be avoided since it could lead to confusion.

CHAPTER 6
STATEMENTS AND ASSIGNMENTS

6.1 STATEMENTS

The statement is the basic operational unit in ALGOL-60. It describes an operation to be performed at run time, such as an assignment.

6.2 ASSIGNMENTS

Assignments convey the value produced by the execution of an expression to a destination variable of the appropriate type. This is done by writing the destination identifier, followed first by the symbols : and = and then by the expression to be evaluated. Thus

```
X := Y + Z
```

causes the result of the addition of the values contained in the variables Y and Z to be placed in the variable X.

When an assignment is made to a variable type differing from that of the result of the expression, a type conversion is performed. Integer, real and long real expressions may be assigned to variables of any of these three types, but not to any other types. Boolean and string expressions can only be assigned to a variable of their own type.

If a real or long real value is assigned to an integer type variable, a rounding process occurs.

```
I := X
```

results in an integral value equal to

```
ENTIER(X + 0.5)
```

being assigned to I.

When an integer expression is assigned to a real or long real variable, a conversion to that type is performed. Real to long real conversion simply consists of zeroing the low-order precision word of

the long real result after assignment of the real result to the high-order part of the long real variable. Long real to real assignments truncate the low-order part of the long real expression, after appropriate rounding.

6.3 MULTIPLE ASSIGNMENTS

A value may be assigned simultaneously to several variables of the same type by a multiple assignment. This takes a form such as

```
P := R := S := X + Y - Z
```

where the result of adding Y to X and subtracting Z is assigned to P, R, and S simultaneously.

All identifiers on the left-hand side of a multiple assignment must be of the same type. If the user wishes to assign a value to two or more different types of variables, the "assignment within expression" (embedded assignment) feature must be used, as below.

A parenthesized assignment may be substituted for any operand in an expression. For example,

```
X := (Y := P+Q)/Z
```

This causes the embedded assignment to be made after the inner expression P+Q is evaluated. Where a type conversion is performed as part of an embedded assignment, the operand type is the same as that assigned to the variable in the embedded assignment. Thus

```
X := (I := 3.4)
```

sets I equal to 3 and X equal to 3.0.

6.4 EVALUATION OF EXPRESSIONS

All expressions in DECsystem-10 ALGOL are evaluated observing the normal algebraic rules of precedence, including bracketing.

Within the precedence structure, expressions are always evaluated from left to right. For example, if X is a scalar, and F a function procedure (see Chapter 11) that alters X,

```
X := X+F
```

may have a different effect than

```
X := F+X
```

This is known as a "side effect".

Consider also:

```
A[I] := (I := I+1)
```

The subscript I is always evaluated before I is incremented, as it is to the left of the embedded assignment, within the statement. Thus the above expression is equivalent to

```
J := I; I := I+1; A[J] := I
```

The user can always predict the order of evaluation of an expression and can count on such things as

```
X := (P := P+Q)/(P+R)
```

being evaluated correctly, thus giving the same result as

```
P := P+Q
X := P/(P+R);
```

6.5 COMPOUND STATEMENTS

A compound statement consists of a number of statements, preceded by BEGIN, separated by semicolons, and terminated by END. ALGOL statements, unlike those in FORTRAN, are terminated by a semicolon not by the end of a line of text.

For example:

```
BEGIN
        I := 3; J := 4;
        K := I + J;
        X := K
END
```

is a compound statement. Semicolons do not have to appear after the BEGIN or before the END; BEGIN and END act as a type of bracket.

The usefulness of compound statements will become apparent in later chapters.

CHAPTER 7
CONTROL TRANSFERS, LABELS, AND CONDITIONAL STATEMENTS

7.1 LABELS

A label is a method of marking a place in a program so that control can be transferred to that point from elsewhere in the program.

DECsystem-10 ALGOL uses identifiers as labels. These identifiers are placed before statements and are followed by a colon. Numeric labels are permitted in the Revised Report, but are not implemented in DECsystem-10 ALGOL. Most implementations of ALGOL-60 do not allow integer labels.

For example:

```
COMP: X := X + Y
```

is a statement labeled by COMP.

More than one label can be attached to a statement if required; thus,

```
LAB1: LAB2: Y := 0
```

7.2 UNCONDITIONAL CONTROL TRANSFERS

A transfer of control, or "jump", to a statement in a program is effected by a GOTO statement. This statement consists of the word GOTO followed by the name of the label attached to the relevant statement. The two words GO TO can be used instead of the word GOTO in any statement where GOTO can be used. Thus:

```
BEGIN    INTEGER I,J,K;
LAB:     I := J := 3;
         K := I + J;
         GOTO LAB
END
```

is an example of a somewhat tedious program. Clearly, to write any reasonable program, it is necessary to be able to jump conditionally.

7.3 CONDITIONAL STATEMENTS

Conditional statements provide a method to make the execution of either a statement or a compound statement dependent on some condition in the program, such as the value of a variable. The simplest form of a conditional statement is

```
IF B THEN S
```

where B is some Boolean expression, and S is a statement. For example:

```
IF X < 0 THEN I := I + 1
```

Here, X < 0 is the Boolean expression and I := I + 1 is the statement which is obeyed if and only if the Boolean condition is true, that is, if X is negative.

A more general form of a conditional statement is

```
IF B THEN S1 ELSE S2
```

In this case, the statement S1 is obeyed if and only if the Boolean expression B is true, and S2 is obeyed if an only if it is false. In order to eliminate the "dangling ELSE ambiguity" (a construction in which an ELSE could be paired with either of two THENs, S1 must not be a conditional, FOR, or WHILE statement which ends in an ELSE clause. (Refer to Chapter 14 for more complete information.)

A control transfer, a type of statement, can appear in a conditional statement. Thus:

```
BEGIN   INTEGER I;
        I := 0;
LAB:    I := I + 1;
        IF I < 100 THEN GOTO LAB
END
```

is a simple way of counting to one hundred. More sophisticated methods are shown in Chapter 14.

CHAPTER 8
FOR AND WHILE STATEMENTS

8.1 FOR STATEMENTS

The FOR statement enables the user to iterate a portion of the program in a fashion similar to, but more sophisticated than, FORTRAN's DO loop.

The general format is

```
FOR V := FORLIST DO S
```

where V is a variable and S is a statement (compound or otherwise).

FORLIST can consist of any number of FOR elements (separated by commas). A FOR element takes one of the following forms:

a. An expression:
   E
b. A STEP-UNTIL element taking the form:
   E1 STEP E2 UNTIL E3
c. A WHILE element taking the form:
   E WHILE B
   where B is some Boolean expression.

Any number of FOR elements may appear in a FOR statement; they are executed serially. Consider the following examples:

```
FOR I := 3,5,10 DO .....
FOR X := 2.5,5.0,10.0 DO .....
FOR J := 1,2,5 STEP 5 UNTIL 20 DO .....
```

8.1.1 STEP-UNTIL Element

This particular form deserves closer inspection, because it is not quite as simple as it appears. For example, consider

```
FOR I := 1 STEP I UNTIL N DO S
```

The statement S is obeyed with I taking an initial value of 1, and being incremented by I until the final value N is achieved. The question is, "Is the I after the STEP recalculated during each turn around the loop, or does it have a constant value equal to the initial value of I?"

The answer is slightly more complicated. Consider the general case

```
FOR V := E1 STEP E2 UNTIL E3 DO S
```

This is defined to have exactly the same effect as

```
        V := E1;
L1:     IF (V - E3)*SIGN(E2) > 0 THEN GOTO L2;
        S;
        V := V + E2;
        GOTO L1;
L2:
```

Clearly, the value of I following the STEP in the previous example is evaluated, if necessary, twice during each turn around the loop, once in the sign test at L1, and again to update V. ALGOL allows the user to modify V, E1, E2, and E3 freely throughout the loop, and takes account of all these changes in the evaluation of the loop.

NOTE

PDP-10 ALGOL allows the user the abbreviated form

```
FOR V := E1 UNTIL E3 DO S
```

instead of

```
FOR V := E1 STEP 1 UNTIL E3 DO S
```

8.1.2 WHILE Element

A FOR statement with a single WHILE element takes the form

```
FOR V := E WHILE B DO S
```

This is interpreted as follows:

```
L1:     IF NOT B THEN GOTO L2;
        V := E;
        S;
        GOTO L1;
L2:
```

Once again, the complexity of the loop may be affected by changing V and E within the loop.

8.2 WHILE STATEMENT

The WHILE statement is an enhancement of ALGOL-60 provided in DECsystem-10 ALGOL. It takes the general form

```
WHILE B DO S
```

and is interpreted as follows:

```
L1:     IF NOT B THEN GOTO L2;
        S;
        GOTO L1;
L2:
```

8.3 GENERAL NOTES

1. Within a FOR statement of any kind, the user can change the controlling variable or any other variable appearing within the action of the loop. Such changes predictably affect the execution of the loop by the rules given above.
2. On exit from a FOR statement either by jumping out of the loop or by exhausting the FOR elements, the controlling variable has a well-defined value equal to the last assigned value of the controlling variable. This may not be true of other ALGOL-60 implementations.

CHAPTER 9
ARRAYS

9.1 GENERAL

Arrays are essentially collections of variables of the same type, allowing the user to address them individually by means of a common name and a unique subscript or subscripts. In the simplest case, an array is a vector and is known as a one-dimensional array. A matrix is a two-dimensional array, etc.

There is no limit to the number of subscripts allowed, other than those imposed by the ability of the computer to store the array.

9.2 ARRAY DECLARATIONS

Arrays may be of type integer, real, long real, Boolean, or string. They are declared in a similar fashion to scalar variables, except the size of the array must be stated. For each subscript that the array possesses, a lower and an upper bound, called the "bound pair" for that subscript, must be given.

For example, to declare two one-dimensional integer arrays A and B with lower bound 1 and upper bound 5:

```
INTEGER ARRAY A,B[1:5]
```

Note that the lower and upper bounds are enclosed in square brackets and separated by a colon.

When there are two or more subscripts, the declaration is similar, and the bound pairs are separated by commas. Thus

```
LONG REAL ARRAY P,Q,R[-5:2,0:10]
```

declares three real arrays, P, Q and R, with the first subscript bounded by -5 and 2 and the second subscript bounded by 0 and 10.

It is possible to declare arrays of different sizes in the same statement provided they are of the same type:

```
REAL ARRAY A[1:10], B,C[1:10,1:12]
```

Note also that in the case of real arrays, the REAL may be omitted in the declaration, and is assumed by default, thus:

```
ARRAY A[1:10], B,C[1:10,1:12]
```

The bounds in an array need not be static, as in the examples above. In general, they may be any arithmetic expressions, which are evaluated to give an integral value for the individual bound pairs. The use of such dynamic array declarations will become apparent later.

9.3 ARRAY ELEMENTS

An individual element of an array can be referred to by following the name of the array by a list of subscripts in square brackets. The number of subscripts must be identical to the number in the array declaration. Thus, a typical element of A used in the last declaration might be

A[5] or A[9] or generally, A[I]

where I is some integer expression or, in general, any expression whatsoever, with the limitation that its value when used as a subscript and evaluated as an integer is in the range 1 through 10, the bounds of the array A.

