CONTROL DATA
CORPORATION

CONTROL DATA®

CYBER 170 COMPUTER SYSTEM MODELS 172, 173, 174, 175

CYBER 70 COMPUTER SYSTEM MODELS 72, 73, 74

6000 COMPUTER SYSTEMS

BASIC LANGUAGE VERSION 2 REFERENCE MANUAL

CONTROL DATA
CORPORATION

CONTROL DATA®
CYBER 170 COMPUTER SYSTEM MODELS 172,173,174,175
CYBER 70 COMPUTER SYSTEM MODELS 72,73,74
6000 COMPUTER SYSTEMS

BASIC LANGUAGE VERSION 2
REFERENCE MANUAL

REVISION RECORD

REVISION	DESCRIPTION
A	Original printing.
(7-73)	
B	Includes corrections to Revision A and the Network Operating System (NOS) update.
(10-74)	
Publication No. 19980300	

Address comments concerning this manual to:
Control Data Corporation
Technical Publications Department
4201 North Lexington Avenue
Arden Hills, Minnesota 55112

or use Comment Sheet in the back of this manual.

PREFACE

This manual describes the BASIC subsystem which permits the SCOPE, KRONOS® and the Network Operating System (NOS) time - sharing terminal user to enter, compile, and execute interactive programs. The BASIC 2.1 compiler is a modification to the CONTROL DATA BASIC Version 2.0 compiler for CYBER 70, Models 72, 73 and 74, CYBER 170, Models 172, 173, 174 and 175, and 6000 Series computers. This version is an extension of the original BASIC language which was designed and implemented at the Dartmouth College Computation Center.

Although BASIC is normally used interactively from a remote terminal, BASIC programs can be compiled and executed as batch programs.

This manual does not contain a description of the complete system hardware or software. For additional information pertaining to the CDC CYBER 70, CYBER 170, and 6000 Series computers; SCOPE 3.4, KRONOS 2.1, NOS 1.0 operating systems; and time-sharing job processing, refer to the following manuals:

Control Data Publication	Publication No.
Control Data 6400/6500/6600 Computer Systems Reference Manual	60100000
INTERCOM Reference Manual, Version 4.2	60307100
KRONOS 2.1 Reference Manual	60407000
SCOPE Reference Manual, Version 3.4	60307200
KRONOS 2.1 TEXT EDITOR Reference Manual	60408200
KRONOS 2.1 TIME SHARING Reference Manual	60407600
NOS V1.0 Reference Manual	60435400
NOS V1.0 TIME SHARING Reference Manual	60435500
NOS V1.0 TEXT EDITOR Reference Manual	60436100
CYBER 70/Model 72 System Description, Volume 1	60347000
CYBER 70/Model 73 System Description, Volume 1	60347200
CYBER 70/Model 74 System Description, Volume 1	60347400

Control Data Publication	Publication No.
CYBER 170/Models 172, 173, 174 System Description,Volume 1	19981200
CYBER 170/Model 175 System Description,Volume 1	60420000

CONTENTS

INTRODUCTION

BASIC is an all-purpose programming language that includes features which render it well-suited for scientific, business, and educational applications. BASIC provides a small but powerful set of easy-to-learn statements that are " English-like" and written in free format. Some of the more important features provided by BASIC are:

- Numeric and character string manipulation.
- Array definition and redimensioning.
- Access to trigonometric, matrix, and string functions.
- Facility for writing user-defined functions.
- Facility for writing subroutines.
- Matrix I/O for one and two dimensional arrays.
- Output format determination.
- File manipulation of coded and binary files.
- Error detection and processing during program execution.

This document is intended to describe these and other BASIC features to both non-programmer and experienced programmer. The information in this manual is therefore provided in three major sections:

- Section 1 - is a primer or introduction to the BASIC language directed at the non-programmer.
- Section 2 - includes reference information that is an expansion of Section 1, and reference information directed at the experienced programmer.
- Appendixes - include information that support Sections 1 and 2, and summary information.

BASIC PRIMER 1.0

Modern digital computers are designed for a wide range of applications. However, all digital computers have certain common characteristics. They all perform tasks specified by a set of instructions.

A set of sequential instructions designed to solve a specific problem is called a program. Such a program might perform a simple task such as adding or subtracting two numbers, or printing a single letter or digit. Typically, a program for a complete scientific computation could require a few thousand computer instructions.

Computer programs process or manipulate information called data. A program can be used to perform calculations using data and to print out the results. Most programs permit new data to be input each time the program is used. The three phases of a program operation are: Input, Computation, and Output.

When a program is performing tasks in a computer, the process is called "program execution" or " running" a program.

PROGRAMMING AND LANGUAGES

Computers can execute thousands and even millions of computer instructions each second; therefore, computer instructions must be structured in a form suited to the computers architecture. To write a program using computer instructions in the form used directly by the computer (machine instructions) is tedious and time-consuming. To simplify writing programs, computer specialists have developed several "high level", easy-to-use programming languages and associated compilers and translators to convert these high level languages to machine instructions. BASIC, the Beginner's All Purpose Symbolic Instruction Code, is one such high level language. BASIC was originally developed by professors Kemeny and Kurtz at Dartmouth College.

The BASIC language is used on Control Data CYBER 70 Series, CYBER 170 Series, or 6000 series time-sharing computer systems. This time-sharing capability permits the simultaneous use of the computer by more than one user without apparent restriction. The compilation and execution of each user program is controlled and monitored through use of an " operating system" which is the case of the CYBER or 6000 computers can be SCOPE V3.4, KRONOS V2.1, or the Network Operating System (NOS) V1.0.

This section describes to a non-programmer how to write, enter and execute a BASIC program by solving a sample problem with a BASIC program and describing the BASIC statements used in solving the problem.

After studying this section the reader will be familiar enough with the BASIC language to write BASIC programs and to understand the more detailed description of the BASIC language provided in section 2.

STATEMENT OF THE PROBLEM

The following general description of a manufacturing system is the problem to be solved using BASIC. In the following problem F represents fixed costs per year associated with production, C represents variable costs incurred per unit, and V represents the annual volume of production (and sales) in units. Then the total cost incurred per year is T = F + CV. If the revenue per unit made (and sold) is R per unit, then the total annual revenue is R1 = RV. The profit obtained on an annual basis is the difference between R1 and T, if that result is positive. A loss occurs if R1 - T is negative. The break-even point is reached when the volume is sufficient to make R1 = T.

A company operates with fixed costs of $1 million per year, variable costs of $10 per unit, and a revenue of $30 per unit of production.

a. What is the break-even point?

b. If the predicted sales are 25,000 units, what is the exptected profit or loss?

c. What is the expected profit or loss for sales of 50,000, 75,000 and 100,000 units?

The following BASIC program was written to solve parts A and B of the problem. The solution to part C is provided later in this section.

SAMPLE BASIC PROGRAM

```
001 REM THIS IS A BREAKEVEN PRØGRAM
002 REM THE FØLLØWING VARIABLES ARE USED
003 REM FIXED ANNUAL CØST           F
004 REM VARIABLE CØST PER UNIT      C
005 REM SALES REVENUE PER UNIT      R
006 REM SALES VØLUME                V
007 REM BREAK-EVEN PØINT (VØLUME)   V1
008 REM TØTAL CØST                  T
009 REM TØTAL REVENUE               R1
010 REM PRØFIT/LØSS                 P
011 REM
012 REM
013 REM ASSIGN VALUES TØ VARIABLES F,C,R,V
020 LET F=1000000
030 LET C=10
040 LET R=30
050 LET V=25000
060 REM
070 REM CØMPUTE BREAK-EVEN PØINT
080 LET V1=F/(R-C)
090 PRINT "BREAK-EVEN PØINT=";V1;"VØLUME UNITS"
100 REM
110 REM CØMPUTE TØTAL CØST
120 LET T=F+C*V
130 REM
140 REM CØMPUTE TØTAL REVENUE
150 LET R1=R*V
160 REM
170 REM CØMPUTE PRØFIT/LØSS
180 LET P=R1-T
200 IFV>V1 THEN 230
210 PRINT "LØSS = $";P,"VØLUME =";V;"UNITS"
220 GØTØ 240
230 PRINT "PRØFIT=$";P,"VØLUME=";V;"UNITS"
240 END
```

ANALYSIS OF STATEMENTS

REM Statement

Observe that each line of the following example includes a number and these numbers are presented in ascending order.

```
001 REM THIS IS A BREAK-EVEN PRØGRAM
002 REM THE FØLLØWING VARIABLES ARE USED:
003 REM FIXED ANNUAL CØST           F
004 REM VARIABLE CØST PER UNIT      C
005 REM SALES REVENUE PER UNIT      R
006 REM SALES VØLUME                V
007 REM BREAK-EVEN PØINT (VØLUME) V1
008 REM TØTAL CØST                  T
009 REM TØTAL REVENUE               R1
010 REM PRØFIT/LØSS                 P
011 REM
012 REM
```

These numbers are called line numbers and identify individual lines of a BASIC program. Each line of a BASIC program is called a statement, and each statement must begin with a line number. Line numbers normally indicate the sequence in which the computer is to perform (execute) the statements. Statements may be typed in any order; however, before the computer executes the statements, it sorts them into the sequence indicated by their line numbers.

The line number is followed by a word or an abbreviation (abbreviation in the above insert) which identifies the type of BASIC statement. The sample program contains six statement types; the above insert contains one, REM. The REM statement allows the user to insert remarks which increase readability and comprehension in a program; it has no effect during execution of the program. A maximum of 72 characters can be included in a REM statement.

In the sample program, the REM statements identify the type of program, the variables used, and their identifiers. These identifiers are used later in program computations.

LET Statement

The LET statement specifies that the variable to the left of the equals sign be set to a value or the value of the formula or expression to the right of the equals sign.

Examples:

1. Constant Value Assignment - Statements 20 through 50 in the sample program assign a value to variables F, C, R, and V which are later used in computing the break-even point. The value for F, C, R, and V represent dollars and the value for V represents units.

```
013 REM ASSIGN VALUES TØ VARIABLES F,C,R,V
020 LET F = 1000000
030 LET C = 10
040 LET R = 30
050 LET V = 25000
060 REM
```

2. Formulas Value Assignment - In the sample program statements 120, 150, and 180 compute total cost, total revenue and profit or loss, respectively and assign these values to variables T, R1 and P. (The symbol * specifies multiplication.) BASIC conforms to the normal algebraic rules for order of arithmetic computation, e.g., multiply before addition, etc.

```
110 REM CØMPUTE TØTAL CØST
120 LET T = F + C * V
130 REM
140 REM CØMPUTE TØTAL REVENUE
150 LET R1 = R * V
160 REM
170 REM CØMPUTE PRØFIT / LØSS
180 LET P = R1 - T
```

Statement 120 directs the computer to multiply V(25000) by C(10) and add the product (250000) to F(1000000) giving a sum of 1250000. This sum is assigned to the variable T.

In computing total revenue, the volume (V) is multiplied by the revenue per unit (R) (25000 * 30), and the product (750000) is assigned to R1.

To determine profit or loss, the total cost (T) is subtracted from the total revenue (R1): (750000 - 1250000) and the remainder (-500000) is assigned to P.

PRINT Statement

The PRINT statement can be used to: (1) print out a value; (2) print a message; (3) print a combination of 1 and 2; (4) or print a blank line. BASIC normally separates an output line into five print zones; each 15 characters wide. Spacing is controlled with commas and semicolons embedded in the PRINT statement. The comma is used to space over to the next print zone (insert blank spaces between items); the semicolon permits items to be printed with no additional blanks between them. When printing headings or labels, enclose the heading or label in quotes in the PRINT statement. To print a blank line, simply use the PRINT statement without specifying what to print; e.g., PRINT.

Example:

Statements 080 and 090, illustrate the assignment of a value derived from a formula to a variable via the LET statement and the use of the PRINT statement in printing an identifying label along with the derived value.

```
070 REM CØMPUTE BREAK-EVEN PØINT
080 LET V1 = F/(R-C)
090 PRINT "BREAK-EVEN PØINT =";V1;"VØLUME UNITS"
100 REM
```

Statement 080 directs the computer to subtract C from R (30-10) and, using the remainder (20) as a divisor, divide F(1000000). The quotient (50000) is then assigned to the variable V1. (The symbol / indicates divide.) Statement 090 directs the computer to print the value of V1, and the "BREAK-EVEN POINT" identifying label. The unit of measure for V1 is labelled " VOLUME UNITS". When executed this PRINT statement produces:

```
BREAKEVEN-PØINT =5000 VØLUME UNITS
```

IF, GOTO and END Statements

If sales volume V is greater then (>) break-even volume, then a profit is earned. If the sales volume is less than (<) break-even volume, a loss is incurred.

```
200 IFV>V1 THEN 230
210 PRINT "LØSS = $";P,"VØLUME =";V;"UNITS"
220 GØTØ 240
230 PRINT "PRØFIT=$";P,"VØLUME=";V;"UNITS"
240 END
```

The IF statement (line 200) directs the program execution to the statement at line 230 if the condition V is greater than V1 (V> V1) is met or to the following statement (line 210) if the condition is not met. This illustrates how execution sequence by line number is altered.

The purpose of the IF statement (line 200) is to select the print label PROFIT or LOSS to be printed with the values associated with variables P and V.

In this example, the PRINT at 210 is executed because V = 25000 and V1 = 50000. After executing the PRINT statement, the computer then executes statement 220. Statement 220 is a GOTO statement that directs the computer to continue execution at statement 240.

The END statement directs the computer to stop executing the BASIC program. Its corresponding line number must be the highest in the program.

Here is the complete break-even program with solutions to parts A and B of the problem.

```
001 REM THIS IS A BREAKEVEN PRØGRAM
002 REM THE FØLLØWING VARIABLES ARE USED
003 REM FIXED ANNUAL CØST           F
004 REM VARIABLE CØST PER UNIT      C
005 REM SALES REVENUE PER UNIT      R
006 REM SALES VØLUME                V
007 REM BREAK-EVEN PØINT (VØLUME)   V1
008 REM TØTAL CØST                  T
009 REM TØTAL REVENUE               R1
010 REM PRØFIT/LØSS                 P
011 REM
012 REM
013 REM ASSIGN VALUES TØ VARIABLES F,C,R,V
020 LET F=1000000
030 LET C=10
040 LET R=30
050 LET V=25000
060 REM
070 REM COMPUTE BREAK-EVEN PØINT
080 LET V1=F/(R-C)
090 PRINT "BREAK-EVEN PØINT=";V1;"VØLUME UNITS"
100 REM
110 REM CØMPUTE TØTAL CØST
120 LET T=F+C*V
130 REM
140 REM CØMPUTE TØTAL REVENUE
150 LET R1=R*V
160 REM
170 REM CØMPUTE PRØFIT/LØSS
180 LET P=R1-T
200 IFV>V1 THEN 230
210 PRINT "LØSS = $";P,"VØLUME =";V;"UNITS"
220 GØTØ 240
230 PRINT "PRØFIT=$";P,"VØLUME=";V;"UNITS"
240 END
```

After the program is entered into the computer, the BASIC compiler is directed to execute the program. Below is the computer output after program execution.

```
BREAK-EVEN PØINT = 50000 VØLUME UNITS
LØSS = $-500000              VØLUME = 25000 UNITS
```

EXPRESSIONS IN BASIC

An expression may be simple, i.e., consists of one term (A) or complex, i.e., consists of two or more terms connected by operations (A+B-C). Expressions evaluate to a single value which can later be used in computation or can be used in determining program execution sequence (see line 200), etc. There are three types of expressions in BASIC: Arithmetic, Relational, and String. String expressions are discussed in section 2 of this manual; Arithmetic and Relational expressions are discussed in the following paragraphs and in section 2.

ARITHMETIC EXPRESSIONS

Arithmetic expressions are formed from numeric variables, numeric constants, function references and arithmetic operators. BASIC provides the following arithmetic operators:

Symbol	Meaning
↑ or * *	Exponentiation
/	Division
*	Multiplication
+	Addition
-	Subtraction

In the sample program, line numbers (080, 120, 150, 180), operators (+, -, *, and /) are used. The exponentiation operator raises a number to a specified power. For example, 2**3 means 2 raised to the third power or 2^3.

The arithmetic operators have a hierarchy for evaluation: exponentiation; then multiplication and division; then addition and subtraction. The hierarchy is altered by the use of parentheses. When using parentheses in BASIC, the rules of algebra apply. For example, 2*3+2 = 8 and 2*(3+2) = 10.

When using numbers in BASIC, commas cannot be used for separation of groups of numbers instead of for decimal grouping within a number (e.g., ten million must be written 10000000 and not 10,000,000).

A numeric variable (i.e., F, C, R and V in the sample program) is named with a single alphabetic character or an alphabetic character followed by a digit. The detailed rules for using numbers and variables are included in section 2.1.

BASIC provides several mathematical functions that may be requested within an arithmetic expression (e.g., SINE, COSINE, SQUARE ROOT, etc.). Functions are described in section 2.4.

RELATIONAL EXPRESSIONS

Relational expressions are formed by combining variables and/or arithmetic expressions constants via relational operators. Relational expressions are used in IF statements to compare two values. BASIC provides the following relational operators:

Symbol	Meaning
=	Equal to
<> or ><	Not equal to
>	Greater than
>= or =>	Greater than or equal to
<	Less than
< = or =<	Less than or equal to

An example of the use of the relational operator, > (e.g., V>V1) can be found in line 200 of the sample program. For more detail and the rules for using relational operators, see section 2.1.

DEFINING AND READING DATA

DATA AND READ STATEMENTS

In the break-even program, values are assigned to variables by using the LET statements as follows:

```
013 REM ASSIGN VALUES TØ VARIABLES F,C,R,V
020 LET F = 1000000
030 LET C = 10
040 LET R = 30
050 LET V = 25000
060 REM
```

A more efficient method of assigning values to variables is provided through use of the READ statement and the DATA statement. We can delete statements 020 through 050 and add the following:

```
035 DATA 1000000,10,30,25000
037 READ F,C,R,V
```

The DATA statement creates a block of data which is internal to the program. Within the DATA statement, values must be separated by commas. In the above insert the DATA statement precedes the READ statement. This is not required because the DATA statement may be placed anywhere in the program. The READ statement is used to access the values contained in the internal data block. The variables in the READ statement are assigned values sequentially from the data block; i.e., F = 1000000, C = 10, R = 30, and V = 25000. This method is more efficient from the programmers standpoint because when the user desires to change the input data for his program, only the associated DATA statements need to be changed.

Example:

The following break-even program is revised to include READ and DATA statements.

```
001 REM THIS IS A BREAK-EVEN PRØGRAM
002 REM THE FØLLØWING VARIABLES ARE USED:
003 REM FIXED ANNUAL CØST          F
004 REM VARIABLE CØST PER UNIT     C
005 REM SALES REVENUE PER UNIT     R
006 REM SALES VØLUME               V
007 REM BREAK-EVEN PØINT (VØLUME) V1
008 REM TØTAL CØST                 T
009 REM TØTAL REVENUE              R1
010 REM PRØFIT/LØSS                P
011 REM
012 REM
013 REM ASSIGN VALUES TØ VARIABLES F,C,R,V      } New Inserts
035 DATA 1000000,10,30,25000                    }
037 READ F,C,R,V
060 REM
070 REM CØMPUTE BREAK-EVEN PØINT
080 LET V1 = F/(R-C)
090 PRINT "BREAK-EVEN PØINT =";V1;"VØLUME UNITS"
100 REM
110 REM CØMPUTE TØTAL CØST
120 LET T = F + C * V
130 REM
140 REM CØMPUTE TØTAL REVENUE
150 LET R1 = R * V
160 REM
170 REM CØMPUTE PRØFIT / LØSS
180 LET P = R1 - T
200 IF V > V1 THEN 230
210 PRINT "LØSS = $";P,"VØLUME =";V;"UNITS"
220 GØTØ 240
230 PRINT " PRØFIT = $";P,"VØLUME =";V;"UNITS"
240 END
```

When executed this program produces:

```
BREAK-EVEN PØINT= 50000 VØLUME UNITS
LØSS = $-500000               VØLUME = 25000 UNITS
```

LOOPING IN BASIC

We are frequently interested in solving a problem, using BASIC, in which a specified sequence of statements is executed a number of times. Each time it is executed, a variable is assigned a different value. In programming this is done by using a technique called looping. The following statements provide two methods for looping.

- IF and GOTO statements.
- FOR and NEXT statements.

IF AND GOTO STATEMENTS

In the original problem, part C requested the profit/loss for sales of 50000, 25000, 75000, and 100000 units. Through use of the IF statement (104) and GOTO statement (236), a loop is inserted to solve parts A and B of the problem for these four values.

```
              001 REM THIS IS A BREAK-EVEN PRØGRAM
              002 REM THE FØLLØWING VARIABLES ARE USED:
              003 REM FIXED ANNUAL CØST             F
              004 REM VARIABLE CØST PER UNIT        C
              005 REM SALES REVENUE PER UNIT        R
              006 REM SALES VØLUME                  V
              007 REM BREAK-EVEN PØINT (VØLUME)     V1
              008 REM TØTAL CØST                    T
              009 REM TØTAL REVENUE                 R1
              010 REM PRØFIT/LØSS                   P
              011 REM
              012 REM
              013 REM ASSIGN VALUES TØ VARIABLES F,C,R
              035 DATA 1000000,10,30
              037 READ F,C,R
              060 REM
              070 REM CØMPUTE BREAK-EVEN PØINT
              080 LET V1 = F/(R-C)
              090 PRINT "BREAK-EVEN PØINT =";V1;"VØLUME UNITS"
              100 REM
              102 LET V = 25000
  ┌─────────→ 104 IF V > 100000 THEN 240
  │         │ 110 REM CØMPUTE TØTAL CØST
  │         │ 120 LET T = F + C * V
  │         │ 130 REM
  │         │ 140 REM CØMPUTE TØTAL REVENUE
  │         │ 150 LET R1 = R * V
  │         │ 160 REM
  │ LOOP    │ 170 REM CØMPUTE PRØFIT / LØSS
  │         │ 180 LET P = R1 - T
  │         │ 200 IF V > V1 THEN 230
  │         │ 210 PRINT "LØSS = $";P,"VØLUME =";V;"UNITS"
  │         │ 220 GØTØ 235
  │         │ 230 PRINT " PRØFIT = $";P,"VØLUME =";V;"UNITS"
  │         │ 235 LET V = V + 25000
  └─────────↓ 236 GØTØ 104
              240 END
```

In this example, V is assigned the initial value of 25000 (statement 102). Statement 104 then compares V to 100000. If $V > 100000$, control is transferred to statement 240 and the loop ends. If V is not greater than 100000, statements 110 through 236 are executed in the normal sequence. Statement 235 increments V by 25000 and statement 236 transfers control back to statement 104. Statement 104 compares the new value of V to 100000 to determine whether or not to go through the loop again. Looping continues until the loop is executed with V = 100000.

For each value of V, T (total cost), R1 (total revenue), P (profit/loss) is computed; LOSS or PROFIT is printed depending on the value of V. This, then is the loop.

During the first pass through the loop V = 25000, the second pass V = 50000, the third pass V = 75000 and the fourth pass V = 100000. The printed output from the program shows the break-even point and the profit /loss for the four volume levels.

```
BREAK-EVEN PØINT = 50000 VØLUME UNITS
LØSS = $-500000               VØLUME = 25000 UNITS
LØSS = $ 0     VØLUME = 50000 UNITS
 PRØFIT = $ 500000            VØLUME = 75000 UNITS
 PRØFIT = $ 1000000           VØLUME = 100000 UNITS
```

FOR AND NEXT STATEMENTS

The following sample program shows a loop created by using the FOR statement (line 101) and NEXT statement (line 235):

```
001 REM THIS IS A BREAK-EVEN PRØGRAM
002 REM THE FØLLØWING VARIABLES ARE USED:
003 REM FIXED ANNUAL CØST            F
004 REM VARIABLE CØST PER UNIT       C
005 REM SALES REVENUE PER UNIT       R
006 REM SALES VØLUME                 V
007 REM BREAK-EVEN PØINT (VØLUME)  V1
008 REM TØTAL CØST                   T
009 REM TØTAL REVENUE               R1
010 REM PRØFIT/LØSS                  P
011 REM
012 REM
013 REM ASSIGN VALUES TØ VARIABLES F,C,R
035 DATA 1000000,10,30
037 READ F,C,R
060 REM
070 REM CØMPUTE BREAK-EVEN PØINT
080 LET V1 = F/(R-C)
090 PRINT "BREAK-EVEN PØINT =";V1;"VØLUME UNITS"
100 REM
101 FØR V = 25000 TØ 100000 STEP 25000
110 REM CØMPUTE TØTAL CØST
120 LET T = F + C * V
130 REM
140 REM CØMPUTE TØTAL REVENUE
150 LET R1 = R * V
160 REM
170 REM CØMPUTE PRØFIT / LØSS
180 LET P = R1 - T
200 IF V > V1 THEN 230
210 PRINT "LØSS = $";P,"VØLUME =";V;"UNITS"
220 GØTØ 235
230 PRINT " PRØFIT = $";P,"VØLUME =";V;"UNITS"
235 NEXT V
240 END
```

The FOR statement establishes the first value of V (25000), the final allowable value of V (10000), and the step value (25000). Statements between the FOR statement and NEXT statement are repeatedly executed until V is greater than the final allowable V value. The value of V is incremented by the step value each time the NEXT statement is executed. Output from the program is identical to the output produced when the IF and GOTO statements controlled the loop.

LISTS AND TABLES

For some problems, it is desirable to present data or the solution in the form of a list or table. Such lists and tables are called arrays. An array is an ordered collection of items (data elements) arranged in a multi-dimensional structure. A one-dimensional array, or list, is called a vector and a two-dimensional array, or table, is called a matrix. These terms have been borrowed from mathematical terminology because vectors and matrices in BASIC obey other special properties expected by mathematicians. Arrays with three dimensions may also be used.

Variables are used to name arrays. The individual elements of an array are identified by the use of subscripts and are called subscripted variables. The subscripts, one for each dimension of the array, are position indicators which locate elements within the array. Subscripts are enclosed by parentheses and separated by commas. The first matrix subscript designates a row, and the second designates a column.

Example:

In the following 3 by 4 matrix, A, the element designated by A(2, 3) is underlined.

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & \underline{7} & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix}$$

In the BREAK-EVEN program, where the profit/loss for four different sales volumes was computed, V (volume), P (profit), T (cost), and R1 (revenue) can be organized in array form, each array with 4 elements. For each volume, an associated revenue, cost, and profit is computed.

In the sample program on the following page, the DIM statement was used to specify each array as containing four elements (statements 39, 40, 41, and 42); however, the use of this statement was not required. To specify an array of up to 10 elements only the selected variable name and associated subscripts are required. If the array is to contain more than 10 elements then the DIM statement is required. The advantage of using the DIM in this situation is the conservation of space, because the use of a variable and subscript results in an automatic allocation of space for 10 array elements by BASIC.
For additional information pertaining to the DIM statement, see section 2.

```
001 REM THIS IS A BREAK-EVEN PRØGRAM
002 REM THE FØLLØWING VARIABLES ARE USED:
003 REM FIXED ANNUAL CØST          F
004 REM VARIABLE CØST PER UNIT     C
005 REM SALES REVENUE PER UNIT     R
006 REM SALES VØLUME               V
007 REM BREAK-EVEN PØINT (VØLUME) V1
008 REM TØTAL CØST                 T
009 REM TØTAL REVENUE              R1
010 REM PRØFIT/LØSS                P
011 REM
012 REM
013 REM ASSIGN VALUES TØ VARIABLES F,C,R
035 DATA 1000000,10,30
037 READ F,C,R
038 REM DEFINE ARRAYS FØR VARIABLES V,P,T,R1
039 DIM V(4)
040 DIM P(4)
041 DIM T(4)
042 DIM R1(4)
060 REM
070 REM CØMPUTE BREAK-EVEN PØINT
080 LET V1 = F/(R-C)
090 PRINT "BREAK-EVEN PØINT =";V1;"VØLUME UNITS"
095 REM INITIALIZE ARRAY V,CØMPUTE P,T,R1
096 I = 0
101 FØR J = 1 TØ 4
102 I = I + 25000
103 V(J) = I
130 REM
140 REM CØMPUTE TØTAL REVENUE
141 T(J) = F + C * V(J)
160 REM CØMPUTE TØTAL REVENUE
161 R1(J) = R * V(J)
170 REM CØMPUTE PRØFIT / LØSS
181 P(J) = R1(J) - T(J)
183 NEXT J
201 PRINT " VØLUME",V(1),V(2),V(3),V(4)
202 PRINT " REVENUE",R1(1),R1(2),R1(3),R1(4)
203 PRINT " CØST",T(1),T(2),T(3),T(4)
204 PRINT " PRØFIT",P(1),P(2),P(3),P(4)
240 END
```

The DIM statement (line 039) reserves space for an array named V. The amount of space reserved is determined by the subscript; in this case, 4. This means that array V has four elements. Arrays P, T and R1 are similar.

element 1	element 2	element 3	element 4

ARRAY V

Following is the method used for initializing (placing data into) the array:

```
095 REM INITIALIZE ARRAY V,CØMPUTE P,T,R1
096 I = 0
101 FØR J = 1 TØ 4
102 I = I + 25000
103 V(J) = I
130 REM
```

"I" is used to initialize the volume array, V. Initially I is set to 0 (zero), and is incremented within the FOR loop (line 102) by 25000 for each increment of J. "J" is a variable used as a subscript to address the individual elements of array V. When J is 1, the first element is addressed. The statement at line 103 places the current value of I into the array V at the location identified by the current value of J. J is also used as a subscript for addressing the elements of arrays P, T and R1.

```
140 REM CØMPUTE TØTAL REVENUE
141 T(J) = F + C * V(J)
160 REM CØMPUTE TØTAL REVENUE
161 R1(J) = R * V(J)
170 REM CØMPUTE PRØFIT / LØSS
181 P(J) = R1(J) - T(J)
183 NEXT J
```

After completing the loop between statements 101 and 183, all of the arrays contain the results of the computation. The PRINT statements (line numbers 201, 202, 203 and 204) will print the individual elements of each array.

```
201 PRINT " VØLUME",V(1),V(2),V(3),V(4)
202 PRINT " REVENUE",R1(1),R1(2),R1(3),R1(4)
203 PRINT " CØST",T(1),T(2),T(3),T(4)
204 PRINT " PRØFIT",P(1),P(2),P(3),P(4)
240 END
```

The program output displays the contents of each array as follows:

```
BREAK-EVEN PØINT = 50000 VØLUME UNITS
 VØLUME         25000          50000          75000          100000
 REVENUE        750000         1500000        2250000        3000000
 CØST           1250000        1500000        1750000        2000000
 PRØFIT        -500000         0              500000         1000000
```

TERMINAL INPUT AND OUTPUT

Sometimes it is desirable to enter data while a program is in execution. For example, let us assume that the break-even program has been generalized to permit several different products with different fixed costs, variable costs and revenue per unit. The program can be modified to request the values for these variables as it executes.

The INPUT statement is used in a BASIC program when the user wishes to enter data to be used by the program via the terminal keyboard. When the INPUT statement is executed, the user is queried, via a displayed " ? ", for data. Execution stops until the requested data is entered. Data entered via the keyboard is assigned sequentially to variables listed as INPUT statement arguments (variable).

When more than one item is requested by one INPUT statement, the exact number of items must be entered and items must be separated by commas. If not enough data or too much data is entered, diagnostics are issued by BASIC. The user must take the specified action before execution can resume.

Example:

```
          001 REM THIS IS A BREAK-EVEN PRØGRAM
          002 REM THE FØLLØWING VARIABLES ARE USED:
          003 REM FIXED ANNUAL CØST           F
          004 REM VARIABLE CØST PER UNIT      C
          005 REM SALES REVENUE PER UNIT      R
          006 REM SALES VØLUME                V
          007 REM BREAK-EVEN PØINT(VØLUME)    V1
          008 REM TØTAL CØST                  T
          009 REM TØTAL REVENUE               R1
          010 REM PRØFIT/LØSS                 P
          011 REM
          012 REM
          013 REM ASSIGNS VALUES TØ VARIABLES F,C,R
New     } 015 PRINT "INPUT: FIXED CØSTS   VARIABLE CØSTS   REVENUE PER UNIT"
Inserts } 036 INPUT F,C,R
          039 DIM V(4)
          040 DIM P(4)
          041 DIM T(4)
          042 DIM R(4)
          060 REM
          070 REM CØMPUTE BREAK-EVEN PØINT
          080 LET V1=F/(R-C)
          090 PRINT "BREAK-EVEN PØINT=";V1;"VØLUME UNITS"
          095 REM INITIALIZE ARRAY V,CØMPUTE P,T,R1
          096 I=0
          101 FØR J=1TØ 4
          102 I=I+25000
          103 V(J)=I
          110 REM
          140 REM CØMPUTE TØTAL CØST
          141 T(J)=F+C*V(J)
          160 REM CØMPUTE TØTAL REVENUE
          161 R1(J)=R*V(J)
          170 REM CØMPUTE PRØFIT/LØSS
          181 P(J)=R1(J)-T(J)
          183 NEXT J
          201 PRINT "VØLUME",V(1),V(2),V(3),V(4)
          202 PRINT  "REVENUE",R1(1),R1(2),R1(3),R1(4)
          203 PRINT "CØST",T(1),T(2),T(3),T(4)
          204 PRINT "PRØFIT",P(1),P(2),P(3),P(4)
          240 END
```

The statement at line number 15 prints a message on the terminal indicating the values and their sequence to be typed in when the INPUT statement (36) makes the request.

```
013 REM ASSIGNS VALUES TØ VARIABLES F,C,R
015 PRINT "INPUT: FIXED CØSTS   VARIABLE CØSTS   REVENUE PER UNIT"
036 INPUT F,C,R
```

The output of the PRINT statement (15) is shown below in addition to the result of the INPUT statement (36); the question mark. Note that only two values were entered and that the " NOT ENOUGH DATA" diagnostic was issued. The additional value was then entered.

```
INPUT: FIXED CØSTS   VARIABLE CØSTS   REVENUE PER UNIT
? 1000000,10
 NØT ENØUGH DATA, TYPE IN MØRE    AT  36
? 30
```

Below is the program output. Revenue, cost, and profit were computed based on the data entered at the terminal.

```
BREAK-EVEN PØINT= 50000 VØLUME UNITS
VØLUME          25000         50000         75000         100000
REVENUE         750000        1500000       2250000       3000000
CØST            1250000       1500000       1750000       2000000
PRØFIT         -500000        0             500000        1000000
```

Refer to section 2.4 and the appendix for more information pertaining to INPUT/OUTPUT.

USING BASIC

The previous paragraphs have described BASIC statements and how to organize these statements into a BASIC program. The following describes how to enter a program into a computer and how to execute that program.

BASIC is primarily a terminal oriented language; however, programs in card deck form can be entered and executed (batch mode). The following paragraphs describe the method for entering and executing BASIC programs interactively through use of a Teletype (TTY) or Cathode Ray Tube (CRT) terminal. For a description of BASIC program card deck structures and batch mode operations, see section 2.9.

When operating interactively, the user must write the program in a file, as shown in the examples which follow, and execute from the file. To correct a syntax, semantic, or logic error, the user need only enter the line number which contains the error followed by the corrected code. When the corrected line is entered and the terminal RETURN key is pressed, the existing program statement is replaced. To delete a line, enter the line number and press the RETURN key.

BASIC can be run interactively under anyone of these operating systems: KRONOS and NOS, the usage of which is described in the following paragraphs, and SCOPE (INTERCOM) which is described at the end of this section.

USING BASIC FROM KRONOS/NOS

KRONOS/NOS provides multi-user access to one large-scale computer. BASIC programs can be submitted from a remote time-sharing terminal. To access a central computer from the terminal, the user must link up with the computer. The method used to establish connection between the terminal and the central site computer varies depending on the type of terminal equipment and the connection provided by the telephone company.