As an example of the use of arrays, consider the declaration

```
REAL ARRAY D,E,F [1:10,1:10]
```

and suppose that it was required to set F equal to the matrix product of D and E:

```
FOR I := 1 UNTIL 10 DO
FOR J := 1 UNTIL 10 DO
BEGIN   X := 0;
        FOR K := 1 UNTIL 10 DO X := X + D[I,K]*E[K,J];
        F[I,J] := X
END
```

Note that X is used to accumulate the inner product of the multiplication for all values of I and J. It would be very inefficient not to use such a variable, because F would otherwise be needlessly involved in the inner loop of the computation.

Also, note that an element of an array of a particular type may be used anywhere that a scalar variable of the same type may be used, even in such places as the controlling variable in a FOR statement.

CHAPTER 10
BLOCK STRUCTURE

10.1 GENERAL

ALGOL program structure is somewhat more complicated than other high-level languages, such as FORTRAN. An ALGOL program consists of a number of "blocks" arranged hierarchically; a block consists of the words BEGIN and END enclosing declarations and (optionally) statements.

Thus:

```
BEGIN
      BEGIN
      END
      BEGIN
                  BEGIN
                  END
      END
END
```

is an ALGOL program, assuming appropriate declarations and statements in the blocks.

The block structure offers the user many interesting features not available in non-block structured languages. For instance, the user may declare an identifier that appears to conflict with another identifier in an enclosing block. Thus:

```
BEGIN      INTEGER I;
           BEGIN INTEGER I;
           END
END
```

In fact, there is no conflict as there are two different Is. The only I that statements in the outer block can "see" is the one in the outer block. Similarly, any statements in the inner block will always use the I in that block. Such a declaration in an inner block is known as a "local" variable; it takes precedence over declarations occurring at an outer or more "global" level. In general, all variables can be "seen" from any point in a program that is either in the same block as the declaration or in a block that is enclosed by the block in which the declaration of the variable occurred. Note that a more local variable is always taken in preference to a relatively global variable. Consider the following example:

```
BEGIN     INTEGER I,J;
          [1]
          BEGIN     INTEGER J,K
                    [2]
          END;
          BEGIN     INTEGER I,K
                    [3]
          END
END
```

Any statements occurring at point [1] can see the declarations of I and J, which are local, but cannot see the declarations of J and K in the first inner block, or the declarations of I and K in the second inner block. At [2], the local variables J and K can be seen, as can the global variable I in the outer block. The global variable J is not seen because the local variable J takes precedence over it; the variables I and K in the second inner block are not seen at all. A similar situation occurs at [3]; here both local variables I and K, as well as the global variable J, are seen.

Note that the "scope" of a variable is the set of all places in a program where it can be seen and therefore used. This term will be used frequently throughout this text.

In general, it is more efficient to use local variables in preference to global ones. This statement is also true of most ALGOL-60 implementations. Where a non-local variable is used frequently, it is advisable to assign its value to a local variable and use that in preference. For example:

```
BEGIN     INTEGER I;
          .....
          I := .....
          BEGIN     INTEGER II;
                    II := I;
                    .....
                    ..... II .....
                    END
                    .....
END
```

Here, in the inner block, a local variable II is used, and assigned the value of the global variable I for use throughout the local block.

10.2 ARRAYS WITH DYNAMIC BOUNDS

The concept of the scope of a variable can be applied most usefully to arrays. In DECsystem-10 ALGOL, all arrays are constructed at execution time, that is, no fixed space is reserved for them by the compiler, irrespective of whether their bounds are static or dynamic. When a declaration of an array is encountered within a block, the space required to construct it is obtained and the array is laid out. When the end of the block enclosing the array is reached, that is, the array variable is no longer within scope, the space utilized by the array is recovered and can be used later for other arrays.

Consider the case of a problem in which the size of an array to be used in a calculation is dependent on the data to be processed. The programmer has the choice of making the array large enough to cope with the worst case (in many languages he does not have any choice at all) or constructing the array with dynamic bounds to suit the size required by the particular data. The first method has the disadvantage of wasting space on many occasions; the latter method only has the minor disadvantage of the overhead needed to construct the array. Such overhead is very small compared to the running time of most programs; therefore, the second method is more desirable.

Consider the following example:

```
BEGIN     INTEGER N;
L:        N := .....
          .....
          BEGIN ARRAY A[1:N,1:N];
          .....
          END;
          GOTO L
END
```

A value for N is calculated in this example, possibly dependent on some data read into the program, and used to declare the array A, which is used to process the data in the inner block. When the end of the inner block is reached, the space used by A is recovered and control passes to L, where another value for N is calculated, and the process repeated.

CHAPTER 11
PROCEDURES

Procedures are similar in concept to the FORTRAN subroutine, although more sophisticated and general in their possible applications.

A "procedure" is a portion of an ALGOL program that is given a name to identify it and can be "called" from any part of a program which is in the scope of the body of the procedure. A procedure can execute a number of statements, or it can return a value if it is a function procedure. In addition, it may or may not have parameters.

In DECsystem-10 ALGOL, a procedure can be one of the following types: integer, real, long real, Boolean or string, or it may be typeless. The formal parameters of a procedure, known as "dummy variables" in FORTRAN, can be one of the following types: integer, real, long real, Boolean or string, as scalars, arrays or procedures, or label. There are seventeen different types of parameters. In addition, all of these parameters may appear in two different modes, neither of which is the same as FORTRAN's method of handling parameters.

11.1 PARAMETERS CALLED BY "VALUE"

Calling parameters by "value" is the most common and, with the exception of arrays, the most efficient way to pass a parameter to a procedure. The value of the expression presented in a procedure call, known as the actual parameter, is evaluated on entry to the procedure and assigned to a formal parameter within the procedure. This formal parameter acts exactly as if it were a local variable of the procedure which is initialized with the value of the actual parameter supplied in the call to the procedure.

Since, in the case of arrays or strings, a new copy of the array or string is made, this type of parameter-passing for arrays and strings (if they are very long) should be avoided unless it is specifically required.

11.2 PARAMETERS CALLED BY "NAME"

Calling parameters by "name" is a very sophisticated method of passing a parameter to an ALGOL procedure. Whenever the formal parameter associated with the actual parameter in a procedure body

appears in the body of the procedure, the actual parameter is re-evaluated as if it appeared in the procedure body at that point. For example, if the actual parameter were an array element such as

```
A[I]
```

it would be re-evaluated using the value of I available each time the formal parameter is used, not the value of I at the time the procedure body is entered.

Table 11-1 shows the different types of formal parameters, together with valid actual parameters that can be substituted in a procedure call.

Table 11-1
Parameters in a Procedure Call

Formal Parameter Type	Permissible Actual Parameter
Integer Real Long Real	Any arithmetic expression
Boolean	Any Boolean expression
String	Any string expression (refer to Chapter 13)
Label	A label or switch element (refer to Chapter 12 and Paragraph 14.4)
Switch	A switch
Integer Array	An array of type integer*
Real Array (or Array)	An array of type real*
Long Real Array	An array of type long real*
Boolean Array	An array of type Boolean
String Array	An array of type string
Procedure	A non-type procedure
Integer Procedure Real Procedure Long Real Procedure	A procedure of type integer, real, or long real
Boolean Procedure	A procedure of type Boolean
String Procedure	A procedure of type string

*In the case where the array parameter is called by value, any arithmetic type (integer, real, or long real) array is allowed as an actual parameter. A type conversion takes place during the copying process.

11.3 PROCEDURE HEADINGS

Procedure headings identify the type of procedure and the number and type of its parameters. They precede the body of the procedure.

A procedure heading consists of:

a. The type of procedure (omitted in the case of typeless procedures).

b. The word PROCEDURE followed by the name of the procedure.

c. A semicolon if the procedure has no parameters; otherwise

d. A list of the formal parameters, enclosed in parentheses, and followed by a semicolon.

e. Specifications of the formal parameters. Omitting formal parameter specifications, this looks like

```
LONG REAL PROCEDURE LR;

BOOLEAN PROCEDURE BOOLCON (I,J,K);

PROCEDURE CALC(THETA,X);
```

The formal parameter specification that follows consists of a list of descriptions of the formal parameters, appearing in any order, and a value specification if any of the parameters are to be called by value. (If this is omitted, the parameters, by default, will be called by name.) For example, the specification of the formal parameters for the second example above might be:

```
VALUE I,J; INTEGER I,J,K;
```

meaning that all three formal parameters are of type integer (scalars), and I and J are to be called by value, while K is to be called by name. A typical formal parameter specification for the third example might be:

```
REAL PROCEDURE THETA; ARRAY X;
```

NOTE

The value specifications, where they appear, must precede the formal parameter specifications.

11.4 PROCEDURE BODIES

The body of a procedure is that part that follows the procedure heading. It consists of a single statement, a compound statement, or a block. In the last-mentioned case, there may be declarations of local variables within the block, and also other blocks or procedures. Consider the following examples of realistic procedures:

a. A real procedure, squareroot, to calculate the square root of a real quantity. The first parameter is the argument; the second is a label that is used as an escape if the argument is found to be negative. The result of the procedure is the square root of the argument. Note how the result of the calculation is assigned to the procedure by placing the name of the procedure on the left-hand side of an assignment.

```
REAL PROCEDURE SQUAREROOT(X,L);
        VALUE X; REAL X; LABEL L;
BEGIN   REAL Y,Z;
        IF X < 0 THEN GOTO L;
        Y := (1 + X)/2;
IT:     Z := (X/Y + Y)/2;
        IF ABS(Z - Y) < 1&-6 THEN GOTO OK;
        Y := Z; GOTO IT;
OK:     SQUAREROOT := Z
END
```

The previous example uses the Newton-Rapheson method of finding the square root of a number: taking an initial approximation (1 + X)/2 and iterating until the difference between successive approximations is less than 1&-6. Although this is a very simple procedure, it is more enlightening with the aid of some commentary. The DECsystem-10 ALGOL alternative method of commentary (refer to Chapter 2) is used for brevity:

```
REAL PROCEDURE SQUAREROOT(X,L);
  VALUE X; REAL X; LABEL L;
BEGIN ! CALCULATES THE VALUE OF SQRT(X)
      USING THE NEWTON-RAPHESON METHOD.
      L IS USED FOR AN ESCAPE IF X < 0;
  REAL Y,Z;
  IF X < 0 THEN GOTO L;  ! EXIT IF X < 0;
  Y := (1+X)/2;          ! FIRST APPROXIMATION;
IT:
  Z := (X/Y + Y)/2;      ! ITERATE;
  IF ABS(Z-Y) < 1&-6
     THEN GOTO OK;       ! TEST FOR CONVERGENCE;
  Y := Z; GOTO IT;       ! OTHERWISE CONTINUE;
```

(continued on next page)

```
OK:

  SQUAREROOT := Z;        ! FINAL RESULT;

END
```

b. This function evaluates the sum of the values of any real procedure G over the integers 1 ..... N, where N is also a parameter of the procedure.

```
REAL PROCEDURE SUM(G,N);

        VALUE N; REAL PROCEDURE G; INTEGER N;

BEGIN   INTEGER I; REAL X;

        X := 0;

        FOR I := 1 UNTIL N DO X := X + G(N);

        SUM := X

END
```

Notice in this example how the formal parameter G is invoked so that the actual procedure that is substituted for G is called.

11.5 PROCEDURE CALLS

In the preceding example, the procedure G was "called". Since G is a function procedure, it is only necessary for its name to appear in an expression for the procedure to be entered with the actual parameters specified substituted for the formal parameters.

The procedure squareroot can be called in a similar way, for example:

```
P := SQUAREROOT(Z + 0.5)
```

causes the square root of Z + 0.5 to be calculated.