When the connection is completed, the computer responds:

```
yy/mm/dd        hh.mm.ss
_____________   TIME SHARING SYSTEM*
```

and requests:

```
USER NUMBER:
```

* The particular message typed at your installation may be different.

When this occurs, perform the following:

Step 1: Submit the user number on the same line:

```
    xxxxxxx (CR) ‡
```

The user number consists of up to seven alphanumeric characters. The system then requests:

```
    PASSWORD
    ■■■■■■■■
```

Step 2: Enter the password:

```
    zzzzzzz (CR)
```

The password must consist of one to seven alphanumeric characters. To provide a greater measure of security, type the password in the area the system has blacked out. If a password is not needed, type:

```
    (CR)
```

If the user number and password are not acceptable, the system types:

```
    ILLEGAL LOGIN, TRY AGAIN.
    USER NUMBER:
```

If the user number and password are acceptable, the system responds:

```
    TERMINAL:   nnn, TTY (where nnn is the particular terminal no.
                          being used).
    RECOVER/SYSTEM:
```

Step 3: Enter the desired subsystem on the same line:

```
    BASIC (CR)
```

Because all interactive programs run under KRONOS/NOS reside in a file, the system queries the user as to the applicable file type by responding:

```
    OLD, NEW, OR LIB FILE:
```

‡ Throughout this section, the convention (CR), Carriage Return, is used to denote the RETURN key on the keyboard.

Step 4: Submit the appropriate status:

OLD $\textcircled{CR}$	for a file that was previously created and saved.
NEW $\textcircled{CR}$	for a new file.
LIB $\textcircled{CR}$	for a file from the system library.

The system responds:

```
FILE NAME:
```

Step 5: Enter the file name:

nnnnnnn $\textcircled{CR}$

The file name consists of up to seven alphanumeric characters. If an OLD or LIB file does not exist, the system responds:

```
FILE NOT FOUND.
```

If the file name entered contains illegal characters, the system responds:

```
FILE NAME ERROR.
```

After the system finds the specified file, it responds:

```
READY.
```

The following example illustrates a sample log-in:

```
74/07/19.  13.19.28.
TIME SHARING SYSTEM
USER NUMBER: ABCDEFG
PASSWORD
[T|U|V|W|X|Y|Z]■■ ———→ The password is blocked out
TERMINAL:60,TTY            and cannot be seen by the user.
RECOVER/SYSTEM:BASIC       It is shown for purposes of
OLD,NEW,OR LIB FILE:NEW, EX4  illustration only.

READY.
```

Step 6: Enter the new BASIC program. Each line must begin with a 1-5 digit line number and end with (CR) . BASIC statements need not be typed in correct order, because KRONOS and NOS automatically sequence them according to line number.

Step 7: To execute the program, type:

```
RUN (CR)     or     RNH (CR)
```

This command initiates compilation and execution of the BASIC program.

The output of a BASIC program is in the form:

```
yy/mm/dd     hh.mm.ss

PROGRAM      nnnnnnn
      .
      .
(data printed by the program - error messages, if program
errors occurred)
      .
      .
RUN COMPLETE.
```

Step 8: When a run is completed, the user has the following options:

- Continue processing - build and execute new programs; modify existing program and rerun; rerun the same program.

  or

- Terminate the terminal session via the following commands:

  ```
  BYE (CR)
      or
  GOODBYE (CR)
  ```

  Either command releases all local files and prints the following:

  ```
  xxxxxxx  LOG  OFF      hh.mm.ss.
  xxxxxxx  CPU           s.sss  SEC.
  ```

  where:

xxxxxxx	user number.
s.sss	number of seconds of central processor time used. (This is not the amount of time used between LOGIN and BYE.)

Sample Terminal Session

The following example was run at a TTY terminal. User responses are underlined. The user must press the carriage return key CR after typing in each response.

```
 74/09/09. 12.06.37.
KRØNØS TIME SHARING SYSTEM - VER. 2.1-03/AA.        log-in procedure-type
USER NUMBER: 412                                    user number and password.
PASSWØRD
[redacted]
TERMINAL:     73,TTY
RECØVER /SYSTEM:BASIC                               Requests BASIC subsystem.
ØLD, NEW, ØR LIB FILE:  NEW,EX4                     Program is from NEW file.
READY.

10 PRINT "TYPE A NUMBER"
20 INPUT X                                          Program statements-
30 LET F=1                                          consist of a line number
40 FØR I=1TØX                                       followed by a space,
50 LET F=F*I                                        followed by the appropriate
60 PRINT "FACTØRIAL ";X;"IS ";F                     statement.
70 GØTØ 10
80 END


RUN                                                 Compile and execute program.

 74/09/09. 12.13.26.
PRØGRAM   EX4

 FØR WITHØUT NEXT  AT  40                           BASIC issues diagnostic.

CP        0.023 SECS.

RUN CØMPLETE.


55 NEXT I                                           Adding statements to correct
25 IF X=0 THEN 80                                   program.
RUN
 74/09/09. 12.14.40.                                Compile and execute again.
PRØGRAM   EX4

TYPE A NUMBER
? 3                                                 User input 3 as value for X.
FACTØRIAL  3 IS  6
TYPE A NUMBER
? 0                                                 X = 0 and program terminates
                                                    at line 80.

CP        0.036 SECS.

RUN CØMPLETE.
```

```
LIST

 74/09/09. 12.15.41.
PRØGRAM   EX4

10 PRINT "TYPE A NUMBER"
20 INPUT X
25 IF X=0 THEN 80
30 LET F=1
40 FØR I=1TØX
50 LET F=F*I
55 NEXT I
60 PRINT "FACTØRIAL ";X;"IS ";F
70 GØTØ 10
80 END
READY.

SAVE,EX4
READY.
```

KRONOS/NOS command to list program.

Program listing.

KRONOS/NOS command saves program with file name EX4 for later use.

For a detailed description of the KRONOS/NOS commands used is this example and other available commands, see the KRONOS /NOS Reference Manual.

In this example, the user saved the program as a file name EX4. The program in this file is now stored as an indirect access permanent file which can later be accessed by use of the OLD command, e.g.

```
ØLD,EX4
READY.

LIST

 74/09/09. 12.17.03.
PRØGRAM   EX4

10 PRINT "TYPE A NUMBER"
20 INPUT X
25 IF X=0 THEN 80
30 LET F=1
40 FØR I=1TØX
50 LET F=F*I
55 NEXT I
60 PRINT "FACTØRIAL ";X;"IS ";F
70 GØTØ 10
80 END
READY.
```

makes a copy of the file accessable to the user.
requests a list of the file contents.

At this time, the user can add, delete, or change program statements, See USING BASIC.

```
RUN
                                                Compiles and executes the
 74/09/09. 12.18.05.                            new program.
PRØGRAM   EX4

TYPE A NUMBER                                   User enters 6 as value for X.
? 6
FACTØRIAL  6 IS  720
TYPE A NUMBER
? 0


CP        0.037 SECS.

RUN CØMPLETE.

BYE                                             Logs off.

LN76      LØG ØFF.  12.18.37.
LN76      CP        0.096 SEC.
```

If the user wishes to store the changed program, this can be accomplished by the REPLACE command which replaces the old program with the corrected program.

```
REPLACE, EX4                                    Stores the updated program
                                                in file EX4. If the user
                                                logs off before replacing
                                                EX4, the corrected
                                                version is lost but the old
                                                version of EX4 remains in
                                                tact.
```

USING BASIC UNDER SCOPE

To access a central computer from a terminal, the user must link up with the computer system. The method of establishing the connection between the terminal and the central site computer varies depending on the type of terminal equipment and the connection provided by the telephone company. When connected to the terminal, the system responds:

```
CONTROL DATA INTERCOM  4.1
DATE              mm/dd/yy
TIME              hh.mm.ss
PLEASE LOGIN
```

Step 1: The terminal user logs into the system by entering:

```
LOGIN
```

The system responds:

```
ENTER USER NAME-
```

Step 2: Enter your user name. The user name may be any combination of up to ten letters or digits and must <u>not</u> be followed by a period.

When the user name has been entered at a TTY terminal, the system responds:

```
■■■■■■■■■ ■ ENTER PASSWORD-
```

At a 200 USER or display terminal, the system responds:

```
ENTER PASSWORD-
```

The user then enters his password. A password is any combination of up to ten letters or digits which must not terminate with a period. On a teletype listing, the password is overprinted on the ten-character, blocked-out line to preserve privacy. The display terminal screen is automatically cleared on acceptance of the entered password to preserve privacy.

When the user name and password are accepted, the user id (a two-character user code) and the time at which the user logged in, followed by the equipment number (multiplexer equipment status table ordinal) and the port number at which he logged in, are displayed at the terminal.

e.g.

```
ENTER USER NAME- USERA
 ■■■■■■■■■■■■ ENTER PASSWORD-

19/07/74      LOGGED IN AT 17.47.26
              WITH USER-ID AB
              EQUIP/PORT   52/03
COMMAND-
```

Step 3: After the user successfully logs-in, the system responds with COMMAND and the user enters the command "EDITOR".

```
COMMAND - EDITOR
```

The user is now in text edit mode.

Step 4: Enter FORMAT, BASIC.
When this command is entered, a format specification is automatically established at the terminal which permits the user to enter lines in BASIC language format.

Step 5: Enter the BASIC program statements-line number followed by BASIC statement.

e.g.10 LET X = 5

Each line must begin with a 1-5 digit line number and end with (CR) .
BASIC statements need not be typed in correct order because the EDITOR automatically sequences them according to line number.

Step 6: Once the entire program is entered, compile and execute the program by typing:

<u>RUN BASIC</u>

After execution is completed the output is printed or if a program error occurs, the appropriate error message is displayed.

Step 7: When the run is completed, the user can select one of the following options:

- Continue processing - build and execute new programs; modify existing programs and rerun; or rerun the same program.
- Terminate the terminal session by saving the program and entering the BYE and LOGOUT commands or by entering the BYE BYE and LOGOUT commands. When the BYE or BYE BYE commands are entered, user returns to INTERCOM mode from EDITOR mode.

When in INTERCOM mode, the system responds with:

COMMAND

At this time the user enters the LOGOUT command which releases local files which the user may have created under EDITOR. Only permanent files are retained between the time of a LOGOUT and any subsequent LOGIN. The user is disassociated from INTERCOM until a subsequent LOGIN command is entered. INTERCOM displays the date and time the user is logged-out. The LOGOUT command is not allowed when the user is under control of EDITOR.

Example:

```
COMMAND- L0G0UT
CP TIME       9.458
PP TIME       8.181
C0NNECT TIME     0 HRS.  8 MIN.
 19/06/73  L0GGED 0UT AT 08.31.23.[
```

The order of the date (month, day, year) may be changed as an installation option. The time of LOGOUT is given in hours, minutes, seconds (24-hour clock); CP/PP time is given in seconds. The user should disconnect his terminal from INTERCOM by turning it off, or by hanging up the data set receiver.

Sample Terminal Session

After the user has logged in, he can create and execute BASIC programs. The following sample BASIC program, run under the INTERCOM system, illustrates how to run a BASIC program. The program was entered at a TTY terminal. User responses are underlined. After typing the response, user must depress the carriage return key (CR) .

```
CØMMAND- EDITØR
..FØRMAT,BASIC
..10 PRINT "TYPE A NUMBER";
20 INPUT X
30 LET F=1
40 FØR I=1TØ X
50 F=F*1
60 PRINT "FACTØRIAL ";X;"IS ";F
70 PRINT
80 GØTØ 10
90 END
```

User calls EDITOR.
User requests BASIC format specifications and following EDITOR command mode response, enters a BASIC program line by line.

```
RUN,BASIC

 BASIC CØMPILATIØN ERRØRS
FØR WITHØUT NEXT  AT   40

..
```

Compiles and executes BASIC program.
BASIC issues diagnostic messages.

```
 55 NEXT I
 25 IF X=0 THEN 80
 RUN,BASIC
```

Statement 55 is added to satisfy looping requirements.
Statement 25 is added to provide an exit from the program.
User calls BASIC compiler-again requests compile and execution of the BASIC program

```
TYPE A NUMBER ?3
FACTØRIAL  3 IS  6
TYPE A NUMBER ?0
..LIST,ALL
```

Zero causes exit from execution and return to EDITOR command mode.
User requests listing of his program in the edit file.

```
10=10 PRINT "TYPE A NUMBER";
20=20 INPUT X
30=30
25=25 IF X=0 THEN 80
30=30 LET F=1
40=40 FØR I=1TØ X
50=50 F=F*1
55=55 NEXT I
60=60 PRINT "FACTØRIAL ";X;"IS ";F
70=70 GØTØ 10
80=80 END
```

```
..SAVE,BASPRØG                  User requests contents of
                                edit file be saved as a local
                                file named BASPROG until
                                LOGOUT.

..STØRE,BASPRØG                 Store BASPROG as a perma-
                                nent file.

..BYE                           User requests a return to
                                INTERCOM command mode.
COMMAND-
```

By saving the file (BASPROG) the user reserves it for later use during the terminal session (before logging out). To permanently save the file, it must be stored as a permanent file (STORE, BASPROG). To retrieve and execute this program later the following command sequence must follow the user login sequence.

```
CØMMAND- EDITØR
..FETCH,BASPRØG                 Retrieve file made permanent earlier tells
..EDIT,BASPRØG                  EDITOR that BASPROG is to be edit file.
..RUN,BASIC                     Compile and execute program.

TYPE A NUMBER ?6
FACTØRIAL  6 IS  720
TYPE A NUMBER ?0
```

For a detailed description of INTERCOM and EDITOR commands used in this example and other available commands, see section 2.8 or the INTERCOM Reference Manual,

BASIC LANGUAGE REFERENCE 2.0

INTRODUCTION

MODES OF OPERATION

Although BASIC is normally used interactively from a remote terminal under an operating system, BASIC programs can also be compiled and executed as batch programs.

CHARACTER SET

Table 2-1 lists the BASIC character set.

TABLE 2-1. BASIC CHARACTER SET

A - Z	0 - 9
+	△ Blank
-	,
*	.
/	"
(	↑
)	<
$	>
=	?
:	;
	#

In addition to these characters, any character available to the operating system can be used in data and string constants. See appendix A for a description of all available characters.

STATEMENT STRUCTURE

All BASIC statements have the following common characteristics:

1. Each statement begins with a line number.
2. Each statement must be completed on a single line. No line continuation is allowed.
3. Generally, blanks within a BASIC statement have no meaning. For example, the following two statements are equivalent:

   ```
   10 LET A = B + C
   10LETA=B+C
   ```

   The exceptions to this rule are discussed in the applicable areas of the sections which follow.
4. A BASIC statement can include a maximum of 72 characters.
5. On input, a statement on a punched card terminates at column 72; a statement from a terminal terminates when the carriage return key is pressed. No more than 72 characters per statement are translated by the BASIC compiler.

ELEMENTS OF THE BASIC LANGUAGE 2.1

BASIC is designed to manipulate numeric and character string data. Numeric data includes integer, decimal, and exponential constants. String data includes alphanumeric text with or without quotation marks. The following paragraphs describe those elements which comprise numeric and character-string data. Also described are variables, operators and expressions.

CONSTANTS

A constant is a fixed unchanging value. In BASIC, there are numeric and string constants.

NUMERIC CONSTANTS

In BASIC, there are three types of numeric constants:

- Integer
- Decimal
- Exponential

Although each has specific rules governing its use, some rules apply to all.

1. A comma cannot be used to delimit thousands, millions, etc.
2. When a numeric constant is not signed explicitly by a negative or positive sign, the constant is assumed positive.
3. Any number of digits can appear in a numeric constant; a maximum of 14 digits accuracy is used in computation.
4. Whether integer, decimal, or exponential, the absolute value of a constant must be in the range 3.13152×10^{-294} to 1.26501×10^{322}.

Integer Constants

An integer constant is a whole number written without a decimal point.

Examples:

```
-49           25000
+123456789    0
```

Decimal Constants

A decimal constant is any whole number, fraction, or mixed number written with a decimal point. Leading zeros to the left of the decimal point and trailing zeros to the right of the decimal point are ignored; the decimal point can appear anywhere in the number.

Examples:	-4.08	1.91632614	.0000001
	50.5	147.2	+3025.098

Exponential Constants

The representation of extremely large and small numbers is simplified using exponential constants. For example, to write ten billion, in its full form requires 11 digits (10000000000); however, ten billion can also be represented by:

$$1.0 \times 10^{10}$$

In BASIC, this exponential form is expressed by:

```
1.0E10
```

The E indicates, "times ten to the power . . .".

Similarly, a small number such as .00000000923 can be represented by:

$$9.23 \times 10^{-9}$$

In BASIC, this notation can be expressed by:

```
9.23E-9
```

To use exponential constants in a BASIC program, the following rules must be observed:

1. A number must precede the E. The number preceding the E can be any valid integer or decimal constant.
2. The exponent (number that follows the E) can consist of one-to-three digits and a positive or negative sign. If a sign is absent, a positive sign is assumed.
3. Decimal points are not permitted in the exponent.

Examples:

```
-2.517E130
7E+20
4.91872634E-18
```

STRING CONSTANTS

A STRING is a collection of alphanumeric text. In BASIC, this alphanumeric text is normally enclosed in quotation marks to set it off from the rest of the program. This is called " quoted text". Unquoted strings are also permitted; but, can only be used in DATA statements, IMAGE statements or as input data.

Rules:

1. A string enclosed in quotes consists of all characters between quotes.
2. The length of string is 0-78 characters; a zero length or null string is represented by a pair of quotes ("").
3. Valid characters for quoted text are any available character except the quotation mark; i.e., characters are not restricted to the BASIC character set.
4. A blank is a significant character when used in a string.
5. Unquoted string constants must not begin with a digit, plus (+), minus (-), comma (,), period (.), blank, or quote.

Example:

```
" PART 25"
"THIS IS A TEST"
```

The quotation marks are not part of the string constant. For an example of unquoted strings, see section 2.4 (IMAGE statement).

VARIABLES

Variables represent values which are not fixed. Values can be assigned to variables and later changed by other statements or conditions during execution of the BASIC program. Variables can represent numeric or string data. In BASIC, a numeric variable can have an integer, decimal or an exponential value.

SIMPLE NUMERIC VARIABLES

A simple numeric variable represents a numeric value that may change during program execution. Simple numeric variables are named by a single alphabetic character and an optional numeric character. Variable names must not exceed two characters in length.

Examples of simple numeric variables are:

```
A  Z3  C9  E
```

Examples of invalid numeric variable identifiers are:

B23   49   G*   AA

The following rules apply to numeric variables:

1. Numeric variables represent numeric data only.
2. Numeric variables are preset to zero before the program executes.
3. The absolute value of a numeric variable must be in the range 3.13152×10^{-294} to 1.26501×10^{322}.
4. If a value smaller than the minimum is assigned, the variable is set to zero.
5. If a value greater then the maximum is assigned, a fatal diagnostic is issued.

SIMPLE STRING VARIABLES

String variables represent alphanumeric text and are named with a two-character identifier. The first character must be alphabetic, and the second character must be a dollar sign ($).

Examples of valid string variables are:

A$ B$ Y$

The value represented by the string variables can consist of a group or string of 0 to 78 characters. Internally each character is represented by a six-bit numeric code (see appendix A, character sets). The characters at the beginning of the alphabet have code values which are less than the characters at the end of the alphabet. For example, if A$ and B$ represent strings " ABC" and " XYZ" respectively, then A$ has a value less than B$.

SUBSCRIPTED VARIABLES

BASIC permits the use of numeric and string arrays; therefore numeric and string subscripted variables are permitted. A subscripted variable locates the value of a particular element of an array (vector or matrix). It is written as a simple variable identifier followed by a maximum of three subscripts enclosed in parentheses.

The identifier portion is the same format as a simple numeric or string variable; i.e., letter or letter digit for numeric variables, and letter $ for string variables. Each subscript can be written as a numeric constant, a simple or subscripted numeric variable, or as an arithmetic expression. The format of arithmetic expressions is discussed later in this section.

Examples of valid subscripted variables are:

A(1) B2(3) A(B2(3) X(1, N+M, A(3)	numeric subscripted variables
B$(4) L$(1, J+3) C$(1, J+3, A(1))	string subscripted variables

In BASIC, array dimensions are normally declared implicitly by using subscripted variables.

Unless an array has been explicitly defined by a DIM statement (section 2. 2), the first subscripted variable that references an element in an array automatically defines the array. Each dimension in the automatically defined array is set to 10. Thus a one-dimensional array has 10 elements; a two-dimensional array has 10 x 10 = 100 elements, and a three-dimensional array has 10 x 10 x 10 = 1000 elements.

Subscripted variables with one subscript refer to elements in one-dimensional arrays; subscripted variables with two subscripts refer to two-dimensional arrays, and with three subscripts to three-dimensional arrays.

The lower limit on subscripts is normally 1. However, this limit can be zero if the BASE 0 statement is used (see section 2. 2). If BASE 0 is in effect, array elements number from 0 and automatically defined arrays contain 11 elements (0-10) if one dimension; 11 x 11 = 121 elements if two dimensions; and 11 x 11 x 11 = 1331 elements if three dimensions.

If a subscript value greater than 10 is required, or if the programmer wishes to save space by dimensioning an array to have an upper boundary of less than 10, arrays can be declared with a DIM statement (section 2. 2).

The following rules apply to all subscripted variables - numeric and string:

1. A maximum of three subscripts is allowed.
2. A subscript may be any arithmetic expression. The subscript used is the value of the expression truncated to an integer.
3. A subscript must never be less than zero; subscript may be zero only if BASE is set to 0 (section 2. 2).
4. The normal range of subscripts is from 1 to 10. If a larger subscript is required, the variable name must appear in the DIM statement.
5. Simple and subscripted variables with the same name may be used in the same program. For example, T, T(1, 2), T$, and T$(1, 3) can all be used in the same program and remain distinct.
6. Once an array is referred to in a BASIC program, the number of array subscripts can not be changed. For example, T(5) and T(2, 3) cannot be used in the same program.

FORMING EXPRESSIONS IN BASIC

An expression is usually formed from a series of operands and operations; however, an operand such as a constant or variable can be considered an expression. In BASIC there are three types of expressions: arithmetic, relational and string. The value of an arithmetic expression is numeric; the value of a relational expression is either true or false; and the value of a string expression is a string of characters.

ARITHMETIC EXPRESSIONS

Arithmetic expressions consist of a series of operands and operators. Operators may be any arithmetic operator listed in table 3-1; operands may be any numeric constant, simple or subscripted variable, function reference (section 2. 3), or any expression enclosed in parentheses. See section 2. 3 for a list of available mathematical functions.

Rules for Writing Arithmetic Expressions

In the formation of arithmetic expressions, certain rules must be followed:

1. Only numeric operands and numeric operators can be used.
2. Two arithmetic operators cannot appear side by side (e. g., X++Y is not allowed). If a minus sign is used to indicate a negative value in an expression,then parentheses must be used to separate the negative sign and associated operand from the remainder of the expression, e.g.,

Correct	Incorrect
A*(-B)	A*-B

3. Operators cannot be implied, e.g., (X+1) (Y+2) is not allowed. The correct form is (X+1) * (Y+2).

The following are examples of valid arithmetic expressions:

```
A+B*C/D↑E
A1(3,I+4)↑2.6-G3/Z
A+B**C
(A(I,J,K)+3.95)*(G3+B↑(C2+3))
A+SIN(X)+COS(Y)
-3.14*R↑2
C*(-G2(1,3)-D4)
A8(3)↑(-49)+B(2,-X+15)
```

Arithmetic Expression Evaluation

The rules for the evaluation of arithmetic expressions follow:

1. Expressions within parentheses are evaluated first.
2. Operations of higher precedence are performed before those of lower precedence. This precedence (hierarchy) is provided in table 3-1 from highest (1) to lowest (4).
3. Operations of equal priority or precedence are performed in order from left to right.

TABLE 3-1. ARITHMETIC EXPRESSION OPERATOR HIERARCHY

Hierarchy	Operator	Definition
1.	↑ or **	Exponentiation
2.	*, /	Multiplication, division
3.	+, -	Unary +, -
4.	+, -	Addition, subtraction

Examples of expression evaluation are:

Expressions	Evaluation steps
A+B*C/D↑E	1. D↑E = a
	2. B*C = b
	3. b/a = c
	4. A+c = d (final value)
A+(B-C)*3	1. B-C = a
	2. a*3 = b
	3. A+b = c (final value)
-2↑2	1. 2↑2 = a
	2. -a = -4 (final value)
(-2)↑2	1. -2 = a
	2. a↑2 = 4 (final value)

STRING EXPRESSIONS

Because there are no string operators available,string expressions are limited to single string operands which may be:

1. a string constant;
2. a simple or subscripted string variable;
3. a string function reference (see section 2.3 for a list of available string functions.)

There are no operators in a string expression.

Examples:
```
" THIS IS A STRING CONSTANT"
A$
B$(1,3)
CLK$
```

RELATIONAL EXPRESSIONS

A relational expression compares two numeric values or two string values.

A relational expression is made up of two numeric expressions or two string expressions separated by a relational operator. Operators used to form a relational expression are shown in Table 3.2.

TABLE 3.2. RELATIONAL EXPRESSION OPERATORS

Operator	Definition	Operator	Definition
=	equal to	<	less than
<> or ><	not equal to	> = or = >	greater than or equal to
>	greater than	< = or = <	less than or equal to

Rules for Writing Relational Expressions

1. Comparison of a string to numeric expressions is not allowed.
2. Only one relational operator is allowed in an expression.
3. Relational expressions can be used only in IF statements (section 2.2).

Rules for Evaluation Relational Expressions

- Numeric Relational Expressions - The two arithmetic expressions are evaluated and then their resultant values are compared algebraically. For example: The two expressions are evaluated and then compared as specified by the operator to yield a "true" or "false" value.

If A = 2 and B = 3, the expressions below are evaluated as shown:

Relational Expression	Value
A = B	False
A><B	True
A > B	False
A < B	True
A > = B	False
A < = B	True
A*A+3<B*2	False

- String Relation Expressions - The following rules apply:

1. Strings are compared character-by-character in left-to-right order. BASIC compares characters according to their numeric codes; see Appendix A (Ordering Sequence) for the character set.

   e.g., A has a numeric code of "01"
   B has a numeric code of "02"

   Therefore, A is less than B.

2. Strings are equal only if they have exactly the same length and contain the same characters (including blanks) in the same order. <u>Blanks are important.</u>

3. When strings are equal in length, the first pair of corresponding characters that are not equal determines the greater string.

   e.g., "ABXY" is greater than "ABCZ" because the numeric code for X is greater than the numeric code for C.

4. When strings are unequal in length but corresponding characters that can be compared are equal, the longer string is always considered greater.

   e.g., "ABX   " is greater than "ABX"

5. When strings are unequal in length but one of the corresponding characters that can be compared - scanning from left-to-right - is greater, then the string with the first character of greater value is greater string.

   e.g., " X7" is greater than "X6542"
   " X76" is greater than "X75123"

REMARK (PROGRAM COMMENTS)

REM STATEMENT

The REM statement is used to insert explanatory remarks or comments into a program. REM is a nonexecutable statement and has no effect on program execution.

Format:

REM ch

ch	Any comment or explanation within the 72-character statement length limitation. Comments can be continued on additional REM statements.

Example:

```
110 IF J=>X  THEN 200
115 REM  IF J IS EQUAL TO OR
120 REM  GREATER THAN X, THE PROGRAM
125 REM  WILL JUMP TO LINE 200
130 REM  AND TERMINATE
 .
 .
 .
200 END
```

ARRAY DEFINITION

DIM STATEMENT

The DIM statement establishes the dimensions of an array. Arrays require a DIM statement if a subscript value greater than 10 is needed; or to save space, the programmer may use the DIM statement to dimension an array with an upper subscript limit less than 10. The lower boundary (origin) of a subscript is normally 1; the origin can be 0 if the BASE statement is used as defined later in this section.

Format:

DIM $a_1(nc_1, \ldots, nc_3) \ldots, a_n(nc_1, \ldots, nc_3)$

a_1-a_n	Numeric or string array identifier.
nc_1-nc_3	One to three integers separated by commas representing the maximum value of each subscript.

Example:

To declare a two-dimensional array, A, with subscript boundaries of 1 to 20 and 1 to 5:

```
120 DIM A(20,5)
```

This statement reserves space for array A with 20x5 or 100 elements. The subscripted variable A(3,4) references the fourth element in the third row of array A.

DIM statements may be used anywhere in a program; however, if an array variable is declared more than once in the same program, the last declaration is used for the entire program.

Examples of acceptable DIM statements:

1. `100 DIM X$(5,5), B3(1,2), X1(50)`

   This statement reserves space for:

X$	a two-dimensional string array with 5x5=25 elements.
B3	a two-dimensional numeric array with 2 elements.
X1	a one-dimensional numeric array with 50 elements.

2. `50 DIM G2(5,6,7), A0(9,2), P$(2,3)`

   This statement reserves space for:

G2	a three-dimensional numeric array with 5x6x7=210 elements.
A0	a two-dimensional numeric array with 9x2=18 elements.
P$	a two-dimensional string array with 6 elements.

NOTE: Each element of a numeric array requires one computer word (10, 6-bit characters). Each element of a string array requires eight computer words enough for the maximum 78-character string.

BASE STATEMENT

The BASE statement explicitly defines the origin or base of all arrays within a program; it is an optional statement and need not be included in each program. There can be only one BASE statement in a program, and it must precede any DIM statement or any reference to an array. If BASE is not specified, array origin is assumed at 1 (i.e., elements are numbered from 1).

Format:

BASE x

x	0 or 1

Example:

```
100 BASE 0
110 DIM A(3,4,3), B(2,13)
```

When compiled, the BASE statement at line 100 specifies that all arrays are origined at 0 and that subscripts of zero can therefore be used. Because the base has been established at 0, the DIM statement defines array A as a 4x5x4=80 element array and B as a 3x14=42 element array.

VALUE ASSIGNMENT

LET STATEMENT

The LET statement assigns a value to a variable during execution of the BASIC program.

Formats:

LET $v_1 = e$	or	LET $v_1 = v_2 = v_3 \ldots = v_n = e$
$v_1 = e$		$v_1 = v_2 = v_3 \ldots = v_n = e$

e	expression of any complexity
$v_1 - v_n$	can be a numeric, string, simple, or subscripted variable.
NOTE:	String values must be assigned to string variables and numeric values must be assigned to numeric variables.

Example:

1.
10 LET X = 2	Assigns value of 2 to variable X.
20 LET Y$ = "AB"	Assigns the string "AB" expression to Y$.
30 LET Z = F*C+V	Assigns the result of the expression F*C+V to Z.
40 LET A(3) = Y*Z	Assigns the result from expression X*Z to the to the third element of A.

When a LET statement consists of a string of equalities, each variable is assigned the value of the expression. Subscript expressions are evaluated prior to the assignment of the value. All expressions are evaluated according to the rules of operator precedence in section 2.1.

2.

```
                10 LET I = 1
                20 LET Z(I) = I = I+1
```

is equivalent to

```
                10 I = 1
                20 Z(I) = I + 1
                30 I = I + 1
```

TEST AND BRANCH

IF STATEMENT

The IF statement tests conditions, and controls the sequence of operations.

Format:

IF r THEN ln

r	Relational expression
ln	Line number

If the relational expression is evaluated as TRUE, program control transfers to statement ln; if FALSE, the next sequential statement is executed.

Example:

```
        where: I = 8 and J = 4
               120 IF 2*I = >J↑2-1 THEN 165
```

The value 16 is compared to the value 15; because the evaluation result is TRUE, the next statement executed is at line number 165.

GOTO STATEMENT

The simple GOTO statement unconditionally (always) transfers control from one point in a program to another and interrupts the normal sequence of instructions.

Format:

```
    GOTO ln
     ln              Line number
```

GOTO specifies that the statement at the referenced line number is to be executed. Normal sequential execution continues from that point. If a GOTO statement references a non-executable statement such as a DIM statement, execution continues from the first executable statement directly following the referenced non-executable statement. For an example of this statement, see the example in the following paragraph.

ON GOTO STATEMENT

Format:

ON ne GOTO ln_1, ln_2, ln_3, ..., ln_n

ne	Arithmetic expression
ln_1-ln_n	Line numbers

The ON GOTO statement provides for conditional branching depending on the value of an expression. The expression is evaluated and truncated to an integer value and control is transferred to ln_1 if ne = 1, ln_2 if ne = 2, etc. If the value of the expression is negative, zero, or greater than the number of line numbers specified, an execution diagnostic is issued. "ON EXPRESSION OUT OF RANGE AT .. ".

Example:

```
•
•
95 ON SGN(A)+2 GOTO 100, 110, 120
100 PRINT "A IS NEGATIVE"
105 GOTO 130
110 PRINT "A IS ZERO"
115 GOTO 130
120 PRINT "A IS POSITIVE"
130 LET B=A+1
•
•
```

In the example, SGN(A) could have the value = -1, 0, or 1 (section 2.3). The expression SGN(A)+2 could have the value 1, 2, or 3 and control would transfer to statements 100, 110 or 120, respectively. If, for example, A has the value 2.5, then SGN(A)+2 has the value 3 and the order of statement execution is 95, 120, 130, etc.

LOOPING

FOR...NEXT STATEMENTS

The FOR and NEXT statements provide for efficient looping within a program; the FOR statement must appear as the First statement of the loop, and the NEXT statement, must be the Last statement of the loop.

Format:

FOR snv = ne_1 TO ne_2 STEP ne_3 or
FOR snv = ne_1 TO ne_2

NEXT snv

snv	Simple numeric variable (called the control variable; it must be identical in both statements).
ne_1	Any arithmetic expression (called the initial value).
ne_2	Any arithmetic expression (called the final value).
ne_3	Any arithmetic expression (called the step value).

If STEP ne_3 is omitted, BASIC assumes a step value of +1. The following are equivalent statements:

```
10 FOR J = 1  TO 15 STEP 1
10 FOR J = 1  TO 15
```

When the FOR statement is executed, the expressions are evaluated and their values are saved as initial, step, and final values of the loop. The control variable is assigned the initial value and if it does not surpass the final value the statements between the FOR and NEXT statements are executed. When the NEXT statement is encountered, the value of the control variable is adjusted by the step value. A comparison is made between the value of the adjusted control variable and the specified final value; if the control value has not surpassed the final value, looping continues at the statement following the FOR.
If it has, the loop is complete and execution continues with the statement following NEXT. The statements between the FOR and NEXT statements are never executed if the initial value is beyond the final value.

Example:

```
10 FØR X = 1 TØ 11 STEP 2
15 PRINT X
20 NEXT X
30 END
```

This program produces:

```
1
3
5
7
9
11
```

The successive values of X(1, 3, 5, 7, 9 and 11) are output by the PRINT statement as a result of this routine. Statements 15 through 20 are repeated six times; once for each value assigned to X.

The initial, final, and step expressions are evaluated only once, i.e., on entrance into the loop. These values do not change during execution of the loop, even if the program changes the value of the variables within the expressions.

The value of the control variable may be changed by statements within the loop; its latest value is always adjusted by the step value and used in comparison to the final value.

Example:

```
10 FØR X = 1 TØ 10
20 LET X = X + 1          (increments the control variable X)
30 PRINT X
40 NEXT X
50 END
```

This program produces:

```
2
4
6
8
10
```

The FOR statement specifies that X is incremented by an implicit step value of +1 until it exceeds 10; however, statement 20 also adds 1 to X causing the control variable X to be incremented by 2 for each pass through the loop.

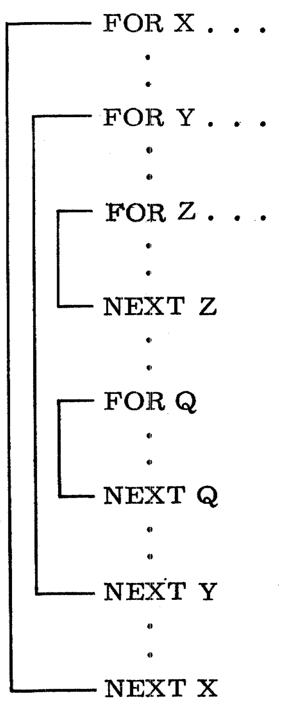
For a positive step value, the initial value must be less than the final value on entrance to the loop. Similarly, for a negative step value, the initial value must be greater than the final value. If not, the loop is not executed and processing continues after the NEXT statement.

The following examples show the effect of the FOR statement on control variables:

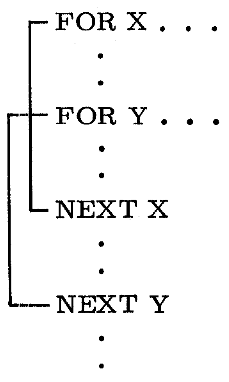
Statement	Values of Control Variables
110 FOR X = -4 TO -2 STEP .5	-4 , -3.5, -3 , -2.5, -2
111 FOR G = 6 TO 3 STEP -1	6, 5, 4, 3
112 FOR X = 5 TO 10 STEP -1	The loop will not be executed; the initial value must be greater than the final value when a negative step value is used.

Loops may be nested (i.e., loops specified within loops) to a maximum depth of 10, but must not intersect each other.

Correct:



Incorrect:



A loop may contain a GOTO or other statements that transfer control outside the range of the loop. In this case, the loop terminates prematurely and the control variable retains its latest value. It is also possible that a GOTO statement may jump to a statement within a FOR loop and, thereby, execute the corresponding NEXT statement before the FOR statement initializes the control variable. This will produce unexpected results. BASIC does not check for this type of situation at compile or execute time.

PROGRAM TERMINATION

STOP STATEMENT

This statement may appear at any point in a program. When it is executed, an immediate exit is made from the program. The STOP statement is equivalent to an unconditional GOTO statement which specifies the line number of the END statement.

Format:

```
    STOP
```

END STATEMENT

This statement signals termination of the BASIC program. Use of the END statement is optional. BASIC assumes END on encountering the last statement. When executed, the program stops and control returns to the operating system. If used, END must be the statement with the highest line number in the program.

Format:

```
    END
```

STRING/NUMBER CONVERSION

CHANGE STATEMENT

One form of the CHANGE statement is used to store a number corresponding to each character of a string into consecutive elements of a numeric array. The first element of the numeric array contains the number of characters in the string.

The second form of the CHANGE statement performs the opposite function. It is used to build a string of characters from numbers stored in consecutive elements of a numeric array. The first element of the array determines the length of the string.

Formats:

1) CHANGE sv TO na

2) CHANGE na TO sv

sv	string variable
na	numeric array variable

When CHANGE is used in format 1, the following occurs:

1. The length of the string is stored in the first element of the array.
2. The numeric code of the first character of the string is stored in the second element of the array; the numeric code of the second character is stored in the third element of the array; etc.
3. If the maximum value, M, allowed for the subscript of the array is less than the number of characters in the string, only the first M-1 characters of the string will be unpacked into separate array elements. The first element of the array will have the value M-1.
4. The original string is unaltered.

When a CHANGE statement in format 2 is executed, the following occurs:

1. If the first element of the array is less than 1, greater than 78, or if it is greater than the dimension of the array, a fatal diagnostic, INVALID LENGTH, is issued.
2. If the first element contains a valid length, then the length of the string is set to this value and the string is built as follows: the character which corresponds to the number stored in the second element of the array becomes the first character of the string; the character which corresponds to the number stored in the third element of the array becomes the second character of the string; etc.
3. If any of the values stored in the numeric array are not valid numeric code, a fatal diagnostic is issued.
4. The numeric array is unaltered.

Appendix A lists all characters and their corresponding numeric codes.

Functions STR$ and VAL can also be used to convert numbers to strings and strings to numbers.

The program:

```
  10 READ M$
  20 CHANGE M$ TØ A       (stores a value 5 as the 1st element of array A.)
  30 A(1) = 3             (changes the value of the 1st element to 3.)
  40 CHANGE A TØ M$       (rebuilds the string with 3 characters "MAR")
  50 PRINT "SHØRTENED STRING =";M$
  60 DATA MARCH
  99 END
```

produces:

```
  SHØRTENED STRING =MAR
```

The program:

```
  10 B(1) =1
  20 B(2) = 48
  30 CHANGE B TØ Q$       (build string Q$ with one character #)
  40 PRINT "WE HAVE STØRED IN Q$ THE CHARACTER:";Q$
  50 END
```

produces:

```
  WE HAVE STØRED IN Q$ THE CHARACTER:#
```

If arrays are origined at 0 by using the BASE 0 statement, the first element of an array is referenced with a subscript of zero.

FUNCTIONS AND SUBROUTINES 2.3

A function is a named procedure or routine that may be used repeatedly from within a program by referencing its name. Each time a function is referenced it executes and returns a single value. A function may be referenced in any expression whenever a variable or numeric constant can be used. BASIC provides a number of predefined functions and permits the user to define his own functions. User functions are defined through use of the DEF statement which is described later in this section.

A subroutine consists of a group of BASIC statements referenced by its first line number and which returns program control to the line following the subroutine reference. Subroutines can be referenced any number of times from any point in the program. In BASIC, subroutines are referenced by the GOSUB statement and the transfer of control from the subroutine to the main program is effected by the RETURN statement. The GOSUB and RETURN statements are described at the end of this section.

PREDEFINED FUNCTIONS

BASIC provides three classes of predefined functions: Mathematical Functions, System Functions, and String Functions.

MATHEMATICAL FUNCTIONS

The following are standard MATHEMATICAL functions which can be evaluated by the BASIC program. In these functions, (x) can be an expression of any complexity and may include other function references. The quantity (x) is an argument (parameter) of the function.

Function	Description
ABS(x)	Finds the absolute value of x.
ATN(x)	Finds the arctangent of x in the principal value range $-\pi/2$ to $+\pi/2$.
COS(x)	Finds the cosine of x; the angle x is expressed in radians.
EXP(x)	Finds the value of e to the power of x.
INT(x)	Finds the largest integer not greater than x. Example: INT(5.95) = 5 and INT(-5.95) = -6.
LGT(x)	Finds the base 10 logarithm of x; $x>0$, otherwise an execution error causes program termination.
LOG(x)	Finds the natural logarithm of x; $x> 0$, otherwise an execution error causes program termination.
RND(x)	See description on next page.
ROF(x) or ROF(x, nc)	Finds the value of x rounded to nc decimal places. If nc is omitted, then x is rounded to the nearest integer.
SGN(x)	Interrogates the sign of x and returns a value of 1 if x is positive; 0 if x is 0; or -1 if x is negative.
SIN(x)	Finds the sine of x; the angle x is expressed in radians.
SQR(x)	Finds the square root of x; $x > 0$, otherwise an execution error causes program termination.
TAN(x)	Finds the tangent of x; the angle x is expressed in radians.

RND Function

The RND function returns a pseudo random number from the set of numbers uniformly distributed over the range $0 \le \text{RND}(x) < 1.0$.

Format:

RND(x)

x	any numeric expression (< 0, 0, > 0)

The value of x affects random number generation as follows:

$x > 0$	A random number sequence is initialized based on the value of x and the first number in the sequence is returned. Each reference to RND with x equal to a particular positive constant value initializes the sequence at the same starting point and returns the same value. Therefore, the same number or the same sequence of numbers can be returned each time RND is referenced and/or each time the program is run if $x > 0$ arguments are used.
$x = 0$	The next number in the established sequence of pseudo random numbers is returned. If the sequence was not previously established by an $x > 0$ RND reference, a standard constant is used to initiate the sequence. The same sequence of random numbers is returned when using RND (0) references each time the program is run, unless the user initializes the same sequence with a different positive (> 0) value each time the program executes. This could be done by using a first reference such as X = RND (CLK (0)).
$x < 0$	The first reference initializes a random number sequence based on the current time of day and returns the first value in that sequence. Subsequent references with $x < 0$, return the next number in the sequence. A program which uses $x < 0$ returns a different value on each reference and a different sequence each time it is run. Although not apparent to the user, the sequence initialized by $x < 0$ is separate from the sequence controlled by $x > 0$ and $x = 0$ references to RND sequences.

Examples:

The following four short examples illustrate the use of each possible value for x (+, -, = 0) and a complete program illustrates the use of the RND(0) option.

1.
```
10 FØR T=1TØ3
20 L=RND(9)
30 E=RND(0)
40 I=RND(0)
50 PRINT L,E,I
60 NEXT T
```

produces:

```
.8125           6.26233E-2      .275991
.8125           6.26233E-2      .275991
.8125           6.26233E-2      .275991
```

RND initialized > 0; the same values are generated during each loop because the sequence is reinitialized by RND(9) each time through the loop.

2.
```
100 FØR I=1TØ3
110 PRINT RND(0),RND(0),RND(0)
120 NEXT I
```

produces:

```
6.78473E-2      .398675         .905878
.178008         .810749         .876414
.679641         .438766         .985444
```

Different values are returned each time through the loop because the sequence is never reinitialized. A standard constant is used to initialize the sequence at the first RND(0) reference.

3.
```
5 FØR A=1TØ3
10 R=RND(-1)
20 S=RND(0)
30 T=RND(-2)
40 U=RND(0)
45 PRINT R,S,T,U
50 NEXT A
60 END
```
produces:

```
1.20398E-2      6.78473E-2      .810629         .398675
.216317         .905878         .546084         .178008
.2636           .810749         .87498          .876414
```

With each execution of RND(<0) different numbers are generated. The sequence is only initialized by time-of-day once per program execution; therefore, a different sequence of numbers is generated each time the program is executed.

4. The following example illustrates a typical use of the RND functions. The program simulates the rolling of a pair of dice (A and B) and outputs the number of occurrences that a particular value is observed. In this program, lines 100 and 110 use the RND function which produces random numbers between 0 and 1. These values are then converted to integer format to simulate the number of dots showing by a roll of the die.

The conversion of these values is accomplished by first multiplying the random value generated by 6 and then adding a value of 1; e.g., if random number generated at line 100 was .62891 -

then: $$\begin{array}{r} .62891 \times 6 = 3.77346 \\ +1.00000 \\ \hline 4.77346 \end{array}$$ INT(4.77346) = 4 (represents 4 dots showing)

The converted values from lines 100 and 110 are summed to represent the total number of dots showing by a roll of the dice, and this value is used in statement 130 to specify an array location. Each of the possible array locations were initialized to zero in statement 50, and in statement 130 the value in the location specified is incremented by 1. If the number of rolls specified was 10 and a result of "4" was generated twice in statement 120, then the location F(4) would contain a value of "2". The values in F(R) locations correlate the value randomly generated with the frequency of occurrence of that value.

Note that this program will produce the same results each time it is run unless a statement such as 85 LET x = RND(CLK(x)) is included or the argument for RND in lines 100 and 110 is changed to some negative value.

```
10 DIM F(12)
40 FØR Q=1TØ12
50 F(Q)=0
60 NEXT Q
70 X=1000
80 PRINT "THE NUMBER ØF RØLLS SELECTED IS";X
90 FØR S=1TØX
100 A=INT(6*RND(0)+1)
110 B=INT(6*RND(0)+1)
120 R=A+B
130 F(R)=F(R)+1
140 NEXT S
150 PRINT
160 PRINT "THE NUMBER ØF SPØTS SHØWING","NUMBER ØF ØCCURENCES"
170 FØR V=2TØ 12
180 PRINT TAB(12);V;TAB(38);F(V)
190 NEXT V
200 END
READY.
```

produces:

```
 THE NUMBER ØF RØLLS SELECTED IS 1000

 THE NUMBER ØF SPØTS SHØWING   NUMBER ØF ØCCURENCES
             2                         26
             3                         49
             4                         81
             5                         111
             6                         122
             7                         161
             8                         166
             9                         114
             10                        92
             11                        60
             12                        18
```

SYSTEM FUNCTIONS

The BASIC supplied SYSTEM functions are:

Function	Description
CLK$	Returns the time of day as a string constant in the form: HH. MM. SS. e.g., 17. 36. 37.
CLK(x)	Returns the time of day in hours and fractions of an hour in a 24-hour scale (x is a dummy argument). e.g., (1) Three minutes and 58 seconds past the hour of 9 is represented: 9.06611 (2) Midnight is represented: 0.00 (3) Noon is represented: 12.000 (4) Two-thirty p.m. is represented: 14.50
DAT$	Returns the date as a string constant in the following form: YY/MM/DD 74/08/11 } KRONOS/NOS MM/DD/YY 10/31/74 } SCOPE
TIM(x)	Returns the total elapsed central processor time in seconds used to-date (x is a dummy argument).

Example:

```
10 X=TIM(1)
20 PRINT "CLK$ TIME ØF";CLK$;"=";CLK(1);"IN CLK(X) TIME"
30 PRINT DAT$
40 Y=TIM(2)
50 PRINT "TØTAL ELAPSED TIME IS";Y-X
60 END
```

This program produces:

```
CLK$ TIME ØF 13.04.03.= 13.0675 IN CLK(X) TIME
 74/09/06.
 TØTAL ELAPSED TIME IS-.004
```

STRING FUNCTIONS

LENGTH Function

The LENGTH function yields the current length in characters of a string.

Format:

LEN (se)

se   is a string constant, expression or string variable.

The resultant value may be assigned to a numeric variable or used in any statement where numeric variables are allowed.

Examples:

1)
```
10 LET S$ = "543"
20 A = LEN(S$)
30 PRINT A
40 END
```

This program produces:

```
3
```

2)
```
DIM B$(20)
.
.
.
.
200 IF LEN(B$(4)) < 24 THEN 225
```

At execution time the length of the string in the 4th element of the string array B$ is compared to 24; if the length is less than 24, control is passed to statement 225.

3)
```
10 LET D$ = "ABCD"
20 PRINT "NØ ØF CHARACTERS";LEN(D$)
30 END
```

The program produces:

```
NØ ØF CHARACTERS 4
```

STRING Function

The STRING function converts a numeric value n to its corresponding string representation as specified by an optional image f.

Formats:

```
STR$(ne) or
STR$(ne, f)
```

ne	numeric constant variable, or arithmetic expression
f	image of the desired format.

The resulting string may be assigned to a string variable or used explicitly in any statement where string variables are allowed. The string is formatted in accordance with the image specified by f. The image f can contain alphanumeric constants and any specification control characters which are allowed in the IMAGE statement. See section 2.4 for a complete discussion of format images.

If f is absent, the string is formatted according to the standard rules for numeric output.

Examples:

1) `B$ = STR$(A(1, 6))`

Execution of this statement assigns B$ the string converted from the numeric value contained in an element of array A. If A(1, 6) contained a value 1234, then string variable B$ is assigned the string constant " 1234".

2)
```
IF "4894" = STR$(A9) THEN 200
A$ = STR$ (I, COSTPRICE = $###.## LESS DISCOUNT)
```

Execution of the first statement provides a comparison between two strings and a corresponding branch to 200. In the second statement, the numeric value of I is formatted according to the image specified; e.g., if the value of I is 203.23 then A$ is assigned the string-COSTPRICE = $203.23 LESS DISCOUNT.

SUBSTRING Function

SUBSTR may be used to extract a substring of specified length and starting position from a string. It may be used to replace the substring of an existing string with a substring specified by the user.

Formats:

SUBSTR(se, ne_1, ne_2)

SUBSTR(se, ne_1)

se	is a string variable representing the string from which the substring is to be extracted, or inserted. It may be a string constant for substring extraction.
ne_1	is a numeric constant, variable, or expression indicating the start of the substring.
ne_2	is a numeric constant, variable, or expression indicating the length of the substring.

Extraction: When referenced as part of a string expression, the SUBSTR function extracts and returns the substring of se which starts at character ne_1 and is ne_2 characters long. If ne_2 is not specified or if greater than the number of available characters in se, all the remaining characters are returned. If the starting position ne_1 is beyond the end of se, a null string is returned.

For substring extraction, SUBSTR may be used in any expression where string variables are allowed.

Examples:

1)

```
10 LET A$ = SUBSTR("DEPARTMENT", 3, 4)
```

Execution of this statement results in A$ being assigned the string "PART".

2)

```
10 A$ = " PART"
20 LET X$ = SUBSTR(A$, 2, 10)
30 LET X$ = SUBSTR(A$, 2)
```

These statements assign the string " ART" to X$. Statement 20 asks for ten characters starting with the second, but there are only three. Statement 30 asks for all characters beginning with the second.

3)

```
40 IF SUBSTR(B$, 1, 4) = "XXXX" THEN 90
```

This statement transfers control to statement 90 if the first four characters of B$ are XXXX.

Insertion

When SUBSTR(se, ne_1, ne_2) is used to the left of the equal-sign in a LET statement, the substring of se which begins at character ne_1 and is ne_2 characters long is replaced by the first ne_2 characters of the string or string expression specified to the right of the equal-sign. The se must be a string variable and may not be a string constant in any case. SUBSTR is a unique function name that permits this form of usage. No other function may be referenced on the left of an equal-sign.

If ne_2 is not specified, the ne_1^{th} and all following characters of se are replaced by the entire string specified on the right side of the equal-sign. The length of se is increased to accomodate the right-hand string if necessary. A diagnostic is generated if the length must extend beyond 78. This feature enables SUBSTR to concatenate two strings.

If the starting position ne_1 is beyond the end of se, blanks are inserted between the end of the original se and the beginning of the inserted substring. Similarly, if ne_2 characters are called for and the right-hand string is less than ne_2 characters long, enough blanks are added to the end of the inserted substring to make it exactly ne_2 characters long.

For substring insertion, SUBSTR must appear on the left side of the equal-sign in a LET statement. It may not appear in any other position or in any other statement.

Examples:

1)
```
10 A$ = " DEDUCTION"
20 LET SUBSTR(A$, 1, 2) = " IN"
```

Execution of these statements results in A$ being assigned to the string "INDUCTION".

2)
```
10 A$ = " HOG"
20 B$ = " TIED"
30 LET SUBSTR(A$, LEN(A$)+1) = B$
```

The execution of these statements results in the string " HOGTIED". This is an example of using SUBSTR to concatenate two strings. All characters in B$ are inserted after the last character of the original A$ because the starting position, ne_1, is set to the length of A$+1 and the length indicator, ne_2, is not specified.

3)
```
50 X$ = " XXX"
66 LET SUBSTR(X$, 6, 6) = " YYY"
```

After executing these statements, X$ identifies string "XXX YYY   ". Two spaces between the X and Y are inserted because the starting position, ne_1 in SUBSTR is 6 and the original X$ is only three characters long. The three trailing blanks are added because the requested length, ne_2 is 6 and the right-hand string is only three characters long.

VALUE Function

The VALUE function converts a string to its numeric value. The function is the inverse of STR$ function.

Format:

VAL(se)

se	is a string constant, variable or expression containing only characters which form a valid number.

The VAL function may be used in any arithmetic expression.

Examples:

```
B9 = VAL(B$(1))
X4 = 2*B4 + VAL("123.7")
IF VAL(B$(I,J)) > 24 THEN 291
```

When the first example is executed and if B$(1) contains a string " 1234", then the numeric value 1234 is assigned to B9.

Similarly, in the latter two examples, numeric values are extracted and used for arithmetic purposes or for comparison with a numeric constant.

USER FUNCTIONS

BASIC permits the user to define his own functions within his program through use of the DEF statement and later reference the function by specifying the name assigned to it on the DEF statement. Results from a function execution (single value) are returned to the function reference.

DEF STATEMENT

Functions mav be defined by the user with the DEF statement.

Format:

DEF FNl (snv) = ne

l	alphabetic character uniquely identifying the function name.
snv	simple numeric variable; the formal parameter.
ne	arithmetic expression; the rule for evaluating the function.

The following rules apply in defining a function:

1. The variable snv is a formal parameter. It may be used elsewhere in the program without affecting the function.
2. The arithmetic expression (ne) may include the formal parameter and it may include other program variables, simple or subscripted.
3. The definition must be complete on one line.
4. The expression in a DEF statement may include references to BASIC or user-defined functions, but not to the function being defined; i.e., recursive definitions are not allowed.
5. A DEF statement may appear anywhere in a BASIC program, not necessarily before the function is used.

Examples:

```
DEF FNA(D) = 1/D
DEF FNB(D) = D- FNA(B)
A  = FNB(A)
```

REFERENCING A FUNCTION

The format of a function reference is:

function name (ne)

function name	name assigned in the DEF statement
ne	any numeric expression

The following rules apply when using a function:

1. The argument (ne) in parentheses in the function call is evaluated and the resultant value is used in the expression of the DEF statement as the value of the formal parameter.

2. The expression of the DEF statement is evaluated (using the current value of any variables in the expression) to yield the function value.

3. When the formal parameter does not appear in the expression of the DEF statement, a function reference must still have an argument (ne), although its value has no effect on the function value.

4. A function may be redefined within a program. When it is referenced, the definition used is the one on the highest numbered line before the line containing the function reference. When a function is referenced at a line before any of its definitions, the definition used is the one with the lowest line number after the function reference.

Example:

```
10 DEF FNA (R) = 3.14159 * R ↑ 2
20 DEF FNC (D) = 3.14159 * D
30 DEF FNV (R) = FNA (R) * R/3
40 PRINT "RADIUS","CIRCUMFERENCE","AREA","VØLUME"
50 FØR R = .1 TØ 1 STEP .1
60 PRINT R, FNC(2*R), FNA(R), FNV(R)
70 NEXT R
80 END
```

This program produces:

```
RADIUS          CIRCUMFERENCE   AREA            VØLUME
 .1              .628318         3.14159E-2      1.04720E-3
 .2              1.25664         .125664         8.37757E-3
 .3              1.88495         .282743         2.82743E-2
 .4              2.51327         .502654         6.70206E-2
 .5              3.14159         .785397         .1309
 .6              3.76991         1.13097         .226194
 .7              4.39823         1.53938         .359188
 .8              5.02654         2.01062         .536165
 .9              5.65486         2.54469         .763406
 1.              6.28318         3.14159         1.0472
```

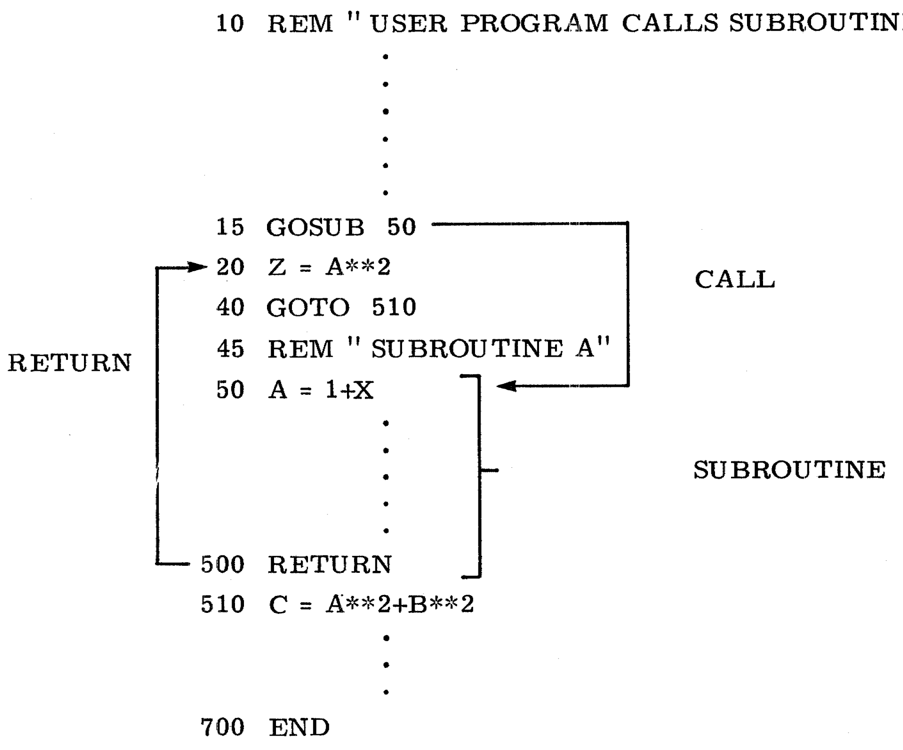
In the example above, FNA computes the area of a circle when given its radius; FNC computes the circumference of a circle when given its diameter; and FNV computes the volume of a sphere when given its radius. Notice that the definition of FNC uses the function FNA.

At line 40 four column headings are printed. The FOR loop prints on successive lines a radius and the corresponding circumference, area, and volume computed by the user-defined functions.

SUBROUTINES

A program may include a number of subroutines, each consisting of any number of BASIC statements terminated by a RETURN statement. Control is transferred to a subroutine from the main program by the GOSUB statement and return is always made to the main program at the next line following the GOSUB statement which transferred control to that subroutine. Subroutine recursion is permitted.

Example:



```
 10 REM " USER PROGRAM CALLS SUBROUTINE A"
         .
         .
         .
         .
         .
         .
 15 GOSUB 50
 20 Z = A**2
 40 GOTO 510
 45 REM " SUBROUTINE A"
 50 A = 1+X
         .
         .
         .
         .
         .
500 RETURN
510 C = A**2+B**2
         .
         .
         .
700 END
```

The above illustrates a subroutine call and return sequence. Lines 50 through 500 contain subroutine A. After execution of subroutine A, control is transferred to line 20 and at line 40 execution sequence is directed to line 510, bypassing the subroutine A statements.

GOSUB STATEMENT

The GOSUB statement directs the program to the first line of a subroutine. Each time this statement is executed the reference line number of the statement following the GOSUB statement is placed at the top of a stack (list). Each execution of the RETURN statement removes the most recent entry in the stack. The stack contains 40 locations; therefore, subroutine references can be nested 40 deep.

Format:

GOSUB	ln
ln	line number of the first statement of the subroutine.

RETURN STATEMENT

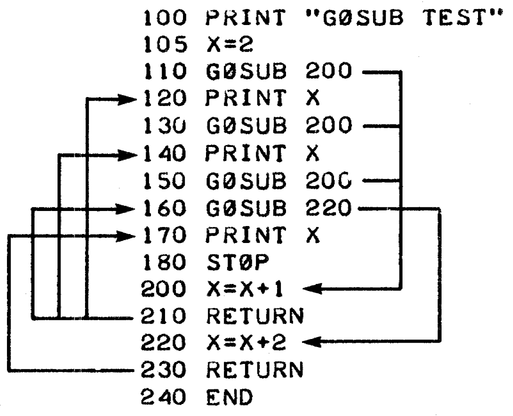
The RETURN statement must appear as the last statement of the subroutine. It directs the program to resume execution at the statement immediately following the previously executed GOSUB statement.

Format:

```
RETURN
```

Example:

The following demonstration program makes use of one subroutine three times and a second subroutine once. The 4-line printout shows a user input of x = 2 and the three results obtained. An analysis of the program statements follows the printout.



```
100 PRINT "GØSUB TEST"
105 X=2
110 GØSUB 200
120 PRINT X
130 GØSUB 200
140 PRINT X
150 GØSUB 200
160 GØSUB 220
170 PRINT X
180 STØP
200 X=X+1
210 RETURN
220 X=X+2
230 RETURN
240 END
```

This program produces:

```
GØSUB TEST
 3
 4
 7
```

Analysis of Program Statements:

100	Prints the heading GOSUB TEST.
105	Assigns a value of " 2 " to x.
110	Control is transferred to the subroutine at 200.
200	X = 2+1 = 3
210	Control is returned to the statement following the most recent GOSUB, i.e., 120.
120	The value 3 for X is printed.
130	Control is transferred to the subroutine at 200.
200	X = 3+1 = 4
210	Control is returned to the statement following the most recent GOSUB, i.e., 140.
140	The value for X is printed.
150	Control is transferred to the subroutine at 200.
200	X = 4+1 = 5
210	Control is returned to the statement following the most recent GOSUB, i.e., 160.
160	Control is transferred to the subroutine at 220.
220	X = 5+2 = 7
230	Control is returned to the statement following the most recent GOSUB, i.e., 170.
170	The value 7 for X is printed.
180	The program terminates.

INPUT AND OUTPUT 2.4

This section describes the BASIC statements related to input and output. Included are the file manipulation statements, special output formatting statements, and statements to read and write data.

FILES AND INTERNAL DATA BLOCK

A file is a named collection of data which a BASIC program can reference and manipulate. A file name (lfn) consists of 1 to 7 alphanumeric characters, the first of which is always a letter (A123). Files used with BASIC are normally located on mass storage. Exceptions are those files connected or assigned to the terminal and internal data block. Terminal files accept and display data directly at a terminal. The internal data block exists within the BASIC program.

When using files, it is important to remember that BASIC programs can read and write data in two formats: binary format (files created by WRITE and DATA statements) and coded format. These formats should never be mixed in a file. Binary format data can be used by the program with no conversion, but it cannot be printed at a terminal or printer. Coded format data can be printed, but it must be converted into binary by BASIC before the program can use it. All data entered by the user on cards, or at his terminal, and all data printed is in coded format. In general, binary data is written only if the data is to be read later, while coded data can be either printed or read later by a BASIC program.

PERMANENT FILE ACCESS

BASIC deals with local (working) files. A local file is any file other than a permanent file; i.e., a file created during the terminal session or, a copy of a permanent file which contains program code to be executed during the session or which contains data to be accessed during the execution of a program created during the session, or a file obtained by copying the permanent file. A local file can be updated (via additions or deletions) at the terminal. Local files which are permanent file copies can be changed without affecting their respective permanent files. Appropriate operating system file commands are used to make a local file permanent and a permanent file local. These commands must be used prior to BASIC program executions. (See sections 2.7, 2.8 and appendix E for a description of these commands.)

The coded files named "INPUT and OUTPUT" have special meaning to BASIC programs. In interactive mode coded data written on the file OUTPUT is automatically printed at the terminal. The file OUTPUT is connected to the terminal. The file INPUT is also connected to the terminal; whenever a BASIC program requests data from the user at a terminal, the data entered is automatically placed in the file INPUT. Files OUTPUT and INPUT are the defaults for the coded input and output statements; i.e., if no file is specified in a coded output (PRINT) statement, the output is displayed at the terminal. A description of the PRINT statement is provided later in this section.

In batch processing, the file OUTPUT is normally the default for coded output statements, and this file is automatically printed after the program has executed. The file INPUT is the default for the coded input statements, and is created from the cards submitted by the user. The specification of the file parameter options is accomplished via the BASIC control card. See section 2.9 for a description of these options.

FILE STATEMENT

Format:

FILE #n_1 = lfn$_1$, #n_2 = lfn$_2$,...#n_3 = lfn$_3$

n_1 - n_3	the file ordinal is any numeric constant variable or expression with a value between 1 and 2^{18-1}
lfn$_1$-lfn$_3$	the file name is a string constant or variable with seven or fewer alphanumeric characters; the first character must be letter.

The file statement is used to associate a number, called the file ordinal, with a file name. After the file statement is executed, all remaining INPUT/OUTPUT statements may reference the named file by its ordinal.

Examples:

```
1)  10 FILE #1  = "OLDM", #11 = "NEWM"
2)  50 FILE #48 = A$
3) 100 FILE #X  = A$
4) 110 FILE #99 = "OUTPUT"
```

In the first example, files OLDM and NEWM are assigned ordinals, 1 and 11, respectively. In the second example, a file whose name is determined during execution of the program, is assigned ordinal 48. In the third example, both file name and ordinal are determined during execution. If the variable X is not an integer, it is truncated. In the fourth example, the ordinal 99 is assigned to the file "OUTPUT", so that all data placed on file 99 is output to the user's terminal or the printer.

All files are allocated at compile time. The compilation of a FILE statement that contains a string variable for a file name, results in the allocation of a file without a name. The first execution of the FILE statement with a value for the string variable, which is not already the name of a file, results in the unnamed file being given that name. Once named, the file always retains that name. The names of files cannot be changed; only their ordinals can be changed. A maximum of 15 files, including INPUT and OUTPUT, can be used in a BASIC program.

DATA STATEMENT

Format:

DATA c_1, c_2, c_3, ... c_n

c_1 - c_n numeric or string constant

The DATA statement is used to create a block of binary format data internal to the BASIC program; this data can then be accessed by a READ statement.

Any number of DATA statements may appear anywhere in the program. The BASIC compiler considers them contiguous statements, and places the data in sequential order in one data block (referred to as an internal data block).

DATA statements are non-executable and have no effect on the results of a program if they are encountered in the normal sequence of execution.

Examples:

1)
```
 5 DATA 5.3, 2, 3, 4, 4.5, -0.003E-5
10 DATA 0.00000589, +55, 384.6, 890
20 DATA "STRING EXAMPLE"
```

2)
```
5 DATA "E",2,3,4,5
10 READ A$,B,C,D,E
20 PRINT A$;E
```

produces:

```
E 5
```

3)
```
5 DATA "STRINGS", 4.3, STRING 2
10 READ A$,B,C$
20 PRINT A$;B;C$
```

produces:

```
STRINGS 4.3 STRING 2
```

Both quoted and unquoted strings are allowed. However, unquoted strings must not begin with a +, -, ., comma, digit, or blank.

RESTORE STATEMENT

Formats:

1) RESTORE
   or
2) RESTORE #ne

   ne   numeric variable or constant or expression which evaluates to a file ordinal which is associated with a file name.

A file or internal data block has a pointer associated with it which indicates the position of the file. For an input file, as the file is being read, the pointer moves ahead indicating the next item of data to read. For an output file, the pointer is always at the end of the file indicating where the next item of information is to be written. The RESTORE statement positions this pointer back to the beginning of the file. If format 1 is specified, the statement refers to the internal block of binary data created by the DATA statements.

Example:

```
210 DATA 1,2,3
220 READ A,B,C
225 RESTØRE
230 READ D
240 PRINT A,B,C,D
250 END
```

produces the following results:

```
 1              2              3              1
```

NODATA STATEMENT

Formats:

1) NODATA ln
2) NODATA #ne, ln

ln	line number
ne	numeric variable, constant, or expression which evaluates to a file ordinal, associated with a file name

The NODATA statement is used to test the location of the file position pointer. If the pointer is at the end of data, control is transferred to the statement whose line number was specified by ln. Thus, the NODATA statement can be used to determine if all the data in a file has been read. If format 1 is used, the statement refers to the block of binary data created by the DATA statements.

A file which has just been written has no data available for reading.

A NODATA statement can reference a file which has never been read.

Example:

```
100 NØDATA 150
110 READ A1,A2,A3
120 PRINT A1,A2,A3
130 GØTØ 100
140 DATA 1,2,3,4,5,6,7,8,9
150 END
```

(when all items in the DATA statement are read, go to line 150)

produces the following results:

```
1                 2                 3
4                 5                 6
7                 8                 9
```

BINARY INPUT/OUTPUT

READ STATEMENT

Formats:

1) READ $v_1, v_2, v_3, \dots v_n$

2) READ #ne, $v_1, v_2, v_3, \dots v_n$

v_1-v_n	variable identifier (numeric string or subscripted)
ne	numeric variable constant or expression which evaluates to a file ordinal which is associated with a file name.

The READ statement is used to read binary data; i.e., files created by WRITE or DATA statements. Each item of data is read from the file, and assigned to the next variable in the READ statement. If format 1 is specified, the binary block of data created by the DATA statement is read. Both string and numeric data can be read by the READ statement, but string data must not be read into numeric variables. When reading from an internal data block, if the program attempts to read strings into numeric variables or numbers into strings, the diagnostic "BAD DATA IN READ" is issued. This diagnostic is not issued when reading from a file; however, the results of such a read are unpredictable. As each item of data is read and assigned to the corresponding variable, the pointer to the next item of data is advanced. If a READ exceeds the end of data, the diagnostic, " END OF DATA AT line number", is given and the program is terminated.