An example of the use of the procedure sum can be used to calculate the sums of the square roots of the first J integers, with the result squared, as follows:

```
X := SUM(SQUAREROOT,J)↑2;
```

Here is a further example of a procedure and its calls:

```
PROCEDURE MATRIXMULT(A,B,C,N);
 VALUE N; ARRAY A,B,C ; INTEGER N
BEGIN   INTEGER I,J,K; REAL X;
        COMMENT THIS PROCEDURE PERFORMS THE MATRIX
        MULTIPLICATION OF B AND C AND PUTS THE RESULT
        IN A.  THE ARRAYS ARE ASSUMED TO BE SQUARE
        AND OF BOUNDS 1:N,1:N;

        FOR I := 1 UNTIL N DO
        FOR J := 1 UNTIL N DO
        BEGIN X := 0;
        FOR K := 1 UNTIL N DO X := X +
                B[I,K]*C[K,J];
                A[I,J] := X
        END
END
```

A typical call for this procedure might be

```
MATRIXMULT(E,F,G,N);
```

or

```
MATRIXMULT(E,F,F,N);
```

Since the arrays are called by name, a call such as MATRIXMULT(E,E,F,N); would give rather interesting results.

This call could be made to work by calling B and C by value. However, this would increase the overhead of the procedure considerably.

11.6 ADVANCED USE OF PROCEDURES

11.6.1 Jensen's Device

This method of using a procedure exploits the power and flexibility of the call-by-name concept. Consider the following example:

```
REAL PROCEDURE SUM(I,N,X); VALUE N; INTEGER I,N; REAL X;
BEGIN   REAL Y;
        Y := 0;
        FOR I := 1 UNTIL N DO Y := Y + X;
        SUM := Y
END
```

On the surface, the procedure appears to calculate the value of N*X. However, consider the call

```
Z := SUM(J,10,A[J]);
```

and remember that J and A[J] are parameters called by name. Since I and consequently J take new values, each X in the loop is evaluated as a particular value of A[J], using the value of J just assigned. Hence the above call calculates

```
A[1] + A[2] + ..... + A[10].
```

Similarly, the call

```
Z := SUM(K,M,A[I,K]*B[K,J]);
```

calculates the (I,J)th inner product of A and B.

11.6.2 Recursion

ALGOL procedures have the inherent ability of recursion, that is, they may call themselves, directly or indirectly, to any reasonable depth. (The only restriction is the amount of core storage available to the object program.)

An often-quoted and very inefficient method of calculating the factorial function of a small positive integer N is:

```
INTEGER PROCEDURE FACTORIAL(N); VALUE N; INTEGER N;
IF N = 1 THEN FACTORIAL := 1
ELSE FACTORIAL := N*FACTORIAL(N-1);
```

Note that this procedure has only a single statement, but no local variables. Therefore, it can be written in a very compact form. A call such as

```
J := FACTORIAL(6);
```

causes the procedure to be entered with N equal to 6. The call to FACTORIAL inside FACTORIAL enters the procedure a second time with N equal to 5, but this N is different from the one to the previous N, which retains its value of 6, as it is stored in a different space. In this particular case, FACTORIAL is entered six times, the last time with N equal to 1.

11.7 LAYOUT OF DECLARATIONS WITHIN BLOCKS

Declarations must always be made at the head of a block, before any assignments, procedure calls, etc., in the following order: 1) scalars and arrays and 2) procedures and switches (see Chapter 12).

Any procedure bodies that occur in a block should follow the declarations at the head of the block, although this is only enforced when necessary. Consider the following example:

```
BEGIN
PROCEDURE P(X); VALUE X; REAL X;
BEGIN    INTEGER J;
         .....
         J := I;
         .....
END;
INTEGER I;
```

The assignment of I to J within the body of P utilizes the I that is declared following the body of P, rather than some global I. However, the compiler has not yet seen this I and, therefore, cannot take any rational action. In a case such as this, the user must declare I before the body of P:

```
BEGIN    INTEGER I;
PROCEDURE P(X); VALUE X; REAL X;
BEGIN    INTEGER J;
         .....
         J := I;
         .....
END;
```

If the user neglects to declare I before P, the compiler can easily detect the condition, because either I is unknown at the time of the assignment to J, or else there is a more global I available, whereupon an error message will occur when the declaration of I is found following the body of P.

11.8 FORWARD REFERENCES

Although most ALGOL-60 compilers operate in two or more passes, the DECsystem-10 ALGOL compiler operates in one pass. Consequently, it has to make some minor restrictions to ALGOL-60 in order not to restrict the user in other ways.

A forward reference for a procedure has to be given when a procedure is called (either directly, or indirectly, by passing its name as an actual parameter in a procedure call) before its body is encountered by the compiler. In most cases the user can avoid this situation by a minor re-ordering of the program. However, in rare cases like the following, where procedure P calls procedure Q, and vice versa, a forward reference, as shown, must be given.

```
BEGIN
FORWARD REAL PROCEDURE Q;
PROCEDURE P(X); VALUE X; REAL X;
BEGIN REAL Y;
          .....
          Y := Q(X);
          .....
END;
REAL PROCEDURE Q(Z); VALUE Z; REAL Z;
BEGIN REAL F;
          .....
          F := P(Z);
          .....
END;
.....
```

In general, a forward reference consists of the word FORWARD, followed by the type of the procedure (omitted if the procedure is typeless), the word PROCEDURE, and the name of the procedure. For example:

```
FORWARD LONG REAL PROCEDURE INTEGRATE
```

or

```
FORWARD PROCEDURE PROBLEM
```

Note that the forward reference must occur in the same block as the procedure body to which it refers.

A forward reference has to be given for a label in the following very rare case:

a. The label is used as an actual parameter in a procedure call, and has not yet appeared in the program.

b. A variable of identical name has appeared in the program and is in the scope of the procedure call.

For example:

```
BEGIN REAL L;
        .....
BEGIN FORWARD L;
        .....
        P(L);
        .....
L: .....
END;
        .....
```

In this case, a forward reference for L must be given.

11.9 EXTERNAL PROCEDURES

If it is required to compile a procedure independently of a program (see Paragraph 18.1.1), an EXTERNAL declaration must be made in the program instead of the procedure. The form of this is the same as that of a FORWARD declaration, but with the word FORWARD replaced by EXTERNAL. For example:

```
EXTERNAL INTEGER PROCEDURE CALC
```

Such an EXTERNAL declaration can be made in any block within the program, and has the same scope as if the procedure appeared at that point.

11.10 ADDITIONAL METHODS OF COMMENTARY

Two further ways of writing commentary are available to the user in addition to COMMENT and ! described in Section 2.4.

11.10.1 Comment After END

Following the delimiter word END, the user may add commentary, terminated by a semicolon, with the following restrictions:

1. The commentary may only contain letters and digits.
2. If the reserved delimiter word mode of compilation is employed, any words appearing in the comment may not be delimiter words.

For example:

```
END OF PROC INVERT;
```

11.10.2 Comments Within Procedure Headings

This method of commentary allows the user to comment formal parameters in a procedure heading. This is done by enclosing the commentary, which may consist of letters only, between the symbols) and :(and omitting the comma on the left of the formal parameter. This cannot apply to the first formal parameter.

The example in Section 11.6.1 can thus be rewritten:

```
REAL PROCEDURE SUM(I) COUNT:(N) INCREMENT:(X);
```

In a similar fashion, a call to such a procedure can be commented. The following example uses the call to SUM in Section 11.6.1:

```
Z:=SUM(K) COUNTER:(M) CROSS PRODUCT: (A[I,K]*B[K,J]);
```

CHAPTER 12
SWITCHES

12.1 GENERAL

Switches enable the user to jump to one of a number of labels, depending on the value of an arithmetic expression. In addition, they provide an automatic detection when such an expression is out of range for the switch.

12.2 SWITCH DECLARATIONS

A switch declaration takes the form of the word SWITCH followed by (a) the name of the switch, (b) an assignment (:=), and (c) a list of labels, called switch elements, all of which must be in the scope of the switch declaration. For example:

```
SWITCH SW := LAB,L1,L2,OK,STOP
```

A switch name must follow the usual rules of scope with regard to its use and, therefore, must not conflict with any local variable of the same name.

In addition to the example above, a switch element itself may be one of the labels in the switch declaration.

12.3 USE OF SWITCHES

A jump to a particular label in a switch declaration is made by following the word GOTO with the name of the switch and an arithmetic expression in square brackets. Thus:

```
GOTO SW[I]
```

This causes control to pass to the I'th label in the switch declaration, unless I is negative or zero, or is larger than the number of switches in the switch declaration. In either case, there is no transfer of control. If the expression in square brackets is not integral, it is evaluated and rounded as usual.

Consider the following more complicated example:

```
SWITCH SW := LAB,L1,L2,OK,STOP;
SWITCH TW := L3,SW[J],L4;
.....
GOTO TW[I];
```

If I has the value 3, a jump to L4 occurs. If I has the value 2 and J has the value 1, a jump to LAB occurs, via SW.

More sophisticated switch elements are described in Chapter 14.

CHAPTER 13
STRINGS

13.1 GENERAL

In DECsystem-10 ALGOL, the concept of a string has been considerably extended from the somewhat limited feature of ALGOL-60.

A string is a type of variable that may be scalar, array, or procedure. For example:

```
STRING S,T;
STRING ARRAY SA[1:10];
STRING PROCEDURE B(X); VALUE X; REAL X;
```

13.2 STRING EXPRESSIONS AND ASSIGNMENTS

String expressions are limited to a single string variable, a string procedure call, or string constant; there are no string operators other than the comparison operators described in Paragraph 13.5. Such a string expression can only be assigned to another string variable. For example:

```
S := T;
SA[I] := SA[3];
SA[2] := B(Z);
T := "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
```

13.3 BYTE STRINGS

The function of a string variable is to "possess" (or point to) a byte string. Byte strings are merely strings of bytes of some particular byte size, between one and thirty-six bits. Byte strings can be handled very efficiently by DECsystem-10 hardware. They form a flexible storage medium for strings of bits, characters, or any useful quantity.

String constants are a particular example of byte strings. They have a byte size of seven and consist of the ASCII characters of the string constant packed end-to-end.

Byte strings can be of any reasonable length; in fact, the permissible length is sufficient to allow a string of one bit bytes to stretch throughout the entire DECsystem-10 core storage. When a string variable possesses a byte string, the length of the byte string, and the size of the bytes in it, are stored in the string variable.

When one string is assigned to another, for example:

```
S := T;
```

where S and T are both string variables, S also possesses the byte string that T possessed prior to the assignment. Note that possession of a byte string is not a monopoly: several string variables can possess the same byte string and operate on it independently. It is important to remember that the assignment of one string variable to another does not involve making a copy of the byte string that the first string variables possesses.

When a string constant is assigned to a string variable, for example,

```
S := "ABCD";
```

the effect is as if an anonymous string variable had already possessed the byte string and assignment of this anonymous byte string were made to S.

13.4 BYTE SUBSCRIPTING

String variables would not be very useful if it were not possible to access the individual bytes of a byte string possessed by a string variable. This is done by means of "byte subscripting" the string variable. A byte subscript consists of a decimal point, followed by a subscript in square brackets, for example:

```
S.[I]
```

This notation means the I'th byte in the byte string that is possessed by the string variable S. I may, of course, be any expression, and is evaluated in exactly the same way as an array subscript.

A byte-subscripted string variable may appear on the left-hand or right-hand side of an assignment. When it is on the right-hand side, or generally appears as an operand in an arithmetic expression, it yields an integral value equal to the value of the particular byte in the byte string. For example,

```
J := S.[I]
```

sets J equal to the value of the I'th byte in the byte string possessed by the string variable S.

When a byte-subscripted string variable appears on the left-hand side of an assignment, it causes the value of the expression on the right-hand side of the assignment (rounded to an integer if necessary, and truncated if it is too large for the particular byte size) to be stored as the new value of the particular byte addressed. For example,

```
S.[K] := J
```

causes the K'th byte in the byte string possessed by the string variable S to be set to the value of J. When a string variable is a particular element of a string array, byte subscripting follows the usual array subscripts. Thus, assuming the declarations at the start of this chapter, the user can write such things as

```
SA[J].[I+1] := S.[K-1] + 1
```

Note that string constants but not string functions may be byte subscripted.

13.5 STRING COMPARISONS

Two byte strings can be compared with each other using the usual comparison operators. Thus the user can write

```
IF S < T THEN GOTO L
```

where S and T are string variables, string constants, or calls to a string procedure. The comparison is performed by comparing the byte strings that the string variables possess, byte by byte; the "lesser" string being the one with the first lower value byte, working from left to right. Thus "ABCD" is less than "ABCE", and "ABCD" is less than "ABCDE".

13.6 LIBRARY PROCEDURES

Refer to Chapter 17 for a detailed description of the DECsystem-10 ALGOL Library.

13.6.1 Concatenation

Strings can be concatenated to form chains, rings, or trees of string variables by forging a link between one string variable and another. This process is independent of any byte string possessed by the string variables involved.

Whenever two strings are linked together, the byte subscripting of the first extends to the second. A link between two unattached strings can be made by a call to the procedure LINK or LINKR (join to the right). Thus, if S and T are strings,

```
LINK(S,T);
```

forges a link from S to T. If the assignments

```
S := "ABCD"; T := "EFGH";
```

are also made, then S.[5] is now the same as T.[1].

A further string, for example, U, can be added to the chain by the user of LINKR, thus

```
LINKR(S,U);
```

or

```
LINKR(T,U);
```

and also

```
LINK(T,U);
```

The difference between LINK and LINKR is that LINK joins the second string to the first replacing any existing link, LINKR attaches the second string to the end of the existing chain.

While only one link can be forged between any pair of strings, more than one string can be linked to another. Thus

```
LINK(S,U); LINK (T,U);
```

causes S and T to be linked independently to U.

Some simple examples of the kind of structures that can be generated are:

a. A ring:

```
LINK(S,T);
LINK(T,U);
LINK(U,S);
```

b. A figure six:

```
LINK(S,T);
LINK(T,U);
LINK(U,T);
```

c. A circle with two stems:

```
LINK(S,T);
LINK(T,U);
LINK(U,T);
LINK(V,U);
```

A link may be removed from a string by omitting the second parameter in LINK or LINKR. Thus

```
LINK(S)
```

removes the link between S and T.

The string procedure TAIL enables the user to move along a structure of strings. Its first parameter is a string that is taken as the head of the structure. The second parameter is integral and specifies the number of links to be skipped in the chain. Thus in example b. above,

```
V := TAIL(S,1);
```

sets V to be the same as T, and

```
W := TAIL (S,2)
```

sets W to be the same as U.

If the second parameter is zero, or greater than or equal to the number of non-repetitive links in the structure, the result is the string at the bottom of the chain; in this case U, as it links to T, which has already been encountered while searching down the chain S - T - U.

The length of any byte string (excluding any possessed by concatenated strings) is yielded by the integer procedure LENGTH, that takes a string as its only parameter.

Thus:

```
I := LENGTH(S);
```

sets I equal to the number of bytes in the byte string possessed by S.

13.6.2 Byte String Copying

A new byte string can be generated from an existing byte string by means of the string procedure COPY.

COPY may have one, two, or three parameters.

a. If there is only one parameter, for example,

```
STRING S,T;
T := "ABCD";
S := COPY(T);
```

a new byte string is generated, identical to that possessed by T, and assigned to the string variable S. If any strings are concatenated with T, the byte strings possessed by these string variables are also copied into the new byte string.

b. If there are two parameters, for example,

```
S := COPY(T,M);
```

where M is some arithmetic expression, the 1st through Mth bytes of the byte string possessed by T are copied.

c. If there are three parameters, for example,

```
S := COPY(T,M,N);
```

the Mth through Nth bytes of the byte string possessed by T are copied.

13.6.3 New Byte Strings

A new byte string can be generated by means of the string procedure NEWSTRING. This procedure has two parameters: the number of bytes required in the new string and the byte size required. Thus

```
S := NEWSTRING (100,7);
```

causes a byte string consisting of 100 7-bit bytes to be generated and possessed by S. All of the bytes in the byte string are preset to a value of zero.

A dynamically-created byte string (i.e., one produced by the COPY or NEWSTRING procedure) can be deleted and the space utilized by it retrieved. This is accomplished by means of the procedure DELETE, which takes as its single parameter the string which possesses the byte string. For example:

```
DELETE (S);
```

causes the byte string in the previous example to be deleted.

CHAPTER 14
CONDITIONAL EXPRESSIONS AND STATEMENTS

14.1 GENERAL

ALGOL-60 allows great flexibility in the construction of expressions and conditions.

Consider, for example, if a user wanted to set a variable I equal to 0 or 1 according to the value of a Boolean variable B, he could write:

```
I := 0;
IF B THEN I := 1;
```

Also, consider the case where a user wants to perform some action, depending on the value of B:

```
IF B THEN X1 := Y; IF NOT B THEN X2 := Y;
```

14.2 CONDITIONAL OPERANDS

ALGOL-60 allows the user to substitute a conditional operand for any operand in an expression by the use of a construction involving IF ..... THEN ..... ELSE.

For instance, the first example above can be rewritten

```
I := IF B THEN 0 ELSE 1;
```

Clearly, this is more compact and of great use in cases such as:

```
J := J + (IF K < 1 THEN 1-K ELSE K-1);
```

Note that the conditional operand must be bracketed. It may be unbracketed only when it forms the complete expression itself.

In general, a conditional operand may replace an operand in any arithmetic or Boolean expression. It may also be used in place of a label as the element in a switch list, for example:

```
SWITCH SW := L1, IF B THEN L2 ELSE L3, L4;
```

It is also permitted, of course, in an array subscript (and also in a byte subscript), for example:

```
X := A[I, IF L = 0 THEN J ELSE J+1];
```

Since a conditional operand may replace any operand in an expression, it may also replace operands in conditional expressions. Consider the following example:

```
IF IF B THEN B1 ELSE B2 THEN I := I + 1;
```

This looks complicated but is really quite simple if brackets are inserted for clarity. Thus:

```
IF (IF B THEN B1 ELSE B2) THEN I := I + 1;
```

14.3 CONDITIONAL STATEMENTS

The reader was introduced to conditional statements of the form

```
IF B THEN S1 ELSE S2
```

in Chapter 7. The full power of this type of statement can now be demonstrated.

First, S1 and S2 can be compound statements or blocks. For example:

```
IF I < 0 THEN
BEGIN          I := -I; B := FALSE
END ELSE
BEGIN          I := I + 1; GOTO L2
END
```

Second, the whole structure of the IF ..... THEN ..... ELSE statement can be made more powerful by using conditional statements within themselves. For example:

```
IF X < 0 THEN X := 0 ELSE IF B THEN GOTO L;
```

This is equivalent to the simple sequence of statements:

```
        IF NOT X < 0 THEN GOTO L1;
        X := 0; GOTO L2;
L1:     IF NOT B THEN GOTO L2;
        GOTO L;
L2:
```

Clearly the former method of expression is both briefer and more elegant.

Conditional statements take the general form

```
IF B THEN S1 ELSE S2
```

where S1 and S2 may themselves be conditional statements with the provision that if there is ambiguity, bracketing using BEGIN and END must be used to remove it. Consider the following illegal example:

```
IF B THEN IF X = 0 THEN Y := Z ELSE P := Q;
```

This could be interpreted as

```
IF B THEN BEGIN IF X = 0 THEN Y := Z END ELSE P := Q;
```

or

```
IF B THEN BEGIN IF X = 0 THEN Y := Z ELSE P := Q END;
```

The first case is interpreted as:

```
IF NOT B THEN GOTO L1;
IF NOT X = 0 THEN GOTO L2;
Y := Z; GOTO L2;
L1: P := Q;
L2:
```

The second case is interpreted as:

```
        IF NOT B THEN GOTO L2;
        IF NOT X = 0 THEN GOTO L1;
        Y := Z; GOTO L2;
L1:     P := Q;
L2:
```

ALGOL-60 forbids such ambiguities by forbidding the sequence THEN IF ..... THEN ..... ELSE.

14.4 DESIGNATIONAL EXPRESSIONS

A designational expression is something that acts as an argument in a GOTO statement, either directly or indirectly, via a formal procedure parameter of type label. It may simply be a label or a switch element. Thus the following are designational expressions:

```
L
IF B THEN L1 ELSE L2
IF X < 0 THEN SW[I] ELSE IF X+Y >= Z THEN TW[J] ELSE L
```

These designational expressions would be used in the following manner:

```
GOTO L;
GOTO IF B THEN L1 ELSE L2;
GOTO IF X < 0 THEN SW[I] ELSE IF X+Y >= Z THEN TW[J] ELSE L;
```

CHAPTER 15
OWN VARIABLES

15.1 GENERAL

Own variables are a special kind of ALGOL variable, and may be of type integer, real, long real, Boolean or string, either scalar or array. They have the following properties:

a. Although they follow the normal scope rules, they are not recursive, the same copy of each variable being used in all occurrences of a procedure or block.

b. The values they contain when control passes out of a block are retained and are still available when the block is re-entered.

c. They are initialized to zero before execution of the program. (FALSE in the case of Boolean own variables.) OWN STRINGS are initialized to possess no byte string.

Own variables are declared by writing the usual declaration with the word OWN preceding it. For example:

```
OWN INTEGER I,J,K;

OWN REAL ARRAY THETA[1:M]
```

15.2 OWN ARRAYS

Own arrays are implemented in a completely dynamic fashion in DECsystem-10 ALGOL. The declaration proceeds according to the following rules.

a. If this is the first time the array is declared, space is obtained and then the array laid out. If the array has been laid out before, proceed to Step b.

b. The bounds are examined to see if they are identical to those of the previous construction of this array. If they are the same, the array is left unaltered; otherwise, proceed to Step c.

c. A new array is constructed and those elements that it has in common, if any, with the old array are copied into it; the remaining elements are zeroed. The old array is then deleted and the space used by it is recovered for future use.

For example, if an own array A is declared as follows:

```
OWN REAL ARRAY AL[1:M,M:N];
```

where M = 2 and N = 5 the first time, and M = 1 and N = 4 the second time, the elements [1,2], [1,3] and [1,4] are copied over, and the remaining elements of the new array are zeroed.

CHAPTER 16
DATA TRANSMISSION

16.1 GENERAL

Data transmission encompasses the input and output of data between the user's program and peripheral devices, such as disk, DECtape, magnetic tape, card reader, card punch, and line printer. The DECsystem-10 ALGOL object-time system, in conjunction with the ALGOL library, provides the user with a set of basic procedures for handling data from most DECsystem-10 devices in a uniform fashion. The user may also perform input/output operations with virtual peripherals that manifest themselves as byte strings in the user's program.