Example:

1)
```
100 DATA 1,2,3
110 READ A,B,C
120 PRINT A,B,C
130 END
```

produces:

```
1                    2                    3
```

2)
```
100 DATA 1,2,3
110 READ A,B,C,D
120 PRINT A,B,C,D
130 END
```

produces:

```
END ØF DATA    AT   110
```

WRITE STATEMENT

Formats:

WRITE #ne, e_1, e_2, e_3, . . . e_n

e_1-e_n	expression, variable or constant (numeric or string)
ne	numeric variable, constant, or expression which evaluates to a file ordinal which is associated with a file name.

The WRITE statement places the data referenced on a specified file which is written as one contiguous block. All WRITE's tô the file contribute to the same block. The data is written in binary format at the current position of the data pointer. Normally, data placed on a file by a WRITE statement will be read by a READ statement contained in the same program or another BASIC program. Binary data cannot be read by an INPUT statement.

Example:

```
95 FILE #1="ØLDM"
100 LET A=B=C=1
110 WRITE #1,A,B,C
120 RESTØRE #1
130 READ #1,D,E,F
140 PRINT A,B,C
150 END
```

produces:

```
1              1              1
```

CODED INPUT

INPUT STATEMENT

The INPUT statement is used to read coded data from a file or to permit the user to enter data during execution from the terminal.

Formats:

1) INPUT $v_1, v_2, \ldots v_n$

2) INPUT #ne, $v_1, v_2, \ldots v_n$

v_1-v_n	string or numeric variable
ne	numeric variable constant or expression which evaluates to a file ordinal which is associated with a file name

When a BASIC program is run interactively from a terminal, the INPUT statement without an ordinal (format 1) reads data into the program from the terminal. One item is input for each variable of the INPUT statement.

Each time an INPUT statement is executed, a question mark is displayed at the current print position of the terminal line. The user must then enter data to satisfy the input request. The data entered must correspond one-for-one with the variables in the INPUT statement. Numbers must be entered for numeric variables and quoted or unquoted strings must be entered for string variables. Unless DELIMIT is in effect, numeric constants may be separated by commas or blanks; string constants must be separated by commas.

A carriage return marks the end-of-data to be entered. If insufficient data is entered, a diagnostic message " NOT ENOUGH DATA, TYPE IN MORE" is issued. The user should continue entering data until the input requirement is satisfied. A diagnostic message followed by a question mark is issued if too much data, or data unacceptable to BASIC, is entered. The user must then retype the entire data list.

Example:

```
10 INPUT X,Y
20 REM QUESTIØN MARK WILL MEAN TWØ VALUES ARE REQUIRED
30 PRINT "X=";X,"Y=";Y
40 END
```

produce: (user responses are underlined)

```
?0
NØT ENØUGH DATA, TYPE IN MØRE     AT  10
?2,4
TØØ MUCH DATA -RETYPE AT  10
?1,2
X= 1              Y= 2
```

A PRINT statement (30) associated with the INPUT statement (10) eliminates any confusion as to how many and what type of data items to enter.

Example:

The statements

```
80 PRINT "WHAT IS THE VALUE ØF X";
85 INPUT X
```

produce:

```
WHAT IS THE VALUE ØF X ? 9
```

The user responds by typing one numeric constant immediately after the question mark.

Following are the rules for entering the data from a terminal:

1. INPUT items are delimited by commas unless a DELIMIT statement is in effect. Numeric items are also delimited by blanks.
2. A Carriage Return marks the end of data entry.
3. If insufficient data is entered, BASIC will issue a request for more.
4. If too much data or unacceptable data is entered, BASIC will request that the data be reentered.
5. All trailing blanks are eliminated from the input line.
6. Redundant delimiters preceeding or following data items are ignored.

When used in batch mode, the format 1 INPUT statement reads from the file named INPUT; i.e., in the same way as an INPUT statement with an ordinal associated with a file named "INPUT" (default input file). The data pointer is advanced for each item read.

When format 2 is used, then input is essentially the same as input from a terminal except that too much data and not enough data conditions are not applicable.

Following are the rules for inputting data from a file:

1. A file is considered one block of data. Items are read sequentially - one at a time.
2. Items are delimited by commas unless DELIMIT is in effect. Numeric items can be delimited by blanks.
3. End-of-line (EOL) always acts as a delimiter. However, it does not mark the end of data as carriage return does for a terminal line. End-of-line (EOL) is a special indicator on coded files which marks the end of each line. It is the logical equivalent of carriage return on a terminal. EOL's are automatically written on coded files created by BASIC (see CODED OUTPUT).
    - If more data is requested than is available in the current line, data is taken from the following line.
    - If more data exists in the current line than is needed, no diagnostic is issued. The next INPUT begins where the current INPUT terminates.
4. The program terminates and the diagnostic, END OF DATA ON FILE is issued if the program attempts to INPUT more data than exists in the file.
5. All trailing blanks are eliminated from each input line.
6. Redundant leading and trailing delimiters are ignored.

DELIMIT STATEMENT

Formats:

1) DELIMIT (ch_1), (ch_2), (ch_3)
2) DELIMIT #ne(ch_1), (ch_2), (ch_3)

ch_1-ch_3	any character or CR (carriage return)
ne	numeric variable or constant used as a file ordinal which is associated with a file name

DELIMIT specifies the character or characters to be used as input item separators. The characters specified override default separators, comma and blank. Any characters, or the mnemonic CR (carriage return), may be specified as separators. If CR is specified, the entire line is accepted as one data item.

Zero, one, two, or three characters may be specified in a DELIMIT statement. If no characters are specified, the default delimiters (comma and blank) are restored. If no file is specified in the DELIMIT statement, the delimiters apply to the file INPUT, and thus to all input from the terminal.

Examples:

1)
```
 5 FILE #1 = " DATAIN"
10 DELIMIT (CR)
55 DELIMIT #1( ), (;)
```

In the first example, if the program is run interactively from the terminal, all requests for input from the terminal read all information typed up to the carriage return (CR) into a single variable. It should be noted that this form would most likely be used to read data into a string variable. In the second example, a blank and semicolon (;) are interpreted as delimiters (numeric and string) when inputting from a file with an ordinal of 1 (DATAIN).

2)
```
110 DELIMIT
155 DELIMIT #1
```

The above two examples restore the default delimiters (blank and comma).

To show concisely the difference between DELIMIT not in effect and DELIMIT in effect, a comparison of the two cases follows.

Normal Case (DELIMIT not in effect):

1. Carriage return on a terminal (or end-of-line on files) is always treated as a delimiter.

   - When the input is from a terminal and CR (Carriage Return) is encountered before the input list is satisfied, the message " NOT ENOUGH DATA, TYPE MORE" is issued.

   - When input from a terminal and data exists on the input line after the input list is satisfied, the message " TOO MUCH DATA, RETYPE... " is issued.

   - When the input is from a file and end-of-line is encountered before the input list is satisfied, it is treated as a delimiter (item separator) and input continues from the next line.

   - When the input is from a file and data exists on the input line after the input list is satisfied, no diagnostic is issued. The next INPUT starts reading where this INPUT ends.

   - If a delimiter is encountered after the input list is exhausted, it is ignored.

2. Leading and trailing blanks on the input line are ignored.

3. Comma is the delimiter for all input items (numbers, quoted strings and unquoted strings).

4. Blanks are delimiters for numbers, but not for strings.

DELIMIT in effect:

1. Default delimiters are turned off except carriage return CR or end-of-line. Only explicitly named characters act as delimiters.

   - Comma and blank do not delimit items unless they are specified in a DELIMIT statement.

   - Quotes have no special meaning.

   - All characters including quotes and leading blanks are valid string characters.

   - All strings are considered to be unquoted. There are no string boundary characters equivalent to quotes.

2. Carriage return (end-of-line on files) is always a delimiter and need not be explicitly defined in a DELIMIT statement.

3. Trailing blanks on an input line are ignored unless CR is explicitly defined as a delimiter.

4. Leading blanks are not ignored unless the item being input is a number.

CODED OUTPUT

This section describes the statement used to create coded output; i.e., the PRINT statement, as well as, statements related to the formatting of coded output.

PRINT STATEMENT

Formats:

1) $\left\{\begin{array}{l}\text{PRINT } e_1de_2d\ldots e_nd \\ \text{PRINT USING ln, } e_1de_2d\ldots e_nd\end{array}\right.$ or

2) $\left\{\begin{array}{l}\text{PRINT \#ne, } e_1de_2d\ldots e_nd \\ \text{PRINT \#ne USING ln, } e_1de_2d\ldots e_nd\end{array}\right.$ or

e	expression, variable, constant (numeric or string)
d	delimiter (comma or semicolon); specification of the final delimiter is optional
ne	numeric variable, constant, or expression which evaluates to a file ordinal which is associated with a file name
ln	line number of IMAGE statement (described later in this section)

The PRINT statement is used to write coded data on a file. As previously described, if the coded file OUTPUT is used, then the data appears on the user's terminal if the program is being run as in interactive job from a terminal, or on the printer if the program is being run as a batch job. The file OUTPUT is the default file. The default file name for PRINT can be changed by using the Land K options on the BASIC batch control card. See section 2.9.

The PRINT statements with the USING option cause the output to be formatted to an IMAGE statement. This is discussed in detail in the IMAGE statement description.

In this discussion and most examples that follow, the PRINT statement is discussed for format 1, which does not specify a file. It should be understood that the same output would appear on a file if one were specified.

Examples:

```
20 PRINT A, B, SIN(A)
```

prints the values of A, B and SIN(A).

```
30 PRINT #N, " VALUES ARE X AND X SQUARED", X;X*X
```

prints the string constant and the values of X and X*X on the file whose ordinal is N.

```
40 PRINT USING 140, 14.3, 143.0
```

prints 14.3 and 143.0 according to the format image at line 140.

```
50 PRINT
```

prints a blank line.

To inhibit automatic printer and terminal carriage spacing, a blank is always appended to the beginning of each line to be output by format 1 PRINT statements. This blank is not normally prefixed to lines output by format 2 PRINT statements, unless the file ordinal referenced is that of the default print file " OUTPUT" or default file specified by the L or K option of the batch BASIC control card.

DEFAULT OUTPUT FORMATS

Unless a USING clause is used or the SETDIGITS statement (described later) is in effect, all numbers and strings printed are printed in standard default formats. These formats and the meaning of the print item delimiters are explained below.

Numeric Formats

Numeric values are formatted in one of the three standard formats shown in table 2.4-1 where:

- n represents a numeric digit.
- each format is preceded by a minus sign (for negative values) or a blank (for positive values)
- each format is terminated by one trailing blank.
- leading zeros are suppressed.
- trailing zeros after a decimal are suppressed.
- numbers are left-justified.
- the final digit in format 2 is obtained by rounding.

TABLE 2.4-1. STANDARD NUMERIC OUTPUT FORMAT

INTERNAL VALUE	OUTPUT FORMAT USED
● exact integers of less than ten digits	nnnnnnnnn
● non-integers whose integer portion is not more than six digits ● non-integers whose integer portion is zero and whose six most significant digits immediately follow the decimal	nnnnnnn (where one n represents a decimal point)
● all other numbers	n.nnnnnE±nnn

Example:

```
10 A1 = 0
20 B1 = -124
30 C1 = 123456789
40 D1 = 123456.789
50 E1 = .7623481
60 F1 = -.00192
70 G1 = 1234567890
80 H1 = 1234567.8
90 J1 = .07623488
100 K1 = -.0000192
110 PRINT "INTERNAL VALUE "
120 PRINT "0","-124","123456789","123456.789",".7623481"
130 PRINT
140 PRINT "ØUTPUT FØRMAT"
150 PRINT A1,B1,C1,D1,E1
160 PRINT
170 PRINT "INTERNAL VALUE "
180 PRINT "-.00192","1234567890","1234567.8",".07623488","-.0000192"
190 PRINT
200 PRINT "ØUTPUT FØRMAT"
210 PRINT F1,G1,H1,J1,K1
220 END
```

produces:

```
INTERNAL VALUE
0               -124            123456789       123456.789      .7623481

ØUTPUT FØRMAT
 0              -124             123456789       123457.         .762348

INTERNAL VALUE
-.00192         1234567890      1234567.8       .07623488       -.0000192

ØUTPUT FØRMAT
-.00192          1.23457E+9      1.23457E+6      7.62349E-2     -1.92000E-5
```

String Formats

String constants are printed exactly as they appear in the PRINT statement without the quotation marks.

Example:

```
115 LET X = Y = Z =2
117 PRINT "ANSWER","X AND Z =";Z,"X*Y*Z=";X*Y*Z
118 END
```

produces:

```
ANSWER          X AND Z = 2     X*Y*Z= 8
```

If statement 117 is changed to:

```
117 PRINT "ANSWER;X AND Z =;Z;X*Y*Z=;X*Y*Z"
```

the program produces:

```
ANSWER;X AND Z =;Z;X*Y*Z=;X*Y*Z
```

PRINT ZONING

The print line normally is divided into five zones of 15 spaces each. A comma, used as a separator or a final delimiter, signals BASIC to move to the next zone of the print line, or to the first zone of the next print line when the last zone is filled.

When a semicolon is used as a separator, it has no spacing effect; i.e., print line zoning effect is inhibited. Because the numbers are printed preceded by a blank or a minus sign, and followed by another blank, two positive numbers will be separated by two blanks.

Example:

1)

```
10 A1 = 123
20 B2 = 256
200 PRINT "0123456789"
300 PRINT A1;B2
400 END
```

produces:

```
0123456789
 123  256
```

When a semicolon is used to separate strings, they are printed consecutively without any preceding or intervening blanks.

2)

```
10 PRINT "THIS IS";"AN EXAMPLE"
30 PRINT "THIS IS","AN EXAMPLE"
40 END
```

produces:

```
THIS ISAN EXAMPLE
THIS IS        AN EXAMPLE
```

Commas and semicolons can be intermixed in any PRINT statement. When commas are used as separators with numeric data, each number occupies one zone: but with string data, each string may occupy more than one zone.

If a PRINT statement does not end with a delimiter (either semicolon or comma), subsequent printing commences at the beginning of a new line. If a PRINT statement does end in a delimiter, subsequent printing continues on the same line until the line is filled. If a semicolon is used as the final delimiter, the next item printed starts in the next available space. If a comma is the last delimiter, the next item printed starts at the beginning of the next zone.

3)

```
490 LET X = 1
500 PRINT "VALUE ØF X=";X
510 END
```

produces:

```
VALUE ØF X= 1
```

4)

```
300 PRINT 300,400,500,600
320 PRINT 700,800
330 END
```

produces:

```
300               400               500               600
700               800
```

5)

```
300 PRINT 300;400;500;600;700;800;900
310 END
```

produces:

```
 300   400   500   600   700   800   900
```

6)

```
10 FØR I = 1 TØ 10
15 PRINT I
20 NEXT I
30 END
```

produces:

```
1
2
3
4
5
6
7
8
9
10
```

If line 15 is changed to:

```
15 PRINT I,
```

the above program produces:

```
1                 2                 3                 4                 5
6                 7                 8                 9                 10
```

If line 15 is changed to:

```
15 PRINT I;
```

the program produces:

```
1  2  3  4  5  6  7  8  9  10
```

TAB FUNCTION

A TAB function used in a PRINT statement causes the printing to start at the position indicated by an argument. All arguments are considered modulo the line width. Print positions are counted from 0. Line width is normally 75; however, it may be changed by the use of the MARGIN statement (discussed later in this section). If the argument is less than the current print position, TAB has no effect. The semicolon should be used as a separator when the TAB function is used, because the comma causes a move to the next print zone. The TAB function is legal only in the PRINT statements.

Format:

TAB (ne)

ne	a constant, variable or expression indicating print position number

Examples:

1)

```
20 PRINT TAB(10);"1";TAB(20);"2";TAB(30);"3"
30 PRINT "0123456789012345678901234567890"
40 END
```

produces:

```
          1         2         3
0123456789012345678901234567890
```

2)

```
100 I1 = 12345678
110 I2 = 123456789
120 I3 = 12345678901
130 D1 = 123.4
140 D2 = 123.456
145 D3 = 123.4567
150 PRINT I1,TAB(30);D1
160 PRINT I2,TAB(30);D2
170 PRINT I3,TAB(30);D3
180 END
```

produces:

```
12345678                       123.4
123456789                      123.456
1.23457E+10                    123.457
```

IMAGE STATEMENT

The IMAGE statement is used with PRINT USING to explicitly describe the desired output format.

Format:

:literal $f_1f_2f_3...f_n$

literal		quoted or unquoted string (optional)
f_1-f_n		any string constant format specification in quotes;
	or	any unquoted string constant not containing +, -, ., #or↑,
	or	a format specification made up of the characters +, -, ., #and↑.

NOTE: The IMAGE statement does not contain a key word as other BASIC statements. The IMAGE statement is identified by the leading colon.

A format specification describes the exact printed format of the value and the literal to be printed by the PRINT statement. The literal is printed as written; it is the format of the value which is determined by the IMAGE statement. When the format is filled by string data, the specification determines only the number of characters to be included from the string. When the format is filled by numeric data, the format specification directs the placement of the value in the field, and the number of digits retained in the converted value. The PRINT statement is described earlier in this section.

A format specification is built from the following table of specification control characters.

SPECIFICATION CONTROL CHARACTERS

CHARACTER	POSITION/OPTION	NO. /FORMAT	EFFECT
+	First character, optional	One only	When outputting numbers, specifies that the output field is to be signed; a minus sign or a plus sign precedes the first significant digit of the output field. When outputting strings, specifies concatenation.
-	First character, optional	One only	For numeric data, specifies that the output field is to be signed if the value is negative, and unsigned if the value is positive; a minus sign precedes the first significant digit of the output field.
#	Any position, required	At least one	For numeric data, specifies a possible digit from the converted numeric data. For string data, specifies a possible character for the output string.
	Any position, optional	One only	Specifies the placement of the decimal point in the output field; this character remains in output field.
↑	Last five, optional	Must be five	Specifies the exponent portion of a field to be output in exponential format. The associated value is output in E-format with letter 'E', an exponent sign and 3 digits for the exponent value.

The specification control characters are combined to provide four classes of format specification. The pictorial representation of the line to be printed is built by combining the necessary format specifications and character strings.

The classes of format specification are:

- Integer
- Fixed Point
- Floating Point
- Alphanumeric

Integer Format

An integer format consists of an optional plus or minus sign and one or more pound signs (#).

The associated value is right-justified within the defined format.

Any non-integer value is truncated before printing.

If the specification includes a minus sign and the associated value is negative, a minus sign is printed immediately preceding the first significant digit.

If the specification includes a plus sign, the sign of the associated value (+ or -) is printed preceding the first significant digit.

If there is no place reserved for the sign and the value is negative, the left-most position is used for the sign.

If the associated value and its sign are larger than the format specification, an asterisk is printed and the format is widened to the right.

Examples:

1)

```
10 PRINT USING 20,879
20 : (##)
30 END
```

produces:

```
(*879)
```

2)

```
100 READ Z1,X2,C3
120 PRINT USING 130,Z1,X2,C3
130 : ####  ###  +#####
135 NØDATA 999
140 GØTØ 100
900 DATA 123,45,-3.856,45.8,548.34,-19
999 END
```

produces:

```
123   45       -3
 45  548      -19
```

Fixed Point Format

A fixed point specification consists of an optional + or - and a string of not less than 1 nor more than 14 pound signs (#) with a decimal point either preceding, embedded in, or terminating the pound signs.

- If the format contains a plus sign (+), the sign of the associated value (+ or -) precedes the most significant digit.
- If the specification includes a minus sign and the associated value is negative, a minus sign precedes the most significant digit.

If the number of digits to the left of the decimal point is greater than the number of positions allowed in the format specification, an asterisk is printed, and the format is expanded to accommodate the value by shifting the remaining fields to the right.

If the value is negative and if the number of digits to the left of the decimal point is equal to the number of positions allowed in the format specification, the number of digits to the right of the decimal point is reduced by one to provide room for the minus sign.

The associated value is right-justified within the format specification and the decimal point inserted as specified in the format specification.

The associated value is rounded and truncated according to the number of pound signs to the right of the decimal point in the format specification.

Examples:

1)

```
100 : .####      #.##      -#####.
110 READ Z1,X2,C3
120 PRINT USING 100,Z1,X2,C3
130 NØDATA 999
140 GØTØ 110
500 DATA -.134255,3,123.45
510 DATA .00177,-1.9999,-4596.432
999 END
```

produces:

```
-.134      3.00         123.
.0018      -2.0       -4596.
```

To correct the problem of insufficient room for the number of digits to the left of the decimal point and a sign, the number of pound signs (#) can be increased by one or the format control characters, plus or minus, may be used to provide space for the sign. Using the previous example and changing line 100 to:

2)

```
100 :  +.####     -#.##      +#####.
```

produces:

```
-.1343      3.00         +123.
+.0018     -2.00        -4596.
```

Floating Point Format:

A floating point format consists of an optional plus or minus sign followed by a string of pound signs (#), with a decimal point preceding or embedded following the #'s, and followed by five ↑ characters.

As with the fixed point format, a plus sign (+) or a minus sign (-) can be used in the format specification to reserve a place for the sign. A plus sign always causes a sign to be printed; a minus sign causes a sign to be printed only if the value is negative. If there is no place reserved for the sign, and the value is negative, the number of places to the right of the decimal point is decreased by one, and the minus sign is printed.

The associated value is shifted to fit the format specification and the exponent is adjusted to reflect the shifting.

After the associated value is shifted, it is rounded to match the number of pound signs following the decimal point.

Examples:

1)
```
100 READ Z1,X2,C3
110 PRINT USING 120,Z1,X2,C3
120 : ##.##↑↑↑↑↑    #.#######↑↑↑↑↑     .####↑↑↑↑↑
130 NØDATA 999
140 GØTØ 100
400 DATA 94.614,3.1415926536,-011466
410 DATA 4.118,439.8143698,1.00149
999 END
```

produces:

```
94.61E+000    3.1415927E+000    -.115E+005
41.18E-001    4.3981437E+002    .1001E+001
```

In the above example, the first value in column three has had the number of digits to the right of the decimal point reduced to accommodate the minus sign. Changing line 120 to:

```
120 : ##.#↑↑↑↑↑    #.#######↑↑↑↑↑     +.####↑↑↑↑↑
```

produces:

```
94.6E+000    3.1415927E+000    -.1147E+005
41.2E-001    4.3981437E+002    +.1001E+001
```

If the number of variables to be output is less than the number of field specifications in the IMAGE line, the print line ends at the first unused specification.

The following examples illustrate the use of the IMAGE statement with the PRINT USING statement previously described.

2)

```
100 R1(1) = 225
105 A4 = R1(1) + (1/4) * R1(1)
110 X4 = .4515642
120 D2 = A4/3.13 *X4
150 :ØVER $### BUT LESS THAN $####    PAY $####.##
160 PRINT USING 150,R1(1),A4,INT(X4*100 + .5001)
170 PRINT USING 150,5*50,340,D2
999 END
```

produces:

```
ØVER $225 BUT LESS THAN $ 281    PAY $   45.00
ØVER $250 BUT LESS THAN $ 340    PAY $   40.58
```

Using the previous example and changing line 170 to:

```
170 PRINT USING 150,5*50,340
```

produces:

```
ØVER $225 BUT LESS THAN $ 281    PAY $   45.00
ØVER $250 BUT LESS THAN $ 340    PAY $
```

If the number of variables to be output is greater than the number of format specifications in the IMAGE line, the format specifications are reused until all the variables have been output.

3)

```
10 FØR I = 1 TØ 3
20 PRINT USING 40,SQR(I),I
30 NEXT I
40 :###.### IS SQR ØF        Image has seven trailing blanks.
99 END
```

produces:

```
1.000 IS SQR ØF          1.000 IS SQR ØF
1.414 IS SQR ØF          2.000 IS SQR ØF
1.732 IS SQR ØF          3.000 IS SQR ØF
```

Alphanumeric Format

An alphanumeric format consists of one or more pound signs (#).

Strings are left-justified with blank padding to the right when the string is shorter than the format specification.

Strings which are longer than the format specification are truncated on the right.

Examples:

1)

```
10 A$="NEW"
20 B$="VALUES"
30 PRINT USING 50,A$
40 PRINT USING 50,B$
50 : STRING IS /####/
RUN
```

produces:

```
STRING IS /NEW /
STRING IS /VALU/
```

Strings may be concatenated (linked together) for printing by using the specification control character plus (+) between pound signs (#).

2)

```
10 A$="BAT"
20 B$="MAN"
30 : ###+###
40 PRINT USING 30,A$,B$

RUN
```

produces:

```
BATMAN
```

SETDIGITS STATEMENT

Format:

SETDIGITS ne

ne a numeric constant, variable or expression

The SETDIGITS statement may be used to specify the number of significant digits to be output in subsequent PRINT statements when the default formatting is used. Use of the SETDIGITS statement allows the user to obtain data printed up to 14 significant digits.

The value assigned by SETDIGITS is truncated to an integer in the range 1 to 14. Numbers are printed within the defined significance until:

a. the end of the program;

b. another SETDIGITS statement is encountered.

Note that any relevant sign or exponent is still printed even if a SETDIGITS value of 1 is in effect.

Example:

```
100 LET A = 55.45454545
110 PRINT "A =55.45454545 AND IS NØRMALLY ØUTPUT AS";A
120 PRINT "SETDIGITS","VALUE ØUTPUT"
130 FØR N = 1 TØ 10
140 SETDIGITS N
150 PRINT N,A
160 NEXT N
170 END
```

produces:

```
A =55.45454545 AND IS NØRMALLY ØUTPUT AS 55.4545
SETDIGITS        VALUE ØUTPUT
 1                6E+1
 2                55.
 3                55.5
 4                55.45
 5                55.455
 6                55.4545
 7                55.45455
 8                55.454545
 9                55.4545455
 10               55.45454545
```

MARGIN STATEMENT

Formats:

MARGIN ne_2

MARGIN #ne_1, ne_2

ne_1	a numeric constant variable or expression which evaluates to a file ordinal which is associated with a file name.
ne_2	a numeric constant variable or expression which is truncated to the nearest integer, and which must be between 15 and 160.

This statement defines the right-hand margin and overrides the default margin of 75. The use of the MARGIN statement permits the building of lines up to 160 characters in length for printing on wide carriage terminals or printers.

When the MARGIN statement is used without specifying a file, it applies to the terminal or the printer.

The MARGIN value (ne_2) may be varied from 15 to 160, and its value affects all PRINT statements to the associated file or the terminal until another MARGIN statement is executed. If a string is longer than the defined margin, it is broken into pieces so that as many full lines as required are used.

Examples:

1) 200 MARGIN #6, 136

2) 310 MARGIN I*J/K

In the first example, the right margin is set at 136 for output to a file with an ordinal of 6 previously specified by a FILE statement. In the second example, the expression is evaluated, truncated and used as the right margin value for PRINT statements.

3)
```
100 MARGIN 17
110 PRINT "ABCDEFGHIJKLMNØPQRSTUVWXYZ",1.75,88
```

produces:

```
ABCDEFGHIJKLMNØPQ
RSTUVWXYZ
 1.75
 88
```

MATRIX OPERATIONS 2.5

Although it is possible to construct programs to perform matrix operations with the ordinary statements of the language, BASIC also provides a set of statements explicitly for matrix operations.

Matrix operations are restricted to one- and two-dimensional numeric arrays. A one-dimensional array is always treated as a column vector. To obtain a row vector, a two-dimensional array must be specified with only one row.

Example:

```
100 DIM X(19), Y(1, 11)
```

This statement generates matrix X with one column of 19 elements, and matrix Y with one row of 11 elements.

It is not necessary to provide dimensional specification for each matrix used in a BASIC program. This can be accomplished by a DIM statement (Explicit Dimensioning) or within the matrix statement (Implicit Dimensioning). In BASIC a default limit of 10 x 10 is permitted for implicitly dimensioned matrices.
Example:
The following accomplish the same:

EXPLICIT DIMENSIONING	IMPLICIT DIMENSIONING
DIM A(2, 3), B(4, 5) . . . MAT READ A, B	MAT READ A(2, 3), B(4, 5)

MATRIX ARITHMETIC

Matrix Statement	Arithmetic Operation
MAT $m_1 = m_2 + m_3$	Addition
MAT $m_1 = m_2 - m_3$	Subtraction
MAT $m_1 = m_2 * m_3$	Multiplication
MAT $m_1 =$ (expression)$* m_2$	Scalar multiplication by value of an expression

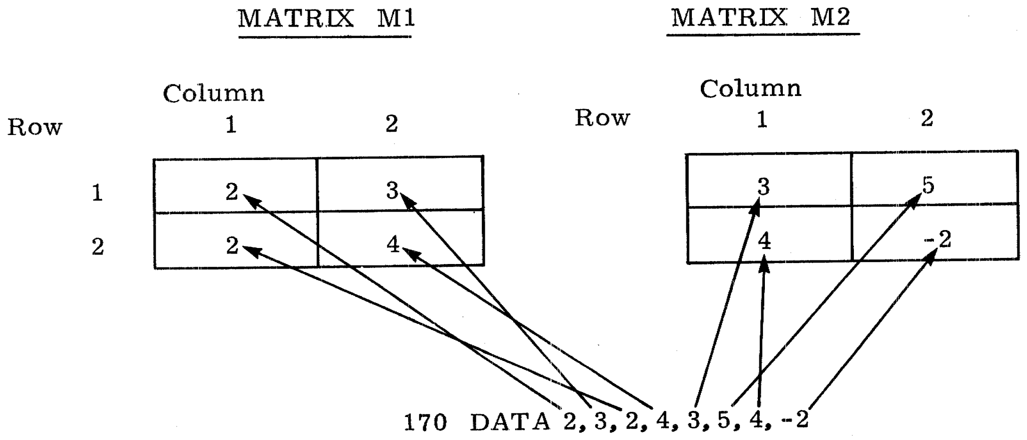
In each of the above statements, m may be any matrix identifier. The array dimensions must conform for each operation, and none of the operands of a matrix multiplication can be used as the result of that matrix multiplication; for example, MAT $m_1 = m_1 * m_2$ is not allowed.

. Example:

For a description of the READ and PRINT statements see the following paragraphs:

```
100 DIM M3(2,2),M4(2,2),M5(2,2),M6(2,2)
110 MAT READ M1(2,2),M2(2,2)
120 MAT M3=M1+M2
130 MAT M4=M2-M1
140 MAT M5=M1*M2
150 MAT M6=(2)*M1
160 MAT PRINT M1;M2;M3;M4;M5;M6
170 DATA 2,3,2,4,3,5,4,-2
180 END
```

At line 110, when READ is executed, the matrices dimensioned at line 100 are filled with the data in the internal storage block, line 170.



This program produces

```
         {   2   3
M1       {
         {   2   4        +
         {   3   5
M2       {
         {   4  -2        +
         {   5   8
line 120 {
         {   6   2
         {   1   2
line 130 {
         {   2  -6
         {  18   4
line 140 {
         {  22   2
         {   4                  6
line 150 {
         {   4                  8
```

Note that the two input matrices (M1 and M2) are dimensioned in the READ statement while the four computed matrices (M3, M4, M5 and M6) are dimensioned in a DIM statement.

MATRIX FUNCTIONS

Matrices can be inverted or transposed by using the inversion (INV) and transpose (TRN) functions. The matrices within a statement must conform, and inversion and transposition in place is not allowed. The maximum size of an inversion matrix is 50 x 50.

Formats:

MAT m = INV(a)

MAT m = TRN(a)

m	- matrix identifier
a	- numeric array variable

Example:

For a description of the MAT READ and MAT PRINT statements, see the following paragraphs:

```
100 DIM I(2,2),T(2,2)
110 MAT READ C(2,2)
120 MAT I=INV(C)
130 MAT T=TRN(C)
140 DATA 2,3,42,4
150 MAT PRINT I;T;
160 END
RUN
 2 -1.5

-1  1


 2  2

 3  4
```

The following matrix statements may be used to generate a matrix of all zeros, all ones, or an identity matrix (assign ones to the elements along the principal diagonal and zeros elsewhere).

MAT m = ZER[(ne$_1$ [, ne$_2$])]	-	generates a matrix of all zeros.
MAT m = CON[(ne$_1$ [, ne$_2$])]	-	generates a matrix of ones.
MAT m = IDN [(ne$_1$ [, ne$_2$])]	-	generates an identity matrix.

m	matrix identifier
ne	numeric expression, constant, or variable. This value specifies the desired dimensions.

Items enclosed in brackets are optional.

Example:

```
10 Y=1
11 X=2
12 MAT Z=ZER (Y*4,6)
13 MAT B=IDN(X**2,X**2)
14 MAT A=CØN
15 MAT PRINT Z;B;A;
16 END
READY.

RUN

 0  0  0  0  0  0

 0  0  0  0  0  0

 0  0  0  0  0  0

 0  0  0  0  0  0


 1  0  0  0

 0  1  0  0

 0  0  1  0

 0  0  0  1


 1  1  1  1  1  1  1  1  1  1

 1  1  1  1  1  1  1  1  1  1

 1  1  1  1  1  1  1  1  1  1

 1  1  1  1  1  1  1  1  1  1

 1  1  1  1  1  1  1  1  1  1

 1  1  1  1  1  1  1  1  1  1

 1  1  1  1  1  1  1  1  1  1

 1  1  1  1  1  1  1  1  1  1

 1  1  1  1  1  1  1  1  1  1

 1  1  1  1  1  1  1  1  1  1
```

MATRIX REDIMENSIONING

Matrices can be redimensioned, i.e., length may be reset through use of the MAT READ statements, MAT INPUT statements or DIM statement. Expressions can be used to redimension the matrix dynamically at execution time. Expressions are evaluated and used to reset the upper limit to the number of elements in the matrix. A matrix cannot be redimensioned larger than its initial value (see the example which follows) or redimensioned to change the number of dimensions. A matrix used in an identity statement must be redimensioned to a square.

Example:

This example illustrates an attempt to redimension a matrix larger than its initial dimension. For a description of the MAT READ and MAT PRINT statements see the following paragraphs:

```
100 DIM M1(4,4)
110 MAT READ M1(2,2)
120 DATA 1,2,3,4,5,6,7,8,9,0,1,2,3,4,5,6,7,8,9,01,2,3,4,5,6,7,7,9,0
130 MAT PRINT M1
140 DIM M1(3,3)
150 MAT READ M1(3,3)
160 MAT PRINT M1
170 MAT READ M1(5,5)
180 MAT PRINT M1
190 END

RUN
```

produces:

```
1               2

3               4


5               6               7

8               9               0

1               2               3


MATRIX DIMENSIØN ERRØR   AT   170
```

MATRIX INPUT/OUTPUT

The following matrix input/output conventions are used by BASIC. In the formats, the parameters m_1, m_2 and m_3 represent the names of matrices. Examples using matrix read and write statements are given in the first part of this section. An example of matrix file manipulation is given in the last part of this section.

MATRIX READ STATEMENT

Formats:

MAT READ m_1, m_2, m_3
MAT READ #ne, m_1, m_2, m_3

ne	numeric expression, constant, or variable
m_1-m_3	matrix identifier.

These statements completely fill the matrices specified from the internal data file or the specified file, respectively. The numeric value which results from the evaluation of argument "ne" is used as an ordinal and identifies the requested file. To read a file with this statement it must be in binary form; i.e., it must have been created by a MAT WRITE or WRITE statement. Matrices are read in row order.

MATRIX INPUT STATEMENT

Formats:

MAT INPUT $m_1, m_2, m_3, \ldots$

MAT INPUT #ne, $m_1, m_2, m_3, \ldots$

ne	-	numeric expression, constant, or variable
m_1-m_3	-	matrix identifier.

With the MAT INPUT statement, matrices can be created at the terminal, one row at a time. A question mark displayed at the terminal indicates that the user should type in one row of the matrix. The system continues to display question marks until the requirements of the matrix are satisfied. If the user attempts to enter less than or more than one row of a matrix, a diagnostic is issued. The user must then complete the row by typing in the missing data (if less than one row was entered) or reenter the entire row if too much data was entered.

The MAT INPUT #ne statement accepts matrices in row order from the specified file, which must have been created with a MAT PRINT statement.

Matrix redimensioning is allowed for either type of matrix input statement.

Example:

```
10 MAT INPUT A(3,3)
20 MAT PRINT A;
30 END
RUN
```

This produces the printout:

```
? 1,2,3,4
 TØØ MUCH DATA -RETYPE AT   10
? 1,2,3
? 4,5,6
? 7,8
 NØT ENØUGH DATA, TYPE IN MØRE     AT   10
? 9
 1   2   3

 4   5   6

 7   8   9
```

MATRIX PRINT STATEMENT

Format:

MAT PRINT m_1d m_2d m_3d . . . d

MAT PRINT #ne, m_1d m_2d m_3d . . . d

ne	numeric constant, variable, or expression
m_1-m_3	matrix identifier
d	delimiter (comma or semicolon); the final delimiter is optional.

These statements cause matrices to be printed out in row order using the same rules as the normal PRINT and PRINT #ne statements. The delimiter specifies the print zoning control for each element of the matrix. A blank line is automatically generated after each row.

MATRIX WRITE STATEMENT

Format:

MAT WRITE #ne, m_1, m_2, m_3 . . .

ne	numeric expression, constant, or variable
m	matrix identifier.

This statement writes the specified matrices in row order on the specified file. Files created by MAT WRITE are in binary form and can only be read by READ or MAT READ statements.

MATRIX FILE MANIPULATION

The following program creates a file containing two matrices. This file is read in and printed by the same program. A subsequent program references and prints out ALPHA. Following the programs is an analysis of program statements.

DEMO1

```
40 FILE #1="ALPHA"
50 MAT READ A(2,2),B(3)
60 MAT WRITE #1,A,B
70 DATA 1,2,3,4,5,6,7
80 RESTØRE #1
81 MAT READ #1,A,B
82 RESTØRE #1
83 MAT PRINT A;B
90 END
READY.

RUN
```

This produces the printout:

```
1   2

3   4


5

6

7
```

DEMO2

```
NEW,DEMØ2
READY.

NØDRØP
READY.

95 FILE #1="ALPHA"
100 MAT READ #1,A(2,2),B(3)
110 RESTØRE #1
120 MAT PRINT A;B
130 END
READY.

RUN

 1   2

 3   4


 5

 6

 7
```

Analysis of Program Statements

DEMO1

40 The numeric constant 1 is assigned to file ALPHA.

50 The READ statement dimensions two matrices, A and B, which it fills with values from the data statement (70).

60 A file, ALPHA, is written with the two matrices read in above. Compare this with using the simple WRITE in which the values had to be entered with a FOR-- NEXT loop.

70 The DATA statement supplies the four values for matrix A and the following three values for matrix B.

80 Once the data is written, the pointer is at the end of the file. To move the pointer to the beginning of data (rewind), use is made of the RESTORE statement.

81 The two matrices in the secondary file area are read back into the program.

82 The file is rewound for future use.

83 The values read in from file ALPHA are printed out in matrix format.

90 Specifies end of the program.

DEMO2

NODROP This system command must be used immediately after the specification of a new program in order to retain secondary files. Without this command, ALPHA would be lost as soon as the first statement of DEMO2 was entered.

95 The numeric value 1 is assigned to file ALPHA.

100 The file ALPHA is read in as two matrices.

110 ALPHA is rewound for future use.

120 A and B are printed out in matrix format.

If file ALPHA is to be referenced by a third program, the NODROP command will have to be used as before.

ERROR PROCESSING 2.6

The following statements are used for detecting and processing errors which occur during program execution.

ON ERROR STATEMENT

Format:

1) ON ERROR GOTO ln
2) ON ERROR THEN ln
3) ON ERROR

Where ln is a line number constant.

Formats 1 and 2 specify that control is to transfer to statement ln, if a subsequent run-time error occurs. This statement in conjunction with the Error Statement Line number (ESL) function and the Error Statement Message (ESM) function which allows the BASIC program to respond to run-time errors.

The statements at line ln can use ESL and ESM functions to determine where the error occurred and what the error was. After the error location and type is determined, appropriate action to process the error can be taken; execution can be reinitiated at any point in the program.

Normal error processing is suppressed when the ON ERROR statement (Formats 1 and 2) has been executed. Normal error processing is reinstated by the execution of ON ERROR without target (Format 3) statement or after control has been transferred to the specified statement ln as a result of an error.

JUMP STATEMENT

The JUMP statement transfers control to the statement at the line number determined by the value of an arithmetic expression.

Format:

```
    JUMP ne
       ne        numeric expression, constant or variable.
```

The expression is evaluated and truncated to an integer value; control is transferred to the statement at the resultant line number provided it exists. If the statement does not exist, a diagnostic is issued. A JUMP statement cannot refer to a REM statement.

The JUMP statement is designed to be used in error processing routines where line numbers are assigned to variables by use of the NXL and ESL functions. It should never be used in place of a GOTO statement.

ESL FUNCTION

The Error Statement Line number (ESL) function yields the line number of the statement which caused the most recent program execution error.

Format:

```
    ESL(x)
       x         is a dummy variable.
```

When an execution error has not occurred or after the execution of an ON ERROR statement, the function yields the value -1. Thus in processing errors the function value should be saved before issuing another ON ERROR statement.

Example:

```
    Y = ESL(x)
```

The line number of the error statement is saved in Y.

ESM FUNCTION

The Error Statement Message (ESM) function yields the error number associated with the most recent program execution error.

Format:

```
ESM(x)
```

x	is a dummy variable.

When an execution error has not occurred or after the execution of an ON ERROR statement, the function yields the value -1. The function value should be saved before issuing another ON ERROR statement.

Example:

```
Z = ESM(x)
```

The message number of the error is saved in Z.

All errors and their corresponding error numbers are listed in appendix B.

NXL FUNCTION

The Next Line Number (NXL) function yields the line number of the next statement after the statement whose line number is specified in the argument.

Format:

```
NXL(x)
```

x	is a constant, variable, function, or numeric expression.

The value of x is truncated to an integer. This function can be used to determine at which statement execution is to resume in the event of an error. Function ESL(x) will return the line number of the statement which caused the error and NXL(ESL(x)) will return the statement number of the statement following the one which caused the error.

The NXL function cannot refer to a REM statement or return the line number of a REM statement. An attempt to refer to a REM statement or to any non-existent statement results in a fatal error "ILLEGAL LABEL".

JUMP NXL(ESL(X))

Execution of this statement passes the control back to the statement following the statement responsible for the error in the program.

The following example illustrates the use of the error processing method provided by BASIC.

The program:

```
090 ØN ERRØR GØTØ 210
100 LET X = 2
130 PRINT " READ ERRØR WILL BE PRØCESSED BY PRØGRAM"
140 READ X1,Y2,X3
150 PRINT "VALUES READ WERE";X1;",";Y2;"AND";X3
160 STØP
200 REM ERRØR PRØCESSING RØUTINE
210 X = ESL(X)
220 Y = ESM(X)
250 IF X = 140 THEN 300
260 PRINT "ERRØR NØT IN STATEMENT 140"
270 STØP
300 PRINT "ERRØR NUMBER #";Y;"DETECTED AT LINE #";X
310 JUMP NXL(X)
400 DATA 2.0,3.0,"STRING"
410 END
```

produces:

```
 READ ERRØR WILL BE PRØCESSED BY PRØGRAM
ERRØR NUMBER # 126 DETECTED AT LINE # 140
VALUES READ WERE 2 , 3 AND 0
```

090	Execution of this statement suppresses normal error processing and ensures that on a subsequent error, control will be transferred to statement 210.
140	If an error occurs in reading the data, control is transferred to statement 210. Normal error processing is then reinstated, i.e., if during further execution another error occurs, the program aborts.
210, 220	Value 140 is saved in X and value 126 is saved in Y. Further action can be taken based on the user requirements for processing errors. Value 126 is the error message number.
250	If error occurred in statement 140, execution control is transferred to statement 300.
310	A jump is made to statement 150 and normal execution continues. In case of another error during execution, the job will abort.

TERMINAL OPERATIONS UNDER SCOPE 2.7

SCOPE SYSTEM

The INTERCOM system, operating in conjunction with SCOPE permits multi-user access to CYBER and 6000 Series computers. INTERCOM commands and directives permit the terminal user to process BASIC programs interactively or to submit BASIC programs for execution from a remote terminal. The remote terminal may be any terminal supported by its respective INTERCOM version.

This section provides a description of appropriate Teletype (TTY) and CRT terminals supported by INTERCOM. Also provided are a description of TEXT EDIT and INTERCOM commands which are most frequently used by the BASIC programmer. Through use of EDITOR, the terminal user creates both data files and programs. To assist the user, a discussion pertaining to the creation and use of SCOPE data files and an example of a complete terminal session are included.

TERMINAL KEYBOARDS

The following paragraphs describe and illustrate the keyboards of those TTY and CRT terminals which are supported by INTERCOM. A dial-in procedure for the TTY Models 33 and 35 terminals is also provided.

TTY TERMINAL

The programmer types lines of information to the INTERCOM system on the TTY terminal. INTERCOM responds interactively to the TTY terminal. The Teletype keyboard (figure 2.7-1) for both models 33 and 35 resembles a standard typewriter keyboard. Special characters shown on the upper portion of the keys are entered by holding the SHIFT or CTRL key down while pressing the special character key. INTERCOM requires special function keys as well as special characters. The keys used and interpreted by INTERCOM are described below; information is included as to what is stored, what action is taken, and what is printed on the Teletype listing.

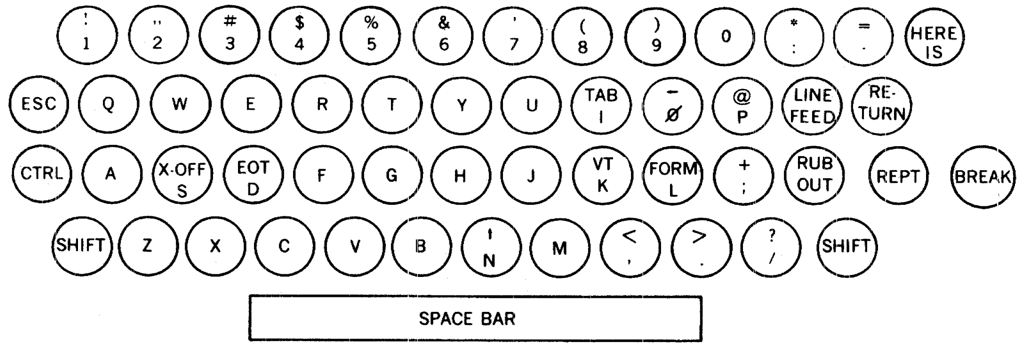


Figure 2.7-1. Typical Teletype Keyboard

SHIFT	SHIFT accesses the characters or functions shown on the upper portion of most Teletype keys. If pressed alone, SHIFT has no effect.
CTRL	This key is used to access the special function keys (TAB, X-OFF, EOT, etc.) and character and line delete functions. If pressed alone, it has no effect.
CTRL X	Pressing CTRL and X deletes the entire entry typed by the user since the previous RETURN. The entry is not erased from the printout, but it is ignored by INTERCOM. No character is stored or printed.
CTRL Z (ESC or ALT MODE)	CTRL Z (on some models ESC or ALT MODE also may be used) is pressed to interrupt current Teletype activity. The user then enters a directive: A (Abort Program), S (Discard Accumulated Output) or a line feed/carriage return to continue. TTY's connected via the 791 LCC require a carriage return to be entered after the desired control function.
CTRL H	Signals INTERCOM to erase the previous character from its input buffer. The Teletype listing is not erased. For example, if the user types FILEY and then backspaces and types S to replace the Y, the listing appears as FILEYS; however, the corrected command FILES is entered in the buffer. No character is stored or printed.

RETURN	The carriage return key signals, to INTERCOM, the end of a message. It also returns the Teletype printer carriage to its left-most position; the computer returns a line feed to advance the carriage to the succeeding line. No character is stored or printed.
LINE FEED	This key spaces to the next line. INTERCOM issues a carriage return to the beginning of the new line. No end-of-message signal is sent. This method provides for entering lines greater than 72 characters. No character is stored or printed.
SPACE	The space bar generates the space character. A blank is stored and printed.
REPEAT	Pressing the REPEAT key along with another character key produces character repetition for as long as the key is pressed; the character key may be released. If the repeated character requires use of the SHIFT or CTRL key, they should be pressed along with the REPEAT key and desired character key.
Alphanumeric	The alphanumeric keys are used to input commands, data, and programs. Each is stored and printed as the key is pressed.

DIAL-IN Procedure

Connecting TTY Models 33/35: If the telephone line is connected directly to the TTY, the following procedure is used:

- Turn on the TTY by setting the rotary power switch to the LINE position or by depressing the ORIG button (on those models having this feature).
- Dial the correct phone number. The system will request answerback, return a header, and request the user number.

There is a slight pause from the time the phone is answered until the TTY begins to type. This is caused because:

- A time delay is allowed, before any data transmission is attempted, to ensure that the line is settled.
- A fixed delay of about three seconds is allowed to ensure that answerback drum transmission is complete. This is necessary for all terminals, as the number of characters on the answerback drum are unknown at this time.

If the teletypewriter is connected to the system by an acoustic coupler, the following procedure is used:

- Turn on the TTY as described above.
- Turn on the acoustic coupler.
- Dial the correct phone number.
- When the connection has been made, that is, a constant high-pitched sound is heard in the receiver, place the receiver in the acoustic coupler. The system will request answerback, return a header, and request user number.

Connecting TTY Model 37: To connect Model 37 to INTERCOM, use the following procedure:

- Turn on the TTY by pressing the DATA button.
- Follow the procedure described for Models 33/35 depending on whether the TTY is connected directly to the system or connected to the system by an acoustic coupler.

CRT TERMINAL

On the terminal keyboard, the programmer types lines of information as messages to INTERCOM. The typed information is displayed on a cathode ray tube (CRT) screen. The INTERCOM system responds interactively to the CRT terminal. The following paragraphs describe the Model 713 terminal and 200 Series display terminals.

Model 713 Terminal

The 713 keyboard (figure 2.7-2) is compatible with the TTY keyboard discussed previously, the differences being the cluster of numeric keys to the right of the keyboard, similar to a calculator keyboard, and the row of function keys above the keyboard. The keys used and interpreted by INTERCOM are described below: information is included as to what is stored, what action is taken, and what the effect is on the display.

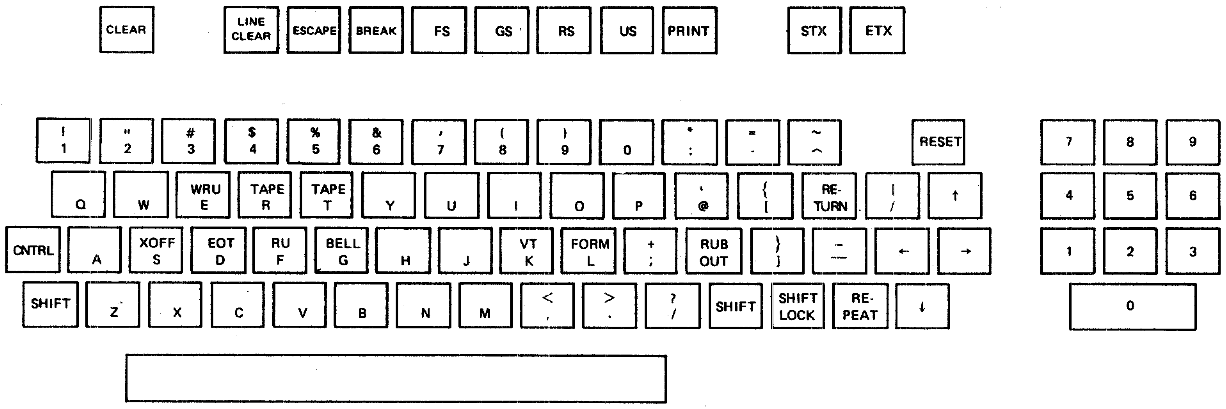


Figure 2.7-2. Model 713 Teletype Compatible Terminal

SHIFT and SHIFT LOCK	Two shift keys and the SHIFT LOCK key enable selection of upper case letters or the upper character on the double-character keys. Pressing SHIFT LOCK locks the keyboard in upper case position. Pressing the SHIFT key adjacent to the SHIFT LOCK key releases the keyboard from upper case operation.
CNTRL	This key is used with several special function keys (X, Z, H, etc.) and edit functions. If pressed alone, it has no effect.

CNTRL H	Pressing CNTRL and H signals INTERCOM to erase the previous character from its input buffer. The cursor moves back one character position without affecting displayed data.
CNTRL X	Pressing CNTRL and X deletes the entire entry typed by the user since the previous RETURN. The entry is not cleared from the display, but it is ignored by INTERCOM. The cursor remains in the same position.
RETURN	Signals, to INTERCOM, the end of an entry. The cursor resets to the first character position of the current line; INTERCOM returns a line feed to advance the cursor to the succeeding line. No character is stored or displayed.
LINE FEED	This key spaces to the next line, INTERCOM issues a return to reset the cursor to the beginning of the new line. No end of message signal is sent. This method allows lines of any length to be entered. The cursor is reset automatically to the beginning of the next line upon reaching the 80th character position of the current physical line.
SPACE	The space bar enters a space character above the cursor. The cursor moves forward one character postition.
CLEAR	Removes all displayed data from the screen. If the terminal is in scroll format mode, the cursor resets to the first character position of the current line. If the terminal is in page mode, the cursor resets to the first character position of the top line.
LINE CLEAR	Removes all displayed data from the cursor position to the end of the line only when the terminal is in page format mode. The cursor does not move and only the line with the cursor is affected. Scroll mode disables this function.
RESET	Pressing the RESET key does not affect displayed data. If the terminal is in scroll format mode, the cursor resets to the first character position of the current line. If in page mode, cursor is reset to the first character position of the top line.
CNTRL Z	CNTRL and Z are pressed to interrupt current Teletype activity. The user then enters a directive: A (Abort Program), S (Discard Accumulated Output) or a line feed/return to continue. The cursor remains in the same position. When Teletypes are connected through an LCC, a carriage return must be entered after the control function, A/S.

200 Series Display Terminals

A 200 Series display terminal is equipped with a cathode ray tube (CRT) display screen, a display controller, and a keyboard. In addition, a 224 Card Reader and/or 222 Line Printer may be included to provide a complete 200 User terminal. A 711 display terminal is similar to a 200 Series CRT, but peripherals may not be attached. The following is a brief description of these terminals. (See figures 2.7-3 and 2.7-4.)

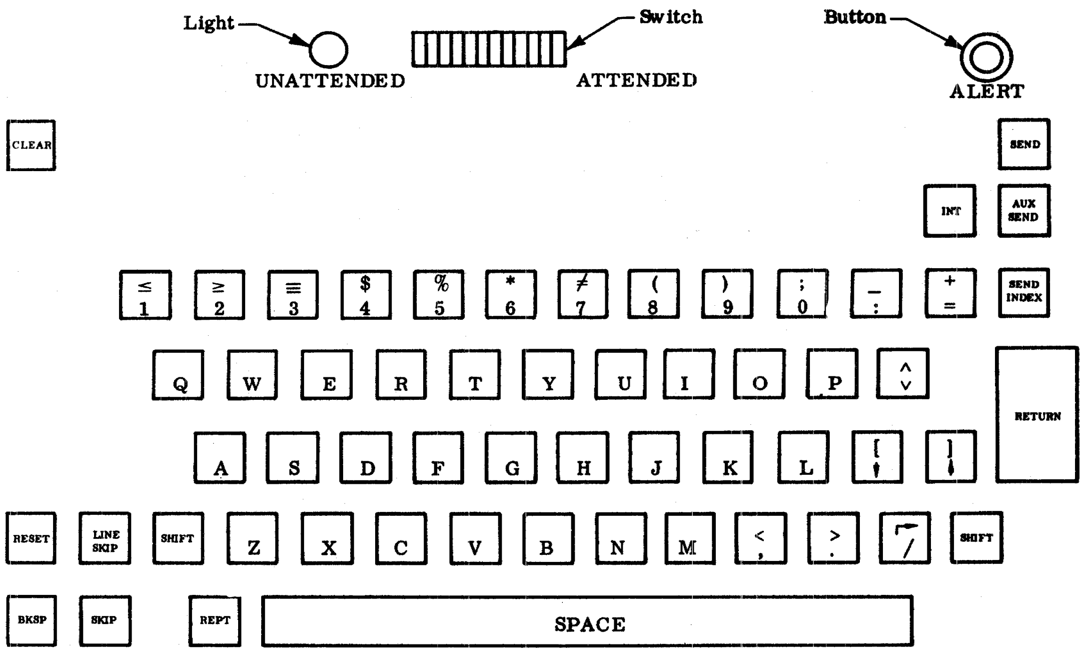


Figure 2.7-3. CRT Display Keyboard (217, 214 Type Terminals)

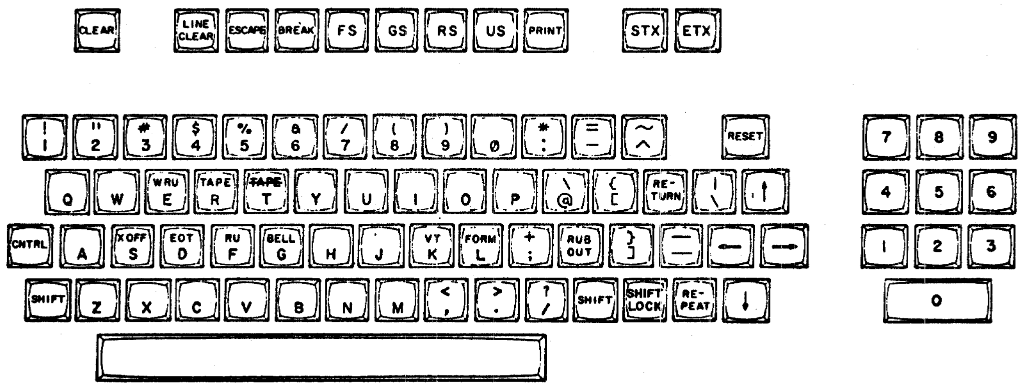


Figure 2.7-4. CRT Display Keyboard (710 Series)

DISPLAY SCREEN MARKERS

Three markers appear automatically on the display screen to facilitate message transmission: the line indicator, the entry marker, and the message terminator.

LINE INDICATOR

This small solid block appears on the screen to the left of the line on which the current input message is to appear. The location of this indicator is dependent on the type of display terminal.

On the 214-11, 214-12, 217-11, and 217-12 display terminals, this small block appears on the screen to the left of the line on which the next message will start. The indicator moves automatically to the next line at the completion of an output operation to the display.

On the 217-13 and 217-14 display terminals, this small block may appear in any character position on the screen. All output messages to the display are terminated with a line indicator at the end of the last output line.

ENTRY MARKER___

The entry marker is the first of a series of dashed underlines; it indicates the next open character position on the input line. The series extends to the right margin of the screen to indicate remaining character positions. Initially, the underlines extend from the line indicator to the right margin of the top line of the screen.

As a symbol is entered, the underline in that position disappears, and the next underline in the series becomes the entry marker. Each time a message is sent to INTERCOM a message terminator appears and the line indicator and entry marker advance to the next line position.

MESSAGE TERMINATOR ▲

A solid delta symbol is displayed to indicate the end of a message. This symbol appears when the SEND key is pressed.

KEYS

The following special keys are used to operate the CRT terminal:

SEND KEY

This key is equivalent to the RETURN key on the TTY terminal. It indicates the end of a message by displaying the message terminator (▲) on the screen. All information between the line indicator (■) and the message terminator is transmitted to INTERCOM.

BKSP KEY

The BKSP key backspaces the entire marker one character position so that the user can type over that position.

CLEAR KEY

Pressing this key causes the contents of the screen to be erased and positions the line indicator and entry marker at the top of the screen. Use of this key has no effect on execution or compilation.

COMMAND MODE

Once the user has logged in, the system responds by displaying:

```
COMMAND-
```

at the terminal. At this point the user may enter any INTERCOM commands including SCOPE control cards.

After a command is entered, control is given to the program which processes that command; control remains with that program until processing terminates or the user voluntarily leaves the program. Command mode is re-established, and INTERCOM is ready to accept another command.

INTERCOM performs two kinds of tasks for the user. Under command mode, INTERCOM loads a library (utility) program that processes the given command. As most SCOPE control cards are also INTERCOM commands, such control card messages may be sent from the terminal and executed. The syntax of INTERCOM commands is similar to that for SCOPE control cards.

For a description of the INTERCOM log in sequence, see USING BASIC from INTERCOM at the end of this section.

INTERCOM COMMANDS

INTERCOM commands of particular relevance to the BASIC user follow. Although the EDITOR command is part of the INTERCOM command repertoire, a discussion of this command and edit utility is discussed later in this section.

BASIC COMMAND

The BASIC command allows the user to execute a BASIC program without using and EDITOR utility routine. See section 2.9 of this manual for a discussion of parameters associated with this command.

Format:

BASIC (p_1, p_2, ...p_n)

With this command input from or output to system or user files can be specified. To permit terminal I/O, files INPUT and OUTPUT must be explicitly connected by use of the CONNECT command (discussed later).

BRESEQ COMMAND

BRESEQ is an INTERCOM Version 4.1 oriented program to resequence a BASIC program. The user controls how the file is resequenced; he can specify the starting value and increment for resequencing. The user enters:

Format:

```
BRESEQ (file,param1,param2)
```

The file to be resequenced is specified by file. The parameters are optional; param1 is the starting value for the file, and param2 is the increment. If no values are specified, param1 is set by default to 100, param2 has a default value of 10. If only one parameter is specified, it is assumed to be the starting line number for the new file; and the increment is set to 10 by default.

To resequence a file named MYPROG, starting at line number 9000, in increments of 10, the user enters:

```
BRESEQ(MYPROG,9000,10)
```

The following entry would accomplish the same:

```
BRESEQ(MYPROG,9000)
```

Only the initial line number is specified; the increment is 10 by default.

Using the following format:

```
BRESEQ(MYPROG)
```

the resequenced file would start at line number 100 and have increments of 10.

Since BRESEQ is intended primarily for the INTERCOM user, it uses files formatted by EDITOR. A program entered directly into EDITOR while in "FORMAT, BASIC" will be correctly formatted for resequencing. For a program not entered through EDITOR, a call to the INTERCOM routine CONVERT must be made.

When in "FORMAT, BASIC" and after resequencing a file, the EDITOR directive "EDIT, file-name" must be used to enter the new file into the edit file. Otherwise the unresequenced old file remains. The resequenced file retains its original name.

EFL COMMAND

The user can alter the system field length (in octal words) by typing:

```
EFL, field-length
```

ETL COMMAND

The user can alter the system default time limit (in octal seconds) by typing:

```
ETL, time-limit
```

FILE COMMANDS

The following commands are related to the use of local and permanent files:

CONNECT Command

The user can request that specific files be designated for terminal interaction by entering the CONNECT command.

Format:

```
CONNECT(filename-1, filename-2, ..., filename-n)
```

Input and output will be routed to and from the terminal when the named files are subsequently read or written. The file names may be INPUT and OUTPUT, as well as any other files.

Each time a connected input file is referenced in the source program, the system waits for input from the terminal. Each time a connected output file is referenced in the source program, the output is printed or displayed at the terminal. It is not saved.

When input is expected, the system waits for the user to enter it from the keyboard. For BASIC programs, the INPUT statement displays a question mark at the terminal when user input is expected.

The CONNECT command need not be entered when programs are executed with the EDITOR RUN command; the files INPUT and OUTPUT are connected automatically. To connect any other input/output files to his terminal, the user may enter the CONNECT command.

A user may create his own object program through use of the BASIC command with I, K, and B options specified and later execute the object program with the following:

```
CONNECT(filename-2, INPUT)
XEQ(filename-2)
```

The BASIC command is described in section 2.9 of this manual; XEQ is described later in this section.

DISCARD Command

Used to purge a permanent file saved by the STORE command. If the file is attached by FETCH or ATTACH, only the file name is required.

```
DISCARD, fn, id, pp.
```

fn	-	permanent file name
id, pp	-	1 to 9 alphanumeric characters relating to the STORE command id and pp parameters

DISCONT Command

This command disconnects a file from the terminal.

Format:

```
DISCONT(filename-1, filename-2, . . . , filename-n)
```

A file specified in this statement will no longer be directed to the terminal, but it will be directed to and reside on allocatable mass storage.

FILES Command

The user can obtain a list of files accessible to him by typing:

```
FILES
```

User private files and attached permanent files are listed. User private files are local files created by the individual user. They can be read, altered, or deleted only by the originator. Permanent files are mass storage files, the location and identification of which are always known to the INTERCOM system. Permanent files are protected from unauthorized access according to privacy controls specified by the creators of the files.

Example:

```
COMMAND- FILES.
--LOCAL FILES--
 $INPUT            $OUTPUT          *FILA          $FILB
 LGO                FORTX           $TEST1
COMMAND-
```

a " $ " prefixed to the file name specifies a connected file.
an "*" prefixed to the file name specifies a permanent file.

FETCH Command

Permits the user to access permanent files saved by the STORE command. Through use of this command a permanent file is made local; however, the FETCH command must be used before any reference to the file by SCOPE or INTERCOM commands. Files saved under the CATALOG command cannot be accessed by FETCH unless they were saved in a manner compatible with the STORE command. To modify a FETCH'd file requires a write-in-place; therefore, the user should make modifications on a scratch file, then DISCARD the old file, and STORE the scratch file under the old file name.

Format:

```
FETCH, fn, id, pp.
fn      file name to be fetch'd
id, pp  1-9 alphanumeric characters relating to STORE command.
```

RETURN Command

Unwanted private files can be replaced with this SCOPE control card command. If the file referenced by this command is an attached permanent file, the file is returned to permanent mass storage and deleted from his list of private files.

Format:

```
RETURN, filename-1, filename-2, . . . , filename-n.
```

STORE Command

Catalogs a users local file as a permanent file. When catalogued, the file remains as an attached permanent file.

Format:

```
STORE, fn, id, pp.

fn       a local file name
id, pp   1-9 alphanumeric characters
     -   id  user identifier
     -   pp  privacy code.
```

XEQ Command

With this INTERCOM Version 4.1 command, the user can load binary programs from his own local files or user libraries and submit these programs for execution or construct absolute overlays from them. After user specified programs are loaded from the currently defined library set, the loader attempts to satisfy remaining unsatisfied externals. Specified file names, including user libraries, must be user local file names or local file names of attached permanent files. To initiate program loading, the user enters:

```
XEQ
```

The system responds:

```
OPTION=
```

The user then enters one of the options described below. If he does not enter EXECUTE, NOGO, or a file name, the system again responds:

```
OPTION=
```

This sequence will continue until the user enters EXECUTE, NOGO, or a file name to initiate the loading operation. The user may enter END at any time to exit from the XEQ command.

Command Options

- LOAD=filename-1, filename-2, . . . , filename-n

  Files are specified whose contents are to be loaded. The file names can be specified in one of the following forms:

filename	Installation defined rewind parameter is assumed
filename/R	Rewind before loading
filename/NR	No rewind before loading

  LIBLOAD=libname, ename-1, ename-2, . . . , ename-n.

  The loader examines the directory of the user library specified by libname and selects programs containing the entry points specified by ename.

- SLOAD=filename, pname-1, pname-2, . . . , pname-n

  The loader searches through the specified file and loads only those programs specified by pname. The file name may be entered in one of the following forms:

filename	Installation defined rewind parameter is assumed
filename/R	Rewind before loading
filename/NR	No rewind before loading

- SATISFY=libname-1, libname-2, . . . , libname-n

  The loader searches through the libraries specified by libname to fill unsatisfied external references without completing the load. Further loading may be specified following the SATISFY option. If no parameters are specified, the currently defined library set is assumed.

  filename, param-1, param-2, . . . , param-n.

  The named file is rewound and loaded. Optional execution parameters specified by param are passed to the loaded programs in RA + 2 through RA + 63.

- EXECUTE=ename, param-1, param-2, . . . , param-n

  The loaded programs are executed beginning at the entry point specified by ename; if ename is omitted, execution begins at the last transfer address encountered by the loader. Optional execution parameters specified by param are passed to the loaded program in RA + 2 through RA + 63.

- NOGO=filename, ename-1, ename-2, . . . , ename-n

  Loading is forced to completion without execution. The file name specified becomes the file on which loaded programs are saved as an absolute overlay. The filename parameter is only applicable to non-segmented relocatable loads. Otherwise, it is ignored. Optional entry point names to be included in the overlay header may be specified by ename, but they are ignored if filename is omitted.

LDSET Option

Loading operations in INTERCOM 4.1 are executed by the LDSET statement which provides user control with a variety of options. Options specified through LDSET apply to the current load only; loader completion statement (name call, EXECUTE, or NOGO) terminates their effect. Each LDSET statement can be used to set several options, or several LDSET statements may be used.

Detailed information on the loader directives and options may be obtained from: LOADER Reference Manual and INTERCOM Version 4 Reference Manual.

EDITOR

With the INTERCOM editor utility the user can create, examine, and modify coded sequential files from a remote terminal. Access to text editor is accomplished by entering EDITOR immediately following the system COMMAND request.

Example:

```
        COMMAND-EDITOR
```

Once the user replies by entering EDITOR, he may enter any EDITOR, SCOPE,or INTERCOM commands. The EDITOR command mode response (..) will be displayed after each command is processed. When entering a program or creating a data file, after each line if text is entered, editor responds with a line feed. In either case another command may then be entered.

CREATING A PROGRAM UNDER EDITOR

When creating a program under EDITOR, the user must enter a FORMAT command immediately after the system accepts the EDITOR request (COMMAND-EDITOR). This command establishes installation defined format specifications depending on the language option specified.

Format:

$$\text{FORMAT} , \left\{\begin{array}{l}\text{BASIC}\\ \text{COBOL}\\ \text{COMPASS}\\ \text{FORTRAN}\end{array}\right.$$

The tabular column positions, valid tabulating character, and maximum character count per input line are controlled by this specification. Specifications established with this command remain in effect for the duration of the user's session with EDITOR, or until changed by the user.

Once the FORMAT command is accepted (apparent by two dot displayed on the line following the FORMAT command), the user enters program text in the form "line-number text". If the lines entered exceed the specified character count, they are truncated to the maximum permitted; and a message is displayed at the terminal. When the BASIC option is selected, the EDITOR line number is duplicated and becomes the BASIC statement number and part of the text.

In the "FORMAT, BASIC" mode, use of the EDIT with SEQ, CREATE, ADD, and RESEQ EDITOR commands is not permitted.

Example:

To enter a BASIC program into the edit file under BASIC format, request a listing of the edit file, and save the program as a local file named BASFIL:

```
••FØRMAT,BASIC                          user enters FORMAT, BASIC
••200 FØR X=1 TØ 100                    user enters BASIC statements line
400 PRINT "X=";X                        by line
600 PRINT "X**2=";X**2
800 NEXT X
1000 END
LIST,A                                  user enters LIST, A
   200=200 FØR X=1 TØ 100               system lists contents of edit file
   400=400 PRINT "X=";X
   600=600 PRINT "X**2=";X**2
   800=800 NEXT X
  1000=1000 END
••SAVE,BASFIL                           user enters SAVE, filename
••                                      EDITOR is ready for next command
```

Program Execution

When the BASIC program is ready for execution, it may be submitted to the BASIC compiler by typing.

RUN, BASIC

If the program executes correctly, the results are returned to the terminal. For an example of a BASIC program execution, see the sample terminal session at the end of this section.

USING DATA FILES

Execution aborts when a BASIC program attempts to read a data file which was created in BASIC Format (sequence numbers prefixed to each line of text). To create data files which are acceptable to a BASIC program, the user must select the COBOL, FORTRAN, or COMPASS FORMAT statement option, enter the file data in "line no. text format", and after the file is created, save the file (becomes a permanent file) without sequence numbers using the "SAVE, lfn, NO" command. Any format option other than BASIC assigns only the EDITOR line number to each line of text; this number is then stripped when the no sequence number option (NO) of the SAVE command is selected. To re-edit a permanent file which was saved without sequence numbers, enter the " EDIT , filename" command (described later in this section). This command automatically assigns sequence numbers to each line in the file. The user may now modify the file using edited utility commands and save the modified file using the SAVE, lfn, NO command.