All peripheral devices that the user requires are under his control completely and can be allocated or released at any time throughout the execution of the program. The user can handle up to sixteen devices simultaneously (seventeen, if one of them is the terminal attached to his job), any number of which may be file devices (disk, DECtape) and have an independent file open.

16.2 ALLOCATION OF PERIPHERAL DEVICES

Peripheral devices are allocated to the user's program by calls to the library procedures INPUT or OUTPUT. A call to one of these procedures usually has two parameters. The first is the channel number, an integer in the range 0 to 15, on which the device is to operate. Only one device at a time may be operated on a channel; a channel provides either input or output facilities, except in the case of a terminal, where the input and output functions are performed simultaneously on the same channel. The second parameter is either a string or a string constant. The text contained in the string is the logical name of the device to be allocated to this channel.

The DECsystem-10 Users Handbook should be consulted for an explanation of what constitutes a logical device name. In the simplest case, it may be the actual name of the peripheral device. The device names shown in Table 16-1 are recognized as standard.

Table 16-1
Standard Device Names

Device Name	Peripheral
DSK	Disk
DTA	DECtape
MTA	Magnetic tape
CDR	Card reader
CDP	Card punch
LPT	Line printer
PTR	Paper-tape reader
PTP	Paper-tape punch
PLT	Plotter
TTY	Terminal

For example, to allocate the card reader for use as an input device on channel 5, the user would use the statement

```
INPUT(5,"CDR");
```

or, if S were a string possessing a byte string that had the characters CDR in it,

```
INPUT(5,S);
```

Similarly, if the disk were to be used as an output device on channel 9:

```
OUTPUT(9,"DSK");
```

Note that with the exception of terminals, all devices are allocated to operate in one direction only; thus, if the user wants input and output from the disk, he must use two separate channels.

Terminals are always allocated bidirectionally, irrespective of whether the user uses INPUT or OUTPUT. For example,

```
INPUT(0,"TTY");
```

allocates the user's terminal for input and output on channel 0.

16.2.1 Device Modes

Normally, a device is allocated in ASCII mode, that is, when the user reads a character from the device it is a 7-bit byte representing readable text, such as a stored source program or data. To allocate the device in a different mode, a third parameter is specified in the call to the INPUT or OUTPUT procedure. Thus, to allocate a disk to channel 9 in image binary mode (the mode used for the storage of binary data on a disk), the user can use

```
OUTPUT(9,"DSK",11);
```

The DECsystem-10 Assembly Language Handbook should be consulted for a full explanation of the different modes used with peripheral devices. The INPUT and OUTPUT procedures allow the user to allocate any standard peripheral device in any buffered mode.

16.2.2 Buffering

The INPUT and OUTPUT procedures normally allocate two buffers for each allocated device; terminals are allocated two buffers for input and two for output. The user may desire to use either one or more than one buffer for a device. For example, in a non-compute bound job that uses a lot of disk transfers at odd intervals, four or even eight buffers may be desirable to increase the speed of execution of the program.

The number of buffers to be used can be controlled by adding a fourth parameter to the procedure call. Thus, to allocate a disk on channel 14 in mode 0 with eight buffers, the call is

```
OUTPUT(14,"DSK",0,8);
```

Note that the mode must always be specified in this case, otherwise there would be an ambiguity in the third parameter.

16.3 SELECTING INPUT/OUTPUT CHANNELS

Before a user uses a device to transfer data, assuming that the device has already been allocated to some channel, the appropriate input or output channel must be "selected" for use as the input or output channel. All data input and output always occurs on the currently selected input channel and output channel, respectively. The user may change the selection of channels at any time, switching from one channel to another without loss of data, irrespective of whether complete lines (or records) of data have been read or not. In fact, the DECsystem-10 input/output system does not assume any structure in the data: all input and output channels are regarded as pipelines through which the user pulls or pushes data.

To select an input channel, a call to the procedure SELECTINPUT must be made. This has one parameter, which is the channel number. Thus

```
SELECTINPUT(5);
```

causes input channel 5 to be selected.

Similarly, the procedure SELECTOUTPUT is used to select an output channel.

16.4 FILE DEVICES

Some peripheral devices, such as disk and DECtape, require the opening of a specifically named file before any input or output operations can be performed.

The opening of this file is performed by means of the procedure OPENFILE, which is called after the device has been allocated to a channel. The procedure call has two parameters: the channel number on which the device has been allocated and a string variable possessing a byte string or a string constant, the text of which is the name of the file.

The user can also specify a protection and/or project-programmer number of a file by means of optional third and fourth Boolean or integer parameters. For example, to open a file with protection <177> on disk area [11,50] the user could write

```
OPENFILE (9,"TEST.DAT",%177,%000011000050);
```

When a user has finished with a file, it should be closed. A file is closed by using the procedure CLOSEFILE, with a parameter that is the channel number on which the file is open. Thus,

```
CLOSEFILE(9);
```

closes the file that is open on channel 9.

The user may also rename or delete existing files: if a file is already open, use of OPENFILE causes the file to be renamed with the new name supplied. Thus the sequence

```
OPENFILE (5,"TEST1.DAT");
OPENFILE (5,"TEST2.DAT");
```

causes the file with name TEST1.DAT to be renamed TEST2.DAT.

If the string containing the new name is null, the original file is deleted. Thus,

```
OPENFILE (5,"TEST3.DAT");
OPENFILE (5,"");
```

causes the file TEST3.DAT to be deleted.

16.5 RELEASING DEVICES

The procedure RELEASE is used to release a device from a channel. Thus,

```
RELEASE(5);
```

releases the device allocated to channel 5. If the device is a file device, and a file is still open on the device, it is automatically closed. Releasing a device on a channel causes a channel to become free; if this channel is currently selected for input or output operations, it is deselected.

If an attempt is made to allocate a device to a channel that already has a device allocated, the allocated device is first released and, if a file is open on it, it is closed before releasing the device.

If a user terminates his program without releasing devices on channels, they are automatically released.

16.6 BASIC INPUT/OUTPUT PROCEDURES

16.6.1 Byte Processing Procedures

The following procedures may be used with any device to handle bytes of any standard size (1 to 36 bits). However, because they are normally used with devices supplying or receiving ASCII bytes, they are "symbol" oriented.

a. INSYMBOL(S); - (where S is usually some integer variable) causes the next byte to be read from the currently selected input channel and stored in S.

b. OUTSYMBOL(J); - (where J is usually some integer expression) causes the value of J to be output as a byte to the currently selected output channel. If J is too large for the byte size of the device in use, it is truncated to size.

c. NEXTSYMBOL(S); - acts in exactly the same way as INSYMBOL except that the byte pointer for the input channel is not advanced to the next available byte. This gives the user a look-ahead facility of one byte.

d. SKIPSYMBOL; - causes the next byte from the selected input channel to be read and ignored.

e. BREAKOUTPUT; - causes all bytes in the buffer of a device to be sent immediately to it. This procedure is normally used to conduct a question-and-answer dialogue on a terminal, with the question and answer on the same line. Normally, a block of data is sent to a device only when the buffer is full (the exception being the terminal, where a break is sent at the end of each line).

16.6.2 String Output

A byte string may have its contents transferred to the currently selected output channel by means of the procedure WRITE, whose single parameter is either a string constant or a string variable that possesses the string to be output. For example:

```
WRITE(S);
```

or

```
WRITE("THE MOON IS MADE OF GREEN CHEESE");
```

With exceptions explained in the following paragraphs, all of the bytes in the string are output literally, with the exception, of course, of the quotes in a string constant, which are not in fact stored in the byte string at all. The important thing to remember is that, unlike some other ALGOL implementations, spaces and other non-printing symbols in byte strings are meaningful.

Special editing characters are permitted within square brackets within the text of a byte string. These have a special function:

P	Page throw
C or N	New line (C stands for carriage return, line feed)
T	Tab
S	Space
B	Break output

Any combination of these characters, with optional repetition counts preceding them, can appear within square brackets in a byte string and are output as their special interpretation demands. For example:

```
WRITE("ABCD[P2C5S]EFGH");
```

causes the following to be output:

a. the symbols ABCD followed by a page throw
b. two new lines and five spaces
c. the symbols EFGH.

In order to output the symbols

```
[ ] " or ;
```

they must appear in the form

```
[[ ]] "" or ;;
```

respectively. Thus

```
WRITE("""A[[I]] := 3;;""");
```

causes the text

```
"A[I] := 3;"
```

to be output.

16.6.3 Miscellaneous Symbol Procedures

The procedures SPACE, TAB, PAGE, and NEWLINE cause the appropriate number of spaces, tabs, page throws, or new lines to be output, depending on their single parameter, which is an integer expression. If the parameter is omitted a value of one is assumed. Thus

```
SPACE(5);
```

causes five spaces to be output, whereas

```
SPACE;
```

or

```
SPACE(1);
```

causes one space to be output.

16.6.4 Numeric and String Procedures

Numeric procedures are used to read and print numeric quantities. They will normally be used with a device that is operating in ASCII mode. They are capable of processing integer, real, or long real quantities in fixed-point and floating-point representations.

16.6.4.1 Numeric Input Data - Numeric data for input can be represented in any format that would be acceptable as a numeric constant in a program, irrespective of the type of variable involved. When a number is read, an automatic type conversion is performed, giving a result of the same type as if an assignment of the data represented as a constant in the program had been executed.

There is a minor restriction in that no spaces, tabs, or other non-printing symbols may appear in such numeric data except between the exponent sign (& or @ for real, && or @@ for long real) and the exponent. Otherwise, any symbol that is not itself a part of a numeric quantity may act as a terminator for such a quantity. It is strongly recommended that spaces, tabs, or new lines be used as separators. For example:

```
3.4 -9.6 1.36 -52
0 14.9
```

Note that in reading a numeric quantity, the terminating symbol, that is, the first symbol that is not part of the number, is lost.

DECsystem-10 ALGOL also allows the user to input floating-point data written in FORTRAN format, that is, using E for & or @, and D for && or @@. Note, however, that no other special effects inherent in FORTRAN formatting are introduced.

The procedure READ is used to input numeric data and also strings. This procedure may have any number of parameters, of type integer, real, long real, Boolean, or string.

The effect is as follows:

a. For integer, real and long real variables, a number is read and converted to the type appropriate to the parameter and then assigned to the variable.

b. For Boolean, a number is read as if for an integer variable, and assigned to the variable.

c. For a string variable, the data text is scanned until a quote (") is found, and the text following this up to but not including the next free quote is read in and a byte string generated, which is then possessed by the string variable.

If the sequence "" is found, a single " is stored, and reading of the string continues.

16.6.4.2 Numeric Output Data - Numeric data is output by means of the procedure PRINT. This procedure may have one, two, or three parameters, the first of which is the variable to be printed. This variable may be an integer, real, long real, or Boolean. The second and third parameters determine the format to be used and are integer expressions. If they are omitted, they are assumed to be zero. The effect of the various combinations of the format integers, M and N, is as follows:

M > 0, N > 0:	Fixed-point printing, M places before the decimal point, N places after. A sign, space if positive, - if negative appears before the number. Zeros before the decimal point are replaced by spaces and the sign moved up to the number. This format always outputs M+N+2 symbols.
M > 0, N = 0:	The same as the preceding except that (a) no fractional part appears, and (b) the decimal point is suppressed. This format always outputs M+1 symbols.
M = 0, N > 0:	Floating-point format, consisting of a sign, a decimal digit, a decimal point, N more decimal digits, and an exponent consisting of & for real, && for long real followed by the exponent sign and a two-digit exponent, zero suppressed from the left. This format outputs N+7 symbols for real and N+8 symbols for long real quantities.