DELETE Command

The user enters this command to delete lines in the edit file.

$$\text{\underline{D}ELETE,}\quad \left\{\begin{array}{l}\text{\underline{A}LL}\\ \text{line-1 [,}\left\{\begin{array}{l}\text{line-2}\\ \text{LAST}\end{array}\right\}\text{]}\\ \text{\underline{L}AST}\end{array}\right\}\quad \left[\text{,/text/ }\left[\text{,(col-1 [,col-2])}\right]\text{ [,\underline{U}NIT]}\right]\text{ [,\underline{V}ETO]}$$

<u>A</u>LL	Keyword; all lines in the edit file are deleted or searched for the text search string.
line-1	Line number; 1-6 digits, from 1 to 999999; first or only line to be deleted or searched.
line-2	Line number; 1-6 digits, from 2 to 999999; last line to be deleted or searched in a range beginning at line-1.
<u>L</u>AST	Keyword; as first parameter, causes last line in edit file to be deleted or searched; as second parameter, causes deletion or search of lines beginning at line-1 through the last line in the file.
/text/	Text search string; 1-20 characters delimited by slashes or an equivalent delimiter; file is searched for this text string (search may be restricted to range of line and column numbers). Lines containing this string are deleted from edit file.
col-1	Column number; 1-3 digits, from 1 to 510; first or only column number of text string search. Must be preceded by a left parenthesis and followed by either col-2 or a right parenthesis.
col-2	Column number; 1-3 digits, from 2 to 510; last column number to be searched in a range beginning at col-1. Must be greater in value than col-1 and followed by a right parenthesis. The range must be at least equal to the number of characters specified in the text search string. Column specification is significant only if a text search string is specified. Lines are deleted only if the text string occurs within the range, or if it begins in col-1, when a single column is specified.
<u>U</u>NIT	Keyword; dictates that the text search string appear as a unit within a line; it must be delimited by special characters (including blank) other than letters or digits.
<u>V</u>ETO	Keyword; permits the user to approve deletions before they occur. The line to be deleted is displayed at the terminal; the user may enter:

- Y, YE, or YES to delete the line.
- C, CO, CON, . . . , or CONTINUE to delete the line and any subsequent lines which satisfy the requirements specified in the DELETE command.
- Any character other than Y or C to retain the line.

The DELETE command must include at least one parameters; ALL, LAST, or a line number.

If a text search string is specified in the command, a message reports the number of deletions performed.

    n DELETIONS

n is the number of lines deleted. If more than 20 characters are entered as a text search string, the string is truncated to the first 20 characters. An informative message is displayed at the terminal and VETO automatically takes effect; each line that satisfies the search conditions is displayed; the user may enter the VETO responses given in the DELETE command description.

An interrupt command may be entered to terminate execution of a DELETE command; however, the edit file may be left with the specified lines partially deleted. (For a description of the INTERCOM command see sections 2 and 4 of the INTERCOM Version 4 Reference Manual or section 2 of the INTERCOM Version 3 Reference Manual.)

Examples:

To delete line 100 in the edit file:

```
..DELETE 100                         user enters DELETE, line-1
..                                   EDITOR is ready for next command
```

To delete from lines 200 through the last line, only if the character string AX appears in columns 7 through 72:

```
..DELETE /AX/ (7,72) 200,L           user enters DELETE, /text/,(col-1,
                                     col-2), line-1, LAST
            2 DELETIØNS              system message
..                                   EDITOR is ready for next command
```

To delete all lines from line 100 through line 200:

```
..DELETE 100 200
..
```

user enters DELETE, line-1, line-2.
EDITOR is ready for next command

To delete all lines in the edit file so that a new file may be constructed:

```
..DELETE ALL
..
```

user enters DELETE, ALL
EDITOR is ready for next command

To delete with veto power, all lines in the edit file only if they contain the character C in column 1 as a unit:

```
..DELETE,ALL /C/,(1),U,V
         20=C BEGIN DØ LØØP
N
         50=C END SCAN
N
          0 DELETIØNS
..
```

user enters DELETE, ALL/text/, (col-1), UNIT, VETO
system displays qualifying line
user elects to retain line
system displays qualifying line
user elects to retain line
system message, all qualifying lines displayed, none deleted.
EDITOR is ready for next command

EDIT Command

The user enters the EDIT command to load a local file into the edit file:

EDIT, filename

filename	Name of file to be edited; required immediately following the command verb.

The file name may be any coded sequential file to which the user has read access, including local and attached permanent files. The file to be loaded is called the source file; it is not modified by execution of the EDIT command.

When BASIC format specifications are in effect, the keyword SEQUENCE is not allowed.

If the user enters the EDIT command when the edit file contains information that has not been saved as a local file since it was last modified, EDITOR ignores the command and displays the message:

WARNING-EDIT FILE NOT SAVED

The user may then save the edit file, or if the contents of the edit file need not be retained, simply re-enter the EDIT command. In the latter case, the contents of the edit file are destroyed.

Multi-record files and multi-file sets may be loaded for editing, but they appear in the edit file as one record. On encountering an end-of-record in the source file, the character string *EOR is assigned a sequential line number and written in the edit file to indicate an end-of-record condition. If the end-of-record is of level nn (and nn$\neq$ zero) a character string *EOR, nn is generated. For end-of-file, the character string *EOF is generated.

Example:

To load the local file named AFIL into the edit file:

```
..EDIT,AFIL                        user enters EDIT, filename
..                                 EDITOR is ready for the next
                                   command
```

LIST Command

This command permits the user to list edit file lines at the terminal.

```
        ┌   ( ALL                  )  ┐           ┌                                          ┐
LIST    │ , < line-1 [, { line-2 } ] >│   [SUP]   │ ,/text/  [,(col-1 [,col-2])]   [,UNIT]   │
        └   ( LAST       { LAST   }  )  ┘           └                                          ┘
```

ALL	Keyword; all lines in the file are listed or searched for the text search string.
line-1	Line number; 1-6 digits, from 1 to 999999; first or only line to be listed or searched.
line-2	Line number; 1-6 digits, from 2 to 999999; last line to be listed or searched in a range beginning at line-1.
LAST	Keyword; if specified as first parameter, the last line in the file is displayed or searched; if specified as second parameter, the listing or search begins at line-1 and continues through the last line in the file.
SUP	Keyword; suppresses EDITOR line numbers from list displayed at terminal.
/text/	Text search string; 1-20 characters delimited by slashes or an equivalent delimiter; file is searched for this text string (search may be restricted to a range of line and column numbers). Lines containing the text string are listed at the terminal.
col-1	Column number; 1-3 digits, from 1 to 510; first or only column number of a text string search. Must be preceded by a left parenthesis and followed by either col-2 or a right parenthesis.
col-2	Column number; 1-3 digits, 2 to 510; last column number of a text string search in a range beginning in col-1. Must be greater in value than col-1 and followed by a right parenthesis. The range of columns must be at least equal to the number of characters in the text string. Column specification is significant only if a text search string is specified. Lines are listed only if the text string occurs within the range, or if it begins in col-1 when only a single column is specified.
UNIT	Keyword; dictates the text search string appear as a unit within a line; the text string must be delimited by special characters (including blank) other than letters or digits.

If the LIST command is entered with no parameters, the current line is listed (line to which the edit file pointer is set).

If more than 20 characters are entered as a text search string, the string is truncated to the first 20 characters. An informative message is displayed at the terminal, followed by the lines that satisfy the search conditions.

Unless the SUP parameter is specified, lines which satisfy the LIST command requirements are displayed in the form:

line number = text line

Examples:

To list all lines in the edit file which contain the variable AX as a unit:

```
..LIST,A /AX/ U                     user enters LIST, ALL, /text/, UNIT
     100=     AX=X**2               system lists all lines which satisfy
     620=     PRINT,AX              command requirements

..                                  EDITOR is ready for next command
```

To list lines 10 through 20 of the edit file, and then the current line:

```
..LIST,10 20                        user enters LIST, line-1, line-2
       10=DATA 10                   system lists appropriate lines
       15=DATA 15
       20=DATA 20
..LIST                              user enters LIST
       20=DATA 20                   system lists current line
..                                  EDITOR is ready for next command
```

SAVE Command

To save the edit file as a local file, the user enters:

```
                                                    ┌   ⎧ ALL             ⎧ line-2 ⎫   ⎫ ┐
SAVE, filename [,NOSEQ] [,OVERWRITE] [MERGE]        │ , ⎨ line-1   [,     ⎨        ⎬ ] ⎬ │
                                                    └   ⎩ LAST            ⎩ LAST   ⎭   ⎭ ┘

               [,/text/ [,(col-1[,col-2])] [,UNIT]] [,VETO]
```

filename	Name under which edit file is saved as a local file; required immediately following the command verb.
NOSEQ	Keyword; causes EDITOR line numbers to be suppressed in local file.
OVERWRITE	Keyword; causes any local file of the same file name to be overwritten.
MERGE	Keyword; causes the entire edit file or selected portions of the edit file to be saved followed by an end-of-record. The named file remains positioned immediately after the end-of-record.
ALL	Keyword; all lines in the file are saved or searched for the text search string.
line-1	Line number; 1-6 digits, from 1 to 999999; first or only line to be saved.
line-2	Line number; 1-6 digits, from 2 to 999999; last line to be saved in a range beginning at line-1.
LAST	Keyword; if specified as first parameter, the last line in the file is saved; if specified as second parameter, the file is saved beginning at line-1 through the last line in the file.
/text/	Text search string; 1-20 characters delimited by slashes or an equivalent delimiter; file is searched for this text string (search may be restricted to a range of line and column numbers). Lines containing the text string are saved.
col-1	Column number; 1-3 digits, from 1 to 510; first or only column number of a text string search. Must be preceded by a left parenthesis and followed by either col-2 or a right parenthesis.

col-2	Column number; 1-3 digits, from 2 to 510; last column number of a text string search beginning in col-1. Must be greater in value than col-1 and followed by a right parenthesis. The range of columns must be at least equal to the number of characters in the text string. Column specification is significant only if a text search string is specified. Lines are saved only if the text string occurs within the column range.
UNIT	Keyword; dictates that the text search string appears as a unit within a line; the text string must be delimited by special characters (including blank) other than letters or digits.
VETO	Keyword; permits the user to approve each line before it is saved on the file specified. The line to be saved is displayed at the terminal; the user may enter: Y,YE, or YES to save the line. C, CO, CON, ..., or CONTINUE to save the line and any subsequent lines which satisfy the requirements specified in the SAVE command. Entry of any character other than Y or C indicates that the displayed line is not to be saved on the file specified.

If the SAVE command is entered with no selective parameters; all lines are saved.

The entire file or part of the file is saved as a sequential mass storage file. The line length in the saved file is determined by the format specification currently in effect at the terminal. Lines will be blank filled or truncated accordingly unless the variable length line specification (CH=999) is in effect. If truncation is necessary, a message is sent indicating the length of the longest line encountered; the user may change the format character count and reenter the SAVE command. The SAVE command does not destroy the edit file.

If the user has not specified the keyword OVERWRITE, and a local file exists with the same file name, the SAVE command is ignored and an error message is displayed. An attached permanent file cannot be overwritten.

TEXT REPLACEMENT Command

To replace text strings in lines of the edit file, the user enters:

$$/\text{text-1}/=/\text{text-2}/ \left[, \left\{ \begin{matrix} \underline{\text{A}}\text{LL} \\ \text{line-1 } [, \left\{ \begin{matrix} \text{line-2} \\ \underline{\text{L}}\text{AST} \end{matrix} \right\}] \\ \underline{\text{L}}\text{AST} \end{matrix} \right\} \right] \left[,(\text{col-1 } [,\text{col-2}]) \right] [,\underline{\text{U}}\text{NIT}] \; [,\underline{\text{V}}\text{ETO}]$$

/text-1/=/text-2/	Text strings; equals sign must be specified with no spaces on either side.
/text-1/	Text search string; 1-20 characters delimited by slashes or an equivalent delimiter. File is searched for this string (search may be restricted to a range of line and column numbers).
/text-2/	Text replacement string; 0-20 characters delimited by slashes or an equivalent delimited. Replace text search string when conditions of the search are satisfied.
$\underline{\text{A}}$LL	Keyword; causes a search of all lines in the edit file.
line-1	Line number; 1-6 digits, from 1 to 999999; first or only line to be searched.
line-2	Line number; 1-6 digits, from 2 to 999999; last line to be searched in a range beginning at line-1.
$\underline{\text{L}}$AST	Keyword; as first parameter, a search is made of the last line in the file; as second parameter, a search is made beginning at line-1 through the last line in the file.
col-1	Column number; 1-3 digits, from 1 to 510; first or only column to be searched. Must be preceded by a left parenthesis and followed by either col-2 or a right parenthesis.
col-2	Column number; 1-3 digits, from 2 to 510; last column to be searched in a range beginning at col-1. Must be greater than col-1 and followed by a right parenthesis. The range must be at least equal to the number of characters in the text search string. Replacement takes place only if the text string occurs within the column range, or if the text string begins in col-1, when a single column is specified.

UNIT	Keyword; dictates that the text search string appear as a unit within a line; the text string must be delimited by special characters (including blank) other than letters or digits.
VETO	Keyword; permits the user to approve text replacement before it occurs. The changed form of the line is displayed at the terminal; the user may enter: Y, YE, or YES to accept the change. C, CO, CON, ..., or CONTINUE to accept the change and any subsequent changes which satisfy the requirements specified in the Text Replacement command. Any character other than Y or C to retain the original line.

If the TEXT REPLACEMENT command is entered with no parameters, the search will be performed on the line to which the current line pointer is set.

The number of replacements performed are reported in a message:

```
n CHANGES
```

where n is the number of changes made. Because more than one replacement may occur in any line, the number of changes displayed may differ from the number of lines changed.

The two text strings specified as the command verb need not contain the same number of characters; the line affected will be expanded or contracted as necessary. If the maximum character count is exceeded, the replacement occurs, and an informative message is displayed. Truncation occurs if a line exceeds 510 characters.

The text replacement string may be entered as a null string (two consecutive slashes, no imbedded blanks). This specification causes the text string to be deleted if all search conditions are satisfied.

If more than 20 characters are entered as a text search or text replacement string, the string is truncated to the first 20 characters. An informative message is displayed at the terminal and VETO automatically takes effect. The changed form of each line that satisfies the search conditions is displayed; the user may enter the VETO responses given in the Text Replacement command description.

The Text Replacement command cannot be used to edit tabulation characters (FORMAT command) into an existing line. If done, the entered character is accepted as a data character; no tabulation occurs.

Examples:

To replace the variable name AX with the name BZ in the current line; AX must be a unit:

```
••/AX/=/BZ/,U                  user enters/text-1/=/text-2/,UNIT
      1 CHANGES                system message
••                             EDITOR is ready for next command
```

To replace the character string TCS, wherever it appears in the edit file, with the string TERMINAL CONTROL SYSTEM. (Two TEXT REPLACEMENT commands must be entered because the text replacement string is greater than 20 characters.) The user requests veto power:

```
••/TCS/=/TERMINAL CØNTRØL SY./,A,V         user enters/text-1/=
                                             /text-2/, ALL, VETO
    60=THE TERMINAL CØNTRØL SY. HAS THE    system displays changed line
Y                                          user accepts change

   190=IN THE TERMINAL CØNTRØL SY. USERS   system displays changed line
YES                                        user accepts change

  2000=***TERMINAL CØNTRØL SY. ABØRT***    system displays changed
N                                          line;user retains original
      2 CHANGES                            line system message;
••/SY./=/SYSTEM/ 60,190 U                  user enters /text-1/=
                                             /text-2/, line-1, line-2,
                                             UNIT system message;
      2 CHANGES                            EDITOR is ready for next
••                                         command
```

To replace the character C with the character * only if C appears as a unit in column 1. All lines are searched:

```
••/C/=/*/ A (1) U              user enters /text-1/ = /text-2/, ALL,
     15 CHANGES                (col-1), UNIT system message
••                             EDITOR is ready for next command
```

To replace the character string PROGRAM in line 2310 with a null string. (line currently appears as 2310=END PROGRAM.) The user requests veto power:

```
..PRØGRAM/=//,2310,V        user enters /text-1/=/text-2/,
                            line-1,VETO system displays changed line
  2310=END
YE                          user accepts change
     1 CHANGES              system message
..                          EDITOR is ready for next command
```

To compile a BASIC program contained in the edit file (an error is encountered by the compiler):

```
..RU B                      user enters RUN, system-name
ILLEGAL STRING IN 330       compilation error message
..                          EDITOR is ready for next command
```

To assemble and execute a BASIC program contained in the edit file (an arithmetic error terminates the job during execution):

```
..RUN,BASIC                                 user enters RUN, system-
ARITHMETIC ERRØR MØDE=1 ADDRESS=023427      name, system message
..                                          EDITOR is ready for next
                                            command
```

LEAVING EDITOR

To exit from the EDITOR routine the user types:

BYE

The user is then returned to INTERCOM.

USING BASIC UNDER SCOPE

To access a central computer from a terminal, the user must link up with the computer system. The method of establishing the connection between the terminal and the central site computer varies depending on the type of terminal equipment and the connection provided by the telephone company. When connected to the terminal, the system responds:

```
CONTROL DATA INTERCOM  4.1
DATE               mm/dd/yy
TIME               hh.mm.ss
PLEASE LOGIN
```

Step 1: The terminal user logs into the system by entering:

```
LOGIN
```

The system responds:

```
ENTER USER NAME-
```

Step 2: Enter your user name. The user name may be any combination of up to ten letters or digits and must <u>not</u> be followed by a period.

When the user name has been entered at a TTY terminal, the system responds:

```
■■■■■■■■■■ ENTER PASSWORD-
```

At a 200 USER or display terminal, the system responds:

```
ENTER PASSWORD-
```

The user then enters his password. A password is any combination of up to ten letters or digits which must not terminate with a period. On a teletype listing, the password is overprinted on the ten-character, blocked-out line to preserve privacy. The display terminal screen is automatically cleared on acceptance of the entered password to preserve privacy.

When the user name and password are accepted, the user id (a two-character user code) and the time at which the user logged in, followed by the equipment number (multiplexer equipment status table ordinal) and the port number at which he logged in, are displayed at the terminal.

e.g.

```
ENTER USER NAME- USERA
■■■■■■■■■■■ ENTER PASSWORD-

19/07/74      LOGGED IN AT 17.47.26
              WITH USER-ID AB
              EQUIP/PORT   52/03
COMMAND-
```

Step 3: After the user successfully logs-in, the system responds with COMMAND and the user enters the command "EDITOR".

```
COMMAND - EDITOR
```

The user is now in text edit mode.

Step 4: Enter FORMAT, BASIC
When this command is entered, a format specification is automatically established at the terminal which permits the user to enter lines in BASIC language format.

Step 5: Enter the BASIC program statements-line number followed by BASIC statement.

e.g. `10 LET X = 5`

Each line must begin with a 1-5 digit line number and end with (CR) . BASIC statements need not be typed in correct order because the EDITOR automatically sequences them according to line number.

Step 6: Once the entire program is entered, compile and execute the program by typing:

```
RUN BASIC
```

After execution is completed the output is printed or if a program error occurs, the appropriate error message is displayed.

Step 7: When the run is completed, the user can select one of the following options:

- Continue processing - build and execute new programs; modify existing programs and rerun; or rerun the same program.
- Terminate the terminal session by saving the program and entering the BYE and LOGOUT commands or by entering the BYE BYE and LOGOUT commands. When the BYE or BYE BYE commands are entered, user returns to INTERCOM mode from EDITOR mode.

When in INTERCOM mode, the system responds with:

COMMAND

At this time the user enters the LOGOUT command which releases local files which the user may have created under EDITOR. Only permanent files are retained between the time of a LOGOUT and any subsequent LOGIN. The user is disassociated from INTERCOM until a subsequent LOGIN command is entered. INTERCOM displays the date and time the user is logged-out. The LOGOUT command is not allowed when the user is under control of EDITOR.

Example:

```
COMMAND- LOGOUT
CP TIME      9.458
PP TIME      8.181
CONNECT TIME     0 HRS.  8 MIN.
 19/06/73  LOGGED OUT AT 08.31.23.[
```

The order of the date (month, day, year) may be changed as an installation option. The time of LOGOUT is given in hours, minutes, seconds (24-hour clock); CP/PP time is given in seconds. The user should disconnect his terminal from INTERCOM by turning it off, or by hanging up the data set receiver.

SAMPLE TERMINAL SESSION

After the user has logged in, he can create and execute BASIC programs. The following sample BASIC program, run under the INTERCOM system, illustrates how to run a BASIC program. The program was entered at a TTY terminal. User responses are underlined. After typing the response, user must depress the carriage return key (CR) .

```
CØMMAND- EDITØR
••FØRMAT,BASIC
••10 PRINT "TYPE A NUMBER";
20 INPUT X
30 LET F=1
40 FØR I=1TØ X
50 F=F*1
60 PRINT "FACTØRIAL ";X;"IS ";F
70 PRINT
80 GØTØ 10
90 END
```

User calls EDITOR.
User requests BASIC format specifications and following EDITOR command mode response, enters a BASIC program line by line.

```
RUN,BASIC

BASIC CØMPILATIØN ERRØRS
FØR WITHØUT NEXT  AT  40

••
```

Compiles and executes BASIC program.
BASIC issues diagnostic messages.

```
55 NEXT I
25 IF X=0 THEN 80
RUN,BASIC
```

Statement 55 is added to satisfy looping requirements.
Statement 25 is added to provide an exit from the program.
User calls BASIC compiler-again requests compile and execution of the BASIC program

```
TYPE A NUMBER ?3
FACTØRIAL  3 IS  6
TYPE A NUMBER ?0
••LIST,ALL
```

Zero causes exit from execution and return to EDITOR command mode.
User requests listing of his program in the edit file.

```
10=10 PRINT "TYPE A NUMBER";
20=20 INPUT X
30=30
25=25 IF X=0 THEN 80
30=30 LET F=1
40=40 FØR I=1TØ X
50=50 F=F*1
55=55 NEXT I
60=60 PRINT "FACTØRIAL ";X;"IS ";F
70=70 GØTØ 10
80=80 END
```

```
..SAVE,BASPRØG
```

User requests contents of edit file be saved as a local file named BASPROG until LOGOUT.

```
..STØRE,BASPRØG
```

Store BASPROG as a permanent file.

```
..BYE
```

User requests a return to INTERCOM command mode.

```
CØMMAND- CØNNECT,BASØUT.
CØMMAND- REWIND,BASPRØG.
CØMMAND- BASIC,I=BASPRØG,K=BASØUT.
```

In INTERCOM command mode, user requests source input, diagnostics, and execution output.

```
TYPE A NUMBER ?  6
FACTØRIAL   6     IS  720
TYPE A NUMBER ?  0
```

Zero causes exit from execution and return to INTERCOM command mode.

```
CØMMAND- LØGØUT.
CP TIME      3.800
PP TIME     15.700

CØNNECT TIME     0 HRS. 10 MIN.
 12/06/74  LOGGED ØUT AT 14.00.43.<
```

User requests to leave INTERCOM.

TERMINAL OPERATION UNDER KRONOS/NOS

This section describes those terminals from which BASIC programs can be submitted; illustrates the method of running BASIC programs under KRONOS and NOS and the use of data files. Because of the similarity between a KRONOS and NOS terminal session, only an example of a KRONOS session is provided in this section.

KRONOS/NOS SYSTEM

KRONOS/NOS allow multi-user access to one large-scale CYBER 70, CYBER 170 abd 6000 Series computers. BASIC programs can be submitted from a remote time-sharing terminal, such as a Model 33 or Model 35 teletypewriter (TTY) or any TTY compatible terminal, and CRT Model 713 terminals.

TTY TERMINAL DESCRIPTION

The programmer types lines of information to the KRONOS/NOS system on the TTY terminal and the operating system responds interactively to the TTY terminal. The TTY keyboard (see figure 2.8-1) is similar to that of the ordinary electric typewriter with the exception of the keys which are described below:

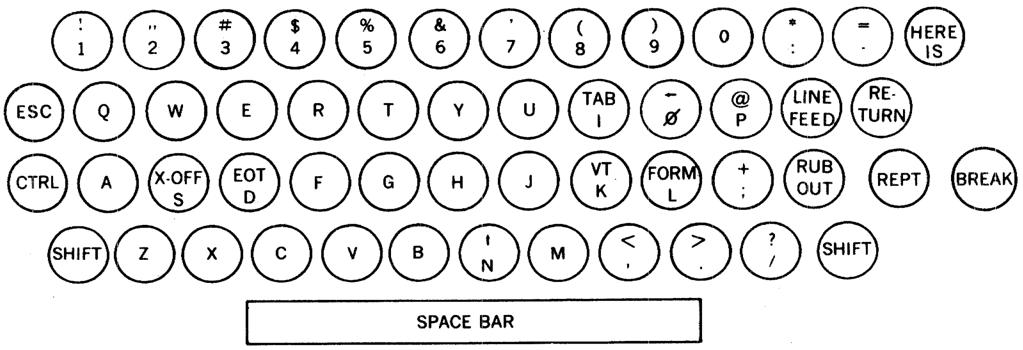


Figure 2.8-1. Typical Teletype Keyboard

Return Key

The user must terminate every line of information by pressing the RETURN key. This signals to the operating system that the message is complete and causes the carriage to return to the beginning of the line. The operating system responds by issuing a line feed command to the terminal. This moves the carriage to the next line at which point the user may begin the next line of input.

Backspace Arrow (←) Key

Entry errors can be corrected before the RETURN key is pressed by using the backspace arrow and typing the correct character.

Example:

```
                The line:  BAX←SJK← ←IC
   is interpreted by the operating system as:  BASIC
```

ESC Key

Entire lines may be deleted before the RETURN key is pressed by pressing the ESC key. This causes the entire line to be ignored. The operating system responds by returning the carriage to the beginning of the line and moving the carriage to the next line position.

Dial-In Procedure

CONNECTING TTY MODELS 33/35

If the telephone line is connected directly to the TTY, the following procedure is used:

- Turn on the TTY by setting the rotary power switch to the LINE position or by depressing the ORIG button, on those models having such a feature.
- Dial the correct phone number. The system will request answerback, return a header, and request the user number.

There is a slight pause from the time the phone is answered until the TTY begins to type. This is caused because:

- A time delay is allowed, before any data transmission is attempted, to ensure that the line is settled.
- A fixed delay of about three seconds is allowed to ensure that answerback drum transmission is complete. This is necessary for all terminals, as the number of characters on the answerback drum are unknown at this time.

If the teletypwriter is connected to the system by an acoustic coupler, the following procedure is used:

- Turn on the TTY as described above.
- Turn on the acoustic coupler.
- Dial the correct phone number.
- When the connection has been made, that is, a constant high-pitched sound is heard in the receiver, place the receiver in the acoustic coupler. The system will request answerback, return a header, and request user number.

CONNECTING TTY MODEL 37

To connect Model 37 to KRONOS, use the following procedure:

- Turn on the TTY by pressing the DATA button.
- Follow the procedure described for Models 33/35 depending on whether the TTY is connected directly to the system or connected to the system by an acoustic coupler.

CONVERSATIONAL DISPLAY TERMINAL, MODEL 713

See section 2.7 for a description of the terminal keyboard.

USING BASIC

The previous paragraphs have described BASIC statements and how to organize these statements into a BASIC program. The following describes how to enter a program into a computer and how to execute that program.

BASIC is primarily a terminal oriented language; however, programs in card deck form can be entered and executed (batch mode). The following paragraphs describe the method for entering and executing BASIC programs interactively through use of a Teletype (TTY) or Cathode Ray Tube (CRT) terminal. For a description of BASIC program card deck structures and batch mode operations, see section 2.9.

When operating interactively, the user must write the program in a file, as shown in the examples which follow, and execute from the file. To correct a syntax, semantic, or logic error, the user need only enter the line number which contains the error followed by the corrected code. When the corrected line is entered and the terminal RETURN key is pressed, the existing program statement is replaced. To delete a line, enter the line number and press the RETURN key.

BASIC can be run interactively under anyone of these operating systems: KRONOS and NOS, the usage of which is described in the following paragraphs, and SCOPE (INTERCOM) which is described at the end of this section.

USING BASIC FROM KRONOS/NOS

KRONOS/NOS provides multi-user access to one large-scale computer. BASIC programs can be submitted from a remote time-sharing terminal. To access a central computer from the terminal, the user must link up with the computer. The method used to establish connection between the terminal and the central site computer varies depending on the type of terminal equipment and the connection provided by the telephone company.

When the connection is completed, the computer responds:

```
yy/mm/dd        hh.mm.ss
____________    TIME SHARING SYSTEM*
and requests:
USER NUMBER:
```

* The particular message typed at your installation may be different.

When this occurs, perform the following:

Step 1: Submit the user number on the same line:

```
xxxxxxx (CR) ‡
```

The user number consists of up to seven alphanumeric characters. The system then requests:

```
PASSWORD
■■■■■■■■
```

Step 2: Enter the password:

```
zzzzzzz (CR)
```

The password must consist of one to seven alphanumeric characters. To provide a greater measure of security, type the password in the area the system has blacked out. If a password is not needed, type:

```
(CR)
```

If the user number and password are not acceptable, the system types:

```
ILLEGAL LOGIN, TRY AGAIN.
USER NUMBER:
```

If the user number and password are acceptable, the system responds:

```
TERMINAL:  nnn, TTY (where nnn is the particular terminal no.
                         being used).
RECOVER/SYSTEM:
```

Step 3: Enter the desired subsystem on the same line:

```
BASIC (CR)
```

Because all interactive programs run under KRONOS/NOS reside in a file, the system queries the user as to the applicable file type by responding:

```
OLD, NEW, OR LIB FILE:
```

‡ Throughout this section, the convention (CR), Carriage Return, is used to denote the RETURN key on the keyboard.

Step 4: Submit the appropriate status:

OLD $\textcircled{CR}$	for a file that was previously created and saved.
NEW $\textcircled{CR}$	for a new file.
LIB $\textcircled{CR}$	for a file from the system library.

The system responds:

```
FILE NAME:
```

Step 5: Enter the file name:

nnnnnnn $\textcircled{CR}$

The file name consists of up to seven alphanumeric characters.
If an OLD or LIB file does not exist, the system responds:

```
FILE NOT FOUND.
```

If the file name entered contains illegal characters, the system responds:

```
FILE NAME ERROR.
```

After the system finds the specified file, it responds:

```
READY.
```

The following example illustrates a sample log-in:

```
74/07/19.  13.19.28.

TIME SHARING SYSTEM
USER NUMBER:  ABCDEFG
PASSWORD
|T|U|V|W|X|Y|Z|■■ ————→ The password is blocked out
TERMINAL:60,TTY          and cannot be seen by the user.
RECOVER/SYSTEM:BASIC     It is shown for purposes of
OLD,NEW,OR LIB FILE:NEW, EX4   illustration only.


READY.
```

Step 6: Enter the new BASIC program. Each line must begin with a 1-5 digit line number and end with (CR) . BASIC statements need not be typed in correct order, because KRONOS and NOS automatically sequence them according to line number.

Step 7: To execute the program, type:

<u>RUN</u> (CR) or <u>RNH</u> (CR)

This command initiates compilation and execution of the BASIC program.

The output of a BASIC program is in the form:

```
yy/mm/dd      hh.mm.ss

PROGRAM       nnnnnnn
     .
     .
(data printed by the program - error messages, if program
errors occurred)
     .
     .
RUN COMPLETE.
```

Step 8: When a run is completed, the user has the following options:

. Continue processing - build and execute new programs, modify existing program and rerun, rerun the same program.

or

. Terminate the terminal session via the following commands:

<u>BYE</u> (CR)

or

<u>GOODBYE</u> (CR)

Either command releases all local files and prints the following:

```
xxxxxxx  LOG  OFF      hh.mm.ss.
xxxxxxx  CPU           s.sss  SEC.
```

where:

xxxxxxx	user number.
s.sss	number of seconds of central processor time used. (This is not the amount of time used between LOGIN and BYE.)

Sample Terminal Session

The following example was run at a TTY terminal. User responses are underlined. The user must press the carriage return key (CR) after typing in each response.

```
 74/09/09. 12.06.37.
KRØNØS TIME SHARING SYSTEM - VER. 2.1-03/AA.
USER NUMBER: 412
PASSWØRD
■■■■■■■■■
```

log-in procedure-type user number and password.

```
TERMINAL:      53, TTY
RECØVER /SYSTEM:BASIC
OLD, NEW, ØR LIB FILE:  NEW,EX4
READY.
```

Requests BASIC subsystem. Program is from NEW file.

```
10 PRINT "TYPE A NUMBER"
20 INPUT X
30 LET F=1
40 FØR I=1TØX
50 LET F=F*I
60 PRINT "FACTØRIAL ";X;"IS ";F
70 GØTØ 10
80 END
```

Program statements-consist of a line number followed by a space, followed by the appropriate statement.

```
RUN
```

Compile and execute program.

```
 74/09/09. 12.13.26.
PRØGRAM    EX4

 FØR WITHØUT NEXT   AT   40

CP         0.023 SECS.

RUN CØMPLETE.
```

BASIC issues diagnostic.

```
55 NEXT I
25 IF X=0 THEN 80
RUN
```

Adding statements to correct program.

```
 74/09/09. 12.14.40.
PRØGRAM    EX4
```

Compile and execute again.

```
TYPE A NUMBER
? 3
FACTØRIAL  3 IS  6
```

User input 3 as value for X.

```
TYPE A NUMBER
? 0

CP         0.036 SECS.

RUN CØMPLETE.
```

X = 0 and program terminates at line 80.

```
LIST

 74/09/09. 12.15.41.
PROGRAM   EX4

10 PRINT "TYPE A NUMBER"
20 INPUT X
25 IF X=0 THEN 80
30 LET F=1
40 FØR I=1TØX
50 LET F=F*I
55 NEXT I
60 PRINT "FACTØRIAL ";X;"IS ";F
70 GØTØ 10
80 END
READY.

SAVE,EX4
READY.
```

KRONOS/NOS command to list program.

Program listing.

KRONOS/NOS command saves program with file name **EX4 for later use.**

For a detailed description of the KRONOS/NOS commands used is this example and other available commands, see the KRONOS /NOS Reference Manual.

In this example, the user saved the program as a file name EX4. The program in this file is now stored as an indirect access permanent file which can later be accessed by use of the OLD command, e.g.

```
ØLD,EX4
READY.

LIST

 74/09/09. 12.17.03.
PRØGRAM   EX4

10 PRINT "TYPE A NUMBER"
20 INPUT X
25 IF X=0 THEN 80
30 LET F=1
40 FØR I=1TØX
50 LET F=F*I
55 NEXT I
60 PRINT "FACTØRIAL ";X;"IS ";F
70 GØTØ 10
80 END
READY.
```

makes a copy of the file accessable to the user.
requests a list of the file contents.

At this time, the user can add, delete, or change program statements, See USING BASIC.

```
 RUN
```

Compiles and executes the new program.

```
 74/09/09. 12.18.05.
PRØGRAM   EX4

TYPE A NUMBER
? 6
FACTØRIAL  6 IS  720
TYPE A NUMBER
? 0


CP       0.037 SECS.

RUN CØMPLETE.
```

User enters 6 as value for X.

```
BYE

LN76       LØG ØFF.  12.18.37.
LN76       CP        0.096 SEC.
```

Logs off.

If the user wishes to store the changed program, this can be accomplished by the REPLACE command which replacesthe old program with the corrected program.

REPLACE, EX4

Stores the updated program in file EX4. If the user logs off before replacing EX4, the corrected version is lost but the old version of EX4 remains in tact.