If only two parameters appear, format M,0 is assumed for integer variables, and format 0,N for real and long real quantities, where M and N are, respectively, the values of the second parameter.

If only one parameter appears, the format is interpreted as 0,0 which assumes standard printing modes of 11,0 for integer quantities, 0,9 for real quantities, and 0,17 for long real quantities.

If the user requests more digits to be printed than are significant in real or long real numbers, the appropriate number of zeros follow a properly rounded printing of the number to the maximum precision available.

16.6.4.3 Octal Input/Output - The procedures READOCTAL and PRINTOCTAL, respectively, allow the user to input and output quantities in octal format.

On input, for single precision variables, up to 12 octal digits are read, preceded by the symbol %, the terminator being any non-numeric symbol. For long real variables, two such octal numbers must be presented for input, each preceded by the symbol %.

On output, 12 octal digits, preceded by the symbol %, are printed for single precision variables. For long real variables, two quantities each with 12 octal digits are printed, with a space separating them.

The foregoing procedures have one scalar parameter which may be of type integer, real, long real or Boolean.

16.7 DEFAULT INPUT/OUTPUT

If the user does not select any input or output channels, input and output occur via an "invisible" channel from and to the user's terminal. Thus, for simple programs where the user wishes to input a few numbers and print a few results, he simply uses READ, types in the data on line through his terminal, and gets back the results from PRINT.

16.8 LOGICAL INPUT/OUTPUT

In addition to the 16 channels used to communicate with peripheral devices, an additional 16 channels, numbered from 16 to 31, are provided. These are input or output channels that use byte strings as a means of storage.

By means of the procedures INPUT or OUTPUT, the user can attach a channel to a byte string possessed by a string variable, and can read and write bytes from and to this byte string, either to or from a peripheral device, or to and from another byte string.

```
INPUT(20,S);
```

or

```
OUTPUT(20,S);
```

cause the byte string possessed by the string variable S to be used as logical channel 20; this channel may subsequently be selected for input or output, as appropriate.

The user is still free, of course, to manipulate the individual bytes within the byte string by means of the byte-subscripting facilities available. Such facilities enable the user to read a file from a peripheral device into a string, process it in any way whatsoever, and output it again.

16.9 SPECIAL OPERATIONS

These procedures are used on channels assigned to magnetic tapes. They consist of the procedures BACKSPACE, ENDFILE and REWIND, each having one parameter, i.e., the channel number on which the operation is to be performed.

Since there is no implicit structure on a magnetic tape, these procedures enable the user to build up formats in any way he chooses.

16.10 I/O CHANNEL STATUS

The status of any input or output channel can be determined at any time by means of the Boolean procedure IOCHAN, which takes as its single parameter an integer quantity which is the channel number. The status returned is bit coded as follows:

Bit	Value	Meaning if Set
18	%400000	Device is physical (i.e., not logical)
19	%200000	Directory device
20	%100000	Terminal device
21	%040000	ASCII mode
22	%020000	Magnetic tape
23	%010000	Plotter
24	%004000	Set for default TTY on channel -1
25	%002000	Spare
26	%001000	Device can do input
27	%000400	Device is initialized for input
28	%000200	File is open for input
29	%000100	End of file encountered
30	%000040	Input ok status
31	%000020	Device can do output
32	%000010	Device is initialized for output
33	%000004	File is open for output
34	%000002	Device quota (exceeded)
35	%000001	Output status ok

Some of these bits are of little use to the user, but, for example, if a device is allocated, and the user does not know whether or not it is a file device, he can use IOCHAN to determine this. The bits of particular use to the user are the input and output end-of-file (note that end-of-file on output is a logical status indicating that, for example, a disk quota is exceeded or a DECtape is full, or in the case of a logical device, the byte string is full).

When IOCHAN is used, the end-of-file flags are always cleared, if set, so that the user may proceed to read a magnetic tape after an end-of-file marker is found.

The following example shows how the user would handle an unknown device whose name is given to the program via the user's terminal:

```
BEGIN
    STRING DEVICE, FILE; INTEGER CHANNEL;
    WRITE ("CHANNEL NO: "); BREAK.OUTPUT;
    READ (CHANNEL);
    WRITE ("[C]DEVICE NAME: "); BREAK.OUTPUT;
    READ (DEVICE);
    OUTPUT (CHANNEL, DEVICE);
    IF IOCHAN (CHANNEL) AND %200000 THEN
      BEGIN
        WRITE ("[C]FILE NAME: "); BREAK.OUTPUT;
        READ (FILE);
        OPENFILE (CHANNEL, FILE)
      END;
          .
          .
          .
END
```

16.11 TRANSFERRING FILES

Once a device has been allocated to an input or an output channel, a complete file of information may be transferred between them automatically by calling the parameter-less procedure TRANSFILE. This procedure copies bytes from one device to another from the currently selected input channel to the currently selected output channel, until an end-of-file status is raised on either the input or output channel. When this occurs, the channels are examined to see if a true file (disk or DECtape) is open on them and, if so, the files are closed. The devices are not released or otherwise disturbed.

CHAPTER 17
THE DECsystem-10 OPERATING ENVIRONMENT

The operating environment of DECsystem-10 ALGOL programs consists of those procedures in the DECsystem-10 ALGOL Library required by the user's program, and the DECsystem-10 ALGOL Object Time System.

The former are those procedures detailed in Chapters 13 and 16, together with those described below. These procedures can be thought of as existing in a block surrounding the user's program, and, therefore, are available when called. Their names, however, are in no sense reserved as are words such as BEGIN.

Note also that these procedures are only present in the user's program when required. They are loaded by the DECsystem-10 Linking Loader when so directed by the DECsystem-10 ALGOL Compiler. The user is not required to take any action to include these procedures, other than make a call to them. A complete list of library procedures is given below.

17.1 MATHEMATICAL PROCEDURES

The following procedures all have one argument, of real type, and yield a real type result.

Procedure Name	Function
SIN	Sine
COS	Cosine
ARCTAN	Arctangent
SQRT	Square root
EXP	Exponential
LN	Logarithm (to base e)
TAN	Tangent
ARCSIN	Arcsine
ARCCOS	Arccosine
SINH	Sinh
COSH	Cosh
TANH	Tanh

The following procedures all have one argument, of long real type, and yield a long real type result. Note that they are formed by adding an L before the equivalent single precision procedure.

Procedure Name	Function
LSIN	Sine
LCOS	Cosine
LARCTAN	Arctangent
LSQRT	Square root
LEXP	Exponential
LLN	Logarithm (to base e)

17.2 STRING PROCEDURE

For details of the procedures LINK, LINKR, TAIL, LENGTH, COPY, NEWSTRING and DELETE, see Paragraph 13.6.

17.3 UTILITY PROCEDURES

17.3.1 Array Dimension Procedures

The integer procedure DIM, which takes as its parameter the name of an array of any type, yields a result that is the number of dimensions of the array. This is most useful when the user passes an array as a parameter and wishes to check if it is, for example, a matrix.

The integer procedures LB and UB also take as first parameters the name of an array; their second parameter is the subscript number. The result is the lower or upper bound, respectively, of the subscript specified by the second parameter. The following procedure uses these to clear real matrices.

```
PROCEDURE ZERO(A); ARRAY A;
BEGIN
  INTEGER I,J;
  IF DIM(A) = 2 THEN
  BEGIN
    L1 := LB(A,1); U1 := UB(A,1);
    L2 := LB(A,2); U2 := UB(A,2);
    FOR I := L1 UNTIL U1 DO
      FOR J := L2 UNTIL U2 DO A[I,J] := 0
  END
END
```

17.3.2 Minima and Maxima Procedures

The integer procedures IMIN and IMAX, the real procedures RMIN and RMAX, and the long real procedures LMIN and LMAX are used, respectively, to determine the minimum or maximum of a number of arguments of the appropriate type. These procedures normally accept up to ten parameters (this figure may be easily changed by altering a parameter in the ALGOL Library).

For example:

```
I := IMIN(J,K);
X := RMAX(Y+Z,RMIN(Y-Z,Q));
```

17.3.3 Field Manipulations

The procedures GFIELD and SFIELD enable the user to manipulate a field within any integer, real, long real, Boolean or string variable. The integer parameters I and J specify a byte of length J bits whose leftmost bit is the I'th bit (counting from zero at the left-hand side). The byte specified may be from 1 to 36 bits in length and may be at any position in the variable.

For single word variables (integer, real, Boolean), I may range from 0 to 35, with the constraint I + J <= 36. For double word variables (long real and string), I may range from 0 to 71, with the constraint I + J <= 72.

The integer procedure GFIELD uses I and J as the second and third parameters; the first parameter is the variable. The result is the value of the byte (right justified) specified by I, J.

Thus

```
K := GFIELD(A,3,5);
```

gives the value of the byte consisting of bits 3 through 7 of A.

The procedure SFIELD sets a byte specified by the second and third parameters I, J to the value specified by the fourth parameter, of type integer. Thus

```
SFIELD(A,3,5,0);
```

zeros the byte specified in the first example.

17.4 DATA TRANSMISSION PROCEDURES

For details of these procedures refer to Chapter 16.

17.5 FORTRAN INTERFACE PROCEDURES[1]

FORTRAN subroutines may be incorporated in ALGOL object programs by loading these subroutines with the ALGOL main program (and any other separate ALGOL procedures).

Such FORTRAN subroutines should be specified by an EXTERNAL declaration in the ALGOL program and can be called by the appropriate use of one of the ALGOL library procedures:

CALL, ICALL, RCALL, DCALL, or LCALL

which are used, respectively, to call nontype, integer, real, long real (double precision), and Boolean (logical) subroutines.

The first parameter in these procedures calls must be the name of the FORTRAN subroutine. Subsequent parameters are taken as the arguments to the procedures.

CALL is used as a single statement, for example:

```
CALL (FORT,X,Y)
```

is equivalent to

```
CALL FORT (X,Y)
```

in a FORTRAN program.

ICALL etc. must appear in the appropriate context in an expression, thus

```
P := Q + ICALL(Z)
```

NOTE

The parameters of CALL, ICALL, etc., are restricted to integer, real, long real, or Boolean expressions or variables; arrays are not permitted.

17.5.1 FORTRAN Input/Output

If FORTRAN input/output is to be used in any FORTRAN subroutine in an ALGOL object program, the ALGOL library procedure

```
FORTIO
```

must first be called. Failure to do this will result in a routine error message.

The user is advised that the simultaneous use of ALGOL and FORTRAN input/output may result in failure of the object program.

---

[1] These features will appear in Version 3 of ALGOL.

CHAPTER 18
RUNNING AND DEBUGGING PROGRAMS

18.1 COMPILATION OF ALGOL PROGRAMS

DECsystem-10 ALGOL programs are compiled by the ALGOL compiler under the standard DECsystem-10 timesharing monitor. The compiler is called by typing

```
R ALGOL
```

at monitor command level.

The DECsystem-10 ALGOL Compiler responds by typing an asterisk on the user's terminal. The user then types a command string to the compiler, specifying the source file(s) from which the program is to be compiled, and the output files for listing and output of relocatable binary. The command string takes the form:

```
OUTPUT-FILE,LISTING-FILE=SOURCE-FILES
```

followed by a carriage-return (ALTMODE cannot be used to terminate a command string).
A file takes one of the forms

```
DEVICE:FILE-NAME.FILE-EXTENSION
```

or

```
DEVICE:FILE.NAME
```

for directory devices (disk and DECtape)

or

```
FILE-NAME.FILE-EXTENSION
```

or

```
FILE-NAME
```

where DSK is assumed to be a default device.

In the case of non-directory devices, the format is simply

DEVICE:

In cases where no FILE-EXTENSIONS are specified, the standard defaults REL for the relocatable binary output file, LST for the listing file, and ALG for the source file are assumed.

SOURCE-FILES

consists of one file or a list of files separated by commas. If a DEVICE is specified for the first file, and not for succeeding files, the second and following files are taken from the same device as the first.

Example:

```
EULER,TTY:←EULER
```

[read source from DSK:EULER.ALG, write relocatable binary on DSK:EULER.REL, and listing on the user's terminal].

```
MTA0:,DSK:SIM26←SIM26,PARAM.TST
```

[read source from DSK:SIM26.ALG, DSK:PARAM.TST, write relocatable binary on device MTA0, and listing on file DSK:SIM26.LST].

Certain switches may be set by the user in the command string. These are:

nD	(where n is an unsigned decimal integer). Set the dynamic storage region (called the "heap") to n words. All input/output buffers, dynamically-created strings, and own arrays are allocated in the heap; hence, this area must be sufficiently large to accommodate all such items. The default size of the heap is 512 words.
E	Line numbers are in columns 73 through 80 of the source program.
L	List the source program (default case).
N	Error messages are not printed on the user's terminal.
Q	Delimiter words are in quotes.
S	Suppress listing of the source program.

These switches are set by preceding them with a / after a file, for example:

```
PROD,PROD/1000D←PROD1/L,PROD2/S
```

causes file PROD1.ALG to be compiled with listing, file PROD2.ALG to be compiled without listing, and causes the size of the heap to be set to 1000 words.

The ALGOL compiler reports all source program errors both on the user's terminal and in the listing device (if it is other than the terminal). After compiling a program, the compiler returns with another asterisk, whereupon the user may compile another program, or type ↑C to return to monitor level.

18.1.1 Compilation of Free-Standing Procedures

DECsystem-10 ALGOL allows the user to compile procedures independent of programs that call them. Such procedures may either follow the main program in the source file (but may not appear before it), or may be in an independent source file either singly or together. The user uses exactly the same process to compile such files.

If the user requires to call those procedures from the main ALGOL program, the appropriate EXTERNAL declarations must be made (refer to Paragraph 11.9).

18.2 LOADING ALGOL PROGRAMS

ALGOL programs are loaded by means of the DECsystem-10 Linking Loader in exactly the same way as programs generated by MACRO-10 and FORTRAN (for details, refer to the DECsystem-10 Assembly Language Handbook).

The loader automatically causes all procedures required from the ALGOL Library (ALGLIB) to be incorporated into the user's program.

For example, consider the source file MAIN.ALG which contains the ALGOL main program and the files SUB1.ALG and SUB2.ALG which contain free-standing procedures.

The user may compile these files to give one relocatable binary file by typing the following command string to the ALGOL compiler,

```
MAIN,MAIN←MAIN,SUB1,SUB2
```

and loading the resulting program by giving the command string

```
MAIN$
```

to the loader. Alternatively, the three source files can be compiled independently by typing three command strings to the ALGOL compiler, for example:

```
MAIN,MAIN←MAIN
SUB1,SUB1←SUB1
SUB2,SUB2←SUB2
```

and giving the loader the command string

```
MAIN,SUB1,SUB2$
```

After a program has been loaded, it may be executed.

18.3 RUNNING ALGOL PROGRAMS

ALGOL programs are executed by typing the console command

```
START
```

or any of its valid abbreviations. If the program executes successfully, it finishes by printing the execution time statistics (core store used and execution and elapsed times) on the user's terminal, and returns to monitor command level.

18.4 CONCISE COMMAND LANGUAGE

The concise command language (CCL) features in the DECsystem-10 monitor may be used to facilitate the compilation and execution of ALGOL programs. They are used in exactly the same way as for programs written in DECsystem-10 FORTRAN. For details, refer to the DECsystem-10 Users Handbook.

18.5 RUN-TIME DIAGNOSTICS AND DEBUGGING

When an error occurs during the execution of a user's ALGOL program, control is passed to the ALGOL Object Time System which monitors the error on the user's terminal. A concise description of the error is given, together with information on the location.

Full details of the debugging system will be documented when they are available.

18.5.1 Facilities to Aid in Program Debugging

18.5.1.1 Checking - The directive

```
CHECKON 1
```

when placed anywhere in a user's program causes all array subscripts from this point onward in the program to be checked at run-time for being in range. The directive

```
CHECKOFF 1
```

nullifies this action. Note that use of this facility causes the generated program to be slightly larger, and to run slower.

NOTE

Most inexplicable errors arising during the execution of an ALGOL program are caused by an array subscript being out of range. Whenever such errors occur, the program should be recompiled with the array bound check feature on, and rerun.

18.5.1.2 Controlling Listing of the Source Program - Normally, a listing of the source program is output with the object program during compilation. The user can suppress this listing entirely by means of the /S compiler switch. However, if the user wishes to suppress only part of the listing and then continue listing, he can control the listing from within his program by means of the statements

```
LISTOFF
LISTON
```

The LISTOFF statement causes listing to be suppressed from the point in the program where LISTOFF was encountered to either the end of the program or until a LISTON statement is encountered. The LISTON statement causes listing to continue after it had been suppressed by a LISTOFF statement. The LISTON and LISTOFF statements have no effect if the /S switch is included in the compiler command string.

18.5.1.3 Setting Line Numbers in Listings - Ordinarily, the lines in the listing file are numbered sequentially starting at 1 and incrementing by 1. The user can, however, change the line numbers by placing sequence numbers in columns 73 through 80 of the source program and compiling with the /E switch. Another way in which the user can change the line numbers is by means of the LINE statement. The statement

```
LINE n
```

causes the next line number to be set to n, which is a decimal integer. The line numbers that follow are incremented by 1 until either another LINE statement is encountered or the program terminates.

CHAPTER 19
TECHNICAL NOTES

These notes concern the authors' particular interpretation of the "Revised Report on the Algorithmic Language ALGOL-60" and its implementation.

a. At all times, strict left-to-right evaluation of statements is employed. Section 3.4.6 of the Revised Report has been construed by some experts to mean that left-to-right evaluation of expressions is not required. However, there are undoubtedly many ALGOL-60 programs in existence that rely on this feature.

b. Section 4.3.5 of the Revised Report requires that a GOTO Statement with a designational expression which is a switch with a subscript out of range be regarded as a dummy statement. Neither DECsystem-10 ALGOL nor any other ALGOL-60 implementations, to the knowledge of the authors, follow this rule if there is a side-effect involved in the evaluation of the subscript.

INDEX

NOTES

NOTES

NOTES

NOTES

NOTES

NOTES

READER'S COMMENTS

DECsystem-10
MATHEMATICAL LANGUAGES HANDBOOK
DEC-10-KRZB-D

Digital Equipment Corporation maintains a continuous effort to improve the quality and usefulness of its publications. To do this effectively we need user feedback: your critical evaluation of this document. Please give specific page and line references when appropriate.

ERRORS NOTED IN THIS PUBLICATION:

_____

_____

_____

_____

_____

SUGGESTIONS FOR IMPROVEMENT OF THIS PUBLICATION:

_____

_____

_____

_____

_____

_____

_____

DEC also strives to keep its customers informed about current DEC software and publications. Thus, the following periodically distributed publications are available upon request. Please check the publication(s) desired.

☐ **PDP-10 User's Bookshelf**, a bibliography of current programming documents.

☐ **Program Library Price List**, a list of available software documents and programs.

Name _____ Date _____

Organization _____

Please describe your position _____

Street _____

City _____ State _____ Zip Code _____

Fold Here

Do Not Tear - Fold Here and Staple

FIRST CLASS
PERMIT NO. 33
MAYNARD, MASS.

BUSINESS REPLY MAIL
NO POSTAGE STAMP NECESSARY IF MAILED IN THE UNITED STATES

Postage will be paid by:

Digital Equipment Corporation
Software Information Services
146 Main Street, Bldg. 3-5
Maynard, Massachusetts 01754

DIGITAL EQUIPMENT CORPORATION digital WORLD-WIDE SALES AND SERVICE

MAIN OFFICE AND PLANT

146 Main Street, Maynard, Massachusetts, U.S.A. 01754 • Telephone: From Metropolitan Boston: 646-8600 • Elsewhere: (617)-897-5111
TWX: 710-347-0212 Cable: DIGITAL MAYN Telex: 94-8457

UNITED STATES

NORTHEAST

REGIONAL OFFICE
275 Wyman Street, Waltham, Massachusetts 02154
Telephone: (617)-890-0320/0330 TWX: 710-324-6919

WALTHAM
15 Lunda Street, Waltham, Massachusetts 02154
Telephone: (617)-891-1030 TWX: 710-324-6919

CAMBRIDGE/BOSTON
899 Main Street, Cambridge, Massachusetts 02139
Telephone: (617)-491-6130 TWX: 710-320-1167

ROCHESTER
130 Allens Creek Road, Rochester, New York 14618
Telephone: (716)-461-1700 TWX: 710-253-3078

CONNECTICUT
240 Pomeroy Ave., Meriden, Conn. 06450
Telephone: (203)-237-8441/7466 TWX: 710-461-0054

MID-ATLANTIC — SOUTHEAST

REGIONAL OFFICE:
U.S. Route 1, Princeton, New Jersey 08540
Telephone: (609)-452-2940 TWX: 510-685-2338

NEW YORK
95 Cedar Lane, Englewood, New Jersey 07631
Telephone: (201)-871-4984, (212)-594-6955, (212)-736-0447
TWX: 710-991-9721

NEW JERSEY
1259 Route 46, Parsippany, New Jersey 07054
Telephone: (201)-335-3300 TWX: 710-987-8319

PRINCETON
U.S. Route 1
Princeton, New Jersey 08540
Telephone: (609) 452-2940 TWX: 510-685-2338

LONG ISLAND
1 Huntington Quadrangle
Suite 1SO7 Huntington Station, New York 11746
Telephone: (516)-694-4131, (212)-895-8095

PHILADELPHIA
Station Square Three, Paoli, Pennsylvania 19301
Telephone: (215)-647-4900/4410 Telex: 510-668-8395

WASHINGTON
Executive Building
6811 Kenilworth Ave., Riverdale, Maryland 20840
Telephone: (301)-779-1600/752-8797 TWX: 710-826-9662

DURHAM/CHAPEL HILL
2704 Chapel Hill Boulevard
Durham, North Carolina 27707
Telephone: (919)-489-3347 TWX: 510-927-0912

ORLANDO
Suite 130, 7001 Lake Ellenor Drive, Orlando, Florida 32809
Telephone: (305)-851-4450 TWX: 810-850-0180

ATLANTA
2815 Clearview Place, Suite 100,
Atlanta, Georgia 30340
Telephone: (404)-451-3734/3735/3736 TWX: 810-757-4223

MID-ATLANTIC — SOUTHEAST (cont.)