KRONOS/NOS DATA FILES

The following example illustrates the creation of a data file used by a KRONOS BASIC program. To create a data file under KRONOS/NOS specifying the name of the new file and enter the TEXT command. The TEXT command permits the user to create the file without sequence numbers. If after the file is created, connections, additions, or deletions are required, enter EDITOR and use TEXT Editor commands. In the following example the program reads a data base file (REST) and update file (SAMI); calculates a new change account balance; and prints an updated statement. Both data files were created by entering the TEXT command and after the system message

"ENTERING TEXT MODE"

inserting data line by line; each line ends by typing CR . Each file was saved under their respective names and later made secondary files accessable by program " TEST1" through use of the GET statement.

For a complete description of the TEXT command, see KRONOS/NOS Time Sharing User's Guides. For a complete description of TEXT EDITOR commands, see KRONOS/NOS TEXT EDITOR Reference Manuals.

DATA FILE " REST"

```
J.BRØWN,1422 EAST ST.,CHARGE NØ.1111,500.00
S.APPLE,3434 CHERRY ST.,CHARGE NØ. 2211,222.22
R.DREW,7896 ALGØ AVE.,CHARGE NØ.1660,133.98
```

DATA FILE " SAMI"

```
000000010
000000020
000000030
```

PROGRAM "TEST1"

```
 74/09/10. 12.04.43.
KRØNØS TIME SHARING SYSTEM - VER. 2.1-03/AA.
USER NUMBER:  LN76,T3X8L1
TERMINAL:     72,TTY
RECØVER /SYSTEM:BASIC
ØLD, NEW, ØR LIB FILE:


ØLD,SAMI
READY.

LIST


 74/09/10. 12.09.34.
PRØGRAM   TEST1

5 FILE #2="SAMI"
10 FILE #1="REST"
20 RESTØRE #1
21 RESTØRE #2
25 FØR I=1TØ3
30 INPUT #1,A$,B$,C$,D
50 INPUT #2,S
60 X=D+S
70 PRINT TAB(2);A$;TAB(12);B$;TAB(32);C$;TAB(52);"BALANCE=$";X
71 NEXT I
READY.

GET,REST
READY.

GET,SAMI
READY.

RUN
```

RESULTS:

```
 74/09/10. 12.10.36.
PRØGRAM   TEST1

  J.BRØWN   1422 EAST ST.         CHARGE NØ.1111       BALANCE=$ 510
  S.APPLE   3434 CHERRY ST.       CHARGE NØ. 2211      BALANCE=$ 242.22
  R.DREW    7896 ALGØ AVE.        CHARGE NØ.1660       BALANCE=$ 163.98
```

BATCH OPERATIONS 2.9

BASIC jobs can be submitted to the KRONOS/NOS or SCOPE operating system input queue for batch processing either at the central site or from a remote terminal.

At any terminal, a job previously stored on disk can be entered into the SCOPE batch queue using the INTERCOM command BATCH. The BATCH command is used to direct the disposition of the file which the user has previously created and saved. Output can be directed to a line printer at the central site or to any terminal with a line printer. Similarly, under KRONOS or NOS a previously stored file can be entered into the KRONOS/NOS batch queue by using the SUBMIT command.

At a remote batch terminal, the BASIC job can also be entered through the terminal card reader for execution by the operating system. Output can be directed to a line printer at the terminal, at another terminal, or at the central site.

Any program file submitted for batch processing must contain the necessary control cards as the first logical record. These control cards may include any applicable operating system control cards.

DECK STRUCTURE

The following examples, figures 2.9-1 and 2.9-2, illustrate the general deck structure for a typical BASIC program. In the control card record:

JOB card	specifies job name, memory and time requirements; also priority, etc.
ACCOUNT card	varies with each installation; validates the user. Required for KRONOS and NOS only.
BASIC card	calls BASIC compiler.
LGO	loads binary decks from LGO file and initiates execution.
7/8/9	end-of-record.

A 6/7/8/9 card specifies end-of-information. A complete description of the BASIC control card follows; a description of KRONOS/NOS batch control cards is contained in the KRONOS 2.1 and NOS Reference Manual. A description of SCOPE batch control cards is contained in the SCOPE Reference Manual.

BASIC CONTROL CARD

Programs submitted for batch processing must include a BASIC control card to call the BASIC compiler from the KRONOS/NOS or SCOPE library.

Formats:

BASIC ($p_1, p_2, \ldots p_n$)

BASIC,$p_1, p_2, \ldots p_n$.

The BASIC compiler accepts the following parameter options:

Parameter	Result
L	Source listing, diagnostics, and execution output on file OUTPUT.
L=lfn	Source listing, diagnostics, and execution output on lfn.
L omitted	Diagnostics and execution output on file OUTPUT.
K	Diagnostics and execution output on file OUTPUT.
K=lfn	Diagnostics and execution output on lfn.
K omitted	Diagnostic and execution output on file OUTPUT.
I	Source input from file INPUT.
I=lfn	Source input from lfn.
I omitted	Source input from file INPUT.
B[‡]	Relocatable code on file LGO.
B=lfn[‡]	Relocatable code on lfn.
B omitted	No relocatable code generated.
A	Assembly listing on file OUTPUT.
A=lfn	Assembly listing on lfn.
A omitted	No assembly listing generated.
N	Inhibits program execution.
N=lfn	Inhibits program execution. Absolute code written to file lfn.
N omitted	Execute in place.

If both K and L options are specified on the control card, K is ignored; the L, A, and B options require a larger field length than I, K, and N options.

‡ Relocatable program has identifier "lfn" if the source input is from file "lfn", or BASICXX if the source input is from file "INPUT".

Examples:

The control card: BASIC.

> results in the compilation and execution of a BASIC program giving a listing of diagnostics and execution output on the OUTPUT file.

The control card: BASIC (L, B, N)

> generates relocatable code on the LGO file, and source and diagnostic listings on the OUTPUT file. Then the program may be executed by the LGO control card (figure 2.9-2).

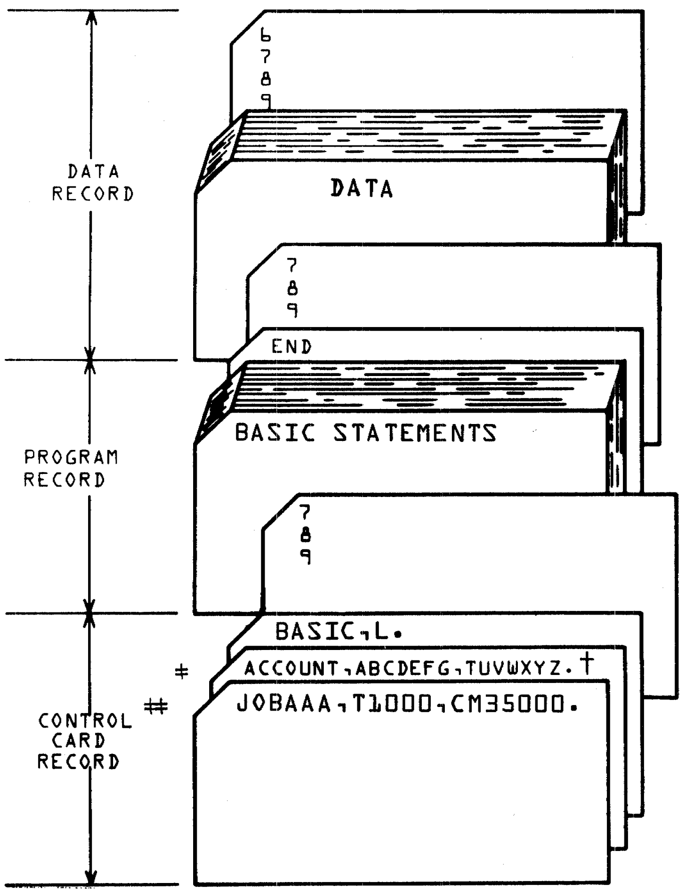


Figure 2.9-1. BASIC Compile and Execute

‡ ACCOUNT card not required for SCOPE.

‡‡ Central memory field length of 35000 is adequate for most BASIC programs. Larger field lengths can be employed for exceptionally large programs.

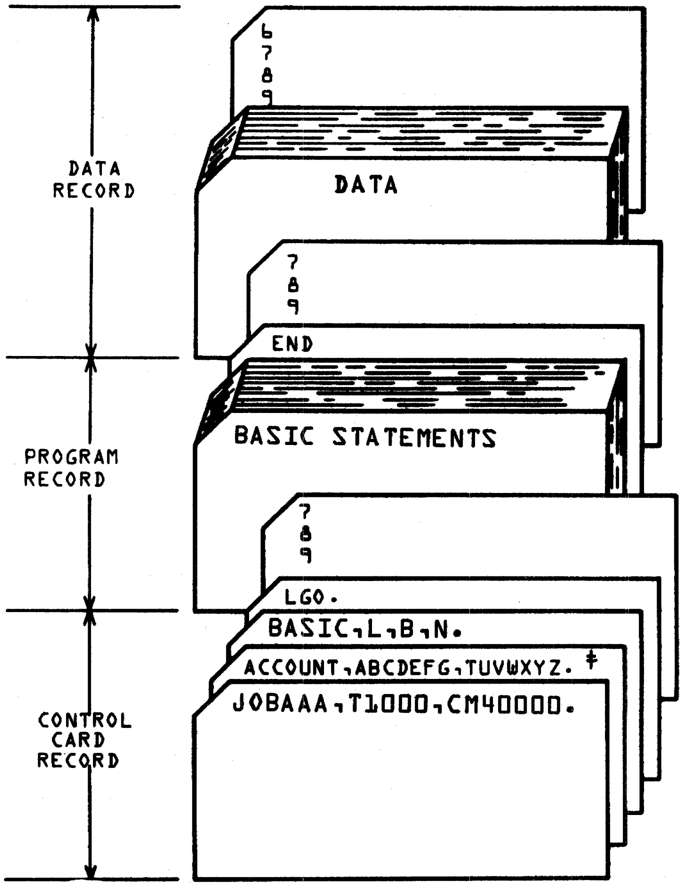


Figure 2.9-2. BASIC Compile, Load and Execute Deck

ǂ ACCOUNT card not required for SCOPE.

BATCH PROCESSING FROM A TERMINAL

BASIC programs can be created at a terminal and submitted for processing in BATCH mode. To accomplish this the user must set up his program in a text editor file which includes control cards.

Using INTERCOM

To create a BATCH job from a terminal, the user must enter the job in an edit file under any format other than BASIC (e.g., COBOL, FORTRAN, COMPASS). This edit file must include control cards, as well as, BASIC program statements. A typical batch deck set up follows:

```
SCOPE JOB CARD        { LN76.
                      {      .
                      {      .
                      {      .
      Control cards   { BASIC, I, L.
                      {      .
                      {      .
                      {      .
      End of Record   { *EOR
                        BASIC STATEMENTS
      End of Record   { *EOR
```

To run a BATCH job from the terminal after creating the edit file, perform the following:

1.	Save the edit file	SAVE, TESTFILE, N
2.	Execute	BATCH, TESTFILE, INPUT, HERE
3.	Identify the output file	FILES or Q, O
4.	Look at output	BATCH, lfn, LOCAL
		PAGE, lfn ← file id found under remote output files when "FILES" was entered

After the PAGE command is entered the system responds with "READY". Enter a 1 and press (CR) to inspect the first page. After the entire page is displayed the user may inspect the next page by entering a "+" (CR).

5.	PRINT output listing at central site	BATCH, lfn, PRINT, user id

Using KRONOS/NOS

The following is an example of a terminal session where a job is created and submitted for batch processing.

Example:

RECOVER/SYSTEM:BATCH	User requests batch mode.
$RFL, 2000	System provides default field length. If additional field length is required enter: RFL, xxxx.
/NEW, GUIDE	User specifies file name "GUIDE".
100 /JOB	
110 /Program Name.	
120 ACCOUNT, account number, password.	Control cards -/JOB must be the first; also all others must end with a period.
. . .	
150 BASIC, I, L.	
160 /EOR	End-of-Record
170 /NOSEQ	Ensure that the BASIC sequence numbers for the statements which follow are not stripped.
BASIC PROGRAM STATEMENTS	
200 /EOF	End-of-File
/SAVE, GUIDE	"GUIDE" becomes a permanent file.
/SUBMIT, GUIDE, B	Directs batch execution of the BASIC program. Output is directed to central site printer as specified by the B parameter.
09.01.36. AB3AACL	System replies with time the job was entered (hr. min. ss.) and system supplied job identifier.
/RETURN, GUIDE	Releases working file GUIDE.
$RETURN, GUIDE.	System reply.
/STATUS, J=AB3AACL	Requests status of a remote batch job initiated by the SUBMIT command. AB3AACL is the alphanumeric name previously assigned by the system to the job.
JOB IN OUTPUT QUEUE.	Typical reply to status request.

The /JOB directive indicates that the file is to be reformatted for batch processing. Some defaults indicated by the directive are:

- remove sequence numbers
- remove internal EOR and EOF marks - does not apply to /EOR and /EOF found in this deck.

See KRONOS/NOS Time Sharing Reference Manual (REFORMATTING SUBMIT FILE) for remaining directive descriptions.

The keyword MERGE enables the user to merge many files or parts of files under one file name. He may load a local file into the edit file and enter the SAVE command with MERGE and any other selective parameters. Specification of NOSEQ is advised to avoid a conflict in sequence numbers. After the SAVE operation, the named file consists of selected lines from the edit file followed by an end-of-record. The saved file remains positioned at the end-of-record.

The user may repeat this procedure until his file is complete. The end-of-record generated between merged files may be later deleted by the user, if necessary.

If more than 20 characters are entered as a text search string, the string is truncated to the first 20 characters. An informative message is displayed at the terminal and VETO automatically takes effect. Each line that satisfies the search conditions is displayed; the user may enter the VETO responses given in the SAVE command description.

Examples:

To save the edit file under the file name FTNPRG:

```
..SAVE, FTNPRG                          user enters SAVE,filename
..                                      EDITOR is ready for the next
                                        command
```

To save the edit file in place of an existing local file named FTNDATA, with a line length of 400 characters and no EDITOR line numbers:

```
                                        user enters FORMAT, CH=nnn
..FØRMAT, CH=400                        user enters SAVE, filename,
..SAVE, FTNDATA, Ø, N                   OVERWRITE, NOSEQ
..                                      EDITOR is ready for the next
                                        command
```

To save all lines between 10 and 100 which have the character * in column 1:

```
                                        user enters SAVE, line-1,line-2,
..SAVE, 10, 100 /*/ (1)                 /text/, col-1
..                                      EDITOR is ready for the next
                                        command
```

CHARACTER SETS A

This appendix describes the SCOPE, KRONOS and NOS character sets which are available to the BASIC user.

SCOPE CHARACTER SETS

Table A-1 lists the standard SCOPE 3.4 character sets: CDC-63, CDC-64, ASCII-63, and ASCII-64. The character set used is installation dependent.

BASIC operates with the ASCII-63 or ASCII-64 character sets only. All of the characters are not part of the BASIC language, but can be used in data and string constants.

If operating in batch mode and if either of the CDC character sets is in use, the user must look up the desired BASIC character in the table A-1 ASCII Graphics Subset column and punch the corresponding character listed in the CDC Graphics column.

When using BASIC, the character ↑ is equivalent to the ∧ (circumflex) or ' (apostrophe) shown in the ASCII Graphics Subset column.

TABLE A-1. SCOPE 3.4 STANDARD CHARACTER SETS

Graphic CDC (200 UT-BCD)	ASCII Graphic Subset (TTY)	Numeric Code	Hollerith Punch (026)	ASCII Punch (029)	CDC Graphic (200 UT-BCD)	ASC Graphic Subset (TTY)	Numeric Code	Hollerith Punch (026)	ASCII Punch (029)
:†	:†	00	8-2	8-2	6	6	33	6	6
A	A	01	12-1	12-1	7	7	34	7	7
B	B	02	12-2	12-2	8	8	35	8	8
C	C	03	12-3	12-3	9	9	36	9	9
D	D	04	12-4	12-4	+	+	37	12	12-8-6
E	E	05	12-5	12-5	-	-	38	11	11
F	F	06	12-6	12-6	*	*	39	11-8-4	11-8-4
G	G	07	12-7	12-7	/	/	40	0-1	0-1
H	H	08	12-8	12-8	(	(	41	0-8-4	12-8-5
I	I	09	12-9	12-9	)	)	42	12-8-4	11-8-5
J	J	10	11-1	11-1	$	$	43	11-8-3	11-8-3
K	K	11	11-2	11-2	=	=	44	8-3	8-6
L	L	12	11-3	11-3	blank	blank	45	no punch	no punch
M	M	13	11-4	11-4	, comma	, comma	46	0-8-3	0-8-3
N	N	14	11-5	11-5	. period	. period	47	12-8-3	12-8-3
O	O	15	11-6	11-6	≡	#	48	0-8-6	8-3
P	P	16	11-7	11-7	[	[	49	8-7	12-8-2
Q	Q	17	11-8	11-8	]	]	50	0-8-2	11-8-2
R	R	18	11-9	11-9	%††	%††	51	8-6	0-8-4
S	S	19	0-2	0-2	≠	" quote	52	8-4	8-7
T	T	20	0-3	0-3	→	underline	53	0-8-5	0-8-5
U	U	21	0-4	0-4	∨	!	54	11-0 or	12-8-7 or
V	V	22	0-5	0-5				11-8-2†††	11-0†††
W	W	23	0-6	0-6	∧	&	55	0-8-7	12
X	X	24	0-7	0-7	↑	' apostrophe(↑)‡	56	11-8-5	8-5
Y	Y	25	0-8	0-8	↓	?	57	11-8-6	0-8-7
Z	Z	26	0-9	0-9	<	<	58	12-0 or	12-8-4 or
0	0	27	0	0				12-8-2†††	12-0†††
1	1	28	1	1	>	>	59	11-8-7	0-8-6
2	2	29	2	2	≤	@	60	8-5	8-4
3	3	30	3	3	≥	\	61	12-8-5	0-8-2
4	4	31	4	4	¬	^circumflex(↑)‡	62	12-8-6	11-8-7
5	5	32	5	5	; semicolon	; semicolon	63	12-8-7	11-8-6

† Twelve or more zero bits at the end of a 60-bit word are interpreted as end-of-line mark rather than two colons.

†† In installations using either 63-graphic set, numeric code 00 has no associated graphic of Hollerith code; numeric code 51 is the colon (8-2 punch).

††† The alternate Hollerith (026) and ASCII (029) punches are accepted for input only.

‡ Character ↑ is equivalent to ASCII Graphic character ∧ or '.

KRONOS/NOS CHARACTER SETS

There are two KRONOS/NOS character sets applicable to BASIC: Table A-2, TTY Character Set, lists the characters for terminal use; Table A-3 lists the character set used in batch operations.

To use the 12-bit ASCII characters in the TTY character set, the user must be in ASCII Mode; see the appropriate Time-Sharing User's Guides, ASCII command.

TABLE A-2. KRONOS/NOS TTY CHARACTER SET

Char	Numeric Code	Char	Numeric Code
(unused)	00	5	32
A	01	6	33
B	02	7	34
C	03	8	35
D	04	9	36
E	05	+	37
F	06	-	38
G	07	*	39
H	08	/	40
I	09	(	41
J	10	)	42
K	11	$	43
L	12	=	44
M	13	(space)	45
N	14	,	46
O	15	.	47
P	16	"	48
Q	17	[	49
R	18	]	50
S	19	:	51
T	20	'	52
U	21	&	53
V	22	CR	54
W	23	LF	55
X	24	↑	56
Y	25	#	57
Z	26	<	58
0	27	>	59
1	28	(unused)	-
2	29	?	61
3	30	(unused)	-
4	31	;	63

†On TTY models having no left arrow, the underline character takes its place.

‡For version 1.0 compatibility.

Char	Numeric Code	Char	Numeric Code

Caution: Each of the following characters are considered by BASIC as one logical character for string manipulation and I/O operations. Each of these characters requires the storage space of two 6-bit characters.

@	3841	DC2	3997
%	3842	ETX	3998
\	3843	DC4	3999
←	3844	NAK	4000
X-ON	3845	SYN	4001
X-OFF	3846	ETB	4002
a	3969	CAN	4003
b	3970	EM	4004
c	3971	VT	4005
d	3972	SOH	4006
e	3973	!	4007
f	3974	SI	4008
g	3975	BS	4009
h	3976	HT	4010
i	3977	EOT	4011
j	3978	GS	4012
k	3979	NUL	4013
l	3980	FF	4014
m	3981	SO	4015
n	3982	STX	4016
o	3983	{	4017
p	3984	}	4018
q	3985	SUB	4019
r	3986	ACK	4020
s	3987	&	4021
t	3988	`	4022
u	3989	¦	4023
v	3990	~	4024
w	3991	#	4025
x	3992	FS	4026
y	3993	RS	4027
z	3994	DEL	4028
DLE	3995	US	4029
BELL	3996	ENQ	4030
		ESC	4031

TABLE A-3. BATCH CHARACTER SET EQUIVALENCES

Char.	Numeric Code	(026) Hollerith	Char.	Numeric Code	(026) Hollerith
A	01	12-1	6	33	6
B	02	12-2	7	34	7
C	03	12-3	8	35	8
D	04	12-4	9	36	9
E	05	12-5	+	37	12
F	06	12-6	-	38	11
G	07	12-7	*	39	11-8-4
H	08	12-8	/	40	0-1
I	09	12-9	(	41	0-8-4
J	10	11-1	)	42	12-8-4
K	11	11-2	$	43	11-8-3
L	12	11-3	=	44	8-3
M	13	11-4	space	45	space
N	14	11-5	,	46	0-8-3
O	15	11-6	.	47	12-8-3
P	16	11-7	†† ≡ (")	48	0-8-6
Q	17	11-8	[	49	8-7
R	18	11-9	]	50	0-8-2
S	19	0-2	:	51	8-2
T	20	0-3	≠	52	8-4
U	21	0-4	→	53	0-8-5
V	22	0-5	∨	54	11-0
W	23	0-6	∧	55	0-8-7
X	24	0-7	†† ↑	56	11-8-5
Y	25	0-8	↓ (#)	57	11-8-6
Z	26	0-9	<	58	12-0
0	27	0	>	59	11-8-7
1	28	1	≤	60 †	8-5
2	29	2	≥	61	12-8-5
3	30	3	¬	62 †	12-8-6
4	31	4	;	63	12-8-7
5	32	5			

†BASIC considers these 6-bit characters as the front end of an ASCII 12-bit character and should be avoided in batch operations.

† 11-0 and 11-8-2 are equivalent

‡ 12-0 and 12-8-2 are equivalent

†† BASIC characters are in parentheses.

DIAGNOSTICS B

All diagnostic messages produced by the BASIC system at compile and execution time are printed in the following format:

message AT line-number

The line number indicates the statement in error.

COMPILE TIME DIAGNOSTICS

The following messages can be produced during program compilation. All compile time diagnostics inhibit execution of the program and BASIC writes a dayfile message: BASIC COMPILATION ERRORS.

Message	Error
BASIC FIELD LENGTH TOO SHORT	Field length too short to begin compiling; increase FL and run again.
BLANK FILE STATEMENT	File ordinal or name missing.
DELIMITER OVERFLOW	More than three delimiters used.
DUPLICATE LINE NO	Same line number used twice.
END NOT LAST	END statement placed prior to the last statement in a BASIC program.
EXCEED LITERAL TBL	Too many literals used.
FOR NEST TOO DEEP	More than ten nested FOR statements.
FOR WITHOUT NEXT	FOR statement has no balancing NEXT statement.
ILLEGAL BOUND	DIM statement declared variable greater than 131071.
ILLEGAL CHARACTER	Unrecognizable character.
ILLEGAL COMPARISON	Numeric quantity compared to string in IF statement.
ILLEGAL FILE NAME	Name is not allowed as a KRONOS or SCOPE file name.
ILLEGAL FILE NUMBER	Number in FILE statement zero or $>(2^{18}-1)$.
ILLEGAL FN NAME	User function name not of the form FNx; x is any alphabetic character.
ILLEGAL LINE REF	Incorrectly written line number, or line number referenced >99999.
ILLEGAL LINE NUMBER	Line number >99999.

Message	Error
ILLEGAL MARGIN	Margin $<$ 15 or $>$160.
ILLEGAL NUMBER	Numeric constant incorrectly written.
ILLEGAL OPERAND	String used in arithmetic expression.
ILLEGAL STATEMENT	Statement does not begin with a recognizable word or is written incorrectly.
ILLEGAL STRING	String constant incorrectly written.
ILLEGAL USING	Using clause incorrect syntax.
ILLEGAL WORD	Unrecognizable word.
INVALID BASE STATEMENT	BASE statement after DIM statement or array reference.
INVALID BASE VALUE	BASE not 0 or 1.
INVALID CHANGE	CHANGE argument is not a string or string variable.
LINES OUT OF ORDER	Line numbers not in ascending order.
MISSING LINE NO	Statement written without a line number.
NEXT WITHOUT FOR	NEXT statement has no balancing FOR statement.
NON FORMAT CALLED	PRINT USING references a statement which is not an image.
OUT OF SPACE*	*Program too large to continue checking source statements; compilation stops.
PROGRAM TOO LARGE	Compiled program too large to execute in the field length given to the BASIC compiler.
PROGRAM TOO LONG*	*Program too long to compile, but the compiler continues to check the source statements.
READ WITHOUT DATA	Program containing a READ statement has no DATA statements.
RECURSIVE FN	Recursive DEF statement; illegal usage.
STRING TOO LONG	String longer than 78 characters.
TOO MANY FILES	The user has more than 15 files including INPUT and OUTPUT.
UNDEFINED FN REF	Undefined user function.
UNDEFINED LINE REF	Line number referenced does not exist.

* If any of these occur, the program should be recompiled with a larger field length. No maximum size is defined for BASIC programs. Limits depend entirely on the field length given to the BASIC system.

The following messages are all caused by incorrect image in IMAGE statement or in STRING FUNCTION (STR$).

Message	Error
EMPTY FORMAT	Must be exactly 5 (↑) in the IMAGE statement.
EXPONENT COUNT ILLEGAL	
FIELD TOO LONG	More than 16 pound signs.
ILLEGAL EXPONENT SIGN	
ILLEGAL FORMAT	Field absent or more than 1, or right parenthesis missing.
ILLEGAL MINUS	
ILLEGAL PERIOD	
ILLEGAL PLUS	
ILLEGAL POUND	
ILLEGAL QUOTES	
ILLEGAL SEQUENCE	
INCOMPLETE FORMAT	
OPEN QUOTES	

EXECUTION TIME DIAGNOSTICS

BASIC allows two modes of execution-time error processing.

During normal error processing the following diagnostics may occur during execution of a program. All errors terminate execution and cause the message BASIC EXECUTION ERROR to be written.

When an error statement is in effect, normal processing is suppressed and the program retains control. The program can then inspect the error number by using the ESM function.

Message	Error Number	Error
ARGUMENT IS POLE IN TAN	153	Argument is $n\pi/2$.
ARGUMENT NEGATIVE IN LOG	154	Must supply positive argument.
ARGUMENT NEGATIVE IN SQR	160	Must supply positive argument.
ARGUMENT TOO LARGE IN COS	152	Argument must be < 2.21069E14.
ARGUMENT TOO LARGE IN SIN	150	Argument must be < 2.21069E14.
ARGUMENT TOO LARGE IN TAN	151	Argument must be < 2.21069E14.
ARGUMENT IS ZERO IN LOG	155	

Message	Error Number	Error
ARRAY TOO SMALL	163	Self-explanatory.
BAD DATA IN READ	126	String read, numeric expected; or vice versa.
BASIC SYSTEM ERROR		Malfunction of BASIC system. Please report the problem.
DIVISION BY ZERO	125	Self-explanatory.
END OF DATA	120	READ statement executed when internal data block is exhausted.
END OF DATA ON FILE	136	READ FILE or INPUT FILE statement executed after file data is exhausted.
GOSUB NEST TOO DEEP	123	More than 40 GOSUB's nested.
ILLEGAL CHARACTER	165	Self-explanatory.
ILLEGAL FILE NAME	139	File name does not correspond to alphanumeric rule.
ILLEGAL FILE NO	138	File number referenced is less than one or exceeds the value $2^{18}-1$.
ILLEGAL I/O ON FILE	137	I/O operation not legal for current mode of file; e.g., READ in a write mode file; READ on a BCD file; MARGIN on a binary file, etc. File must be RESTOREd to change mode.
ILLEGAL LABEL	170	Label referenced in a JUMP statement or NXL function does not exist; is greater than 99999; or is the label of a REM statement.
ILLEGAL MARGIN	131	Margin specified is outside the allowable range 15 to 160.
ILLEGAL INPUT ON FILE	135	INPUT statement attempted to read a file in incorrect format.
ILLEGAL OUTPUT ON FILE	130	Type of input/output statement changed without repositioning the file to beginning of information.
ILLEGAL SUBSTR PARAMETER	169	Parameters specified in SUBSTR function are outside the legal range as determined by the actual string length.
INDEFINITE OPERAND	111	Self-explanatory.
INFINITE AND INDEFINITE OPERAND	113	Self-explanatory.
INFINITE OPERAND	109	Self-explanatory.

Message	Error Number	Error
INVALID LENGTH	164	Length >78 in CHANGE.
MATRIX DIMENSION ERROR	161	Dimension inconsistency in one of the MAT statements or is >50x50 for INV function.
NEARLY SINGULAR MATRIX	162	Attempt to invert a singular or nearly singular matrix.
NEGATIVE NUMBER TO POWER	158	Negative number raised to non-integer exponent.
NO FILE SPACE	140	User must specify another FILE statement.
NON-NUMERIC STRING	167	String is non-numeric in VAL function.
ON EXPRESSION OUT OF RANGE	122	Expression in ON statement is negative, zero, or exceeds the count of line numbers.
POWER TOO LARGE	159	Self-explanatory.
RETURN BEFORE GOSUB	124	RETURN statement has been executed and no GOSUB is in effect.
STRING OVERFLOW	168	Attempt to define a string which is more than 78 characters long.
SUBSCRIPT ERROR	121	Attempt to reference an element outside bounds of an array.
TIME LIMIT EXCEEDED	100	Program exceeded its time limit.
TYPE 2 PROGRAM ABORT	102	PP abort.
TYPE 3 PROGRAM ABORT	103	CP abort.
TYPE 4 PROGRAM ABORT	104	PP call error.
TYPE 5 PROGRAM ABORT	105	Operator drop.
TYPE 6 PROGRAM ABORT	106	Program stop.
TYPE 7 PROGRAM ABORT	107	Unknown abort.
TYPE 8 PROGRAM ABORT	108	Address out of range.
TYPE 10 PROGRAM ABORT	110	Address out of range and infinite operand.
TYPE 12 PROGRAM ABORT	112	Address out of range and indefinite operand.
TYPE 14 PROGRAM ABORT	114	Address out of range and infinite operand and indefinite operand.
UNDEFINED FILE NO	141	Self-explanatory.
UNDEFINED VALUE		Self-explanatory.
ZERO TO A NEGATIVE POWER	157	Self-explanatory.

The following messages may occur after the data typed in response to an INPUT request from the terminal has been checked. The BASIC system will print a question mark and the user should retype the data.

Message	Error Number
ILLEGAL INPUT, RETYPE INPUT	133
TOO MUCH DATA, RETYPE INPUT	132

The following message may occur after the data typed in response to an input request from the terminal has been checked. The user should continue typing data until the requirements of the input request are satisfied.

NOT ENOUGH DATA, TYPE IN MORE	134

For ease of reference, diagnostics have been arranged by the error numbers in ascending order.

Error Number	Error
100	Program exceeded its time limit.
101	Not used.
102	PP abort.
103	CP abort.
104	PP call error.
105	Operator drop.
106	Program stop.
107	Unknown abort.
108	Address out of range.
109	Infinite operand.
110	Address out of range and infinite operand.
111	Indefinite operand.
112	Address out of range and indefinite operand.
113	Infinite and indefinite operand.
114	Address out of range and infinite operand and indefinite operand.
115	Not used.
116	Not used.
117	Not used.
118	Not used.
119	Not used.

Error Number	Error
120	READ statement executed when the internal data block has been exhausted.
121	Attempt to reference an element outside bounds of array.
122	Expression in ON statement is negative, zero, or exceeds the count of line numbers.
123	More than 40 GOSUB's nested.
124	RETURN statement has been executed and no GOSUB is in effect.
125	Division by zero.
126	String read, numeric expected; or vice versa.
127	Not used.
128	Not used.
129	Not used.
130	Type of input/output statement changed without repositioning the file to beginning of information.
131	Margin specified is outside the allowable range 15 to 160.
132	Too much data, retype input.
133	Illegal data, retype input.
134	Not enough data, type in more.
135	INPUT statement attempted to read a file in incorrect format.
136	READ FILE or INPUT FILE statement executed after file data is exhausted.
137	I/O operation not legal for current mode of file; e.g., READ in write mode file; READ on BCD file; MARGIN on binary file; etc. File must be RESTOREd to change mode.
138	File number referenced is less than one or exceeds the value $2^{18}-1$.
139	File name does not correspond to alphanumeric rule of formation.
140	Attempt to specify more than three files by the execution of only one FILE statement.
141	Undefined file number.
142	Not used.

Error Number	Error
143	Not used.
144	Not used.
145	Not used.
146	Not used.
147	Not used.
148	Not used.
149	Not used.
150	Argument too large in SIN.
151	Argument too large in TAN.
152	Argument too large in COS.
153	Argument is POLE in TAN.
154	Argument is negative in LOG.
155	Argument is zero in LOG.
156	Argument is too large in exponent.
157	Zero to a negative power.
158	Negative number to a power.
159	Power too large.
160	Argument is negative in a square root.
161	Matrix dimension error.
162	Nearly singular matrix.
163	Array too small.
164	Invalid length.
165	Illegal character.
166	Not used.
167	String is non-numeric in VAL function.
168	String overflow.
169	Parameters specified in SUBSTR function are outside the legal range as determined by the actual string length.
170	Illegal label does not exist or >99999 in JUMP statement or NXL function.

INDEX OF BASIC STATEMENTS C

The following alphabetical list of BASIC statements gives the formats, functions, and pages on which they appear. Throughout this appendix the following abbreviations are used:

a	array identifier	na	numeric array name
c	numeric or string constant	nc	numeric constant
ch	any character or carriage return $\textcircled{CR}$	ne	numeric expression, constant or variable
d	delimiter	r	relational expression
e	expression, constant or variable	sc	string constant
f	format specification	se	string expression, constant or variable
l	letter	snv	simple numeric variable
lfn	file name	sv	string variable
ln	line number	v	variable identifier (simple, subscripted, numeric, or string)
m	matrix identifier (one-or-two-dimensional array)	x	0 or 1

Items enclosed in brackets [ ] are optional.

Shaded formats are retained for compatibility purposes and should not be used in new programs.