KNOXVILLE
6311 Kingston Pike, Suite 21E
Knoxville, Tennessee 37919
Telephone: (615)-588-6571 TWX: 810-583-0123

CENTRAL

REGIONAL OFFICE:
1850 Frontage Road, Northbrook, Illinois 60062
Telephone: (312)-498-2500 TWX: 910-686-0655

PITTSBURGH
400 Penn Center Boulevard
Pittsburgh, Pennsylvania 15235
Telephone: (412)-243-9404 TWX: 710-797-3657

CHICAGO
1850 Frontage Road, Northbrook, Illinois 60062
Telephone: (312)-498-2500 TWX: 910-686-0655

ANN ARBOR
230 Huron View Boulevard, Ann Arbor, Michigan 48103
Telephone: (313)-761-1150 TWX: 810-223-6053

INDIANAPOLIS
21 Beachway Drive — Suite G
Indianapolis, Indiana 46224
Telephone: (317)-243-8341 TWX: 810-341-3436

MINNEAPOLIS
Suite 111, 8030 Cedar Avenue South,
Minneapolis, Minnesota 55420
Telephone: (612)-854-6562-3-4-5 TWX: 910-576-2818

CLEVELAND
Park Hill Bldg., 35104 Euclid Ave.
Willoughby, Ohio 44094
Telephone: (216)-946-8484 TWX: 810-427-2608

ST. LOUIS
Suite 110, 115 Progress Pky., Maryland Heights,
Missouri 63043
Telephone: (314)-878-4310 TWX: 910-764-0831

DAYTON
3101 Kettering Blvd., Dayton, Ohio 45439
Telephone: (513)-299-7377 TWX: 810-459-1676

MILWAUKEE
8531 W. Capitol Drive, Milwaukee, Wisconsin 53222
Telephone: (414)-463-9110 TWX: 910-262-1199

DALLAS
8855 North Stemmons Freeway
Dallas, Texas 75247
Telephone: (214)-638-4880 TWX: 910-861-4000

HOUSTON
3417 Milam Street, Suite A, Houston, Texas 77002
Telephone: (713)-524-2961 TWX: 910-881-1651

CENTRAL (cont.)

NEW ORLEANS
3100 Ridgelake Drive, Suite 108
Metairie, Louisiana 70002
Telephone: 504-837-0257

WEST

REGIONAL OFFICE
310 Soquel Way, Sunnyvale, California 94086
Telephone: (408)-735-9200

ANAHEIM
801 E. Ball Road, Anaheim, California 92805
Telephone: (714)-776-6932/8730 TWX: 910-591-1189

WEST LOS ANGELES
1510 Cotner Avenue, Los Angeles, California 90025
Telephone: (213)-479-3791/4318 TWX: 910-342-6999

SAN DIEGO
3444 Hancock Street
San Diego, California 92110
Telephone: (714)-298-0591, 0583 TWX: 910-335-1230

SAN FRANCISCO
1400 Terra Bella
Mountain View, California 94040
Telephone: (415)-964-6200 TWX: 910-373-1266

PALO ALTO
560 San Antonio Rd., Palo Alto, California 94306
Telephone: (415)-969-6200 TWX: 910-373-1266

OAKLAND
7850 Edgewater Drive
Oakland, California 94621
Telephone: (415)-635-5453/7830 TWX: 910-366-7238

ALBUQUERQUE
6303 Indian School Road, N.E.
Albuquerque, N.M. 87110
Telephone: (505)-296-5411/5428 TWX: 910-989-0614

DENVER
2305 South Colorado Blvd., Suite #5
Denver, Colorado 80222
Telephone: (303)-757-3332/758-1656/758-1659
TWX: 910-931-2650

SEATTLE
1521 130th N.E., Bellevue, Washington 98005
Telephone: (206)-454-4058/455-5404 TWX: 910-443-2306

SALT LAKE CITY
431 South 3rd East, Salt Lake City, Utah 84111
Telephone: (801)-328-9838 TWX: 910-925-5834

PHOENIX
4358 East Broadway Road
Phoenix, Arizona 85040
Telephone: (602) 268-3488 TWX: 910-950-4691

PORTLAND
Suite 168
5319 S.W. Canyon Court, Portland, Ore. 97221
Telephone: (503) 297-3761/3765

INTERNATIONAL

EUROPEAN HEADQUARTERS

Digital Equipment Corporation International Europe
81 Route de l'Aire
1211 Geneva 26, Switzerland
Telephone: 42 79 50 Telex: 22 683

FRANCE

Equipment Digital S.A.R.L.

PARIS
327 Rue de Charenton, 75 Paris 12 ème, France
Telephone: 344-76-07 Telex: 21339

GRENOBLE
10 rue Auguste Ravier, F-38 Grenoble, France
Telephone: (76) 87 87 32 Telex: 32 882 F (Code 212)

GERMANY

Digital Equipment GmbH

MUNICH
8 Muenchen 13, Wallensteinplatz 2
Telephone: 0811-35031 Telex: 524-226

COLOGNE
5 Koeln, Bismarckstrasse 7,
Telephone: 0221-522181 Telex: 888-2269
Telegram: Flip Chip Koeln

FRANKFURT
6078 Neu-Isenburg 2
Am Forsthaus Gravenbruch 5-7
Telephone: 06102-5526 Telex: 41-76-82

HANNOVER
3 Hannover, Podbielskistrasse 102
Telephone: 0511-69-70-95 Telex: 922-952

AUSTRIA

Digital Equipment Corporation Ges.m.b.H
VIENNA
Mariahilferstrasse 136, 1150 Vienna 15, Austria
Telephone: 85 51 86

UNITED KINGDOM

Digital Equipment Co., Ltd.

U.K. HEADQUARTERS
Arkwright Road, Reading, Berks.
Telephone: 0734-583555 Telex: 84327

READING
The Evening Post Building, Tessa Road
Reading, Berks.

BIRMINGHAM
29/31, Birmingham Road, Sutton Coldfield, Warwicks.
Telephone: (0044) 21-355 5501 Telex: 337 060

MANCHESTER
13 Upper Precinct, Walkden, Manchester M28 5AZ
Telephone: 061-790-8411 Telex: 668666

LONDON
Bilton House, Uxbridge Road, Ealing, London W.5.
Telephone: 01-579-2334 Telex: 22371

EDINBURGH
Shiel House, Craigshill, Livingston,
West Lothian, Scotland
Telephone: 32705 / Telex: 727113

NETHERLANDS

THE HAGUE
Digital Equipment N.V.
Sir Winston Churchilllaan 370
Rijswijk/The Hague, Netherlands
Telephone: 070-995-160 Telex: 32533

BELGIUM

BRUSSELS
Digital Equipment N.V./S.A.
108 Rue D'Arlon
1040 Brussels, Belgium
Telephone: 02-139256 Telex: 25297

SWEDEN

Digital Equipment Aktiebolag
STOCKHOLM
Vretenvagen 2, S-171 54 Solna, Sweden
Telephone: 98 13 90 Telex: 170 50
Cable: Digital Stockholm

NORWAY

Digital Equipment
OSLO
c/o Firma Service
Waldenmarthranesgate 84-B-86
Oslo 1, Norway
Telephone: 37 19 85, 37 02 30 Telex: 166 43

DENMARK

Digital Equipment Corporation
COPENHAGEN
Vesterbrogade 140, 1620 Copenhagen V

SWITZERLAND

Digital Equipment Corporation S.A.
GENEVA
81 Route de l'Aire
1211 Geneva 26, Switzerland
Telephone: 42 79 50 Telex: 22 683
ZURICH
Scheuchzerstrasse 21
CH-8006 Zurich, Switzerland
Telephone: 01/60 35 66 Telex: 56059

ITALY

Digital Equipment S.p.A.
MILAN
Corso Garibaldi 49, 20121 Milano, Italy
Telephone: 872 748 694 394 Telex: 33615

SPAIN

MADRID
Ataio Ingenieros S.A., Enrigue Larreta 12, Madrid 16
Telephone: 215 35 43 / Telex: 27249

BARCELONA
Ataio Ingenieros S.A., Ganduxer 76, Barcelona 6
Telephone: 221 44 66
Digital Equipment Corporation Ttd.

AUSTRALIA

Digital Equipment Australia Pty. Ltd.
SYDNEY
P.O. Box 491, Crows Nest
N.S.W. Australia 3065
Telephone: 439-2566 Telex: AA20740
Cable: Digital, Sydney

MELBOURNE
60 Park Street, South Melbourne, Victoria, 3205
Telephone: 696-142 Telex: AA40616

PERTH
643 Murray Street
West Perth, Western Australia 6005
Telephone: 214-993 Telex: AA92140

BRISBANE
139 Merivale Street, South Brisbane
Queensland, Australia 4101
Telephone 444-047 Telex: AA40616

ADELAIDE
6 Montrose Avenue
Norwood, South Australia 5067
Telephone: 631-339 Telex: AA82825

NEW ZEALAND

Digital Equipment Corporation Ltd.
AUCKLAND
Hilton House, 430 Queen Street, Box 2471 A,
Auckland, New Zealand
Telephone: 75-533

CANADA

Digital Equipment of Canada, Ltd.

CANADIAN HEADQUARTERS
150 Rosamond Street, Carleton Place, Ontario
Telephone: (613)-257-2615 TWX: 610-561-1651

OTTAWA
120 Holland Street, Ottawa 3, Ontario K1Y 0X7
Telephone: (613)-725-2193 TWX: 610-562-8907

TORONTO
230 Lakeshore Road East, Port Credit, Ontario
Telephone: (416)-274-1241 TWX: 610-492-4306

MONTREAL
9675 Cote de Liesse Road
Dorval, Quebec, Canada 760
Telephone: 514-636-9393 TWX: 610-422-4124

EDMONTON
5531 - 103 Street
Edmonton, Alberta, Canada
Telephone: (403)-434-9333 TWX: 610-831-2248

VANCOUVER
Digital Equipment of Canada, Ltd.
2210 West 12th Avenue
Vancouver 9, British Columbia, Canada
Telephone: (604)-736-5616 TWX: 610-929-2006

ARGENTINA

BUENOS AIRES
Coasin S.A.
Virrey del Pino 4071, Buenos Aires
Telephone: 52-3185 Telex: 012-2284

VENEZUELA

CARACAS
Coasin S.A. (Sales only)
Apartado 50939
Salana Grande No. 1, Caracas
Telephone: 72-9637 Cable: INSTRUVEN

CHILE

SANTIAGO
Coasin Chile Ltda. (sales only)
Casilla 14588, Correo 15, Santiago
Telephone: 396713 Cable: COACHIL

JAPAN

TOKYO
Rikei Trading Co., Ltd. (sales only)
Kozato-Kaikan Bldg.
No. 18-14, Nishishimbashi 1-chome
Minato-Ku, Tokyo, Japan
Telephone: 5915246 Telex: 781-4208
Digital Equipment Corporation International
Kowa Building No. 17, Second Floor
2-7 Nishi-Azabu 1-Chome
Minato-Ku, Tokyo, Japan
Telephone: 404-5894/6 Telex: TK-6428

PHILIPPINES

Stanford Computer Corporation
P.O. Box 1608
416 Dasmarinas St., Manila
Telephone: 49-68-96 Telex: 742-0352

INDIA

H.S. Sonawala Mg Director (Sales Only)
HINDITRON SERVICES PUT LTD.
69/A Nepean Sea Road
Bombay, India

digital

decsystem10 handbook series