Statement Format	Function	Page No.
BASE x	Defines the origin of arrays.	2.2-2
CHANGE na TO sv	Creates string from an array containing display code characters.	2.2-9
CHANGE sv TO na	Stores each character of a string into a separate element of an array.	2.2-9
DATA $c_1, c_2, c_3, \ldots c_n$	Creates a block of data internal to the BASIC program.	2.4-4
DEF FNl (snv) = ne	Defines a new function to be used within a BASIC program.	2.3-13
DELIMIT $(ch_1), \ldots (ch_3)$	Defines separators between input items on the terminal.	2.4-12

Statement Format	Function	Page No.
DELIMIT #ne(ch_1),...(ch_3) DELIMIT FILE (lfn) (ch_1),...(ch_3) DELIMIT : lfn : (n_1),...(n_3)	Defines separators between input items on specified file.	2.4-12
DIM $a_1(nc_1, \ldots, nc_3) \ldots, a_n(nc_1, \ldots, nc_3)$	Declares the dimensions of an array variable. nc may be 1 to 3 integers.	2.2-1
END	Terminates a program.	2.2-9
FILE $\#ne_1 = lfn_1, \#ne_2 = lfn_2, \ldots \#ne_n = lfn_n$	Defines file ordinal and equates it to a file name.	2.4-3
FOR snv = ne_1 TO ne_2 [STEP ne_3]	Begins a program loop; this statement must be used with a NEXT statement.	2.2-6
GOSUB ln	Transfers program control to a subroutine beginning at line number indicated.	2.3-17
GOTO ln	Interrupts the normal sequence of program execution and transfers program control to indicated line number.	2.2-5
IF r THEN ln	Transfers program control to indicated line number if certain conditions are met.	2.2-4
(IMAGE) : [literal] $f_1f_2f_3 \ldots f_n$	Specifies output formats.	2.4-23
INPUT $v_1, v_2, \ldots v_n$	Enters data from a terminal.	2.4-9
INPUT #ne $v_1, v_2, \ldots v_n$	Inputs coded data from specified file #ne.	2.4-9
INPUT FILE (lfn) $v_1, v_2, \ldots v_n$	Inputs coded from specified file lfn.	-
INPUT : lfn : $v_1, v_2, \ldots v_n$	Inputs coded data from specified file lfn.	-
JUMP ne	Transfers control to statement where line number = ne.	2.6-2
[LET] $v_1 = v_2 = v_3 \ldots v_n = e$	Assigns a value to a variable during program execution.	2.2-3
MARGIN ne_2	Defines a right-hand margin for output to a terminal.	2.4-32
MARGIN $\#ne_1, ne_2$	Defines right-hand margin for output to a specified file.	2.4-32

Statement Format	Function	Page No.
MARGIN FILE (lfn) ne_2	Defines right-hand margin for output to a specified file.	-
MARGIN : lfn : ne_2	Defines right-hand margin for output to a specified file.	-
MAT $m_1 = m_2 + m_3$	Matrix addition.	2.5-2
MAT $m_1 = m_2 - m_3$	Matrix subtraction.	2.5-2
MAT $m_1 = m_2 * m_3$	Matrix multiplication.	2.5-2
MAT $m_1 = e * m_2$	Matrix scalar multiplication by value of an expression.	2.5-2
MAT m = INV(a)	Inverts a matrix.	2.5-4
MAT m = TRN(a)	Transposes a matrix.	2.5-4
MAT m = ZER[(ne_1[, ne_2])]	Creates a matrix of all zeros.	2.5-4
MAT m = CON[(ne_1[, ne_2])]	Creates a matrix of all ones.	2.5-4
MAT m = IDN [(ne_1[, ne_2])]	Creates an identity matrix.	2.5-4
MAT READ m_1, m_2, m_3...	Reads matrices from internal data file.	2.5-7
MAT READ FILE (lfn) m_1, m_2, m_3...	Reads matrices from specified file lfn in binary format.	-
MAT READ #ne, m_1, m_2, m_3...	Reads matrices from file #ne in binary format.	2.5-7
MAT READ : lfn: m_1, m_2, m_3...	Reads matrices from file lfn in binary format.	-
MAT PRINT $m_1 d m_2 d m_3 d$...d	Prints matrices on a terminal.	2.5-9
MAT PRINT FILE (lfn) $m_1 d m_2 d$...d		
MAT PRINT #ne, $m_1 d m_2 d$...d	Prints matrices in a coded format on specified file.	2.5-9
MAT PRINT : lfn : $m_1 d m_2 d$...d		-
MAT INPUT m_1, m_2, m_3, ...	Inputs matrices from a terminal.	2.5-8
MAT INPUT FILE (lfn) m_1, m_2, m_3, ...	Inputs matrices from specified file lfn (coded format).	-
MAT INPUT #ne, m_1, m_2, m_3, ...	Inputs matrices from a file #ne.	2.5-8
MAT INPUT : lfn : m_1, m_2, m_3, ...	Inputs matrices from a file lfn.	-

Statement Format	Function	Page No.
MAT WRITE FILE (lfn) $m_1, m_2, m_3, \ldots$		-
MAT WRITE #ne, $m_1, m_2, m_3, \ldots$	Writes matrices in binary format on a specified file.	2.5-9
MAT WRITE : lfn : $m_1, m_2, m_3, \ldots$		
NEXT snv	Terminates a program loop or increments the value tested by the loop.	2.2-6
NODATA ln	Tests data pointer for increment beyond end of data block. Branches to ln if data exhausted.	2.4-6
NODATA #ne, ln		2.4-6
NODATA FILE (lfn)ln	Transfers program control to specified line number if file is positioned at end of information.	-
NODATA : lfn : ln		-
ON ne GOTO $ln_1, ln_2, ln_3, \ldots ln_n$	Expression is evaluated and truncated to an integer value; transfers control to line number ln_1 if ne=1, line number ln_2 if ne=2, etc.	2.2-5
ON ERROR GOTO ln	Transfers control to ln on run-time error.	2.6-1
ON ERROR THEN ln	Transfers control to ln on run-time error.	2.6-1
ON ERROR	Turn on normal error processing.	2.6-1
PRINT $e_1de_2d\ldots e_nd$	Prints data at terminal.	2.4-14
PRINT USING ln, $e_1de_2d\ldots e_nd$	Output to be formatted on an IMAGE statement on a terminal.	2.4-14
PRINT #ne, $e_1de_2d\ldots e_nd$	Prints data on specified file.	2.4-14
PRINT #ne USING ln, $e_1de_2d\ldots e_nd$	Output to be formatted to an IMAGE statement on specified file.	2.4-14
PRINT (lfn) $e_1de_2d\ldots e_nd$	Prints data on specified file.	-
PRINT : lfn : USING ln, $e_1de_2d\ldots e_nd$	Output to be formatted to an IMAGE statement on specified file.	-

Statement Format	Function	Page No.
READ $v_1, v_2, v_3, \ldots v_n$	Accesses data created by DATA statements.	2.4-7
READ #ne, $v_1, v_2, v_3, \ldots v_n$	Reads binary data from named file created by WRITE FILE statements.	2.4-7
READ FILE (lfn) $v_1, v_2, v_3, \ldots v_n$		-
READ : lfn : $v_1, v_2, v_3, \ldots v_n$		-
REM ch...ch	Inserts explanatory remarks into a program.	2.2-1
RESTORE	Reinitializes data printer to the first word of the data block.	2.4-5
RESTORE #ne	Sets named file to beginning of information.	2.4-5
RESTORE FILE (lfn)		-
RESTORE: lfn :		-
RETURN	Resumes execution at statement following most recently executed statement.	2.2-17
SETDIGITS ne	Specifies number of significant digits for output.	2.4-31
STOP	Terminates program execution at places other than the END statement.	2.2-9
WRITE #ne, $e_1, e_2, e_3, \ldots e_n$	Writes data in binary format on specified file.	2.4-8
WRITE FILE (lfn) $e_1, e_2, e_3, \ldots e_n$		-
WRITE : lfn : $e_1, e_2, e_3, \ldots e_n$		-

INDEX OF BASIC FUNCTIONS D

Legend:		
nc	-	numeric constant
ne	-	numeric expression, constant or variable
se	-	string expression, constant or variable
sv	-	string or string variable
f	-	format specification

Function	Meaning	Page
ABS(x)	Finds the absolute value of x.	2.3-2
ATN(x)	Finds the arctangent of x in the principal value range - $\pi/2$ to + $\pi/2$.	2.3-2
CLK(x)	Returns the time of day in hours and fractions of an hour in a 24 hour scale.	2.3-7
CLK$	Determines time of day as a string.	2.3-7
COS(x)	Finds the cosine of x expressed in radians.	2.3-2
DAT$	Determines the date as a string.	2.3-7
EXP(x)	Finds the value of e ↑ x.	2.3-2
INT(x)	Finds the largest integer not greater than x. Example: INT(5.95) = 5 and INT(-5.95) = -6.	2.3-2
LEN(sv)	Determines current length of a string.	2.3-8
LGT(x)	Finds the base 10 logarithm of x; x>0, otherwise an execution error will cause the program to terminate.	2.3-2
LOG(x)	Finds the natural logarithm of x; x> 0, otherwise an execution error will cause the program to terminate.	2.3-2
RND(x)	Returns pseudo-random numbers from the set of numbers uniformly distributed over the range $0 \leq$ RND (x) $<$ 1.0. If x> 0 a random number sequence is initialized based on the value of x and the first number of the sequence is returned. If x = 0 the next number in the established sequence of random numbers is returned. If the sequence was not previously established by an x>0 RND reference, a standard constant is used to initiate the sequence. If x < 0 the first RND reference initializes a random number sequence based on the time of day and returns the first value of the sequence. Subsequent x < 0 RND references return the next number in the sequence.	2.3-3

Function	Meaning	Page
ROF(x) or ROF(x,nc)	Finds the value of the first argument rounded to the number of decimal places specified by the second argument. Omission of nc rounds variable x to the nearest integer.	2.3-2
SGN(x)	Assigns a value of 1 if x is positive; 0 if x is 0; or -1 if x is negative.	2.3-2
SIN(x)	Finds the sine of x expressed as an angle in radians.	2.3-2
SQR(x)	Finds the square root of x; $x > 0$, otherwise an execution error will cause the program to be terminated.	2.3-2
STR$(ne) or STR$(ne, f)	Converts numeric value to string representation.	2.3-9
SUBSTR(se, ne_1, ne_2) or SUBSTR(se, ne_1)	Extract or insert a substring.	2.3-10
TAB(ne)	Move print line to position (ne).	2.4-22
TAN(x)	Finds the tangent of x expressed in radians.	2.3-2
TIM(x)	Elapsed time in seconds (x is a dummy argument).	2.3-7
VAL(se)	Converts a string to its numeric value.	2.3-12

KRONOS/NOS FILE HANDLING E

A file is a collection of information with an associated name. A BASIC program is an example of a file. Another file may contain data to be read in and used by a separate BASIC program. Therefore, all or part of the output from a program may be stored in a file instead of being printed at the terminal. This file might then be listed on a teletype or on a high-speed printer, or it may simply be used as data for another program.

KRONOS and NOS recognize two types of files: local and permanent. A local file is one which ceases to exist (is released) when the user disconnects or issues a NEW, OLD or LIB command. Before any file can be used, it must be "made local". A permanent file is one which remains in the KRONOS/NOS permanent file system after the user signs off. There may be both a local and a permanent copy of the same file. When the user signs off, the permanent copy is retained; the local copy is released.

There are two types of permanent files: indirect access and direct access. An indirect access file is so named because it is used"indirectly"; it is always a separate local copy of the permanent file that is used. With a direct access file, the permanent copy itself is made local. (See figure E-1.) An indirect access file is created using the KRONOS system commands NEW and SAVE; a local copy is made by either the OLD or GET commands; it is updated by the REPLACE command and released from use (but not from permanent storage) by the RETURN command. A direct access file is created by the DEFINE command; it is opened for access (made local) by the ATTACH command and released from use by the RETURN command. The PURGE command is used to remove from permanent storage both direct and indirect access files.

When a file is "made local", it becomes either a primary or a secondary file. The local file established by a NEW, OLD or LIBRARY command is always primary. The NEW command creates a primary file; the OLD and LIBRARY commands obtain a primary file from an indirect access file. There can be only one primary file and usually it is the program to be run. When the commands LIST, SAVE or RUN are issued, the operating system assumes it refers to the primary file. The GET, or ATTACH commands establish a secondary file. There may be as many as eight secondary files and usually these are data input and output files. To refer to a secondary file with a KRONOS command, the file name must be specified as in: LIST, F=DAT or SAVE, DAT. In the latter, file DAT is retained as a permanent file; DAT could be a primary or secondary file. When the current primary file is released by entry of OLD, NEW, or LIBRARY commands, all primary and secondary files are released unless the next command encountered is a NODROP command.

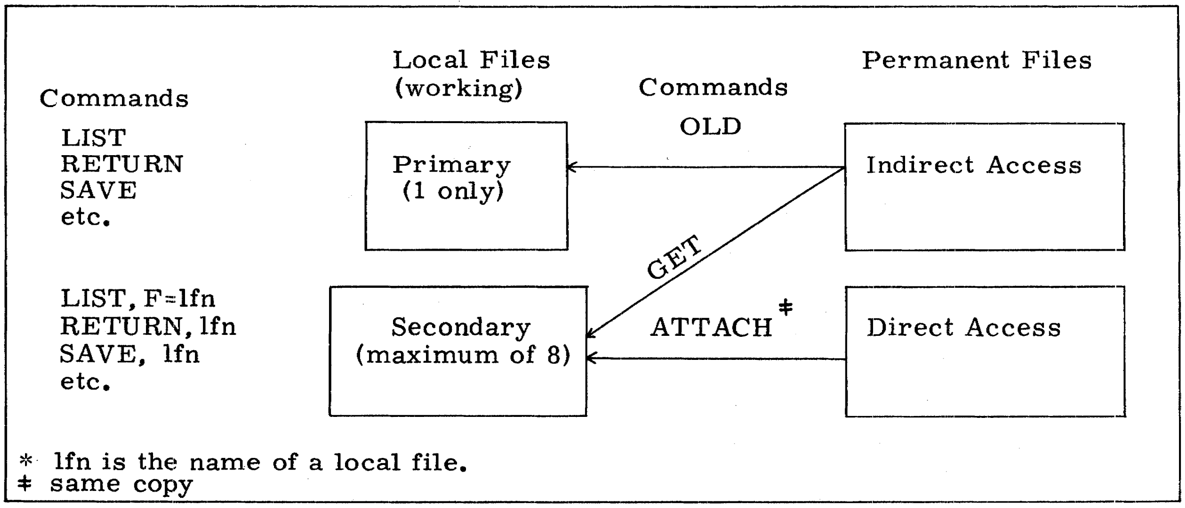


Figure E-1. KRONOS Files

KRONOS/NOS FILE CONTROL COMMANDS

The following includes a brief description of some KRONOS/NOS file manipulation commands. A user can obtain specific information pertaining to his permanent files by using the CATLIST command described in Section 5 of the appropriate KRONOS 2.1 and NOS Time-Sharing User's Reference Manual.

DIRECT ACCESS PERMANENT FILES

- DEFINE, lfn=pfn/CT=n, M=m, NA — stores local file name as a permanent file to be later referenced by permanent file name.

where:

lfn	-	If DEFINE is to be used to create an empty direct access permanent file, lfn is specified only if the user desires to reference the file by a name other than its permanent file name. If DEFINE is to be used to define an existing local file as a direct access file, lfn is the name of the local file. Also, if lfn exists, its position is not altered.
pfn	-	Permanent file name. If pfn is omitted, the system assumes lfn =pfn .

CT - Permanent File Category:

P	=	private
S	=	semi-private
PU	=	public

M - File or User Permission:

W	=	write permission
M	=	modify permission
RM	=	read in modify mode
RA	=	read in append mode
E	=	execute file permission

NA - if a resource is unavailable KRONOS suspends user requests until resource is free.

- ATTACH,lfn=pfn/M=m, NA — obtains a file defined by a DEFINE command for use during program executions.

where:

lfn=pfn - is used when desirable to reference an attached file by other than its permanent file name. If a current working file is referenced as lfn, the contents of that file are lost when the permanent file is attached.

M=m - modify permission. If omitted, the system assumes read permission only.

NA - User wishes to wait for direct access file to become available. If the file is currently being accessed, the user's job is suspended. Enter STOP to terminate the request.

For a description of the remaining parameters, see DEFINE command previously described.

- CHANGE See Indirect Access Permanent Files.

INDIRECT ACCESS PERMANENT FILES

• SAVE, lfn=pfn/M=m, NA	creates an indirect access permanent file; permits the user to retain a copy of the specified working file on the permanent file system.

For a description of the statement parameters, see DEFINE command previously described.

• GET, lfn=pfn/NA	retrieves a copy of a specified indirect access file for use as a working file. If the user wishes to reference the working file by a name other than the pfn, the lfn parameter is used. The current primary file remains primary unless the filename specified by lfn is that of the current primary file. In this case, the contents of the primary file are replaced by a copy of pfn which becomes the new primary file.

For a description of the statement parameters, see DEFINE command previously described.

• OLD,lfn=pfn	requests a copy of the specified permanent file as a primary file. When a specific subsystem is associated with the file, it is selected automatically. This occurs only if the file was originally a primary file and saved while a subsystem other than the subsystem null is active.
• LIBRARY, lfn=pfn	requests a copy of specified indirect access permanent files from the catalog of a special user library; this file becomes a primary file.

For a description of the command parameters, see DEFINE command previously described.

- CHANGE, nfn=ofn/CT=n, M=m, NA — allows use of direct or indirect access permanent file to alter several parameters without further operation on the file (attach, save, etc.). This is valid only for the originator of the file.

where:

nfn	new permanent file name to be assigned
ofn	current permanent file name
CT, M	specify only if these are to be changed. For a description of the command parameters, see DEFINE command.

- REPLACE, lfn=pfn/NA — permits the user to replace the contents of an indirect access permanent file with the contents of a working file. If pfn does not exist, a new permanent file is created.

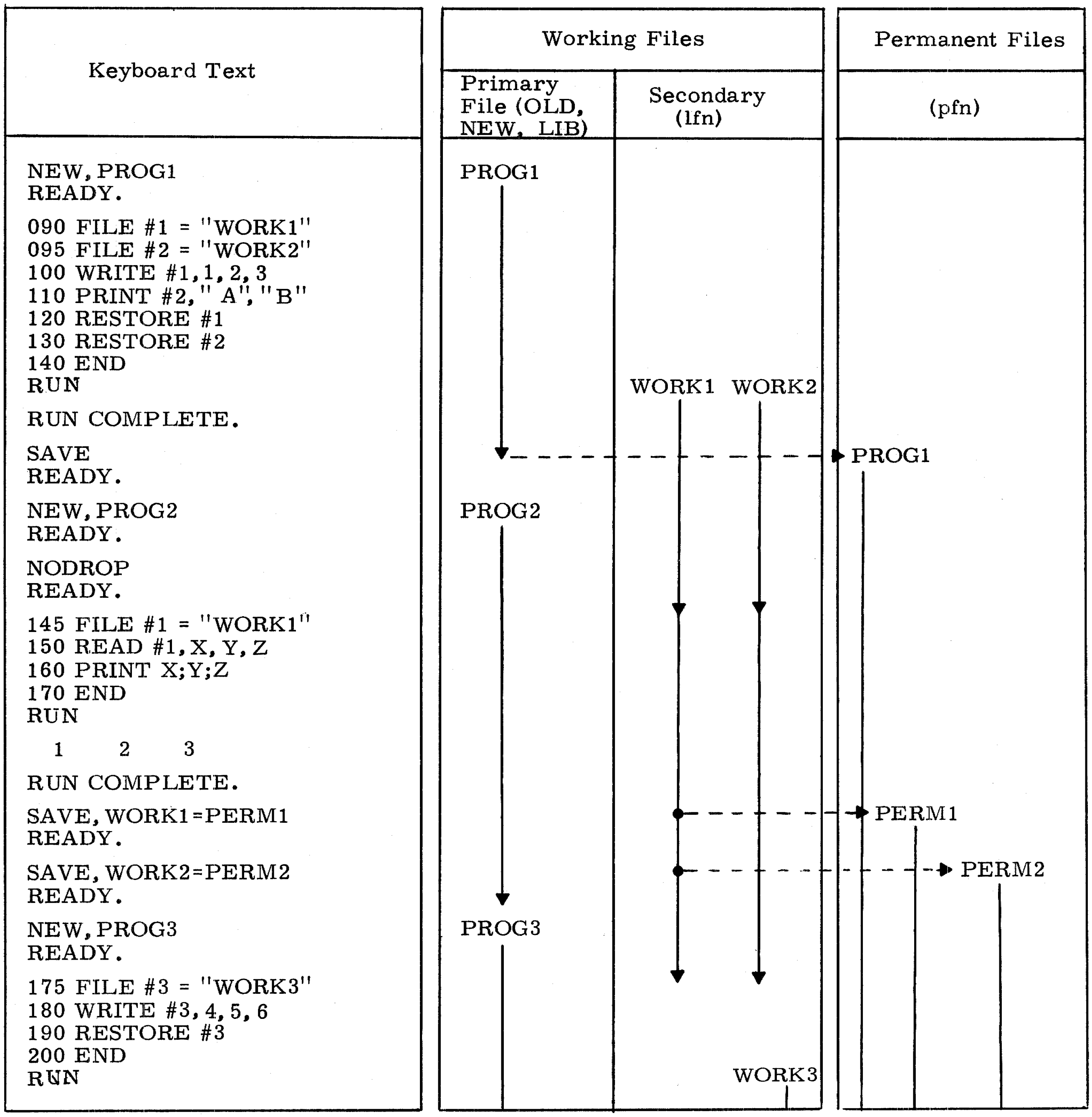
For a description of the command parameters, see DEFINE command previously described.

FILE CONTROL COMMANDS EXAMPLE

The following presents a series of demonstration programs that illustrate the use of system commands to create, reference, list, and purge files via the time-sharing terminal. The example is divided into three main columns. The left-most column contains a transcript of the text entered and received at the terminal. The center column represents the area of working files. It is divided into two sections: the left section shows the life span of each program (primary file) entered; the right section is the area of the remaining local files and shows when temporary files enter the working area and how long they endure. The right-most column is the area of permanent files. It shows when a copy of a working file is made into a permanent file and how long that permanent file exists.

Duration of a file is indicated by a solid vertical line. An arrow point signals termination. The copying of a file from lfn to pfn, or the reverse, is indicated by a broken horizontal line. An arrow point designates destination.

For a complete explanation of system commands, consult the KRONOS and NOS Time-Sharing Users Reference Manuals.

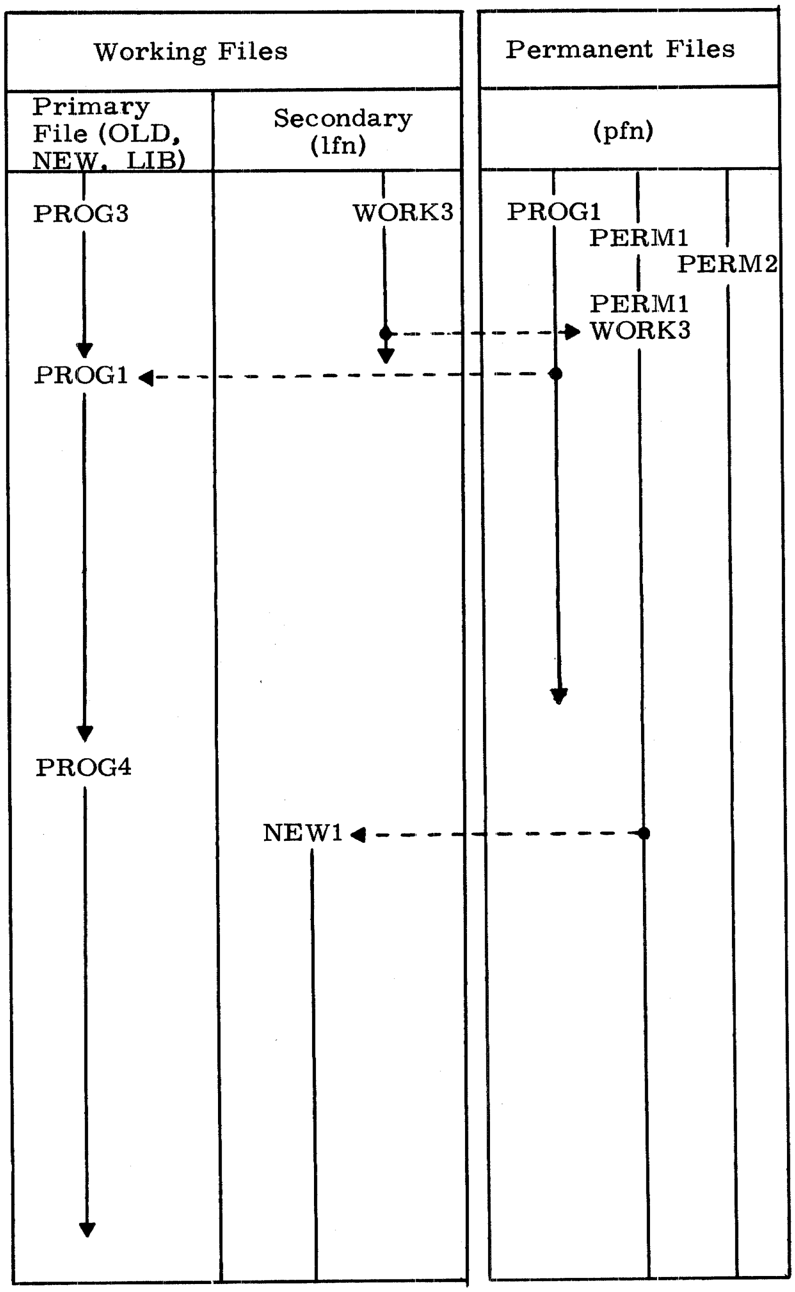


```
NEW,PROG1
READY.
090 FILE #1 = "WORK1"
095 FILE #2 = "WORK2"
100 WRITE #1,1,2,3
110 PRINT #2," A","B"
120 RESTORE #1
130 RESTORE #2
140 END
RUN
RUN COMPLETE.
SAVE
READY.
NEW,PROG2
READY.
NODROP
READY.
145 FILE #1 = "WORK1"
150 READ #1,X,Y,Z
160 PRINT X;Y;Z
170 END
RUN
  1      2      3
RUN COMPLETE.
SAVE,WORK1=PERM1
READY.
SAVE,WORK2=PERM2
READY.
NEW,PROG3
READY.
175 FILE #3 = "WORK3"
180 WRITE #3,4,5,6
190 RESTORE #3
200 END
RUN
```

Keyboard Text

```
RUN COMPLETE.


APPEND,PERM1,WORK3
READY.
OLD,PROG1
READY.
LIST
090 FILE #1 = "WORK1"
095 FILE #2 = "WORK2"
100 WRITE #1,1,2,3
110 PRINT #2,"A","B"
120 RESTORE #1
130 RESTORE #2
140 END
READY.
PURGE,PROG1
READY.
NEW,PROG4
READY.
GET,NEW1=PERM1
READY.
200 FILE #4 = "NEW1"
210 DIM A(15)
220 FOR I=1 TO 15
230 READ #4, A(I)
240 PRINT A(I);
250 NODATA #4,270
260 NEXT I
270 PRINT "ALL OUT"
280 END
RUN
  1      2      3      ALL OUT
RUN COMPLETE.
```



Keyboard Text	Working Files		Permanent Files
	Primary File (OLD, NEW, LIB)	(lfn)	(pfn)
RUN	PROG4	NEW1	PERM1 WORK3
4 5 6 ALL OUT			PERM2
RUN COMPLETE.			
RUN END OF DATA ON FILE AT 230			
RUN COMPLETE.			
CATLIST,LO=0			
CATALOG OF USER007			
FILE NAMES			
PERM2 PERM1			
2 FILE(S)			
READY.			

ANALYSIS OF COMMANDS AND PROGRAM STATEMENTS

NEW, PROG1	The first program to become the primary file is called PROG1. The system responds with READY.
090 095	Assign a file ordinal to file WORK1 and WORK2; these ordinals are used to reference these files in the following program statements.
100	Specifies a binary file called WORK1 with values 1, 2, 3. This first line of coding establishes PROG1 as the primary file.
110	Specifies a coded file called WORK2 with characters A and B.
120	Moves the pointer back to beginning of data for WORK1.
130	Moves the pointer back to beginning of data for WORK2.
RUN	Runs the program. This establishes WORK1 and WORK2 as working files (lfn). The system responds with RUN COMPLETE.
SAVE	This makes the previous program PROG1 a permanent file (pfn).
NEW, PROG2	Specifies a new primary file which will contain new program code. The system responds with READY.
NODROP	This command retains WORK1 and WORK2 in the working area when the new program is entered. Otherwise they would be dropped when the first line of the program was entered. The system responds with READY.
145	The ordinal for WORK1 must be respecified or a diagnostic is returned.
150	Reads in the values from the working file WORK1.
160	Prints out these values.
SAVE, WORK1=PERM1	A copy of WORK1 is made a permanent file with the new name PERM1. The original remains as a working file.
SAVE, WORK2=PERM2	A copy of WORK2 is made a permanent file with the new name PERM2. The original remains as a working file.

NEW, PROG3	Specifies a new primary file, PROG3.
175	A file ordinal is assigned to file WORK3.
180	Specifies a binary work file, WORK3. When this first line of the program is entered (with the carriage return), PROG2 is displaced as the primary file by PROG3. Also, working files WORK1 and WORK2 are dropped at this point. Had a NODROP command been entered just before this line, WORK1 and WORK2 would have been retained as working files.
190	Rewinds WORK3.
RUN	Runs the program and establishes WORK3 as a lfn in the working area.
APPEND, PERM1, WORK3	Adds a copy of WORK3 to the permanent file PERM1. The original of WORK3 remains in working storage.
OLD, PROG1	Makes a copy of permanent file PROG1 and enters it as the primary file. Since the entire program already exists, it cancels all previous working files immediately (not when the first line of the program is entered, as with the NEW command).
LIST	Causes the primary file to be typed out. WORK3 can be listed with the command LIST, F=lfn.
PURGE, PROG1	Removes PROG1 from the permanent files.
NEW, PROG4	Specifies a new program, PROG4.
GET, NEW1=PERM1	Makes a copy of permanent file PERM1 in the working area and gives the copy a temporary name NEW1. (PERM1 has WORK3 appended to it.)
200	Assigns an ordinal to file NEW1.
210	Dimensions a 15-element array. This first line of coding makes PROG4 the primary file.
220	Begins a loop which may go through 15 iterations.
230	Reads one element from file NEW1. Each time through the loop it will read the next element (A(1), A(2), A(3)...)

240	Types out the single element read at line 230.
250	Checks NEW1 to see if there are any more elements to read. When end of data is reached, control is transferred to statement 270.
260	Ends the loop.
270	Types "ALL OUT" to the right of the output data.
RUN	Running the program reads and types out the three values in PERM1. It does not go on to WORK3 because it hits an end-of-record mark (PERM1 and WORK3 are treated as two records in one file designated PERM1).
RUN	Running the program a second time reads the values in the second record WORK3.
RUN	Running the program a third time immediately encounters an end of data and the appropriate message is typed.
CATLIST, LO=0	Gives a catalog of all current permanent files for the user with the user number USER007 as of the year, month and day at x hours, x minutes and x seconds.

SAMPLE BASIC PROGRAMS F

The following sample programs illustrate some common features of BASIC and are not presented as models for programming or mathematical techniques in problem solving.

Example 1:

This program illustrates the use of the DEF and GOSUB statements to calculate the value P1 by evaluation of a series:

Program:

```
10 PRINT "CALCULATE A VALUE FØR PI"
20 PRINT
25 Z=100000
26 PRINT "NUMBER ØF ITERATIØNS";Z
27 PRINT
30 A=1
40 B=3
50 DEF FNA(D)=(1/D)
60 DEF FNB(D)=(D-FNA(B))
70 DEF FNC(D)=(D+FNA(B))
80 FØR I=1TØZ
90 A=FNB(A)
100 GØSUB 150
110 A=FNC(A)
120 GØSUB 150
130 NEXT I
140 GØTØ 170
150 B=B+2
160 RETURN
170 PRINT "PI=";4*A
200 END
READY.

RUN

 74/09/06. 11.43.28.
PRØGRAM   EXAMP

CALCULATE A VALUE FØR PI

NUMBER ØF ITERATIØNS 100000

PI= 3.1416
```

Example 2:

This job illustrates the use of a FOR--NEXT loop to calculate a table of factorials.

Program:

```
10 A=1
50 Z=20
60 FØR I=1TØ Z
70 A=A*I
75 PRINT "FACTØRIAL";I,A
80 NEXT I
100 END
READY.

RUN

 74/09/06. 11.45.31.
PRØGRAM   EXAMP2




FACTØRIAL 1      1
FACTØRIAL 2      2
FACTØRIAL 3      6
FACTØRIAL 4      24
FACTØRIAL 5      120
FACTØRIAL 6      720
FACTØRIAL 7      5040
FACTØRIAL 8      40320
FACTØRIAL 9      362880
FACTØRIAL 10     3628800
FACTØRIAL 11     39916800
FACTØRIAL 12     479001600
FACTØRIAL 13     6.22702E+9
FACTØRIAL 14     8.71783E+10
FACTØRIAL 15     1.30767E+12
FACTØRIAL 16     2.09228E+13
FACTØRIAL 17     3.55687E+14
FACTØRIAL 18     6.40237E+15
FACTØRIAL 19     1.21645E+17
FACTØRIAL 20     2.43290E+18
```

Example 3:

This job illustrates the sorting of a list of names (string variables) into alphabetic order.

Program:

```
5 PRINT "UNSØRTED LIST"
10 READ N
20 FØR I=1TØ N
30 READ A$(I)
40 PRINT A$(I)
50 NEXT I
60 FØR I=1TØ N-1
70 FØR J=I+1TØ N
80 IF A$(I)<A$(J) THEN 120
90 LET T$=A$(I)
100 LET A$(I)=A$(J)
110 LET A$(J)=T$
120 NEXT J
130 NEXT I
135 PRINT
140 PRINT "SØRTED LIST"
150 FØR I=1TØ N
160 PRINT A$(I)
170 NEXT I
180 STØP
200 DATA 8
210 DATA MARY,JØHN,SUE,JØE,JACK,BILL,TED,ANN
READY.

RUN

 74/09/06. 11.48.02.
PRØGRAM   EXAMP3



UNSØRTED LIST
MARY
JØHN
SUE
JØE
JACK
BILL
TED
ANN

SØRTED LIST
ANN
BILL
JACK
JØE
JØHN
MARY
SUE
TED
```

Example 4:

This job illustrates the inversion of a Hilbert Matrix (n by n) using BASIC matrix operations.

Program:

```
10 DIM A(20,20),B(20,20)
20 READ N
30 MAT A=CØN(N,N)
40 MAT B=CØN(N,N)
50 FØR I=1TØ N
60 FØR J=1TØ N
70 LET A(I,J)=1/(I+J-1)
80 NEXT J
90 NEXT I
100 MAT B=INV(A)
110 MAT PRINT B;
190 DATA 4
READY.

RUN

 74/09/06. 11.49.28.
PRØGRAM   EXAMP4




 16.  -120.  240.  -140.

-120.  1200.  -2700.  1680.

 240.  -2700.  6480.  -4200.

-140.  1680.  -4200.  2800.
```

INDEX

COMMENT SHEET

MANUAL TITLE CONTROL DATA® BASIC LANGUAGE REFERENCE MANUAL
VERSION 2.1 FOR KRONOS 2.1, NOS 1.0 AND SCOPE 3.4

PUBLICATION NO. 19980300 REVISION B

FROM: NAME: ______

BUSINESS ADDRESS: ______

COMMENTS:

This form is not intended to be used as an order blank. Your evaluation of this manual will be welcomed by Control Data Corporation. Any errors, suggested additions or deletions, or general comments may be made below. Please include page number references and fill in publication revision level as shown by the last entry on the Record of Revision page at the front of the manual. Customer engineers are urged to use the TAR.

CUT ALONG LINE

AA3419 REV. 11/69 PRINTED IN USA

NO POSTAGE STAMP NECESSARY IF MAILED IN U. S. A.

FOLD ON DOTTED LINES AND STAPLE

STAPLE STAPLE

FOLD FOLD

FIRST CLASS
PERMIT NO. 8241

MINNEAPOLIS, MINN.

BUSINESS REPLY MAIL
NO POSTAGE STAMP NECESSARY IF MAILED IN U.S.A.

POSTAGE WILL BE PAID BY

CONTROL DATA CORPORATION
Technical Publications Department
4201 North Lexington Avenue
Arden Hills, Minnesota 55112

CUT ALONG LINE

FOLD FOLD

CONTROL DATA

CORPORATION

CORPORATE HEADQUARTERS, 8100 34th AVE. SO., MINNEAPOLIS, MINN. 55440
SALES OFFICES AND SERVICE CENTERS IN MAJOR CITIES THROUGHOUT THE WORLD

LITHO IN U.S.A.